Reversing: Secrets of Reverse Engineering

Eldad Eilam
Reversing: Secrets of Reverse Engineering
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
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 We run setup utilities that install numerous files, change system settings, delete or disable older versions and superceded utilities, and modify critical registry files.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

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

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
Chapter 4 Reversing Tools

Different Reversing Approaches
   Offline Code Analysis (Dead-Listing)
   Live Code Analysis
Disassemblers
   IDA Pro
   ILDasm
Debuggers
   User-Mode Debuggers
      OllyDbg
   User Debugging in WinDbg
   IDA Pro
   PEBrowse Professional Interactive
 Kernel-Mode Debuggers
   Kernel Debugging in WinDbg
   Numega SoftICE
   Kernel Debugging on Virtual Machines
Decompilers
System-Monitoring Tools
Patching Tools
   Hex Workshop
Miscellaneous Reversing Tools
   Executable-Dumping Tools
      DUMPBIN
      PEView
   PEBrowse Professional
Conclusion
**Part II  Applied Reversing**

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

**Chapter 6  Deciphering File Formats**

<table>
<thead>
<tr>
<th></th>
<th>199</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptex</td>
<td>200</td>
</tr>
<tr>
<td>Using Cryptex</td>
<td>201</td>
</tr>
<tr>
<td>Reversing Cryptex</td>
<td>202</td>
</tr>
<tr>
<td>The Password Verification Process</td>
<td>207</td>
</tr>
<tr>
<td>Catching the “Bad Password” Message</td>
<td>207</td>
</tr>
<tr>
<td>The Password Transformation Algorithm</td>
<td>210</td>
</tr>
<tr>
<td>Hashing the Password</td>
<td>213</td>
</tr>
<tr>
<td>The Directory Layout</td>
<td>218</td>
</tr>
<tr>
<td>Analyzing the Directory Processing Code</td>
<td>218</td>
</tr>
<tr>
<td>Analyzing a File Entry</td>
<td>223</td>
</tr>
<tr>
<td>Dumping the Directory Layout</td>
<td>227</td>
</tr>
<tr>
<td>The File Extraction Process</td>
<td>228</td>
</tr>
<tr>
<td>Scanning the File List</td>
<td>234</td>
</tr>
<tr>
<td>Decrypting the File</td>
<td>235</td>
</tr>
<tr>
<td>The Floating-Point Sequence</td>
<td>236</td>
</tr>
<tr>
<td>The Decryption Loop</td>
<td>238</td>
</tr>
<tr>
<td>Verifying the Hash Value</td>
<td>239</td>
</tr>
<tr>
<td>The Big Picture</td>
<td>239</td>
</tr>
<tr>
<td>Digging Deeper</td>
<td>241</td>
</tr>
<tr>
<td>Conclusion</td>
<td>242</td>
</tr>
</tbody>
</table>
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
  CVaVariableSet::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
Conclusion  306
### Part III  Cracking

#### Chapter 9  Piracy and Copy Protection

- 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

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

**Appendix A** Deciphering Code Structures 479

**Appendix B** Understanding Compiled Arithmetic 519

**Appendix C** Deciphering Program Data 537

Index 561
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 is 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.