Rootkits: Subverting the Windows Kernel

By Greg Hoglund, James Butler

.....
Publisher: **Addison Wesley Professional**
Pub Date: **July 22, 2005**
ISBN: **0-321-29431-9**
Pages: **352**

Table of Contents | Index

Overview

"It's imperative that everybody working in the field of cyber-security read this book to understand the growing threat of
Magazine

"This material is not only up-to-date, it defines up-to-date. It is truly cutting-edge. As the only book on the subject, ***Ro***
programmer. It's detailed, well researched and the technical information is excellent. The level of technical detail, resear
word: Outstanding." --*Tony Bautts, Security Consultant; CEO, Xtivix, Inc.*

"This book is an essential read for anyone responsible for Windows security. Security professionals, Windows system a
techniques used by rootkit authors. At a time when many IT and security professionals are still worrying about the lates
Hoglund and Mr. Butler open your eyes to some of the most stealthy and significant threats to the Windows operating s
defend the networks and systems for which you are responsible." --*Jennifer Kolde, Security Consultant, Author, an*

"What's worse than being owned? Not knowing it. Find out what it means to be owned by reading Hoglund and Butle
toolset--which includes decompilers, disassemblers, fault-injection engines, kernel debuggers, payload collections, cove
Software left off, this book shows how attackers hide in plain sight. "Rootkits are extremely powerful and are the next
on stealthiness. They hide away from standard system observers, employing hooks, trampolines, and patches to get the
that usually monitor machine behavior can't easily detect them. A rootkit thus provides insider access only to people wh
can hide files and running processes to provide a backdoor into the target machine. "Understanding the ultimate attacke
systems. No authors are better suited to give you a detailed hands-on understanding of rootkits than Hoglund and Butl
CTO, Cigital, coauthor of Exploiting Software *(2004) and* Building Secure Software *(2002), both from Addison-W*

"Greg and Jamie are unquestionably the go-to experts when it comes to subverting the Windows API and creating roo
surrounding rootkits, bringing this information out of the shadows. Anyone even remotely interested in security for Wind
on their must-read list." --*Harlan Carvey, author of* Windows Forensics and Incident Recovery *(Addison-Wesley, 20*

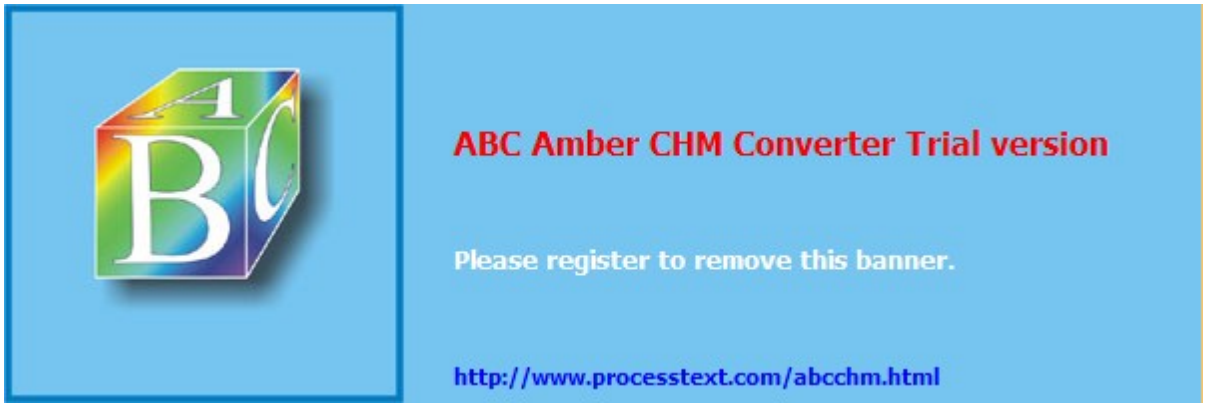
Rootkits are the ultimate backdoor, giving hackers ongoing and virtually undetectable access to the systems they explo
comprehensive guide to rootkits: what they are, how they work, how to build them, and how to detect them. Rootkit.c
course in rootkits. In this book, they reveal never-before-told offensive aspects of rootkit technology--learn how attack

Hoglund and Butler show exactly how to subvert the Windows XP and Windows 2000 kernels, teaching concepts that
Server 2003 to Linux and UNIX. Using extensive downloadable examples, they teach rootkit programming techniques
operating system drivers and debuggers.

After reading this book, readers will be able to

- Understand the role of rootkits in remote command/control and software eavesdropping
- Build kernel rootkits that can make processes, files, and directories invisible
- Master key rootkit programming techniques, including hooking, runtime patching, and directly manipulating ker
- Work with layered drivers to implement keyboard sniffers and file filters
- Detect rootkits and build host-based intrusion prevention software that resists rootkit attacks

Visit rootkit.com for code and programs from this book. The site also contains enhancements to the book's text, such a



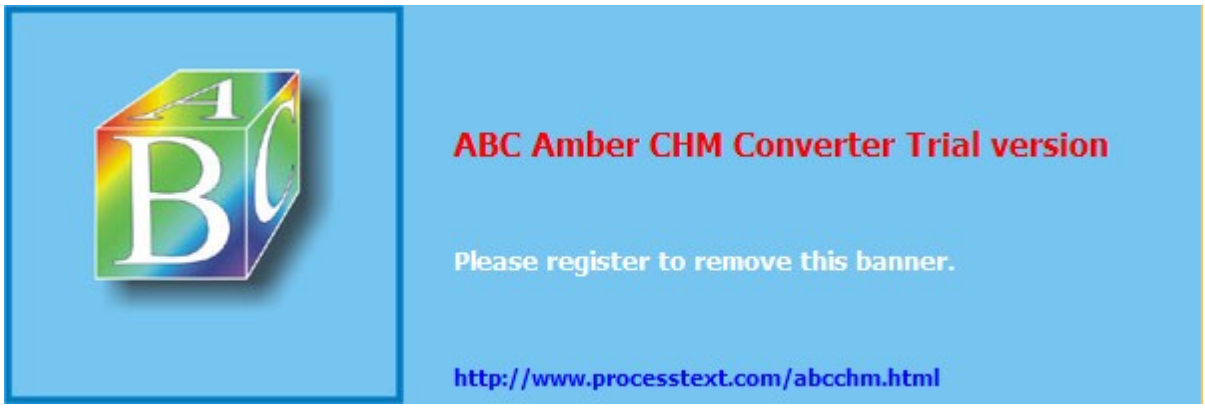
Rootkits: Subverting the Windows Kernel

By Greg Hoglund, James Butler

Publisher: **Addison Wesley Professional**
Pub Date: **July 22, 2005**
ISBN: **0-321-29431-9**
Pages: **352**

Table of Contents | Index

PREV < Day Day Up > NEXT



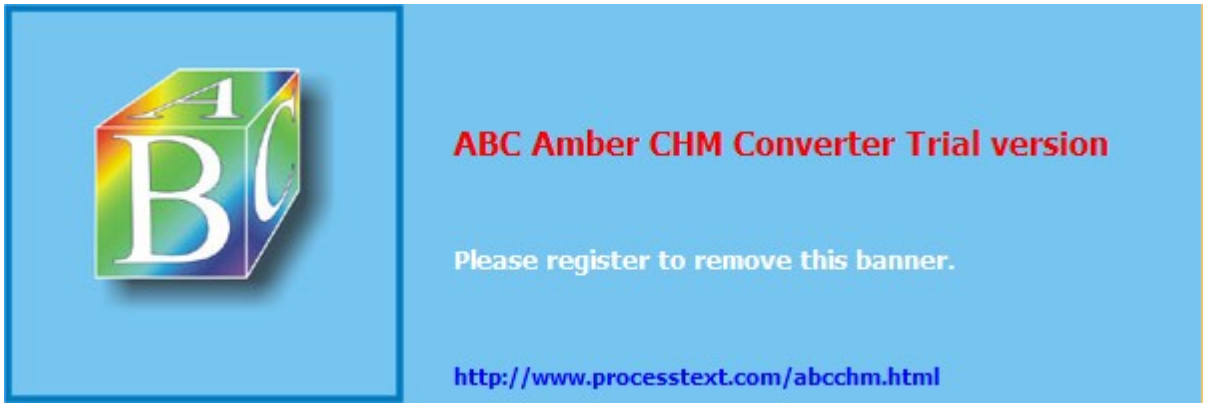
Copyright

Library of Congress Cataloging-in-Publication Data

Hoglund, Greg.
Rootkits: subverting the Windows kernel / Greg Hoglund, James Butler.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-29431-9 (pbk. : alk. paper)
1. Microsoft Windows (Computer file) 2. Computers?Access control. 3. Computer security.
I. Butler, James. II. Title.
QA76.9.A25H637 2005
005.8?dc22 2005013061

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, July 2005

Dedication



Praise for *Rootkits*

"It's imperative that everybody working in the field of cyber-security read this book to understand the growing threat of rootkits."

?*Mark Russinovich, editor*, Windows IT Pro / Windows & .NET Magazine

"This material is not only up-to-date, it defines up-to-date. It is truly cutting-edge. As the only book on the subject, *Rootkits* will be of interest to any Windows security researcher or security programmer. It's detailed, well researched and the technical information is excellent. The level of technical detail, research, and time invested in developing relevant examples is impressive. In one word: Outstanding."

?*Tony Bautts*
Security Consultant; CEO, Xtivix, Inc.

"This book is an essential read for anyone responsible for Windows security. Security professionals, Windows system administrators, and programmers in general will want to understand the techniques used by rootkit authors. At a time when many IT and security professionals are still worrying about the latest e-mail virus or how to get all of this month's security patches installed, Mr. Hoglund and Mr. Butler open your eyes to some of the most stealthy and significant threats to the Windows operating system. Only by understanding these offensive techniques can you properly defend the networks and systems for which you are responsible."

?*Jennifer Kolde*
Security Consultant, Author, and Instructor

"What's worse than being owned? Not knowing it.

"Find out what it means to be owned by reading Hoglund and Butler's first-of-a-kind book on rootkits. At the apex the malicious hacker toolset?which includes decompilers, disassemblers, fault-injection engines, kernel debuggers, payload collections, coverage tools, and flow analysis tools?is the rootkit. Beginning where *Exploiting Software* left off, this book shows how attackers hide in plain sight.

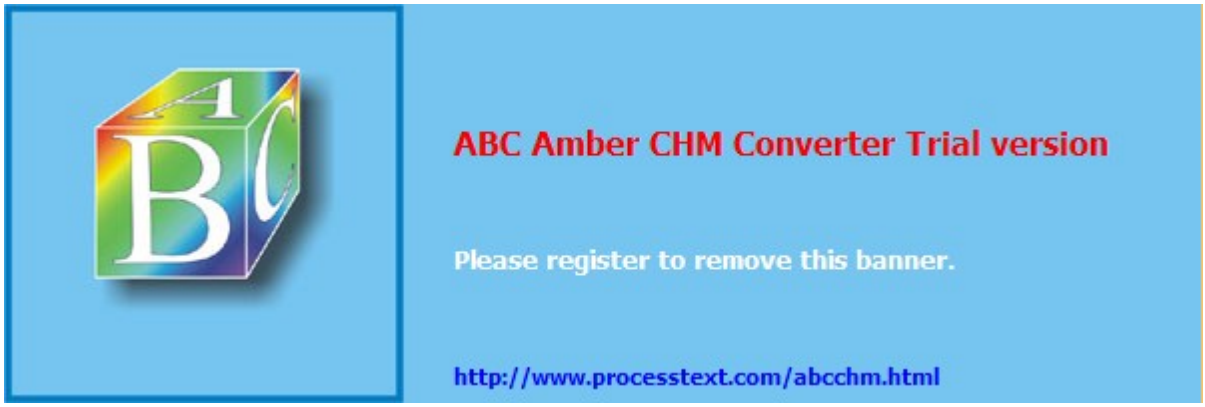
"Rootkits are extremely powerful and are the next wave of attack technology. Like other types of malicious code, rootkits thrive on stealthiness. They hide away from standard system observers, employing hooks, trampolines, and patches to get their work done. Sophisticated rootkits run in such a way that other programs that usually monitor machine behavior can't easily detect them. A rootkit thus provides insider access only to people who know that it is running and available to accept commands. Kernel rootkits can hide files and running processes to provide a backdoor into the target machine.

"Understanding the ultimate attacker's tool provides an important motivator for those of us trying to defend systems. No authors are better suited to give you a detailed hands-on understanding of rootkits than Hoglund and Butler. Better to own this book than to be owned."

?*Gary McGraw, Ph.D., CTO, Cigital, coauthor of Exploiting Software (2004) and Building Secure Software (2002), both from Addison-Wesley*

"Greg and Jamie are unquestionably the go-to experts when it comes to subverting the Windows API and creating rootkits. These two masters come together to pierce the veil of mystery surrounding rootkits, bringing this information out of the shadows. Anyone even remotely interested in security for Windows systems, including forensic analysis, should include this book very high on their must-read list."

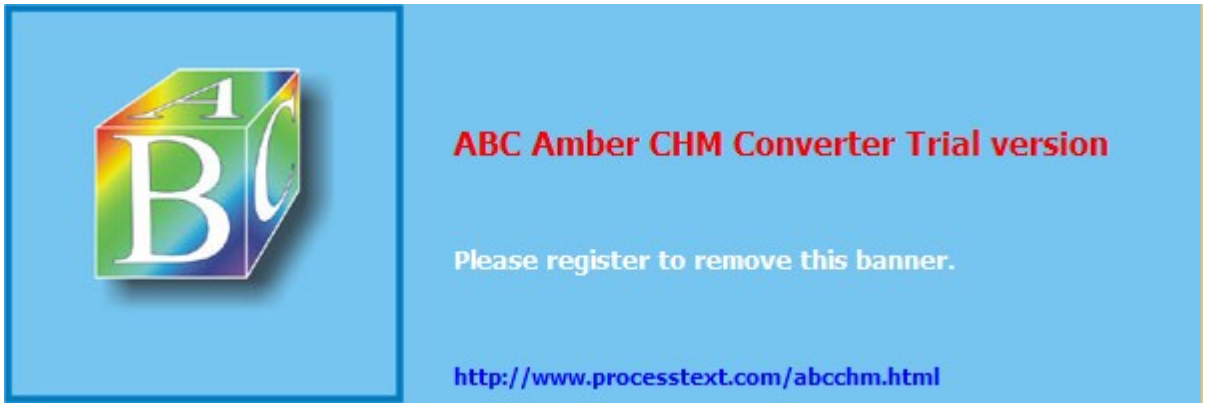
?*Harlan Carvey, author of* Windows Forensics and Incident Recovery (*Addison-Wesley, 2005)*



Preface

A rootkit is *a set of programs and code that allows a permanent and undetectable presence on a computer.*

PREV < Day Day Up > NEXT



Historical Background

We became interested in rootkits because of our professional work in computer security, but the pursuit of the subject quickly expanded into a personal mission (also known as late nights and weekends). This led Hoglund to found rootkit.com, a forum devoted to reverse engineering and rootkit development. Both of us are deeply involved with rootkit.com. Butler first contacted Hoglund online through this Web site because Butler had a new and powerful rootkit called FU that needed testing,[1] Butler sent Hoglund some source code and a pre-compiled binary. However, by accident, he did not send Hoglund the source code to the kernel driver. To Butler's amazement, Hoglund just loaded the pre-compiled rootkit onto his workstation without question, and reported back that FU seemed to be working fine! Our trust in one another has only grown since then.[2]

[1] Butler was not interested in rootkits for malicious purposes. He was instead fascinated with the power of kernel modifications. This led Butler to develop one of the first rootkit-detection programs, VICE.

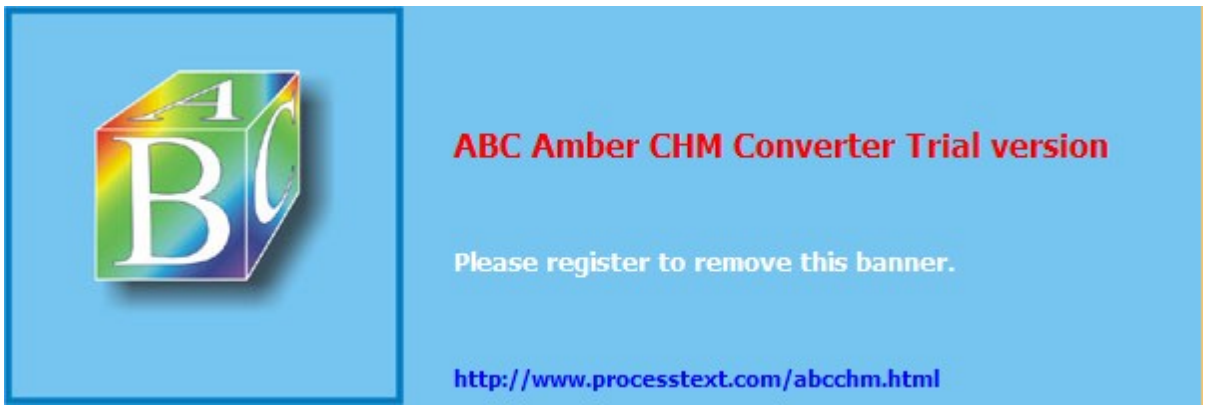
[2] Hoglund still wonders, from time to time, whether that original version of FU is still running on his workstation.

Both of us have long been driven by an almost perverse need to reverse-engineer the Windows kernel. It's like when someone says we can't do something?then we accomplish it. It is very satisfying learning how so-called computer security products work and finding ways around them. This inevitably leads to better protection mechanisms.

The fact that a product claims to provide some level of protection does not necessarily mean it actually does. By playing the part of an attacker, we are always at an advantage. As the attacker we must think of only one thing that a defender didn't consider. Defenders, on the other hand, must think of every possible thing an attacker might do. The numbers work in the attacker's favor.

We teamed up a few years ago to offer the training class "Offensive Aspects of Rootkit Technology." This training started as a single day of material that since has grown to include hundreds of pages of notes and example code. The material for the class eventually became the foundation for this book. We now offer the rootkit training class several times a year at the Black Hat security conference, and also privately.

After training for awhile, we decided to deepen our relationship, and we now work together at HBGary, Inc. At HBGary, we tackle very complex rootkit problems on a daily basis. In this book, we use our experience to cover the threats that face Windows users today, and likely will only increase in the future.

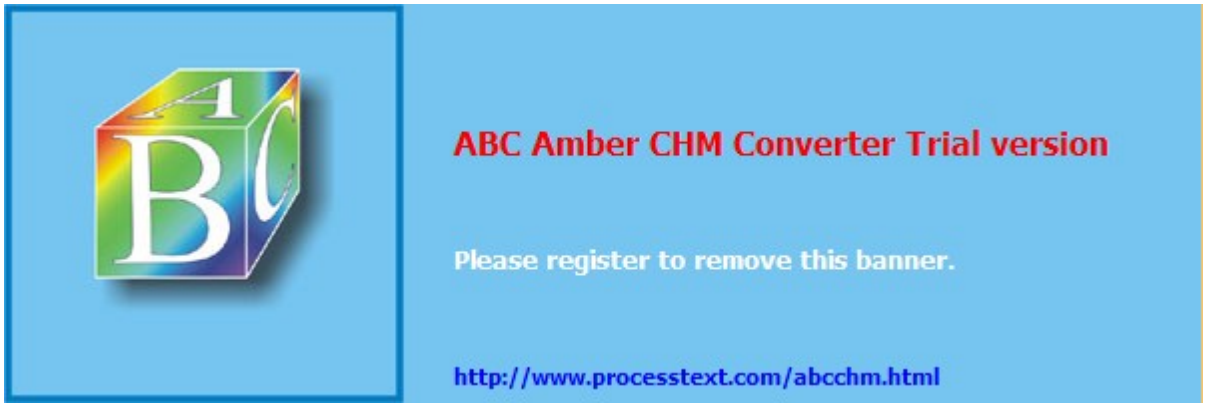


Target Audience

This book is intended for those who are interested in computer security and want a truer perspective concerning security threats. A lot has been written on how intruders gain access to computer systems, but little has been said regarding what can happen once an intruder gains that initial access. Like the title implies, this book will cover what an intruder can do to cover her presence on a compromised machine.

We believe that most software vendors, including Microsoft, do not take rootkits seriously. That is why we are publishing this book. The material in this book is not groundbreaking for someone who has worked with rootkits or operating systems for years, but for most people this book should prove that rootkits are a serious threat. It should prove that your virus scanner or desktop firewall is never good enough. It should prove that a rootkit can get into your computer and stay there for years without you ever knowing about it.

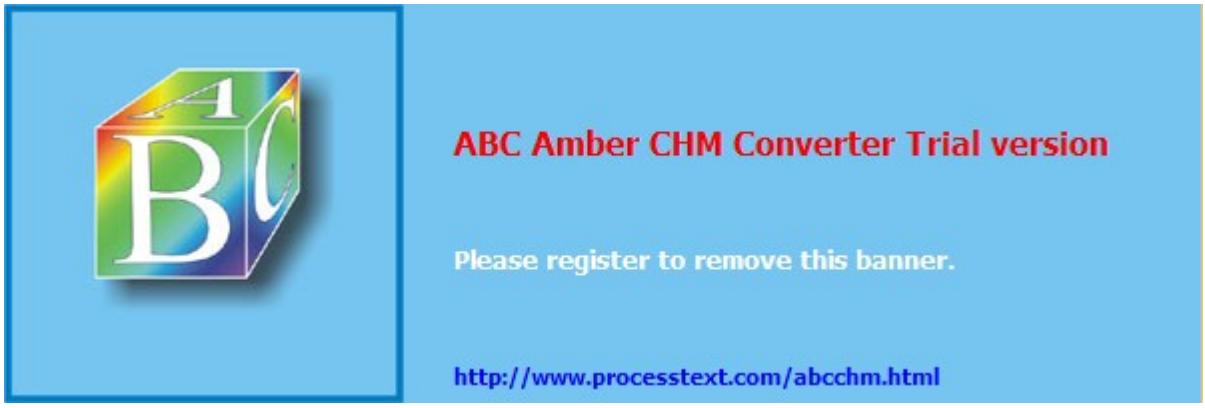
To best convey rootkit information, we wrote most of this book from an attacker's perspective; however, we end the book on a defensive posture. As you begin to learn your attackers' goals and techniques, you will begin to learn your own system's weaknesses and how to mitigate its shortcomings. Reading this book will help you improve the security of your system or help you make informed decisions when it comes to purchasing security software.



Prerequisites

As all of the code samples are written in C, you will gain more insight if you already understand basic C concepts?the most important one being pointers. If you have no programming knowledge, you should still be able to follow along and understand the threats without needing to understand the particular implementation details. Some areas of the book draw on principles from the Windows device driver architecture, but experience writing device drivers is not required. We will walk you through writing your first Windows device driver and build from there.

PREV < Day Day Up > NEXT



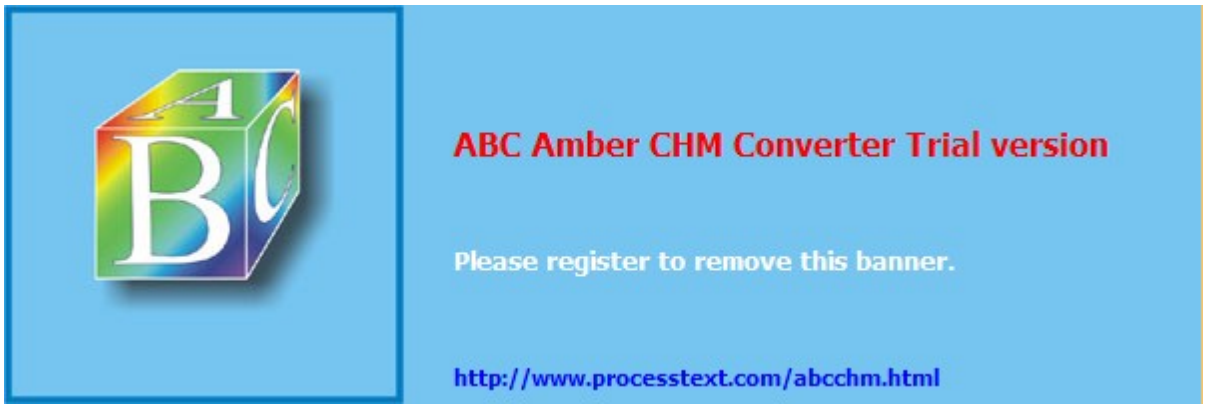
Scope

This book covers Windows rootkits, although most of the concepts apply to other operating systems as well, such as LINUX. We focus on kernel rootkits because these are the most difficult to detect. Many public rootkits for Windows are *userland* rootkits[3] because these are the easiest to implement, since they do not involve the added complexity of understanding how the undocumented kernel works.

[3] Userland rootkits are rootkits that do not employ kernel-level modifications, but instead rely only upon user-program modifications.

This book is not about specific real-world rootkits. Rather, it teaches the generic approaches used by all rootkits. In each chapter, we introduce a basic technique, explain its purposes, and show how it's implemented using code examples. Armed with this information, you should be able to expand the examples in a million different ways to perform a variety of tasks. When working in the kernel, you are really limited only by your imagination.

You can download most of the code in this book from rootkit.com. Throughout the book, we will reference the particular URL for each individual example. Other rootkit authors also publish research at rootkit.com that you may find useful for keeping up with the latest discoveries.



Acknowledgments

We could not have written this book on our own. Many people have helped further our understanding of computer security throughout the years. We would like to thank the community of colleagues and users at rootkit.com. Special thanks also go to all the students who have taken our rootkit class, "Offensive Aspects of Rootkit Technology." We learn something new every time we teach it.

The following people provided helpful reviews of early drafts of this book: Tony Bautts, Richard Bejtlich, Harlan Carvey, Graham Clark, Greg Cummings, Jeremy Epstein, Jennifer Kolde, Marcus Leech, Gary McGraw, and Sherri Sparks. Special thanks to Audrey Doyle, who helped tremendously with developing the book under an extreme time schedule.

Finally, we owe our gratitude to our editor, Karen Gettman, and her assistant, Ebony Haight, at Addison-Wesley. Thank you for being flexible with our crazy schedules and distances of two time zones and 3000+ miles. You were largely successful keeping our attention on the book. Both of you provided everything we needed to be successful writing the book.

Greg and Jamie

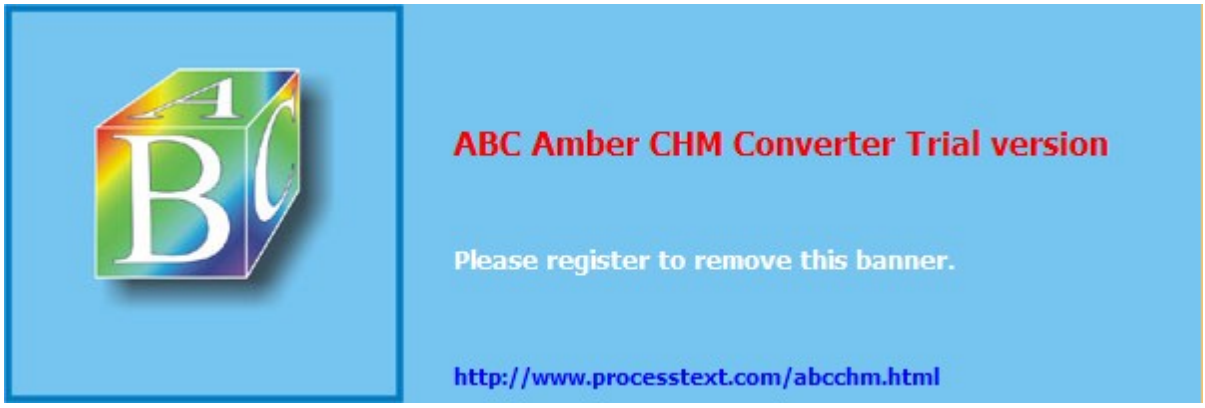
PREV < Day Day Up > NEXT

About the Authors

Greg Hoglund has been a pioneer in the area of software security. He is CEO of HBGary, Inc., a leading provider of software security verification services. After writing one of the first network vulnerability scanners (installed in over half of all Fortune 500 companies), he created and documented the first Windows NT-based rootkit, founding www.rootkit.com in the process. Greg is a frequent speaker at Black Hat, RSA, and other security conferences. He coauthored the bestselling *Exploiting Software: How to Break Code* (Addison-Wesley, 2004).

James Butler, Director of Engineering at HBGary, has a world-class talent for kernel programming and rootkit development and extensive experience in host-based intrusion-detection systems. He is the developer of VICE, a rootkit detection and forensics system. Jamie's previous positions include Senior Security Software Engineer at Enterasys and Computer Scientist at the National Security Agency. He is a frequent trainer and speaker at Black Hat security conferences. He holds a masters of computer science from the University of Maryland, Baltimore County. He has published articles in the *IEEE Information Assurance Workshop, Phrack, USENIX ;login:*, and *Information Management and Computer Security*.

PREV < Day Day Up > NEXT



About the Cover

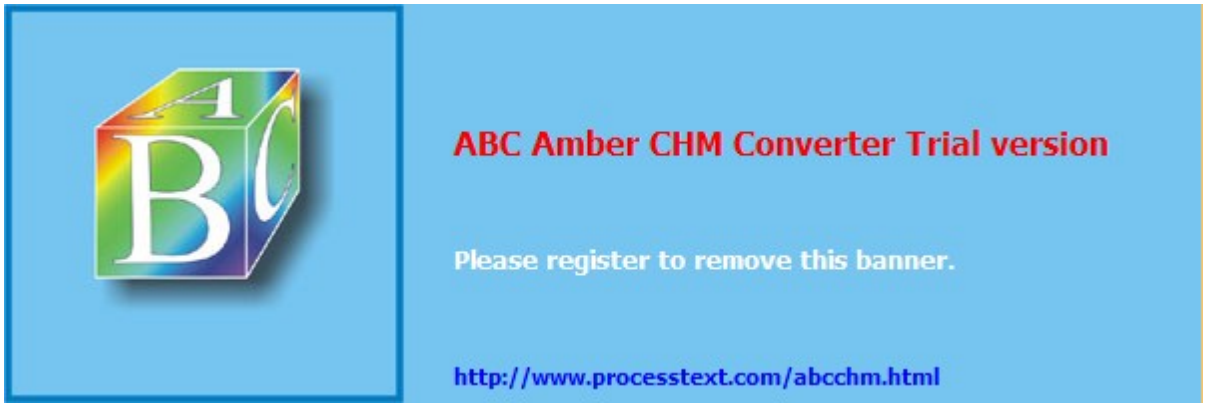
The front cover of this book holds a lot of significance for Jamie and me. We designed this cover ourselves, with the help of a wonderfully talented Brazilian artist named Paulo. The person depicted on the front is a historical Japanese figure called a *Samurai*. (We mean no disrespect by taking some creative license in depicting the character.) We chose him because he represents the artistry of his craft, strength of character, and the fact that his art was essential to his culture and its leaders. He also represents the importance of recognizing the interconnectedness of the world in which we live.

The sword is the tool of the Samurai, the object of his skill. You'll notice that his sword is centered in the picture, and driven into the ground. From the sword springs roots that signify growth and depth of knowledge. The roots become circuits to represent knowledge of computer technology and the tools of the rootkit developer. The kanji characters behind him mean "to gain knowledge."

We think this is an apt description of our work. Jamie and I are continually learning and updating our knowledge. We are pleased to be able to impart what we've learned to others. We want you to see the incredible power that rests in the roots you can create.

Greg Hoglund

PREV < Day Day Up > NEXT



Chapter 1. Leave No Trace

Subtle and insubstantial, the expert leaves no trace; divinely mysterious, he is inaudible. Thus he is the master of his enemy's fate.

SUN TZU

Many books discuss how to penetrate computer systems and software. Many authors have already covered how to run hacker scripts, write buffer-overflow exploits, and craft shellcode. Notable examples include the texts *Exploiting Software*,[1] *The Shellcoder's Handbook*,[2] and *Hacking Exposed*.[3]

[1] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code* (Boston: Addison-Wesley, 2004). *See also* www.exploitingsoftware.com

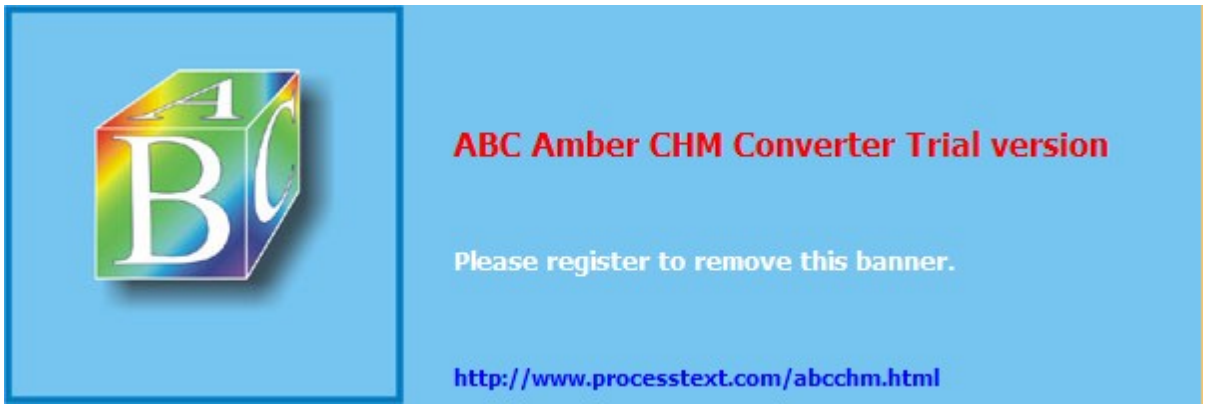
[2] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, *The Shellcoder's Handbook* (New York: John Wiley & Sons, 2004).

[3] S. McClure, J. Scambray, and G. Kurtz, *Hacking Exposed* (New York: McGraw-Hill, 2003).

This book is different. Instead of covering the attacks, this book will teach you how attackers stay in *after* the break-in. With the exception of computer forensics books, few discuss what to do after a successful penetration. In the case of forensics, the discussion is a defensive one: how to detect the attacker and how to reverse-engineer malicious code. In this book we take an offensive approach. This book is about penetrating a computer system without being detected. After all, for a penetration to be successful over time, it cannot be detected.

In this chapter we will introduce you to rootkit technology and the general principals of how it works. Rootkits are only part of the computer-security spectrum, but they are critical for many attacks to be successful.

Rootkits are not, in and of themselves, malicious. However, rootkits can be used by malicious programs. Understanding rootkit technology is critical if you are to defend against modern attacks.



Understanding Attackers' Motives

A *back door* in a computer is a secret way to get access. Back doors have been popularized in many Hollywood movies as a secret password or method for getting access to a highly secure computer system. But back doors are not just for the silver screen—they are very real, and can be used for stealing data, monitoring users, and launching attacks deep into computer networks.

An attacker might leave a back door on a computer for many reasons. Breaking into a computer system is hard work, so once an attacker succeeds, she will want to keep the ground she has gained. She may also want to use the compromised computer to launch additional attacks deeper into the network.

A major reason attackers penetrate computers is to gather intelligence. To gather intelligence, the attacker will want to monitor keystrokes, observe behavior over time, sniff packets from the network, and *exfiltrate*[4] data from the target. All of this requires establishing a back door of some kind. The attacker will want to leave software running on the target system that can perform intelligence gathering.

[4] *Exfiltrate:* To transport out of, to remove from a location; to transport a copy of data from one location to another.

Attackers also penetrate computers to destroy them, in which case the attacker might leave a *logic bomb* on the computer, which she has set to destroy the computer at a specific time. While the bomb waits, it needs to stay undetected. Even if the attacker does not require subsequent back-door access to the system, this is a case where software is left behind and it must remain undetected.

The Role of Stealth

To remain undetected, a back-door program must use stealth. Unfortunately, most publicly available "hacker" back-door programs aren't terribly stealthy. Many things can go wrong. This is mostly because the developers want to build everything including the proverbial kitchen sink into a back-door program. For example, take a look at the Back Orifice or NetBus programs. These back-door programs sport impressive lists of features, some as foolish as ejecting your CD-ROM tray. This is fun for office humor, but not a function that would be used in a professional attack operation.[5] If the attacker is not careful, she may reveal her presence on the network, and the whole operation may sour. Because of this, professional attack operations usually require specific and automated back-door programs—programs that do only one thing and nothing else. This provides assurance of consistent results.

[5] *Professional* in this case indicates a sanctioned operation of some kind, as performed, for example, by law enforcement, pen testers, red teams, or the equivalent.

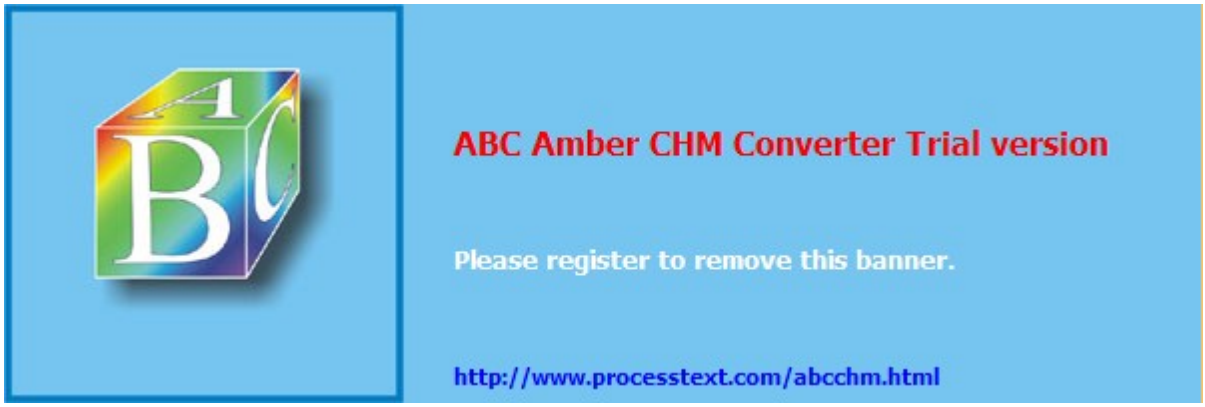
If computer operators suspect that their computer or network has been penetrated, they may perform forensic discovery, looking for unusual activity or back-door programs.[6] The best way to counter forensics is with stealth: If no attack is suspected, then no forensics are likely to be applied to the system. Attackers may use stealth in different ways. Some may simply try to step lightly by keeping network traffic to a minimum and avoiding storing files on the hard drive. Others may store files but employ obfuscation techniques that make forensics more difficult. If stealth is used properly, forensics will never be applied to a compromised system, because the intrusion will not have been detected. Even if an attack is suspected and forensics end up being used a good stealth attack will store data in obfuscated ways to escape detection.

[6] For a good text on computer forensics, *see* D. Farmer and W. Venema, *Forensic Discovery* (Boston: Addison-Wesley, 2004).

When Stealth Doesn't Matter

Sometimes an attacker doesn't need to be stealthy. For instance, if the attacker wants to penetrate a computer only long enough to steal something, such as an e-mail spool, perhaps she doesn't care if the attack is eventually detected.

Another time when stealth is not required is when the attacker simply wants to crash the target computer.



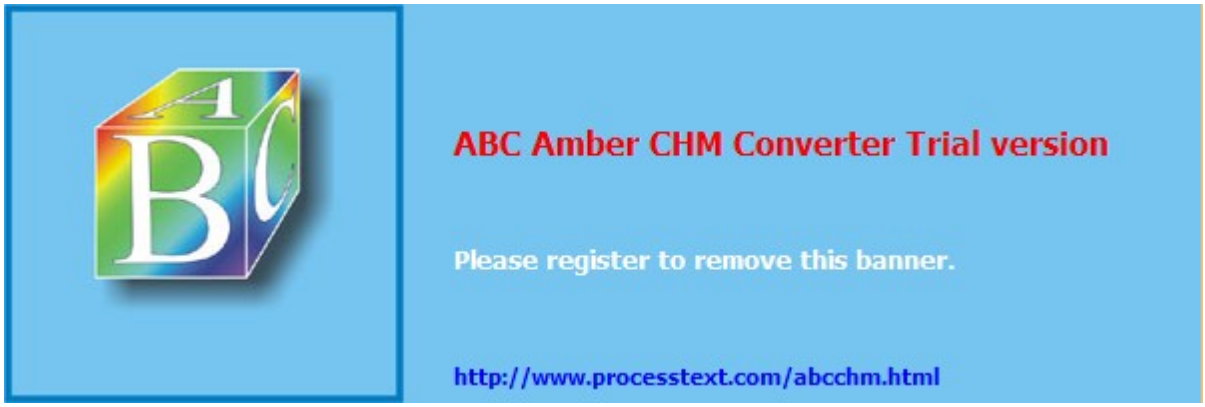
What Is a Rootkit?

The term *rootkit* has been around for more than 10 years. A rootkit is a "kit" consisting of small and useful programs that allow an attacker to maintain access to "root," the most powerful user on a computer. In other words, *a rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer*.

In our definition of "rootkit," the key word is "undetectable." Most of the technology and tricks employed by a rootkit are designed to hide code and data on a system. For example, many rootkits can hide files and directories. Other features in a rootkit are usually for remote access and eavesdropping?for instance, for sniffing packets from the network. When combined, these features deliver a knockout punch to security.

Rootkits are not inherently "bad," and they are not always used by the "bad guys." It is important to understand that a rootkit is just a technology. Good or bad intent derives from the humans who use them. There are plenty of legitimate commercial programs that provide remote administration and even eavesdropping features. Some of these programs even use stealth. In many ways, these programs could be called rootkits. Law enforcement may use the term "rootkit" to refer to a sanctioned back-door program?something installed on a target with legal permission from the state, perhaps via court order. (We cover such uses in the section Legitimate Uses of Rootkits later in this chapter.) Large corporations also use rootkit technology to monitor and enforce their computer-use regulations.

By taking the attacker's perspective, we guide you through your enemies' skills and techniques. This will increase your skills in defending against the rootkit threat. If you are a legitimate developer of rootkit technology, this book will help you build a base of skills that you can expand upon.



Why Do Rootkits Exist?

Rootkits are a relatively recent invention, but spies are as old as war. Rootkits exist for the same reasons that audio bugs exist. People want to see or control what other people are doing. With the huge and growing reliance on data processing, computers are natural targets.

Rootkits are useful only if you want to maintain access to a system. If all you want to do is steal something and leave, there is no reason to leave a rootkit behind. In fact, leaving a rootkit behind always opens you to the risk of detection. If you steal something and clean up the system, you may leave no trace of your operation.

Rootkits provide two primary functions: remote command and control, and software eavesdropping.

Remote Command and Control

Remote command and control (or simply "remote control") can include control over files, causing reboots or "Blue Screens of Death," and accessing the command shell (that is, cmd.exe or /bin/sh). Figure 1-1 shows an example of a rootkit command menu. This command menu will give you an idea of the kinds of features a rootkit might include.

Figure 1-1. Menu for a kernel rootkit.

```
Win2K Rootkit by the team rootkit.com
Version 0.4 alpha
----------------------------------------
command       description
ps            show process list
help          this data
buffertest    debug output
hidedir       hide prefixed file or directory
hideproc      hide prefixed processes
debugint      (BSOD)fire int3
sniffkeys     toggle keyboard sniffer

echo <string> echo the given string
*"(BSOD)" means Blue Screen of Death
  if a kernel debugger is not present!
*"prefixed" means the process or filename
  starts with the letters '_root_'.
*"sniffer" means listening or monitoring software.
```

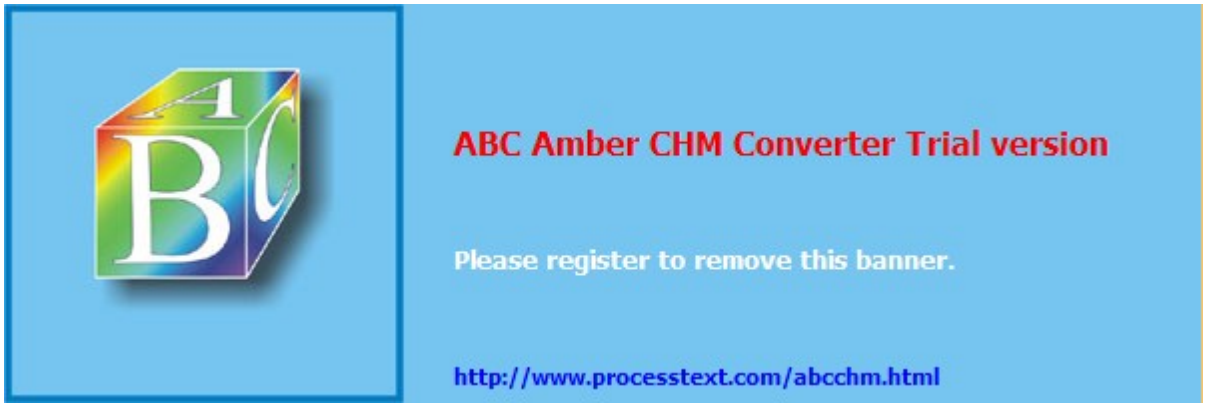
Software Eavesdropping

Software eavesdropping is all about watching what people do. This means sniffing packets, intercepting keystrokes, and reading e-mail. An attacker can use these techniques to capture passwords and decrypted files, or even cryptographic keys.

Cyberwarfare

While rootkits have applications in waging digital warfare, they are not the first application of the concept.

Wars are fought on many fronts, not the least of which is economic. From the end of World War II through the Cold War, the USSR mounted a large intelligence-gathering operation against the U.S. to obtain technology.[7]



How Long Have Rootkits Been Around?

As we noted previously, rootkits are not a new concept. In fact, many of the methods used in modern rootkits are the same methods used in viruses in the 1980s?for example, modifying key system tables, memory, and program logic. In the late 1980s, a virus might have used these techniques to hide from a virus scanner. The viruses during this era used floppy disks and BBS's (bulletin board systems) to spread infected programs.

When Microsoft introduced Windows NT, the memory model was changed so that normal user programs could no longer modify key system tables. A lapse in hard virus technology followed, because no virus authors were using the new Windows kernel.

When the Internet began to catch on, it was dominated by UNIX operating systems. Most computers used variants of UNIX, and viruses were uncommon. However, this is also when network worms were born. With the famous Morris Worm, the computing world woke up to the possibility of software exploits.[11] During the early 1990s, many hackers figured out how to find and exploit buffer overflows, the "nuclear bomb" of all exploits. However, the virus-writing community didn't catch on for almost a decade.

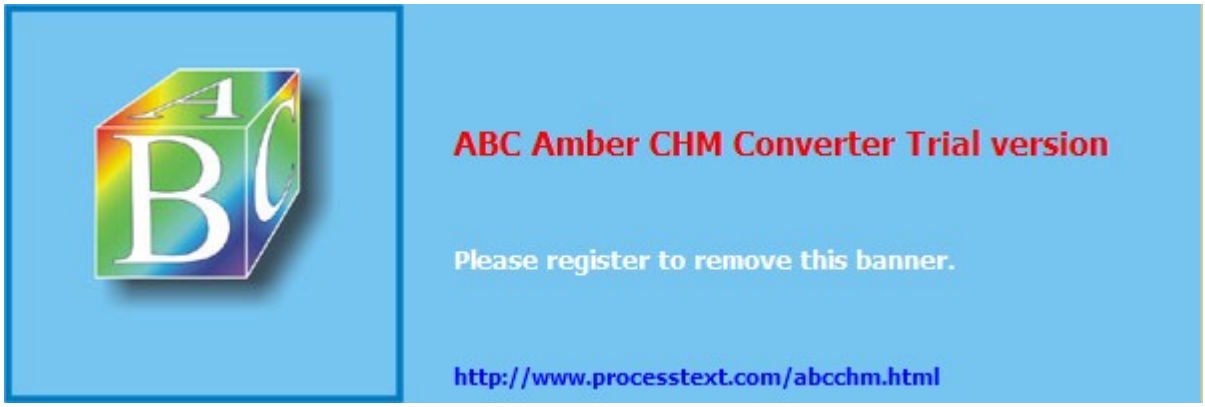
[11] Robert Morris released the first documented Internet worm. For an account of the Morris Worm, *see* K. Hafner and J. Markoff, *Cyberpunk: Outlaws and Hackers on the Computer Frontier* (New York: Simon & Schuster, 1991).

During the early 1990s, a hacker would penetrate a system, set up camp, and then use the freshly compromised computer to launch new attacks. Once a hacker had penetrated a computer, she needed to maintain access. Thus, the first rootkits were born. These original rootkits were merely backdoor programs, and they used very little stealth. In some cases, they replaced key system binaries with modified versions that would hide files and processes. For example, consider a program called ls that lists files and directories. A first-generation rootkit might replace the ls program with a Trojan version that hides any file named hacker_stuff. Then, the hacker would simply store all of her suspect data in a file named hacker_stuff. The modified ls program would keep the data from being revealed.

System administrators at that time responded by writing programs such as Tripwire[12] that could detect whether files had been changed. Using our previous example, a security utility like Tripwire could examine the ls program and determine that it had been altered, and the Trojan would be unmasked.

[12] www.tripwire.org

The natural response was for attackers to move into the kernel of the computer. The first kernel rootkits were written for UNIX machines. Once they infected the kernel, they could subvert any security utility on the computer at that time. In other words, Trojan files were no longer needed: All stealth could be applied by modifying the kernel. This technique was no different from the techniques used by viruses in the late 1980s to hide from anti-virus software.



How Do Rootkits Work?

Rootkits work using a simple concept called *modification.* In general, software is designed to make specific decisions based on very specific data. A rootkit locates and modifies the software so it makes incorrect decisions.

There are many places where modifications can be made in software. Some of them are discussed in the following paragraphs.

Patching

Executable code (sometimes called a *binary*) consists of a series of statements encoded as data bytes. These bytes come in a very specific order, and each means something to the computer. Software logic can be modified if these bytes are modified. This technique is sometimes called *patching*—like placing a patch of a different color on a quilt. Software is not smart; it does only and exactly what it is told to do and nothing else. That is why modification works so well. In fact, under the hood, it's not all that complicated. Byte patching is one of the major techniques used by "crackers" to remove software protections. Other types of byte patches have been used to cheat on video games (for example, to give unlimited gold, health, or other advantages).

Easter Eggs

Software logic modifications may be "built in." A programmer may place a back door in a program she wrote. This back door is not in the documented design, so the software has a hidden feature. This is sometimes called an *Easter Egg,* and can be used like a signature: The programmer leaves something behind to show that she wrote the program. Earlier versions of the widely used program Microsoft Excel contained an easter-egg that allowed a user who found it to play a 3D first-person shooter game similar to Doom[13] embedded inside a spreadsheet cell.

[13] *The Easter Eggs and Curios Database,* www.eggheaven2000.com

Spyware Modifications

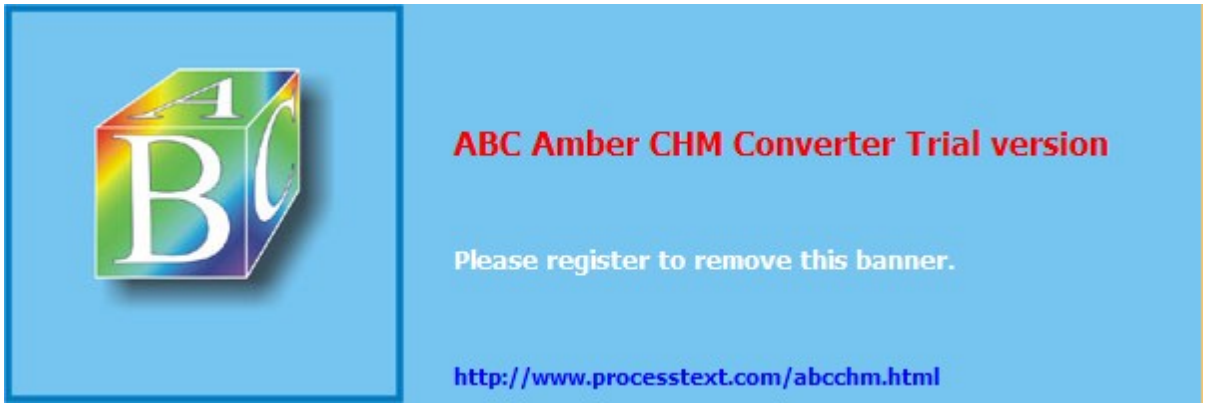
Sometimes a program will modify another program to infect it with "spyware." Some types of spyware track which Web sites are visited by users of the infected computer. Like rootkits, spyware may be difficult to detect. Some types of spyware hook into Web browsers or program shells, making them difficult to remove. They then make the user's life hell by placing links for new mortgages and Viagra on their desktops, and generally reminding them that their browsers are totally insecure.[14]

[14] Many Web browsers fall prey to spyware, and of course Microsoft's Internet Explorer is one of the biggest targets for spyware.

Source-Code Modification

Sometimes software is modified at the source—literally. A programmer can insert malicious lines of source code into a program she authors. This threat has caused some military applications to avoid open-source packages such as Linux. These open-source projects allow almost anyone ("anyone" being "someone you don't know") to add code to the sources. Granted, there is some amount of peer review on important code like BIND, Apache, and Sendmail. But, on the other hand, does anyone really go through the code line by line? (If they do, they don't seem to do it very well when trying to find security holes!) Imagine a back door that is implemented as a bug in the software. For example, a malicious programmer may expose a program to a buffer overflow on purpose. This type of back door can be placed on purpose. Since it's disguised as a bug, it becomes difficult to detect. Furthermore, it offers plausible deniability on the part of the programmer!

Okay, we can hear you saying "Bah! I fully trust all those unknown people out there who authored my software because they are obviously only three degrees of separation from Linus Torvalds[15] and I'd trust Linus with my life!" Fine, but do you trust the skills of the system administrators who run the



What a Rootkit Is Not

Okay, so we've described in detail what a rootkit is and touched on the underlying technology that makes a rootkit possible. We have described how a rootkit is a powerful hacker tool. But, there are many kinds of hacker tools?a rootkit is only one part of a larger collection. Now it's time to explain what a rootkit is *not*.

A Rootkit Is Not an Exploit

Rootkits may be used in conjunction with an exploit, but the rootkit itself is a fairly straightforward set of utility programs. These programs may use undocumented functions and methods, but they typically do not depend on software bugs (such as buffer overflows).

A rootkit will typically be deployed after a successful software exploit. Many hackers have a treasure chest of exploits available, but they may have only one or two rootkit programs. Regardless of which exploit an attacker uses, once she is on the system, she deploys the appropriate rootkit.

Although a rootkit is not an exploit, it may incorporate a software exploit. A rootkit usually requires access to the kernel and contains one or more programs that start when the system is booted. There are only a limited number of ways to get code into the kernel (for example, as a device driver). Many of these methods can be detected forensically.

One novel way to install a rootkit is to use a software exploit. Many software exploits allow arbitrary code or third-party programs to be installed. Imagine that there is a buffer overflow in the kernel (there are documented bugs of this nature) that allows arbitrary code to be executed. Kernel-buffer overflows can exist in almost any device driver (for example, a printer driver). Upon system startup, a loader program can use the buffer overflow to load a rootkit. The loader program does not employ any documented methods for loading or registering a device driver or otherwise installing a rootkit. Instead, the loader exploits the buffer overflow to install the kernel-mode parts of a rootkit.

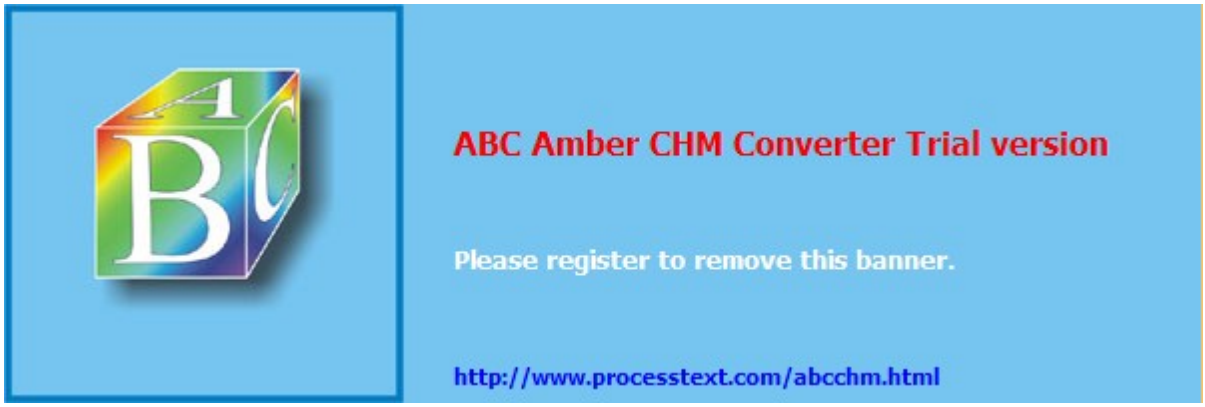
The buffer-overflow exploit is a mechanism for loading code into the kernel. Although most people think of this as a bug, a rootkit developer may treat it as an undocumented feature for loading code into the kernel. Because it is not documented, this "path to the kernel" is not likely to be included as part of a forensic investigation. Even more importantly, it won't be protected by a host-based firewall program. Only someone skilled in advanced reverse engineering would be likely to discover it.

A Rootkit Is Not a Virus

A virus program is a self-propagating automaton. In contrast, a rootkit does not make copies of itself, and it does not have a mind of its own. A rootkit is under the full control of a human attacker, while a virus is not.

In most cases, it would be dangerous and foolish for an attacker to use a virus when she requires stealth and subversion. Beyond the fact that creating and distributing virus programs may be illegal, most virus and worm programs are noisy and out of control. A rootkit enables an attacker to stay in complete control. In the case of a sanctioned penetration (for example, by law enforcement), the attacker needs to ensure that only certain targets are penetrated, or else she may violate a law or exceed the scope of the operation. This kind of operation requires very strict controls, and using a virus would simply be out of the question.

It is possible to design a virus or worm program that spreads via software exploits that are not detected by intrusion-detection systems (for instance, *zero-day* exploits[18]). Such a worm could spread very slowly and be very difficult to detect. It may have been tested in a well-stocked lab environment with a model of the target environment. It may include an "area-of-effect" restriction to keep it from spreading outside of a controlled boundary. And, finally, it may have a "land-mine timer" that causes it to be disabled after a certain amount of time?ensuring that it doesn't cause problems after the mission is over. We'll discuss intrusion-detection systems later in this chapter.



Rootkits and Software Exploits

Software exploitation is an important subject relating to rootkits. (How software can break and be exploited is not covered in this book. If you're interested in software exploitation, we recommend the book *Exploiting Software*.[20])

[20] G. Hoglund and G. McGraw, *Exploiting Software*.

Although a rootkit is not an exploit, it may be employed as part of an exploit tool (for example, in a virus or spyware).

The threat of rootkits is made strong by the fact that software exploits are in great supply. For example, a reasonable conjecture is that at any given time, there are more than a hundred known working exploitable holes in the latest version of Microsoft Windows.[21] For the most part, these exploitable holes are known by Microsoft and are being slowly managed through a quality-assurance and bug-tracking system.[22] Eventually, these bugs are fixed and *silently* patched.[23]

[21] We cannot offer proof for this conjecture, but it is a reasonable assumption derived from knowledge about the problem.

[22] Most software vendors use similar methods to track and repair bugs in their products.

[23] "Silently patched" means the bug is fixed via a software update, but the software vendor never informs the public or any customers that the bug ever existed. For all intents, the bug is treated as "secret" and nobody talks about it. This is standard practice for many large software vendors, in fact.

Some exploitable software bugs are found by independent researchers and never reported to the software vendor. They are deadly because nobody knows about them accept the attacker. This means there is little to no defense against them (no patch is available).

Many exploits that have been publicly known for more than a year are still being widely exploited today. Even if there is a patch available, most system administrators don't apply the patches in a timely fashion. This is especially dangerous since even if no exploit program exists when a security flaw is discovered, an exploit program is typically published within a few days after release of a public advisory or a software patch.

Although Microsoft takes software bugs seriously, integrating changes by any large operating system vendor can take an inordinate amount of time.

When a researcher reports a new bug to Microsoft, she is usually asked not to release public information about the exploit until a patch can be released. Bug fixing is expensive and takes a great deal of time. Some bugs aren't fixed until several months after they are reported.

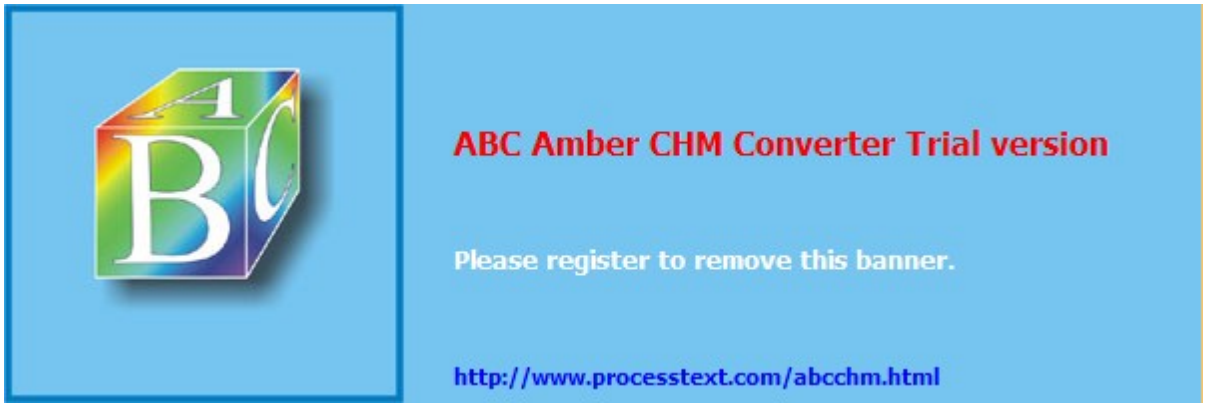
One could argue that keeping bugs secret encourages Microsoft to take too long to release security fixes. As long as the public doesn't know about a bug, there is little incentive to quickly release a patch. To address this tendency, the security company eEye has devised a clever method to make public the fact that a serious vulnerability has been found, but without releasing the details.

Figure 1-2, which comes from eEye's Web site,[24] shows a typical advisory. It details when the bug was reported to a vendor, and by how many days the vendor patch is "overdue," based on the judgment that a timely response would be release of a patch within 60 days. As we have seen in the real world, large software vendors take longer than 60 days. Historically, it seems the only time a patch is released within days is when a real Internet worm is released that uses the exploit.

[24] www.eEye.com

Figure 1-2. Method used by eEye to "pre-release" a security advisory.

[View full size image]



Offensive Rootkit Technologies

A good rootkit should be able to bypass any security measures, such as firewalls or intrusion-detection systems (IDSes). There are two primary types of IDSes: network-based (NIDS) and host-based (HIDS). Sometimes HIDSes are designed to try to stop attacks before they succeed. These "active defense" systems are sometimes referred to as a host-based intrusion-prevention systems (HIPSes). To simplify the discussion, we refer to these systems as HIPS from now on.

HIPS

HIPS technology can be home-grown or bought off-the-shelf. Examples of HIPS software include:

- Blink (eEye Digital Security, www.eEye.com)
- Integrity Protection Driver (IPD, Pedestal Software, www.pedestal.com)
- Entercept (www.networkassociates.com)
- Okena StormWatch (now called Cisco Security Agent, www.cisco.com)
- LIDS (Linux Intrusion Detection System, www.lids.org)
- WatchGuard ServerLock (www.watchguard.com)

For the rootkit, the biggest threat is HIPS technology. A HIPS can sometimes detect a rootkit as it installs itself, and can also intercept a rootkit as it communicates with the network. Many HIPSes will utilize kernel technology and can monitor operating systems. In a nutshell, HIPS is an *anti*-rootkit. This means that anything a rootkit does on the system most likely will be detected and stopped. When using a rootkit against a HIPS-protected system, there are two choices: bypass the HIPS, or pick an easier target.

Chapter 10 in this book covers the development of HIPS technology. The chapter also includes examples of anti-rootkit code. The code can help you understand how to bypass a HIPS and can also assist you in constructing your own rootkit-protection system.

NIDS

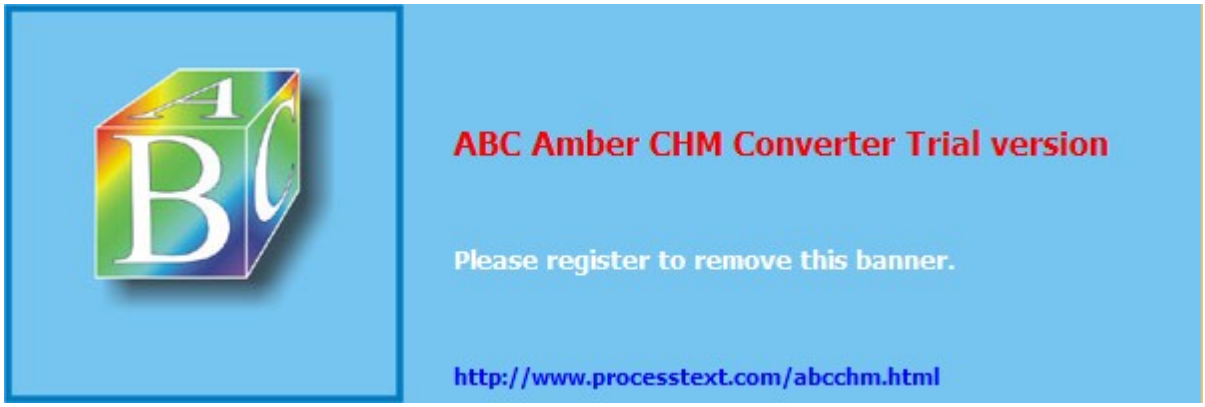
Network-based IDS (NIDS) is also a concern for rootkit developers, but a well-designed rootkit can evade a production NIDS. Although, in theory, statistical analysis can detect covert communication channels, in reality this is rarely done. Network connections to a rootkit will likely use a covert channel hidden within innocent-looking packets. Any important data transfer will be encrypted. Most NIDS deployments deal with large data streams (upward of 300 MB/second), and the little trickle of data going to a rootkit will pass by unnoticed. The NIDS poses a larger detection threat when a publicly known exploit is used in conjunction with a rootkit.[30]

[30] When using a publicly known exploit, an attacker may craft the exploit code to mimic the behavior of an already-released worm (for example, the Blaster worm). Most security administrators will mistake the attack as simply actions of the known worm, and thus fail to recognize a unique attack.

Bypassing the IDS/IPS

To bypass firewalls and IDS/IPS software, there are two approaches: active and passive. Both approaches must be combined to create a robust rootkit. Active offenses operate at runtime and are designed to prevent detection. Just in case someone gets suspicious, passive offenses are applied "behind the scenes" to make forensics as difficult as possible.

Active offenses are modifications to the system hardware and kernel designed to subvert and confuse intrusion-detection software. Active measures are usually required in order to disable HIPS software (such as Okena and Entercept). In general, active offense is used against software which runs in memory



Conclusion

First-generation rootkits were just normal programs. Today, rootkits are typically packaged as device drivers. Over the next few years, advanced rootkits may modify or install into the microcode of a processor, or exist primarily in the microchips of a computer. For example, it is not inconceivable that the bitmap for an FPGA (field programmable gate array) could be modified to include a back door.[32] Of course, this type of rootkit would be crafted for a very specific target. Rootkits that use more generic operating-system services are more likely to be in widespread use.

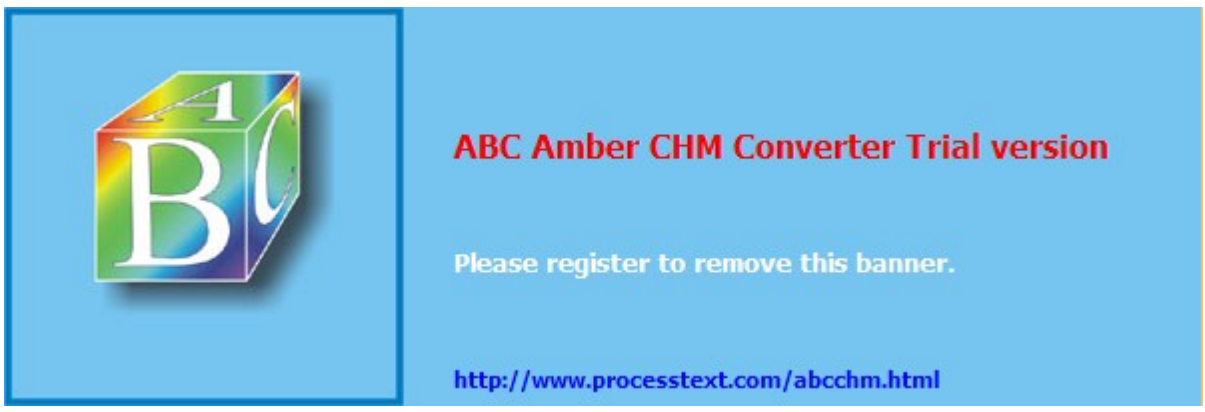
[32] This assumes that there is enough room (in terms of gates) to add features to an FPGA. Hardware manufacturers try to save money on every component, so an FPGA will be as small as possible for the application. There may not be much room left in the gate array for anything new. To insert a rootkit into a tight spot like this may require removal of other features.

The kind of rootkit technology that could hide within an FPGA is not suitable for use by a network worm. Hardware-specific attacks don't work well for worms. The network-worm strategy is facilitated by large-scale, homogenous computing. In other words, network worms work best when all the targeted software is the same. In the world of hardware-specific rootkits, there are many small differences that make multiple-target attacks difficult. It is much more likely that hardware-based attacks would be used against a specific target the attacker can analyze in order to craft a rootkit specifically for that target.

As long as software exploits exist, rootkits will use these exploits. They work together naturally. However, even if such exploits were not possible, rootkits would still exist.

In the next few decades or so, the buffer overflow, currently the "king of all software exploits," will be dead and buried. Advances in type-safe languages, compilers, and virtual-machine technologies will render the buffer overflow ineffective, striking a huge blow against those who rely on remote exploitation. This doesn't mean exploits will go away. The new world of exploiting will be based on logic errors in programs rather than on the architecture flaw of buffer overflow.

With or without remote exploitation, however, rootkits will persist. Rootkits can be placed into systems at many stages, from development to delivery. As long as there are people, people will want to spy on other people. This means rootkits will always have a place in our technology. Backdoor programs and technology subversions are timeless!



Chapter 2. Subverting the Kernel

There was no trace then of the horror which I had myself feltat this curt declaration; but his face showed rather the quiet and interested composure of the chemist who sees the crystals falling into position from his oversaturated solution.

?*THE VALLEY OF FEAR*, SIR ARTHUR CONAN DOYLE

Computers of all shapes and sizes have software installed on them, and most computers have an operating system. The operating system is the core set of software programs that provide services to the other programs on the computer. Many operating systems multitask, allowing multiple programs to be run simultaneously.

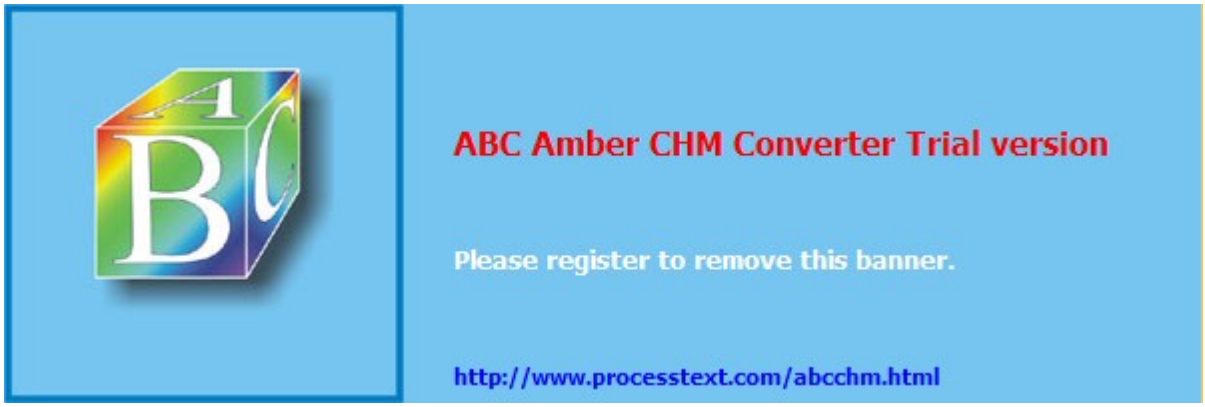
Different computing devices can contain different operating systems. For instance, the most widely used operating system on PCs is Microsoft's Windows. A large number of servers on the Internet run Linux or Sun Solaris, while many others run Windows. Embedded devices typically run the VXWorks operating system, and many cellular phones use Symbian.

Regardless of the devices on which it is installed, every operating system (OS) has one common purpose: to provide a single, consistent interface that application software can use to access the device. These core services control access to the device's file system, network interface, keyboard, mouse, and video/LCD display.

A secondary function of the OS is to provide debugging and diagnostic information about the system. For example, most operating systems can list the running or installed software. Most have logging mechanisms, so that applications can report when they have crashed, when someone fails to login properly, etc.

Although it is possible to write applications that bypass the OS (undocumented, direct-access methods), most developers don't do that. The OS provides the "official" mechanism for access, and frankly, it's much easier to just use the OS. This is why nearly all applications use the OS for these services?and it's why a rootkit that changes the OS will affect nearly all software.

In this chapter we jump right in and start writing our very first rootkit for Windows. We will introduce source code and explain how to set up your development environment. We also cover some basic information about the kernel, and how device drivers work.



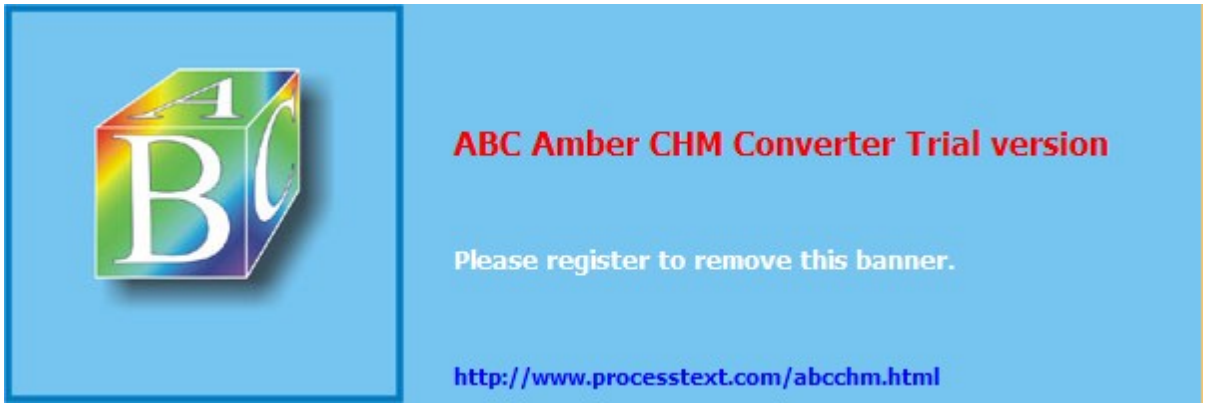
Important Kernel Components

In order to understand how rootkits can be used to subvert an OS kernel, it helps to know which functions the kernel handles. Table 2-1 describes each major functional component of the kernel.

Table 2-1. Functional components of the kernel.

Process management	Processes need CPU time. The kernel contains code to assign this CPU time. If the OS supports threads, the kernel will schedule time to each thread. Data structures in memory keep track of all the threads and processes. By modifying these data structures, an attacker can hide a process.
File access	The file system is one of the most important features an OS provides. Device drivers may be loaded to handle different underlying file systems (such as NTFS). The kernel provides a consistent interface to these file systems. By modifying the code in this part of the kernel, an attacker can hide files and directories.
Security	The kernel is ultimately responsible for enforcing restrictions between processes. Simple systems may not enforce any security at all. For example, many embedded devices allow any process to access the full range of memory. On UNIX and MS-Windows systems, the kernel enforces permissions and separate memory ranges for each process. Just a few changes to the code in this part of the kernel can remove all the security mechanisms.
Memory management	Some hardware platforms, such as the Intel Pentium family, have complex memory-management schemes. A memory address can be mapped to multiple physical locations. For example, one process can read the memory at address 0x00401111 and get the value "HELLO," while another process can read that same memory at address 0x00401111 but get the value "GO AWAY." The same address points to two totally different physical memory locations, each containing different data. (We will discuss more about virtual-to-physical memory mapping in Chapter 3, The Hardware Connection.) This is possible because the two processes are mapped differently. Exploiting the way this works in the kernel can be very useful for hiding data from debuggers or active forensics software.

Now that we have an idea of the functions of the kernel, we will discuss how a rootkit might be designed to modify the kernel.



Rootkit Design

An attacker typically designs a rootkit to affect a particular OS and software set. If the rootkit is designed with direct hardware access, then it will be limited to that specific hardware. Rootkits can be generic to different versions of an OS, but will still be limited to a given OS family. For example, some rootkits in the public domain affect all flavors of Windows NT, 2000, and XP. This is possible only when all the flavors of the OS have similar data structures and behaviors. It would be far less feasible to create a generic rootkit that can infect both Windows and Solaris, for example.

A rootkit may use more than one kernel module or driver program. For instance, an attacker may use one driver to handle all file-hiding operations, and another driver to hide registry keys. Distributing the code across many driver packages is sometimes a Good Thing because it helps keep the code manageable?as long as each driver has a specific purpose. It would be hard for an attacker to manage a monolithic "kitchen-sink" driver that provides every feature known to man.

A complex rootkit project might have many components. It helps to keep things organized in a large project. Although we won't develop any examples that are quite so complex in this book, the following directory structure might be used by a complex rootkit project:

```
/My Rootkit
    /src/File Hider
```

One Rootkit, One System

One rootkit should be enough for any system. A rootkit is invasive and alters data on the system. Although attackers generally keep this invasive alteration to a minimum, installing multiple rootkits may cause alterations of alterations, leading to possible corruption. Rootkits assume, in most cases, that the system is clean. A rootkit may perform checks for anti-hacker software (such as desktop firewalls), but it usually doesn't check for another rootkit. If another rootkit were found to be already installed on the system, the attacker's best strategy might be to "fail out" (that is, stop executing due to an error).

File-hiding code can be complex and should be contained in its own set of source-code files. There are multiple techniques for file hiding, some of which could require a great deal of code. For example, some file-hiding techniques require hooking a large number of function calls. Each hook requires a fair amount of source code.

```
/src/Network Ops
```

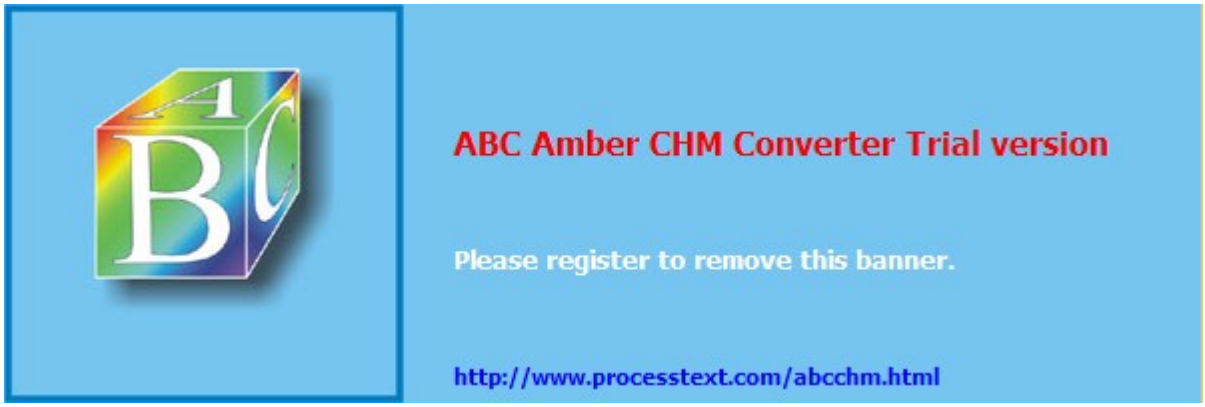
Network operations require NDIS[1] and TDI[2] code on Microsoft Windows. These drivers tend to be large, and they sometimes link to external libraries. Again, it makes sense to confine these features to their own source files.

[1] Network Driver Interface Specification

[2] Transport Driver Interface

```
/src/Registry Hider
```

Registry-hiding operations may require different approaches than file-hiding features. There may be many hooks involved, and perhaps tables or lists of handles that need to be tracked. In practice, registry-key hiding has been problematic due to the way keys and values relate to one another. This has caused some



Introducing Code into the Kernel

The straightforward way to introduce code into the kernel is by using a loadable module (sometimes called a device driver or kernel driver). Most modern operating systems allow kernel extensions to be loaded so that manufacturers of third-party hardware, such as storage systems, video cards, motherboards, and network hardware, can add support for their products. Each operating system usually supplies documentation and support to introduce these drivers into the kernel. This is the easy route, and is the road we will take to introduce code into the kernel.

As its name suggests, a device driver is typically for devices. However, any code can be introduced via a driver. Once you have code running in the kernel, you have full access to all of the privileged memory of the kernel and system processes. With kernel-level access you can modify the code and data structures of any software on the computer.

A typical module would include an entry point and perhaps a cleanup routine. For example, a Linux-loadable module may look something like this:

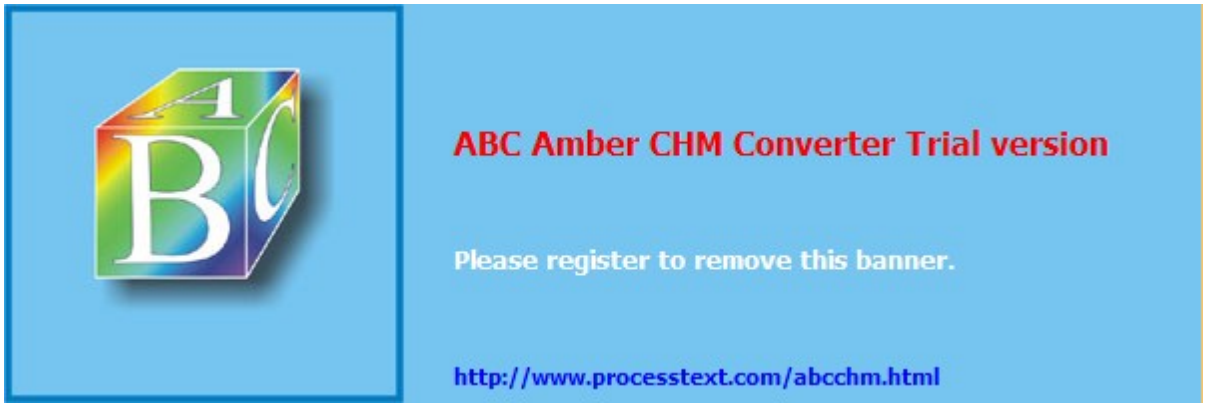
```
int init_module(void)
{
}
void cleanup_module(void)
{
}
```

In some cases, such as with Windows device drivers, the entry point must register function callbacks. In such a case, the module would look like this:

```
NTSTATUS DriverEntry( ... )
{
     theDriver->DriverUnload = MyCleanupRoutine;
}
NTSTATUS MyCleanupRoutine()
{
}
```

A cleanup routine is not always needed, which is why Windows device drivers make this optional. The cleanup routine would be required only if you plan on unloading the driver. In many cases, a rootkit can be placed into a system and left there, without any need to unload it. However, it is helpful during development to have an unload routine because you may want to load newer versions of the rootkit as it evolves. Most example rootkits provided by rootkit.com include unload routines.[3]

[3] A set of basic rootkits known as the "basic_class" can be found at rootkit.com.



Building the Windows Device Driver

Our first example will operate on the Windows XP and 2000 platforms and will be designed as a simple device driver. In reality, this isn't actually a rootkit yet?it's just a simple "hello world" device driver.

```
#include "ntddk.h"
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
IN PUNICODE_STRING theRegistryPath )'
{
   DbgPrint("Hello World!");
   return STATUS_SUCCESS;
}
```

Wow, that one was easy, wasn't it? You can load this code into the kernel, and the debug statement will be posted.[4]

[4] See the section Logging the Debug Statements later in this chapter to learn how to capture debug messages.

Our rootkit will be composed of several items, each of which we describe in the sections that follow.

The Device Driver Development Kit

To build our Windows device driver, we'll need the Driver Development Kit (DDK). DDKs are available from Microsoft for each version of Windows.[5] Chances are you will want the Windows 2003 DDK. You can build drivers for Windows 2000, XP, and 2003 using this version of the DDK.

[5] Information on Windows DDKs is available at: www.microsoft.com/ddk/

The Build Environments

The DDK provides two different build environments: the *checked* and the *free* build environments. You use the checked-build environment when you're developing a device driver, and you use the free-build environment for release code. The checked build results in debugging checks being compiled into your driver. The resulting driver will be much larger than the free-build version. You should use the checked build for most of your development work, and switch to the free build only when you're testing your final product. While exploring the examples in this book, checked builds are fine.

The Files

You will write your driver source code in C, and you will give the filename a .c extension. To start your first project, make a clean directory (a suggestion is C:\myrootkit), and place a mydriver.c file there. Then copy into that file the "hello world" device-driver code shown earlier.

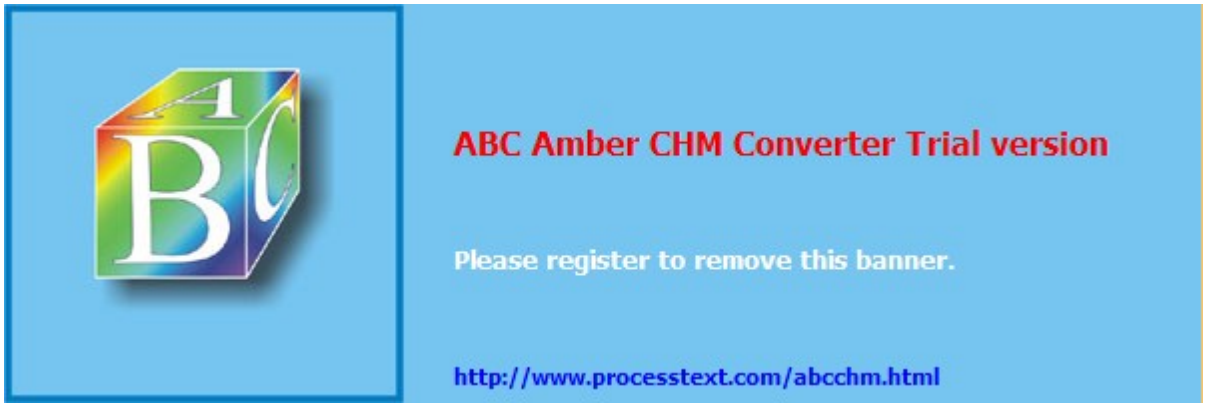
You will also need a SOURCES file and a MAKEFILE file.

The SOURCES File

This file should be named SOURCES in all-capital letters, with no file extension. The SOURCES file should contain the following code:

```
TARGETNAME=MYDRIVER
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=mydriver.c
```

The TARGETNAME variable controls what your driver will be named. Remember that this name may be embedded in the binary itself, so using a TARGETNAME of



Loading and Unloading the Driver

Loading and unloading the driver is easy. For starters, just download the InstDrv tool from rootkit.com.[6]

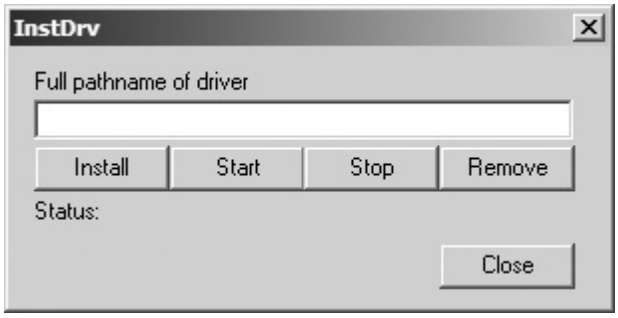
[6] The InstDrv tool was not written by members of rootkit.com; it is hosted there as a convenience.

Rootkit.com

You can find a copy of the InstDrv tool at: www.rootkit.com/vault/hoglund/InstDvr.zip.

This utility will allow you to register and start/stop your driver. Figure 2-1 shows a screenshot of this utility.

Figure 2-1. The InstDrv utility.



When it comes to real-world use, you will certainly need a better method for loading your driver. However, this utility works very well while your rootkit is in development. We cover a real-world deployment program under the section Loading the Rootkit later in this chapter.

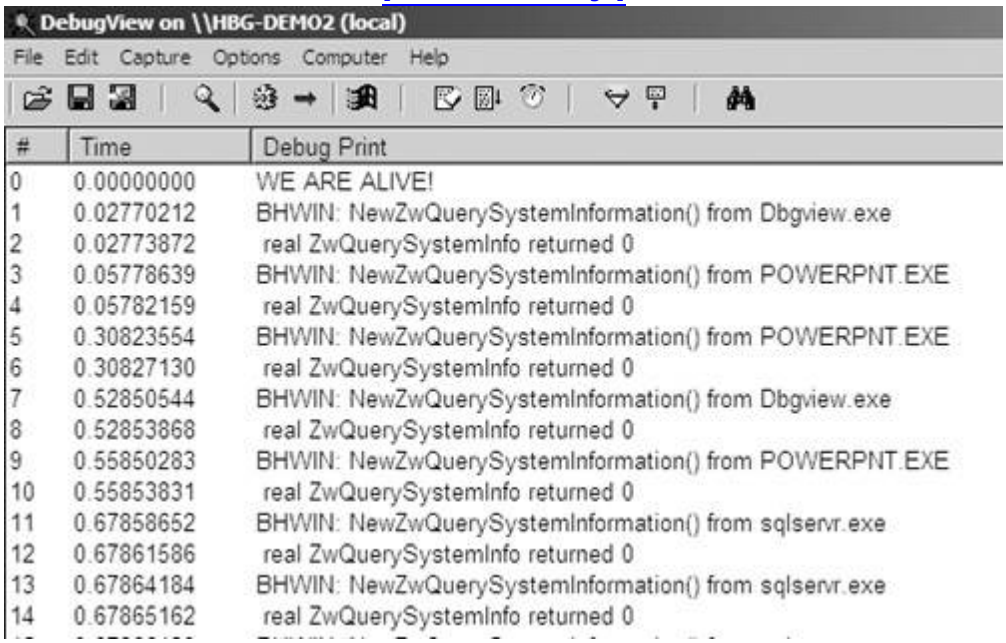
Logging the Debug Statements

Debug statements provide a way for the developer to log important information while a driver executes. In order to log the messages, you need a debug message capturing tool. A useful tool for capturing debug statements is called Debug View, and is available from www.sysinternals.com. This tool is free.

Debug statements can be used to print *tombstones*, markers to indicate that particular lines of code have executed. Using debug statements can sometimes be easier than using a single-step debugger like SoftIce or WinDbg. This is because running a tool to capture debug statements is very easy, while configuring and using a debugger is complex. With debug statements, return codes can be printed or error conditions detailed. Figure 2-2 shows an example of a call-hooking rootkit sending debug output to the system.

Figure 2-2. DebugView captures output from a kernel rootkit.

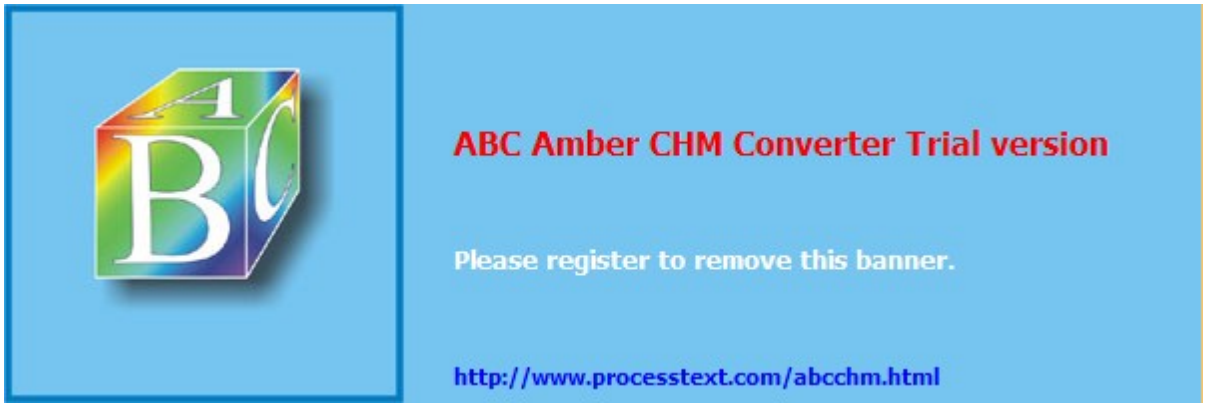
[View full size image]



You can print debug statements with Windows drivers by using the following call:

```
DbgPrint("some string");
```

Many debug or kernel-level logging functions such as DbgPrint are available with most operating systems. For example, under Linux, a loadable module can use the printk(...) function.

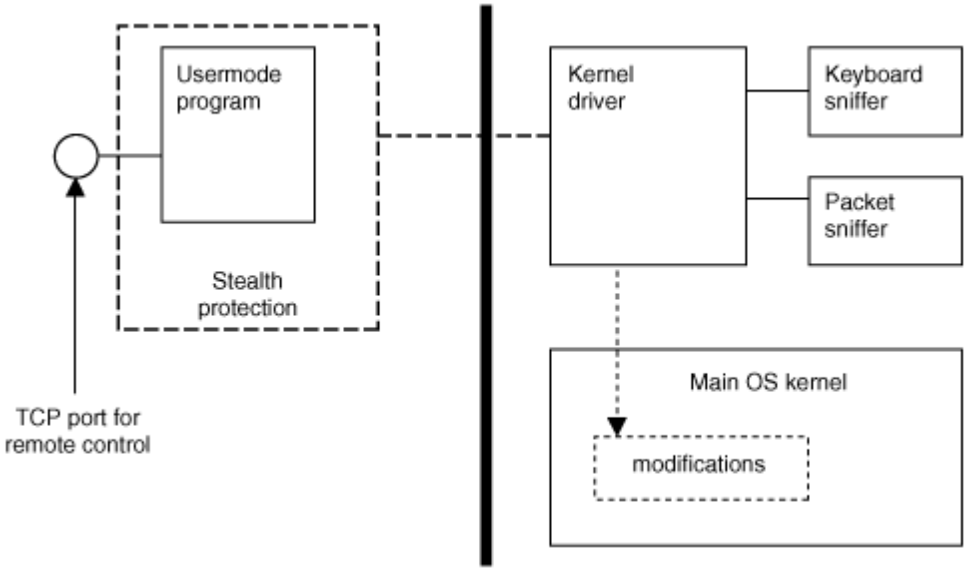


Fusion Rootkits: Bridging User and Kernel Modes

Rootkits can easily contain both user-mode and kernel-mode components (see Figure 2-3). The user-mode part deals with most of the features, such as networking and remote control, and the kernel-mode part deals with stealth and hardware access.

Figure 2-3. A fusion rootkit using both user and kernel components.

[View full size image]



Most rootkits require kernel-level subversion while at the same time offering complex features. Because complex features may contain bugs and require use of system API libraries, the user-mode approach is preferred.

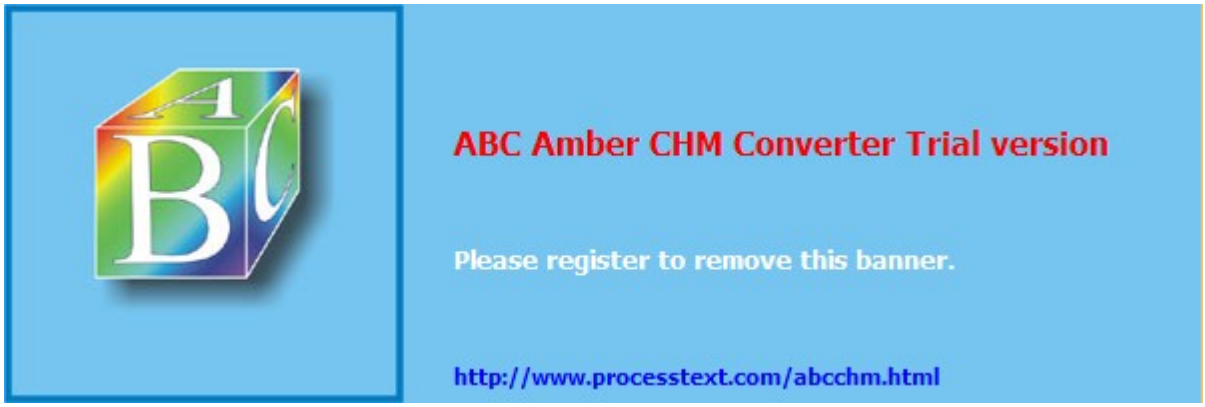
A user-mode program can communicate with a kernel-level driver through a variety of means. One of the most common is the use of I/O Control (IOCTL) commands. IOCTL commands are command messages that can be defined by the programmer. You should understand the following device-driver concepts in order to build a rootkit that has both user- and kernel-mode components.

I/O Request Packets

One of the device-driver concepts to understand is I/O Request Packets (IRPs). In order to communicate with a user-mode program, a Windows device driver typically needs to handle IRPs. These are just data structures which contain buffers of data. A user-mode program can open a file handle and write to it. In the kernel, this write operation is represented as an IRP. So, if a user-mode program writes the string "HELLO DRIVER!" to the file handle, the kernel will create an IRP that contains the buffer and string "HELLO DRIVER!" Communication can take place between the user and kernel modes via these IRPs.

In order to process IRPs, the kernel driver must include functions to handle the IRP. Just as we did in installing the unload routine, we simply set the appropriate function pointers in the driver object:

```
NTSTATUS OnStubDispatch(IN PDEVICE_OBJECT DeviceObject,
                        IN PIRP Irp )
{
      Irp->IoStatus.Status = STATUS_SUCCESS;
      IoCompleteRequest(Irp,
                        IO_NO_INCREMENT );
      return STATUS_SUCCESS;
}
VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
```



Loading the Rootkit

Inevitably, you will need to load the driver from a user-mode program. For example, if you penetrate a computer system, you will want to copy over a deployment program of some kind that, when run, loads the rootkit into the kernel.

A loading program typically will decompress a copy of the .sys file to the hard drive, and then issue the commands to load it into the kernel. Of course, for any of this to work, the program must be running as "administrator."[7]

[7] Or as NT_AUTHORITY/SYSTEM, depending on how you get onto the system.

There are many ways to load a driver into the kernel. We cover two methods?one we call "quick and dirty," and another we call "The Right Way." Either method will work, but read on to learn the details.

The Quick-and-Dirty Way to Load a Driver

Using an undocumented API call, you can load a driver into the kernel without having to create any registry keys. The problem with this approach is that the driver will be pageable. "Pageable" refers to memory that can be swapped to disk. If a driver is pageable, any part of the driver could be paged out (that is, swapped from memory to disk). Sometimes when memory is paged out, it cannot be accessed; an attempt to do so will result in the infamous Blue Screen of Death (a system crash). The only time when this loading method is really safe is when it's specifically designed around the paging problem.

An example of a good rootkit that uses this loading method is migbot, which is available at rootkit.com. The migbot rootkit is very simple, and copies all of the operational code into a non-paged memory pool, so the fact that the driver is paged does not affect anything migbot does.

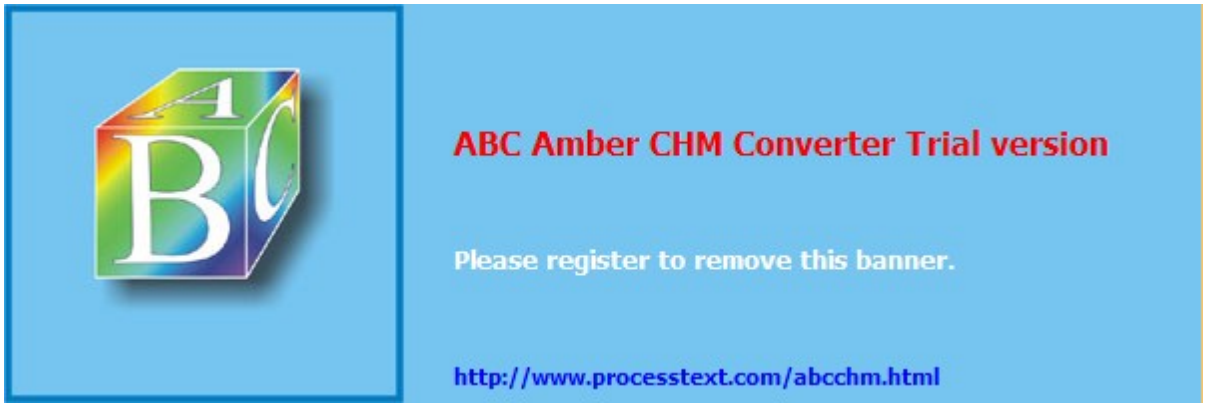
Rootkit.com

You can download the source code for migbot from www.rootkit.com/vault/hoglund/migbot.zip

The loading method is typically referred to as SYSTEM LOAD AND CALL IMAGE because this is the name given to the undocumented API call.

Here is the loading code from migbotloader:

```
//-------------------------------------------------------------
---
// load a sys file as a driver using undocumented method
//-------------------------------------------------------------
---
bool load_sysfile()
{
      SYSTEM_LOAD_AND_CALL_IMAGE GregsImage;
      WCHAR daPath[] = L"\\??\\C:\\MIGBOT.SYS";

////////////////////////////////////////////////////////
      // get DLL entry points

////////////////////////////////////////////////////////
      if(!(RtlInitUnicodeString = (RTLINITUNICODESTRING)
                  GetProcAddress( GetModuleHandle("ntdll.dll")
                  ,"RtlInitUnicodeString"
                  )))
      {
            return false;
```



Decompressing the .sys File from a Resource

Windows PE executables allow multiple sections to be included in the binary file. Each section can be thought of as a folder. This allows developers to include various objects, such as graphics files, within the executable file. Any arbitrary binary objects can be included within the PE executable, including additional files. For instance, an executable can contain both a .sys file and a configuration file with startup parameters for the rootkit. A clever attacker might even create a utility that sets configuration options "on the fly" before an exploit is used with the rootkit.

The following code illustrates how to access a named resource within the PE file and subsequently make a copy of the resource as a file on the hard drive. (The word decompress in the code is imprecise, as the embedded file is not actually compressed.)

```
//-------------------------------------------------------------
---
// build a .sys file on disk from a resource
//-------------------------------------------------------------
---
bool _util_decompress_sysfile(char *theResourceName)
{
      HRSRC aResourceH;
      HGLOBAL aResourceHGlobal;
      unsigned char * aFilePtr;
      unsigned long aFileSize;
      HANDLE file_handle;
```

The subsequent FindResource API call is used to obtain a handle to the embedded file. A resource has a type, in this case BINARY, and a name.

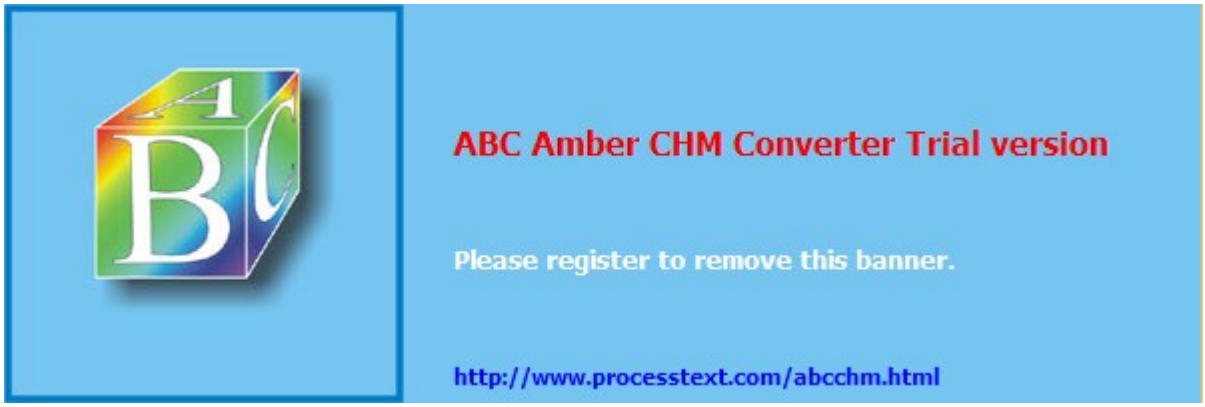
```
////////////////////////////////////////////////////
      // locate a named resource in the current binary EXE

////////////////////////////////////////////////////
      aResourceH = FindResource(NULL, theResourceName,
"BINARY");
      if(!aResourceH)
      {
            return false;
      }
```

The next step is to call LoadResource. This returns a handle that we use in subsequent calls.

```
      aResourceHGlobal = LoadResource(NULL, aResourceH);
      if(!aResourceHGlobal)
      {
            return false;
      }
```

Using the SizeOfResource call, the length of the embedded file is obtained:

```
      aFileSize = SizeofResource(NULL, aResourceH);
      aFilePtr = (unsigned char
*)LockResource(aResourceHGlobal);
```



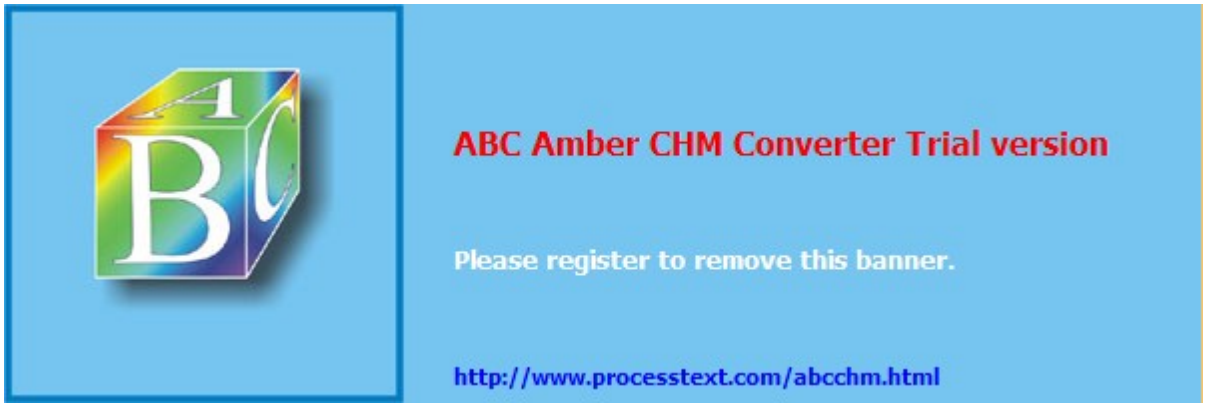
Surviving Reboot

The rootkit driver must be loaded upon system boot. If you think about this problem generally, you will realize that many different software components get loaded when the system boots. As long as a rootkit is connected with one of the boot-time events listed in Table 2-2, it will also load.

Table 2-2. Some ways to load a rootkit at system-boot time.

Using the run key ("old reliable")	The run key (and its derivates) can be used to load any arbitrary program at boot time. This program can decompress the rootkit and load it. The rootkit can hide the run-key value once loaded so that it remains undetected. All virus scanners check this key, so it's a high-risk method. However, once the rootkit has been loaded, the value can be hidden.
Using a Trojan or infected file	Any .sys file or executable that is to be loaded at boot time can be replaced, or the loader code can be inserted similarly to the way a virus can infect a file. Ironically, one of the best things to infect is a virus-scanning or security product. A security product will typically start when the system is booted. A trojan DLL can be inserted into the search path, or an existing DLL can simply be replaced or "infected."
Using .ini files	.ini files can be altered to cause programs to be run. Many programs have initialization files that can run commands on startup or specify DLLs to load. One such file that can be used in this way is called win.ini.
Registering as a driver	The rootkit can register itself as a driver which is loaded on boot. This requires creating a registry key. Again, the key can be hidden once the rootkit has loaded.
Registering as an add-on to an existing application	A favorite method used by spyware is to add an extension to a Web-browsing application (for example, in the guise of a search bar). The extension is loaded when the application loads. This method requires that the application is launched, but if that's likely to occur before the rootkit must be activated, it can be effective for loading the rootkit. A downside to this approach is that many free adware scanners are available, and these may detect the application extension.
Modifying the on-disk kernel	The kernel can be directly modified and saved to disk. A few changes must be made to the boot loader so that the kernel will pass a checksum integrity check. This can be very effective, since the kernel will be permanently modified, and no drivers will need to be registered.
Modifying the boot loader	The boot loader can be modified to apply patches to the kernel before it loads. An advantage is that the kernel file itself will not appear modified if the system is analyzed offline. However, a boot-loader modification can be detected with the right tools.

There are many ways to load at boot time; the list in Table 2-2 is by no means complete. With a little creativity and some time, you should be able to discover additional ways to load.



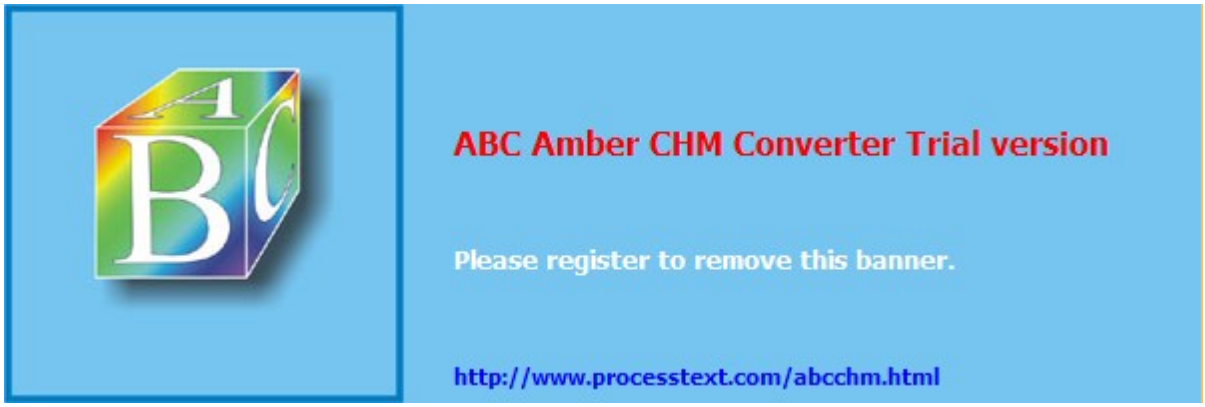
Conclusion

This chapter has armed you with the basics of device-driver development for Windows. We described some of the key areas that can be targeted in the kernel. We also covered the mundane details of setting up your development environment and tools to make rootkit development easier. Finally, we covered the basic requirements for loading, unloading, and starting a driver. We also touched upon deployment methods, and ways to make a driver start on system boot.

The subjects covered in this chapter are required for writing rootkits for MS-Windows. At this point, you should be able to write a simple "hello world" rootkit, and load and unload it from the kernel. You also should be able to write a user-mode program that can communicate with a kernel-mode driver.

In subsequent chapters, we will delve much deeper into the workings of the kernel and the underlying hardware that supports all software. By beginning with the lowest-level structures, you can build correct understandings that enable you to synthesize knowledge of the highest-level elements. This is how you will become a master of rootkits.

PREV < Day Day Up > NEXT



Chapter 3. The Hardware Connection

One Ring to rule them all, One Ring to find them, One Ring to bring them all and in the darkness bind them.

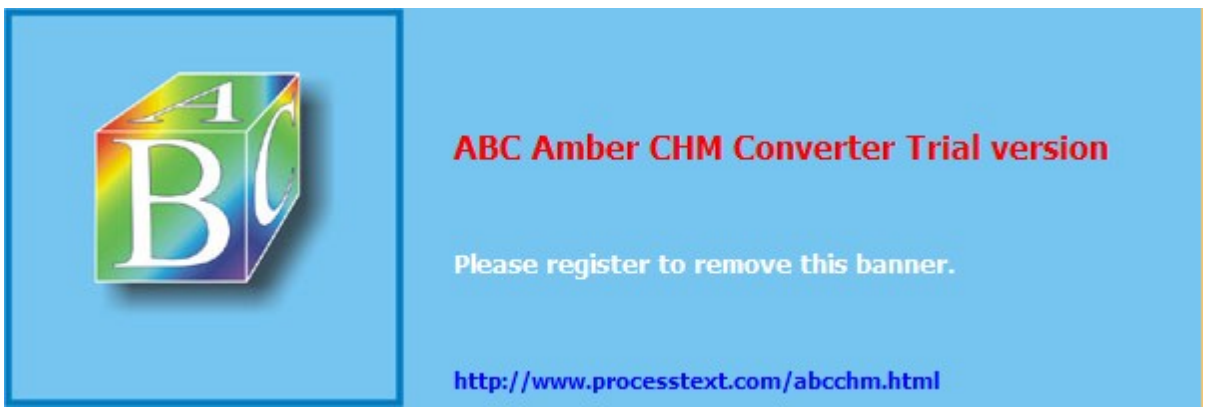
?*THE FELLOWSHIP OF THE RING*,J. R. R. TOLKIEN

Software and hardware go together. Without software, hardware would be lifeless silicon. Without hardware, software cannot exist. Software ultimately controls a computer, but under the hood, it's the hardware that implements the software code.

Furthermore, hardware is the ultimate enforcer of software security. Without hardware support, software would be totally insecure. Many texts cover software development without ever addressing the underlying hardware. This might work for the developers of enterprise applications, but it won't work for rootkit developers. As a rootkit developer, you will be faced with reverse-engineering problems, hand-coded assembly language, and highly technical attacks against software tools installed on the system. Your understanding of the underlying hardware will empower you to tackle these hard problems. Throughout the rest of this book, you will encounter concepts and code that assume you have some amount of hardware understanding. Therefore, we encourage you to read this chapter before moving on.

Ultimately, all access controls are implemented in hardware. For example, the popular notion of process separation is enforced using "rings" on the Intel x86 hardware. If the Intel CPU had no mechanism for access control, then all software executing on the system would be trusted. This would mean that any program that crashed could bring the whole system down with it. Any program would have the ability to read and write to hardware, access any file, or modify the memory of another process. Sound familiar? Even though the Intel family of processors have had access control capabilities for many years, Microsoft did not take advantage of these until the release of Windows NT.

In this chapter we discuss the hardware mechanisms that work behind the scenes to enforce security and memory access in the Windows operating system. We begin our discussion of hardware mechanisms by taking a look at how the Intel x86 family of microprocessors performs access control. We then discuss how the processor keeps track of matters using lookup tables. We also discuss control registers and, more importantly, how memory pages work.



Ring Zero

The Intel x86 family of microchips use a concept called *rings* for access control. There are four rings, with Ring Zero being the most privileged and Ring Three being the least privileged. Internally, each ring is stored as a number; there aren't actually physical rings on the microchip.

All kernel code in the Windows OS runs in Ring Zero. Therefore, rootkits running in the kernel are considered to be running in Ring Zero. User-mode programs, those that don't run in the kernel (for example, your spreadsheet program), are sometimes called *Ring Three programs.* Many operating systems, including Windows and Linux, take advantage of only Rings Zero and Three on the Intel x86 microchips; they do not use Rings One and Two.[1] Since Ring Zero is the most privileged and powerful ring on the system, it's a sign of pride for rootkit developers to claim that their code runs in Ring Zero.

[1] Although Rings One and Two may be used, the architecture of Windows does not require their use.

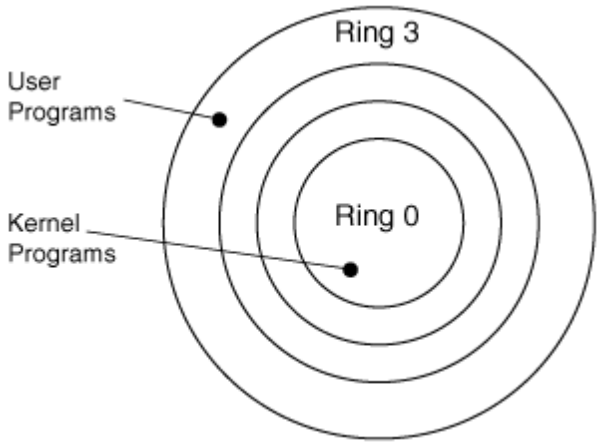
The CPU is responsible for keeping track of which software code and memory is assigned to each ring, and enforcing access restrictions between rings. Usually, each software program is assigned a ring number, and cannot access any rings with lower numbers. For example, a Ring Three program cannot access a Ring Zero program. If a Ring Three program attempts to access Ring Zero memory, the CPU will throw an interrupt. In most such cases, the access will not be allowed by the OS. The attempt might even result in the shutdown of the offending program.

Under the hood, quite a bit of code controls this access restriction. There is also code that allows a program to access lower rings under special circumstances. For example, loading a printer driver into the kernel requires that an administrator program (a Ring Three program) have access to the loaded device drivers (in the Ring Zero kernel). However, once a kernel-mode rootkit is loaded, its code will be executing in Ring Zero, and these access restrictions will cease to be of concern.

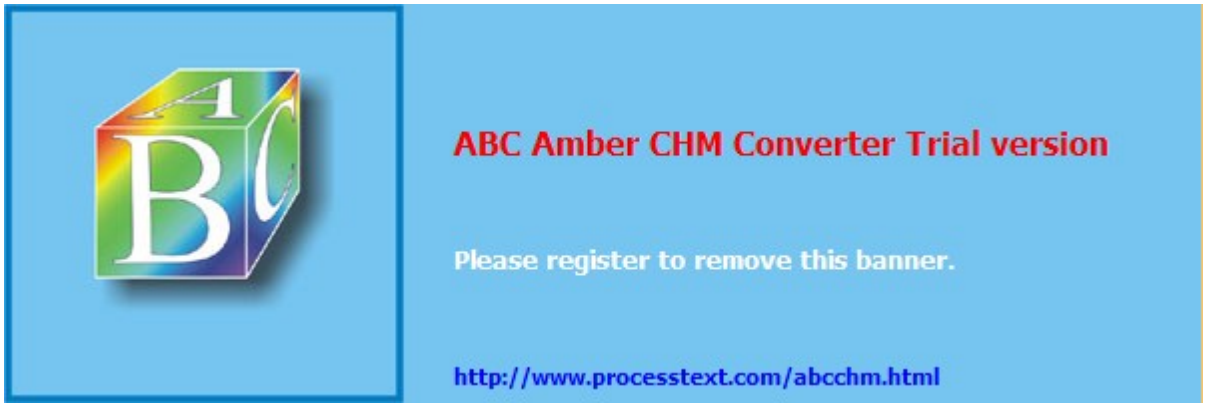
Many tools that might detect rootkits run as administrator programs in Ring Three. A rootkit developer should understand how to leverage the fact that her rootkit has a higher privilege than the administrator tool. For example, the rootkit can use this fact to hide from the tool, or render it inoperative. Also, a rootkit is typically installed using a loader program. (We covered loader programs in Chapter 2.) These loader programs are Ring Three applications. In order to load rootkit into the kernel, these loader programs use special function calls that allow them to access Ring Zero.

Figure 3-1 shows the rings of Intel x86 processors and where user-mode and kernel-mode programs execute within those rings.

Figure 3-1. The rings of Intel x86 processors.



In addition to memory-access restrictions, there are other security provisions. Some instructions are considered privileged, and can be used only in Ring Zero. These instructions are typically used to alter the behavior of the CPU or to directly access hardware. For example, the following x86 instructions are



Tables, Tables, and More Tables

In addition to being responsible for keeping track of rings, the CPU also is responsible for making many other decisions. For example, the CPU must decide what to do when an interrupt is thrown, when a software program crashes, when hardware signals for attention, when user-mode programs try to communicate with kernel-mode programs, and when multi-threaded programs switch threads. Clearly the operating system code must deal with such matters?but the CPU always deals with them first.

For every important event, the CPU must figure out which software routine deals with that event. Since every software routine lives in memory, it makes sense for the CPU to store addresses for important software routines. More specifically, the CPU needs to know where to *find* the address of an important software routine. The CPU cannot store all of the addresses internally, so it must look up the values. It does this by using tables of addresses. When an event occurs, such as an interrupt, the CPU looks up the event in a table and finds a corresponding address for some software to deal with that event. The only information the CPU needs is the base address of these tables in memory.

There are many important CPU tables, including:

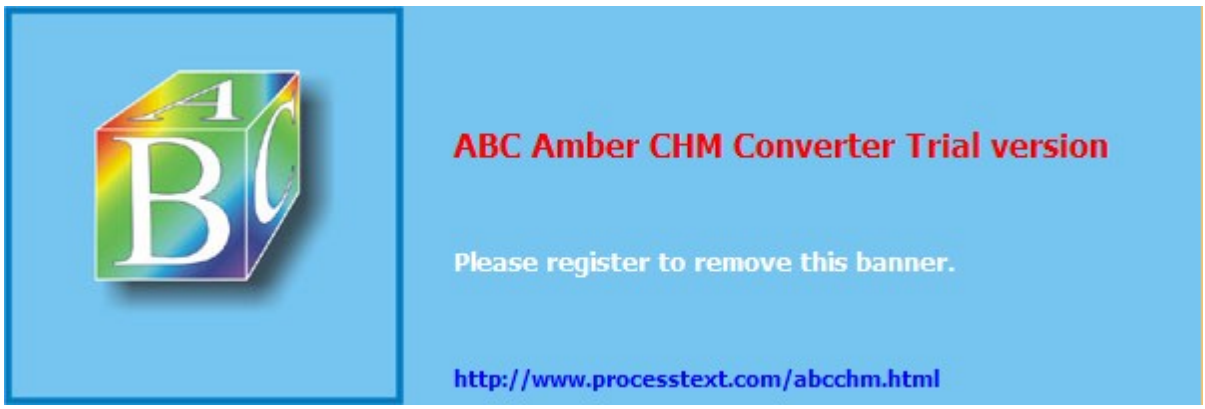
- Global Descriptor Table (GDT), used to map addresses
- Local Descriptor Table (LDT), used to map addresses
- Page Directory, used to map addresses
- Interrupt Descriptor Table (IDT), used to find interrupt handlers

In addition to CPU tables, the operating system itself may also keep tables. These OS-implemented tables are not directly supported by the CPU, so the OS includes special functions and code to manage them.

An important OS-implemented table is:

- System Service Dispatch Table (SSDT), used by the Windows OS for handling system calls

These tables are used in a variety of ways. In the following sections, we make reference to these tables and explore how they work. We also suggest ways a rootkit developer can modify or hook these tables in order to provide stealth or to capture data.



Memory Pages

All memory is separated into pages, as in a book. Each page can hold only a certain number of characters. Each process may have a separate lookup table to find these memory pages.

Imagine that memory is like a giant library of books, where every process has its own separate card catalog for looking things up. The different lookup tables can cause memory to be viewed entirely differently by each process. This is how one process can read memory at address 0x00401122 and see "GREG," while another process can read memory at the same address but see "JAMIE." Each process can have a unique "view" of memory.

Access controls are applied to memory pages. To continue our library metaphor, imagine that the CPU is an overbearing librarian who will allow a process to examine only a few books in the library. To read or write memory, a process must first find the correct "book," and then the exact "page" for the memory in question. If the CPU doesn't approve of the book or page that is requested, access is denied.

The lookup procedure for finding a page in this manner is long and involved; access control is enforced at several stages during this procedure. First, the CPU checks whether the process can open the book in question (the *descriptor* check); next, the CPU checks whether the process can read a certain chapter in the book (the *page directory* check); and finally, the cpu checks whether the process can read a particular page in the chapter (the *page* check). Wow—that is a lot of work!

Only if the process can pass all the security checks will it be allowed to read a page.

Even if the CPU checks are passed, the page may be marked as read-only. This, of course, means the process can read the page, but cannot write to it. In this way, the integrity of the data can be maintained. Rootkit developers are like vandals in this library, attempting to scribble all over these books—so we must learn all we can about manipulating access controls.

Memory Access Check Details

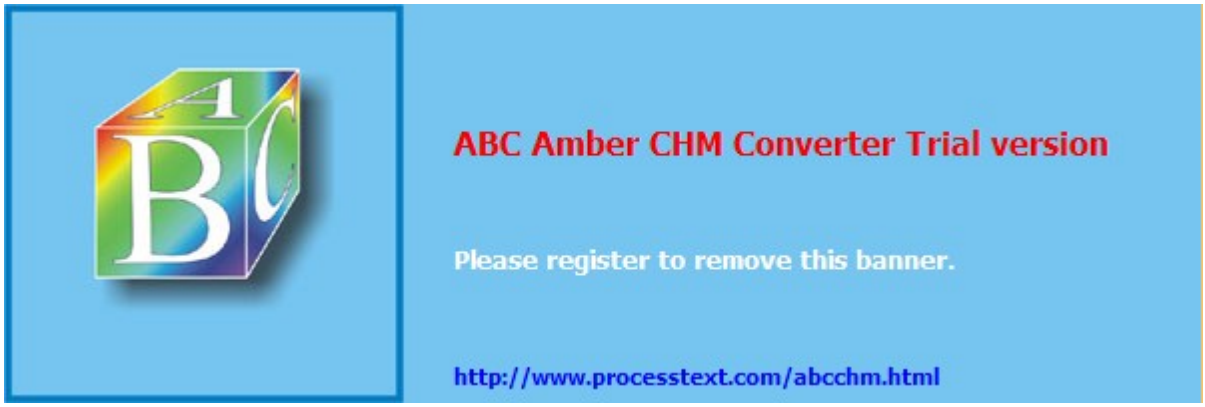
To access a memory page, the x86 processor performs the following checks, in the order shown:

- Descriptor (or *segment*) check: Typically, the global descriptor table (GDT) is accessed and a *segment descriptor* is checked. The segment descriptor contains a value known as the *descriptor privilege level* (DPL). The DPL contains the ring number (zero to three) required of the calling process. If the DPL requirement is lower than the ring level (sometimes called the *current privilege level* [CPL]) for the calling process, access is denied, and the memory check stops here.

- Page directory check: A user/supervisor bit is checked for an entire page table—that is, an entire range of memory pages. If the user/supervisor bit is set to zero, then only "supervisor" programs (Rings Zero, One, and Two) can access the range of memory pages; if the calling process is not a "supervisor," the memory check stops here. If the user/supervisor bit is set to 1, then any program can access the range of memory pages.

- Page check: This check is made for a single memory page. If the page-directory check has succeeded, a page check will be made for the individual page in question. Like the page directory, each individual page has a user/supervisor bit that is checked. If the user/supervisor bit is set to zero, then only "supervisor" programs (Rings Zero, One, and Two) can access the individual page. If the user/supervisor bit is set to 1, then any program can access the individual page. A process can access the page of memory only if it can get all the way to and through this check without any access denials.

The Windows family of operating systems does not really use the descriptor check. Instead, Windows relies *only* on Rings Zero and Three (sometimes called *kernel mode* and *user mode*). This allows the user/supervisor bit in the page table check *alone* to control access to memory. Kernel-mode programs,



The Memory Descriptor Tables

Some of the tables that the CPU uses to keep track of things can contain descriptors. There are several types of descriptors, and they can be inserted or modified by a rootkit.

The Global Descriptor Table

A number of interesting tricks may be implemented via the GDT. The GDT can be used to map different address ranges. It can also be used to cause task switches. The base address of the GDT can be found using the SGDT instruction. You can alter the location of the GDT using the LGDT instruction.

The Local Descriptor Table

The LDT allows each task to have a set of unique descriptors. A bit known as the *table-indicator bit* can select between the GDT and the LDT when a segment is specified. The LDT can contain the same types of descriptors as the GDT.

Code Segments

When accessing code memory, the CPU uses the segment specified in the code segment (CS) register. A code segment can be specified in the descriptor table. Any program, including a rootkit, can change the CS register by issuing a *far call, far jump,* or *far return,* where CS is popped from the top of the stack.[6] It is interesting to note that you can make your code execute only by setting the R bit to zero in the descriptor.

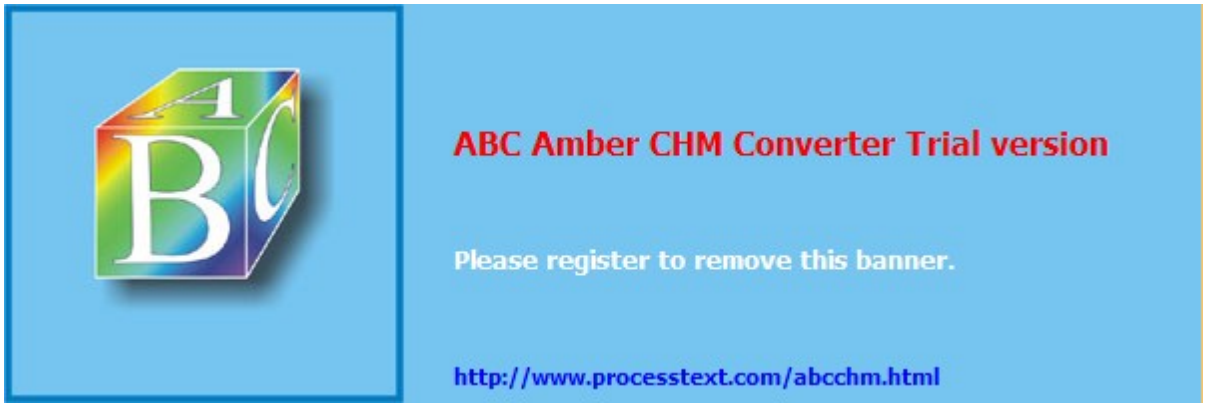
[6] An IRET instruction can also be used.

Call Gates

A special kind of descriptor, called a *call gate,* can be placed in the LDT or the GDT. A program can make a far call with the descriptor set to the call gate. When the call occurs, a new ring level can be specified. A call gate could be used to allow a user-mode program to make a function call into kernel mode. This would be an interesting back door for a rootkit program. The same mechanism can be used with a far jump, but only when the call gate is of the same privilege level or lower than process performing the jump.[7]

[7] The exception is a far jump to a "conforming" code segment.

When a call gate is used, the address is ignored?only the descriptor number matters. The call gate data structure tells the CPU where the code for the called function lives. Optionally, arguments can be read from the stack. For example, a call gate could be created such that the caller puts secret command arguments onto the stack.



The Interrupt Descriptor Table

The *interrupt descriptor table register* (IDTR) stores the base (the start address) of the *interrupt descriptor table* (IDT) in memory. The IDT, used to find the software function employed to handle an interrupt, is very important.[8] Interrupts are used for a variety of low-level functions in a computer. For example, an interrupt is signaled whenever a keystroke is typed on the keyboard.

[8] Also, for interrupt handling to occur on a CPU, the IF bit in that CPU's EFlags register must be set.

The IDT is an array that contains 256 entries?one for each interrupt. That means there can be up to 256 interrupts for each processor. Also, each processor has its own IDTR, and therefore has its own interrupt table. If a computer has multiple CPUs, a rootkit deployed on that computer must take into account that each CPU has its own interrupt table.

When an interrupt occurs, the interrupt number is obtained from the interrupt instruction, or from the programmable interrupt controller (PIC). In either case, the interrupt table is used to find the appropriate software function to call. This function is sometimes called a *vector* or *interrupt service routine (ISR)*.

When the processor is in protected mode, the interrupt table is an array of 256 eight-byte entries. Each entry has the address of the ISR and some other security-related information.

To obtain the address of the interrupt table in memory, you must read the IDTR. This is done using the SIDT (Store Interrupt Descriptor Table) instruction. You can also change the contents of the IDTR by using the LIDT (Load Interrupt Descriptor Table) instruction. More details on this technique can be found in Chapter 8.

One trick employed by rootkits is to create a new interrupt table. This can be used to hide modifications made to the original interrupt table. A virus scanner may check the integrity of the original IDT, but a rootkit can make a copy of the IDT, change the IDTR, and then happily make modifications to the copied IDT without detection.

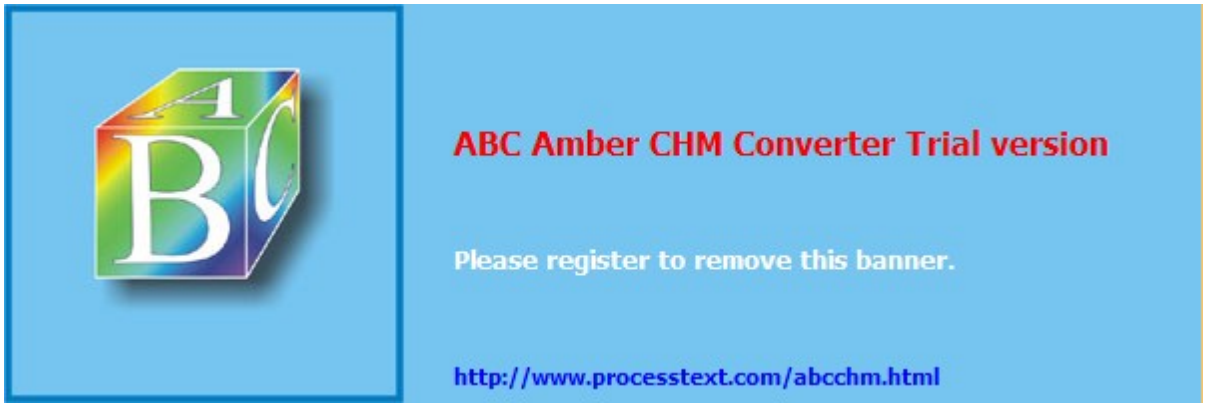
The SIDT instruction stores the contents of the IDTR in the following format:

```
/* sidt returns idt in this format */
typedef struct
{
   unsigned short IDTLimit;
   unsigned short LowIDTbase;
   unsigned short HiIDTbase;
} IDTINFO;
```

Using the data provided by the SIDT instruction, an attacker can then find the base of the IDT and dump its contents.

Remember that the IDT can have up to 256 entries. Each entry in the IDT contains a pointer to an interrupt service routine. The entries have the following structure.

```
// entry in the IDT: this is sometimes called
// an "interrupt gate"

#pragma pack(1)
typedef struct
{
      unsigned short LowOffset;
      unsigned short selector;
      unsigned char unused_lo;
      unsigned char segment_type:4; //0x0E is interrupt gate
```



The System Service Dispatch Table

The system service dispatch table is used to look up the function required to handle a given system call. This facility is implemented in the operating system, not by the CPU. There are two ways a program can make a system call: by using interrupt 0x2E, or by using the SYSENTER instruction.

On Windows XP and beyond, programs typically use the SYSENTER instruction, while older platforms use interrupt 0x2E. The two mechanisms are completely different, although they achieve the same result.

Making a system call results in the function KiSystemService being called in the kernel. This function reads the system-call number from the EAX register, and looks up the call in the SSDT. KiSystemService also copies the arguments for the system call from the user-mode stack onto the kernel-mode stack. The arguments are pointed to by the EDX register. Some rootkits will hook into this processing chain to sniff data, alter data arguments, or redirect the system call. This technique is covered in great detail in Chapter 4.

The Control Registers

Aside from the system tables, a few special registers control important features of the CPU. These registers may be used by rootkits.

Control Register Zero (CR0)

The control register contains bits that control how the processor behaves. A popular method for disabling memory-access protection in the kernel involves modifying a control register known as CR0.

The control register was first introduced in the lowly '286 processor and was previously called the *machine status word*. It was renamed *Control Register Zero* (CR0) with the release of the '386 family of processors. It wasn't until the '486 series of processors that the *write protect* (WP) bit was added to CR0. The WP bit controls whether the processor will allow writes to memory pages marked as read-only. Setting WP to zero disables memory protection. This is very important for kernel rootkits that are intended to write to OS data structures.

The following code shows how to disable and re-enable memory protection using the CR0 trick.

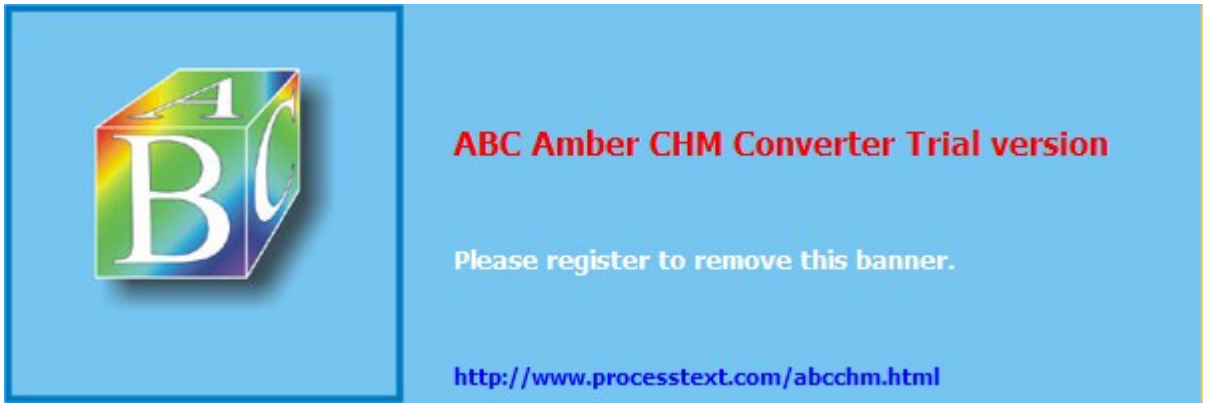
```
// UN-protect memory
__asm
{
      push eax
      mov  eax, CR0
      and  eax, 0FFFEFFFFh
      mov  CR0, eax
      pop  eax
}
// do something
// RE-protect memory
__asm
{
      push eax
      mov  eax, CR0
      or   eax, NOT 0FFFEFFFFh
      mov  CR0, eax
      pop  eax
}
```

Other Control Registers

There are four more control registers, and they handle other functions for the processor. CR1 remains unused or undocumented. CR2 is used when the processor is in protected mode; it stores the last address that caused a page fault. CR3 stores the address of the page directory. CR4 was not implemented until the Pentium series of processors (and later versions of the '486); it handles matters such as when the virtual 8086 mode is enabled?that is, when running an old DOS program on Windows NT. If this mode is enabled, the processor will trap privileged instructions such as CLI, STI, and INT. For the most part, these additional registers are not useful for rootkits.

The EFlags Register

The EFlags register is also important. For one thing, it handles the *trap flag*. When this flag is set, the processor will single-step. A rootkit can use a feature such as single-stepping to detect whether a debugger is running or to hide from virus-scanner software. You can disable interrupts by clearing the *interrupt flag*. Also, the I/O Privilege Level can be used to modify the ring-based protection system used by most Intel-based operating systems.



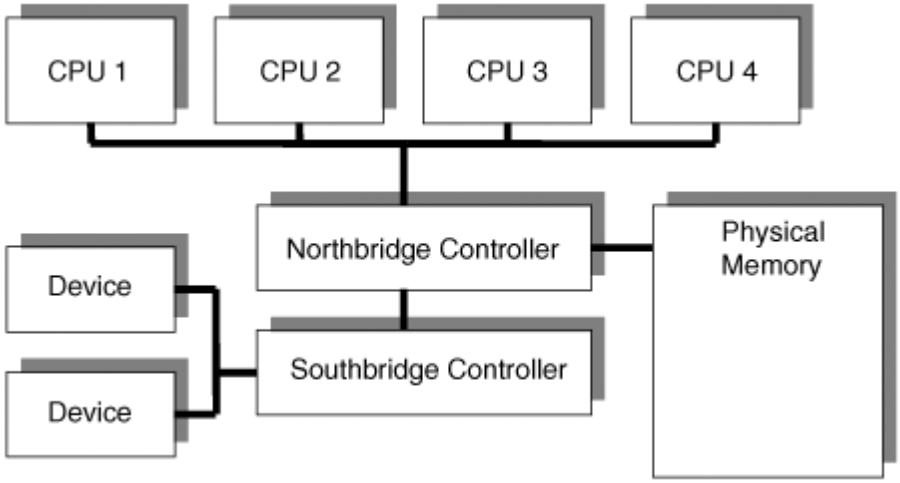
Multiprocessor Systems

Multiprocessor systems (sometimes known as Symmetric Multi-Processing [SMP] systems) and hyper-threaded systems come with their own unique set of problems. The major issue they pose for rootkit developers is synchronization. If you have written multi-threaded applications, you have already come to understand thread safety (we hope!), and what can happen if two threads access a data object at the same time. If you haven't, suffice it to say that if two different operations access the same data object at the same time, the data object will become corrupted. It's like having too many cooks in the kitchen!

Multiple-processor systems are like multi-threaded environments in a way, because code can be executing on two or more CPUs at once. Chapter 7, Direct Kernel Object Manipulation, covers multiprocessor synchronization.

The layout of a typical multiprocessor system is shown in Figure 3-8. As the figure illustrates, multiple CPUs share access to a single memory area, set of controllers, and group of devices.

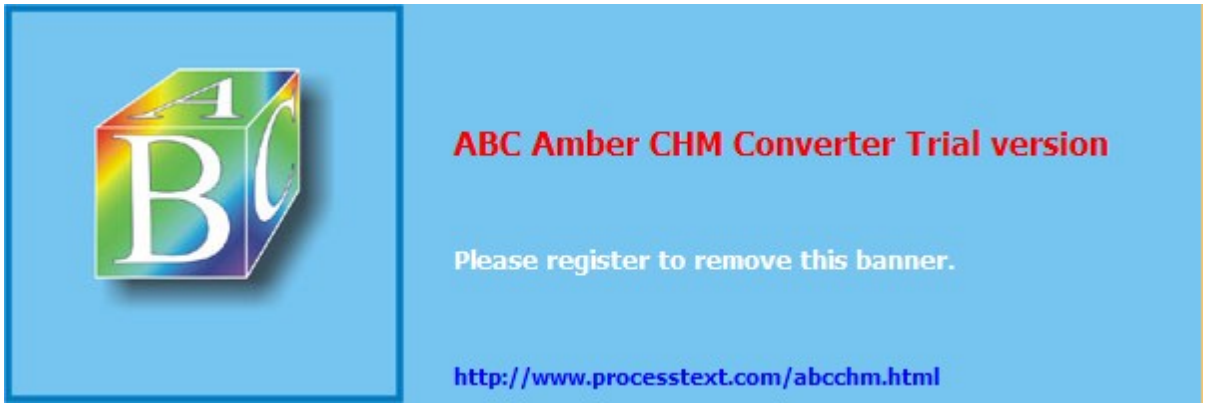
Figure 3-8. A typical multiprocessor bus layout.



Some points to remember about multiprocessor systems:

Every CPU Has its Own Interrupt Table. If you hook the interrupt table, remember to hook it for all the CPUs! If you don't, then your hook will only apply to a single CPU. This may be intentional if you don't need to have 100% control over an interrupt?but this is rare.

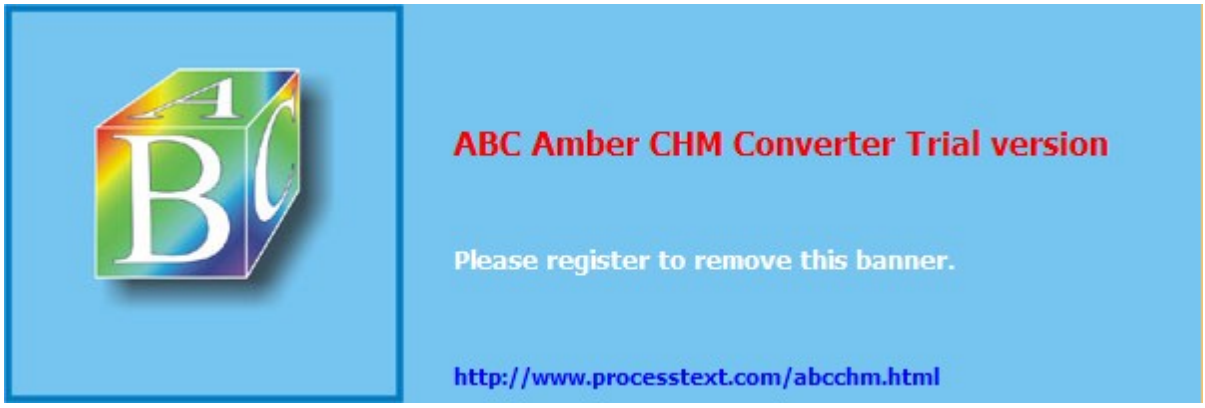
- A driver that works fine on a single processor system may crash (produce a Blue Screen of Death) on a multiprocessor system. You must include multiprocessor systems into your test plan.
- The same driver function can be running in multiple contexts, on multiple CPUs, simultaneously. The only way to make this safe is to use locking and synchronization with shared resources.
- Multiprocessor systems provide interlock routines, Spinlocks, and Mutexes. These are tools provided by the system to help you synchronize access to data. Details on their use can be found in the DDK documentation.
- Don't implement your own locking mechanisms. Use the tools the system already provides. If you really must do it yourself, then you must familiarize yourself with *memory barriers* (KeMemoryBarrier, etc.) and *hardware reordering* of instructions. These topics are beyond the scope of this book.
- Detect which processor you are running on. You can use a call like KeGetCurrentProcessorNumber to determine which processor your code is currently running on. You can also use KeGetActiveProcessors to determine how many active processors are in



Conclusion

This chapter has introduced the hardware-level mechanisms that work behind the scenes to enforce security and memory access in the operating system. It also has covered, in some detail, the use of the interrupt table. This knowledge is a basis upon which you can grow your understanding of computer manipulation. Because the hardware is ultimately responsible for implementing the software, all software is subject to manipulations applied at the hardware level. Thoroughly understanding these concepts is the starting point for true rootkit skills and the ability to subvert any other software running on the system.

PREV < Day Day Up > NEXT



Chapter 4. The Age-Old Art of Hooking

How does the sea become the king of all streams?Because it lies lower than they!Hence it is the king of all streams.

LAO TZU

The two purposes of most rootkits are to allow continued access to the computer and to provide stealth for the intruder. To achieve these objectives, your rootkit must alter the execution path of the operating system or directly attack the data that stores information about processes, drivers, network connections, etc. Chapter 7, Direct Kernel Object Manipulation, discusses the latter approach. In this chapter, we will cover altering the execution path of important reporting functions provided by the operating system. We will begin with a discussion of simple userland hooks in a target process, then advance to covering more global kernel-level hooks. At the end of the chapter, we will present a hybrid method. Keep in mind that the goal is to intercept the normal execution flow and alter the information returned by the operating system's report-ing APIs.

Userland Hooks

In Windows, there are three subsystems on which most processes depend. They are the Win32, POSIX, and OS/2 subsystems. These subsystems comprise a well-documented set of APIs. Through these APIs, a process can request the aid of the OS. Because programs such as Taskmgr.exe, Windows Explorer, and the Registry Editor rely upon these APIs, they are a perfect target for your rootkit.

For example, suppose an application lists all the files in a directory and performs some operation on them. This application may run in user space as a user application or as a service. Assume further that the application is a Win32 application, which implies it will use Kernel32, User32.dll, Gui32.dll, and Advapi.dll to eventually issue calls into the kernel.

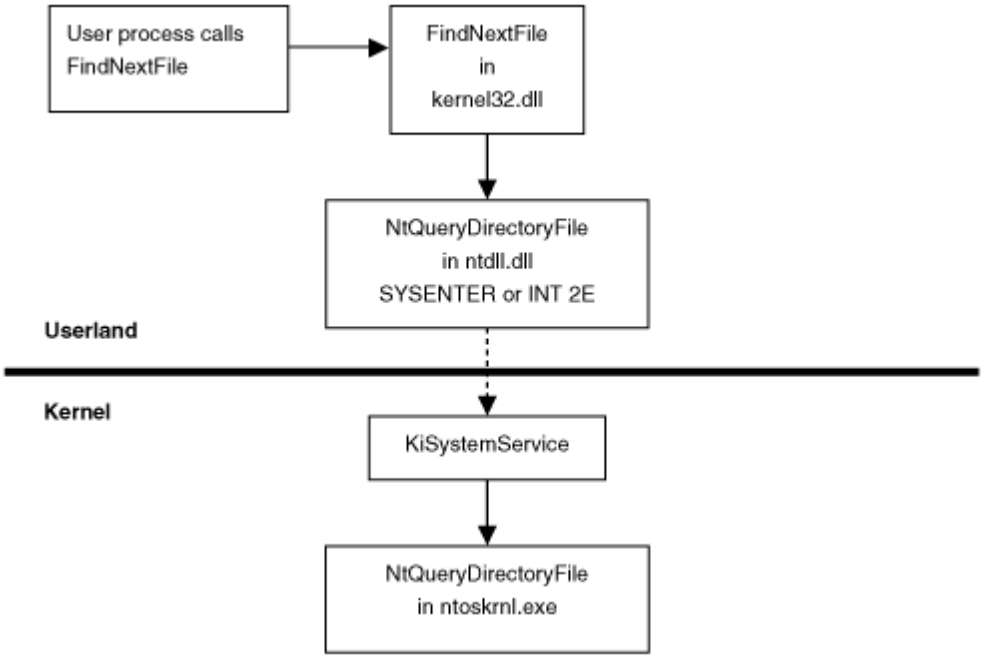
Under Win32, to list all the files in a directory, an application first calls FindFirstFile, which is exported by Kernel32.dll. FindFirstFile returns a handle if it is successful.

This handle is used in subsequent calls to FindNextFile to iterate through all the files and subdirectories in the directory. FindNextFile is also an exported function in Kernel32.dll. To use these functions, the application will load Kernel32.dll at runtime and copy the memory addresses of the functions into its function Import Address Table (IAT). When the application calls FindNextFile, execution in the process jumps to a location in its IAT. Execution in the process then continues to the address of FindNextFile in Kernel32.dll. The same is true for FindFirstFile.

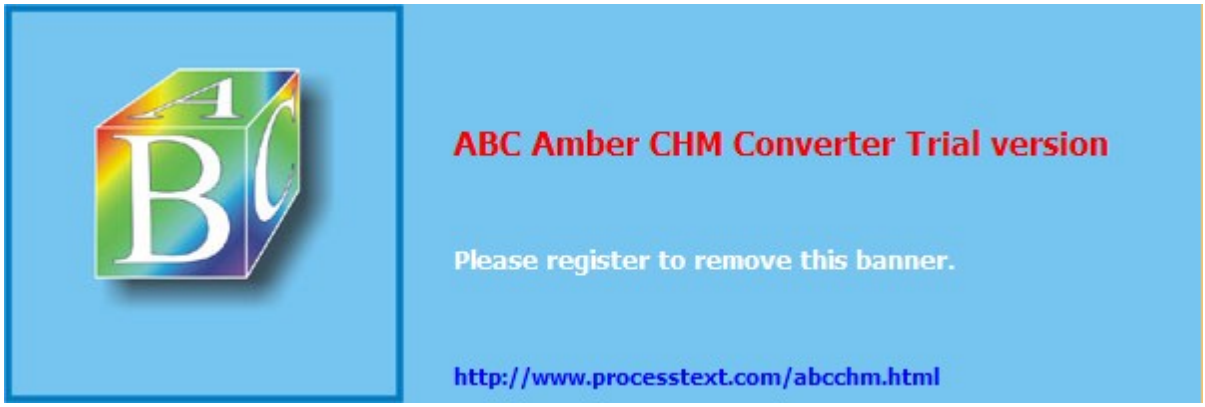
FindNextFile in Kernel32.dll then calls into Ntdll.dll. Ntdll.dll loads the EAX register with the system service number for FindNextFile's equivalent kernel function, which happens to be NtQueryDirectoryFile. Ntdll.dll also loads EDX, with the user space address of the parameters to FindNextFile. Ntdll.dll then issues an INT 2E or a SYSENTER instruction to trap to the kernel. (These traps into the kernel are covered later in this chapter.) This sequence of calls is illustrated in Figure 4-1.

Figure 4-1. FindNextFile execution path.

[View full size image]



Because the application loads Kernel32.dll into its private address space between memory addresses 0x00010000 and 0x7FFE0000, your rootkit can directly overwrite any function in Kernel32.dll or in the application's import table as long as the rootkit can access the address space of the target process. This is called *API hooking*. In our example, your rootkit could overwrite FindNextFile with your own hand-crafted machine code in order to prevent listing of certain files or otherwise change the performance of FindNextFile. The rootkit could also overwrite the import table in the target application so that it points to the rootkit's own function instead of Kernel32.dll's. By hooking APIs, you can hide a process



Kernel Hooks

As explained in the previous section, userland hooks are useful, but they are relatively easy to detect and prevent. (Userland-hook detection is discussed in detail in Chapter 10, Rootkit Detection.) A more elegant solution is to install a kernel memory hook. By using a kernel hook, your rootkit will be on equal footing with any detection software.

Kernel memory is the high virtual address memory region. In the Intel x86 architecture, kernel memory usually resides in the region of memory at 0x80000000 and above. If the /3GB boot configuration switch is used, which allows a process to have 3 GB of virtual memory, the kernel memory starts at 0xC0000000.

As a general rule, processes cannot access kernel memory. The exception to this rule is when a process has debug privileges and goes through certain debugging APIs, or when a call gate has been installed. We will not cover these exceptions here. For more information on call gates refer to the Intel Architecture Manuals.[4]

[4] *IA-32 Intel Architecture Software Developer's Manual,* Volume 3, Section 4.8.

For our purposes, your rootkit will access kernel memory by implementing a device driver.

The kernel provides the ideal place to install a hook. There are many reasons for this, but the two that are most important to remember are that kernel hooks are global (relatively speaking), and that they are harder to detect, because if your rootkit and the protection/detection software are both in Ring Zero, your rootkit has an even playing field on which to evade or disable the protection/detection software. (For more on rings, refer to Chapter 3, The Hardware Connection.)

In this section, we will cover the three most common places to hook, but be aware that you can find others depending on what your rootkit is intended to accomplish.

Hooking the System Service Descriptor Table

The Windows executive runs in kernel mode and provides native support to all of the operating system's subsystems: Win32, POSIX, and OS/2. These native system services' addresses are listed in a kernel structure called the *System Service Dispatch Table* (SSDT).[5] This table can be indexed by system call number to locate the address of the function in memory. Another table, called the System Service Parameter Table (SSPT),[6] specifies the number of bytes for the function parameters for each system service.

[5] P. Dabak, S. Phadke, and M. Borate, *Undocumented Windows NT* (New York: M&T Books, 1999), pp. 117–29.

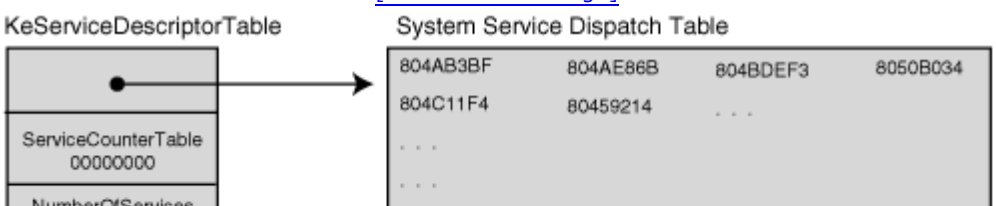
[6] *Ibid.*, pp. 128–9.

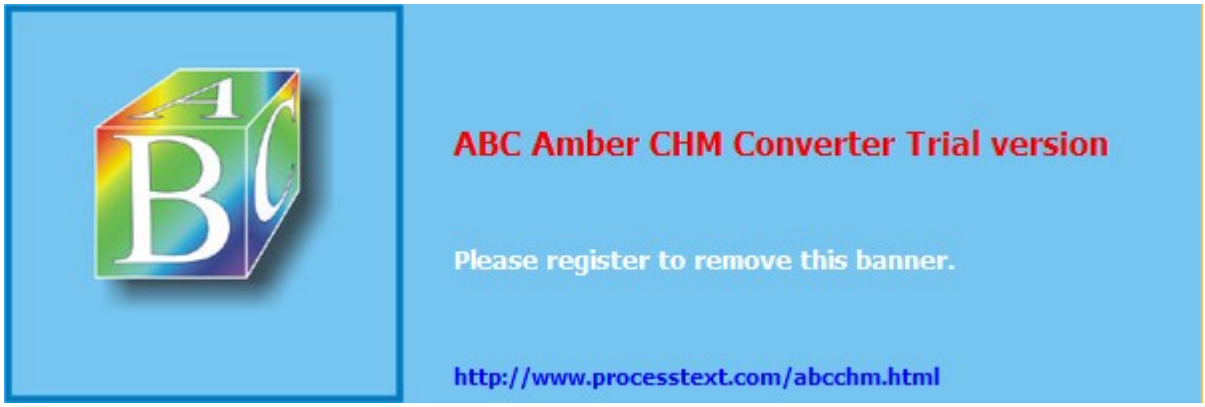
The KeServiceDescriptorTable is a table exported by the kernel. The table contains a pointer to the portion of the SSDT that contains the core system services implemented in Ntoskrnl.exe, which is a major piece of the kernel. The KeServiceDescriptorTable also contains a pointer to the SSPT.

The KeServiceDescriptorTable is depicted in Figure 4-4. The data in this illustration is from Windows 2000 Advanced Server with no service packs applied. The SSDT in Figure 4-4 contains the addresses of individual functions exported by the kernel. Each address is four bytes long.

Figure 4-4. KeServiceDescriptorTable.

[View full size image]





A Hybrid Hooking Approach

Userland hooks have their place. They are usually easier to implement than kernel-mode hooks. Also, some of the functions your rootkit may be designed to filter may not have obvious paths through the kernel.

However, we do not recommend implementing a rootkit using userland hooks. The reason: if a detection mechanism is implemented in the kernel, your rootkit will not be on an even footing with its adversary, the detection software.

Typically, the detection process involves observing the ways in which code is induced to execute in another process's address space. When this mode of detection or prevention is expected, a hybrid approach may be the answer. The hybrid hooking approach is designed to hook a userland process by using an Import Address Table (IAT) hook, but to do so without opening a handle to the target process, using WriteProcessMemory, changing a Registry key, or engaging in other readily detectable activities.

The HybridHook example presented in the following discussion hooks the userland process from a kernel driver.

Getting into a Process's Address Space

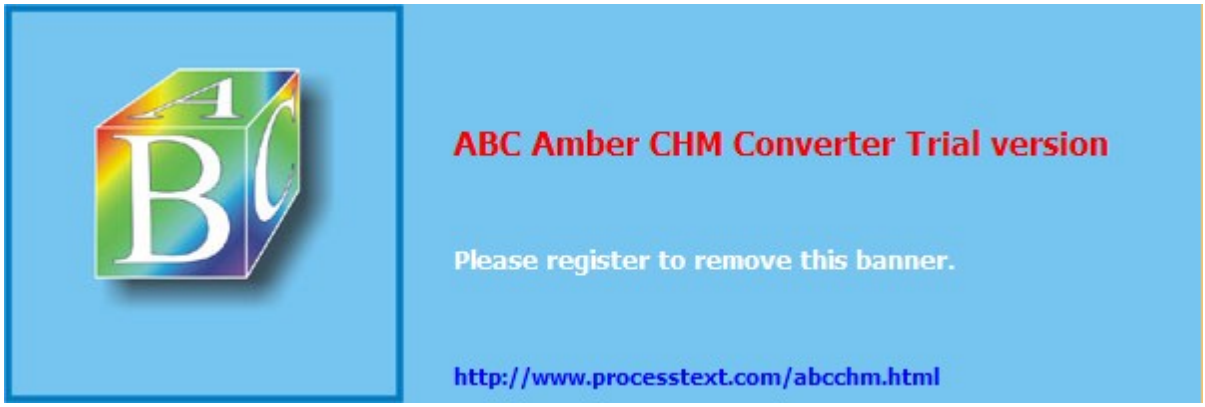
The operating system provides a very useful function if you want to be notified when your target process or DLL is loaded. It is called PsSetImageLoadNotifyRoutine. As the name suggests, this function registers a driver callback routine that will be called every time an image is loaded into memory. The function takes only one parameter, the address of your callback function. Your callback routine should be declared as follows:

```
VOID MyImageLoadNotify(IN PUNICODE_STRING,
                       IN HANDLE,
                       IN PIMAGE_INFO);
```

The UNICODE_STRING contains the name of the module loaded by the kernel. The HANDLE parameter is the Process ID (PID) of the process the module is being loaded into. Your rootkit is already in the memory context of this PID. The IMAGE_INFO structure is full of good information your rootkit will need, such as the base address of the image being loaded into memory. It is defined as follows:

```
typedef struct  _IMAGE_INFO {
    union {
      ULONG  Properties;
      struct {
          ULONG ImageAddressingMode  : 8; //code addressing
mode
          ULONG SystemModeImage      : 1; //system mode image
          ULONG ImageMappedToAllPids : 1; //mapped in all
processes
          ULONG Reserved             : 22;
      };
    };
    PVOID  ImageBase;
    ULONG  ImageSelector;
    ULONG  ImageSize;
    ULONG  ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

In your callback function, you must determine whether this is a module whose IAT you wish to hook. If



Conclusion

In this chapter, we provided a lot of information about hooking tables of function pointers, both in userland and in the kernel. Kernel hooks are preferred, because if a detection/protection software suite is looking for your rootkit, you may employ all the power of the kernel to evade or defeat it. Kernel-level access provides a vast number of places to hide from or ways to defeat the enemy. Since stealth is a primary goal for your rootkit, filtering in some fashion is a must.

Hooking is truly a dual-use technology. It is used by many public rootkits and other malicious software, but it is also used by anti-virus software and other host-protection products.

PREV < Day Day Up > NEXT

Chapter 5. Runtime Patching

All I need to find you, Louis, is follow the corpses of rats.

?*INTERVIEW WITH THE VAMPIRE*, ANNE RICE

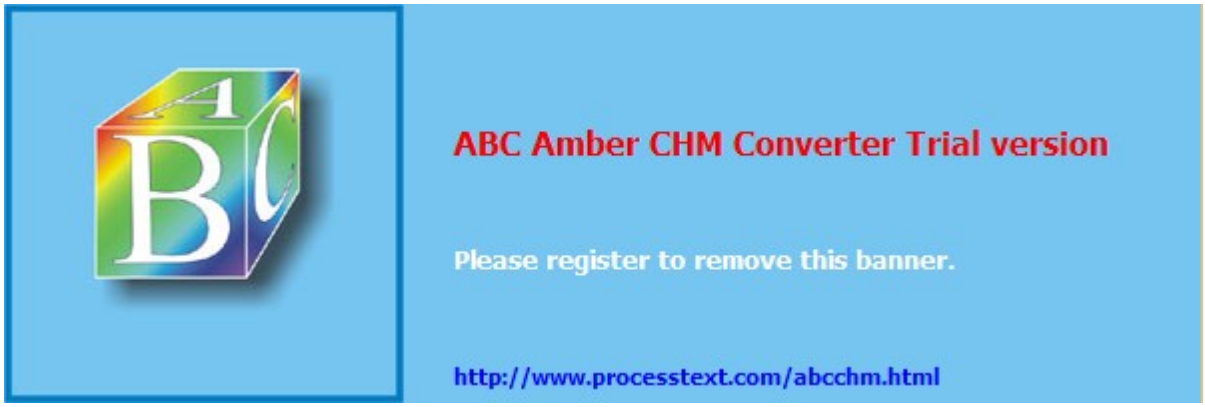
Call hooks and other methods of modifying software logic are powerful for sure, but they're old techniques, they're well published, and they're easily detected by anti-rootkit technology. Runtime patching offers a more-obscure way to achieve the same results. Runtime patching is not new, but in the published material relating to rootkits it typically has not been showcased.

Most material relating to code patches goes back to the days of software cracking and piracy. But applied in rootkits, runtime patching is one of the most advanced techniques possible. Armed with this technique, you should be able to build undetectable rootkits, even against modern intrusion-prevention systems. If you combine runtime patching with low-level hardware manipulation (such as page-table management,) you will be operating on the bleeding edge of rootkits.

The logic of software can be modified in several ways. The most obvious way is to modify the source code and then recompile the software. This is the practice of developers. The second way is to directly modify the bits and bytes that result from compilation?the *binary* software. This is what software crackers do, and is the basic approach to removing copy protection on software. The third way is to modify the data that is stored in memory when the software executes. In-memory data structures control how a program behaves; thus, changing this data changes the program logic. Good examples of this are "game trainers" that alter games to, for example, give the player 10 million gold pieces.

Modifying code logic is simple in comparison to rewriting or replacing files on the system with Trojan devices. By flipping a few bytes here and there, you can turn off most security functions. Of course, you have to be able to read and write the memory where these security functions reside. Since our rootkits operate from the kernel, we have full access to the memory space of the computer, so this typically isn't a problem.

In this chapter you will learn how to modify code logic using one of the strongest methods available: the *direct code-byte patch* method. You also will learn how to combine this with other powerful methods, such as detour patching and jump templates, to develop a very deadly and hard-to-detect rootkit.



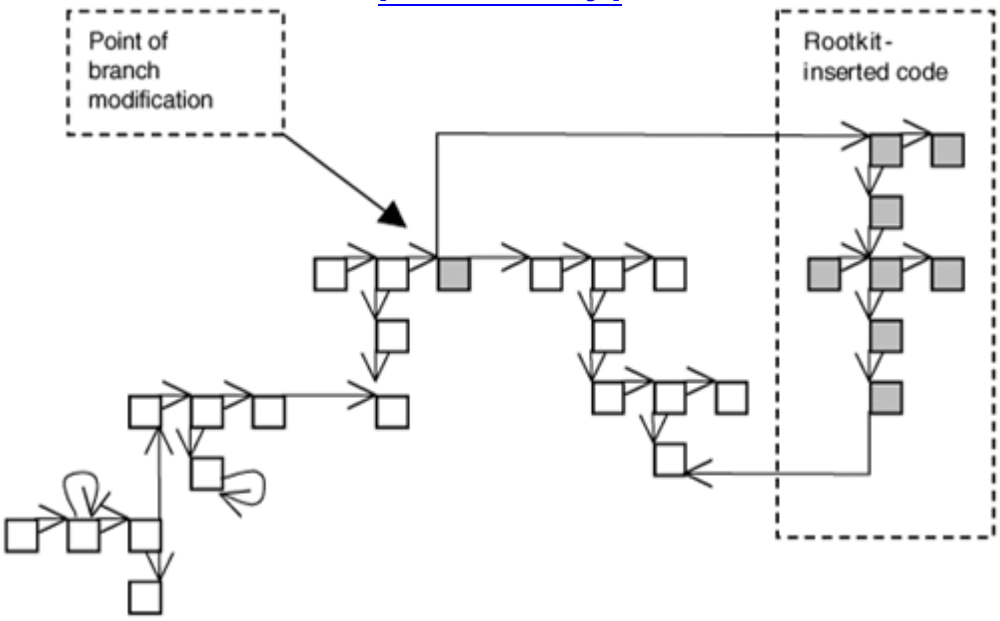
Detour Patching

In Chapter 4, we saw the power of using call hooks as a convenient way to modify program behavior. One downside of the call hook is that it modifies call tables, and this can be detected by anti-virus and anti-rootkit technology. A subtler approach to the problem is to patch the bytes within the function itself by inserting a jump into rootkit code. Additionally, modifying just a single function can affect multiple tables pointing to that function, without the need to keep track of all the tables that point to the function. This technique is called *detour patching,* and can be used to reroute the control flow around a function.

Figure 5-1 illustrates how code is inserted by the rootkit into the control flow.

Figure 5-1. Modification of control flow.

[View full size image]



As with a call hook, we can insert rootkit code to modify arguments before and after a system call or function call. We can also make the original function call as if it had never been patched. Finally, we can rewrite the logic of the function call altogether. For example, we can make the call always return a certain error code.

Detour patching is best illustrated by example. The technique requires several steps which are detailed in the following sections.

Rerouting the Control Flow Using MigBot

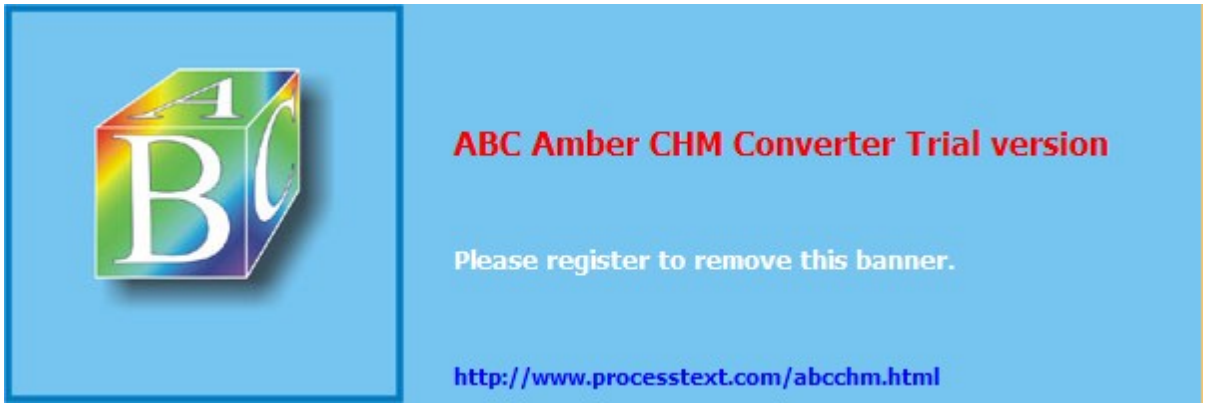
Migbot is an example rootkit that illustrates detour patches on kernel functions.

Rootkit.com

MigBot can be downloaded from rootkit.com at: www.rootkit.com/vault/hoglund/migbot.zip

MigBot reroutes the control flow from two important kernel functions: NtDeviceIoControlFile, and SeAccessCheck.

Rerouting a function requires first finding the function in memory. An advantage of the two functions we have chosen is that they are exported. This makes them easier to locate, because there is a table in the PE header where we can perform a lookup to find them. In the code for MigBot, we simply refer to the functions by their exported names. Because they are exported, there is no need to hunt through PE headers and such.[1]



Jump Templates

We now detail a technique called *jump templates*. This technique can be used in a variety of ways, but we illustrate it with a hook on the interrupt table.

The following example counts the number of times each interrupt is called. Instead of patching the interrupt service routine (ISR) directly, we craft a special bit of code that will be executed for each ISR. To do this, we start with a template. In this case, we make hundreds of copies of the template?one for each ISR. That is, instead of creating a single hook, we create an individual hook for each entry in the IDT.

Rootkit.com

The following example can be downloaded from rootkit.com at the address:
www.rootkit.com/vault/hoglund/basic_interrupt_3.zip

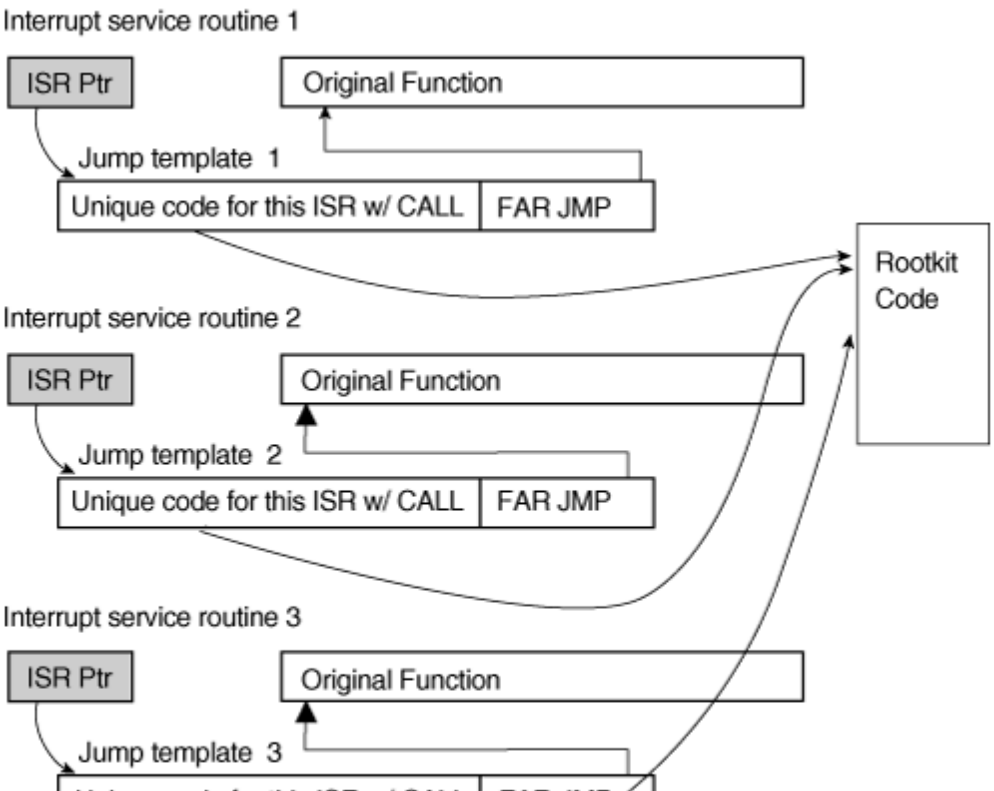
Because each interrupt service routine exists at a different address, and therefore the reentry address is unique for each one, we must introduce a new technique that allows each individual entry to be hooked with unique jump details.

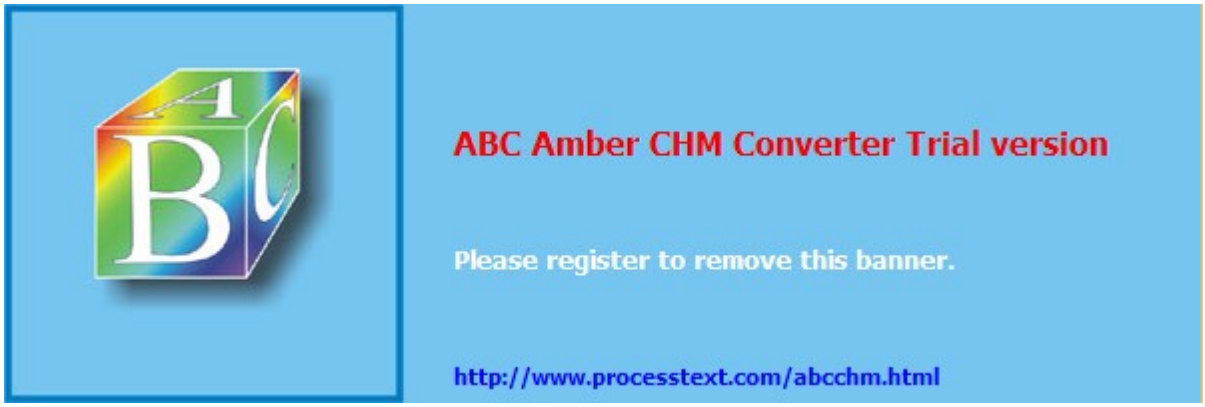
In the previous example, the rootkit code itself jumped back into the original function. That method works only when there is just a single hook. Instead of re-coding the same function hundreds of times we use a jump template to call into the rootkit code and then branch back to the original function.

The jump template is replicated for each interrupt routine. The FAR JMP address in each replicated copy is fixed up uniquely for each corresponding interrupt routine.

Figure 5-4 illustrates this technique. Each template calls the same rootkit code?which in this case is treated like a normal function. A function always returns to its caller, so we don't need to worry about runtime address fixups in the rootkit code. This technique allows specific, unique code to be applied to each ISR hook. In our example, the unique code holds the correct interrupt number for each interrupt handler.

Figure 5-4. Use of jump templates.





Variations on the Method

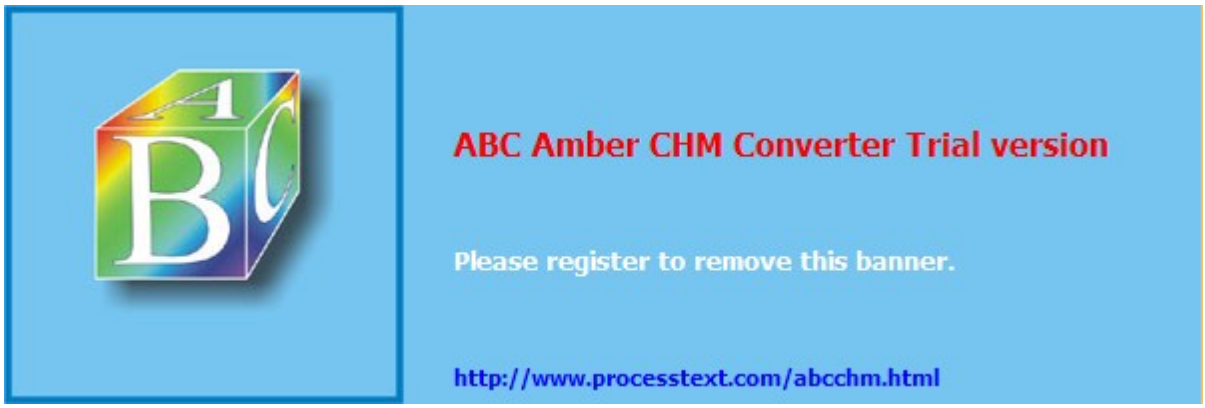
As you've seen, the common place to insert code patches into a function is at the very beginning of the function. This is easy, because functions are easy to find in memory. Of course, we don't need to stop there; we can also patch code bytes deep within the function itself. Deeper code patches provide better stealth and, therefore, aren't as easy to detect. Some rootkit-detection software checks the integrity of only the first 20 bytes of a function. If you place your code modification past the initial 20-byte mark, you remain undetected by that software.

Searching for code bytes to patch can sometimes work well. If the series of code bytes you wish to patch are unique, you can simply search for them in memory and patch them. When the code can simply be searched for, there is no need to use function pointers to find it. If the patch itself is simple, you can sometimes search for unique code bytes that are near the intended patch location. The trick is to find some code bytes that are unique, so they can be searched for without generating false hits.

Authentication functions are also good places to modify code. These can be disabled completely so that they always offer access. A more-complex patch could allow a backdoor password or username.

Patches to general-purpose kernel functions can provide stealth for the installed driver and programs. A fairly interesting place to patch is the loader program that loads the kernel itself. Integrity-checking functions can be patched so that they no longer detect Trojan or modified files. Patches to network functions can be used to sniff packets and other data. Patches to firmware and the BIOS can be hard to detect.

When patching and inserting code, you sometimes need to insert a great number of new instructions. From a driver, the best way to proceed is to allocate non-paged pool memory. For more-esoteric patches, however, you may wish to put your code into unused memory. There are unused sections of memory at the bottom of many memory pages. Using these lower regions of existing pages is sometimes called *cavern infection* (the unused section of memory being known as a *cavern*).



Conclusion

Generally speaking, the direct code-byte patch is one of the strongest methods for modifying program logic. Almost any program code or logic can be modified. Furthermore, the technique is somewhat difficult to detect?at least with current rootkit-detection technology.

Code-byte patches offer an alternative way to implement many of the hooking strategies described in this book. If combined with other powerful techniques, such as direct hardware access and virtual-memory obfuscations, the direct code-byte patch can be used to develop a very deadly and hard-to-detect rootkit.

Overall, runtime patching is a staple technique for modern rootkit development.

Chapter 6. Layered Drivers

If you have a difficult task, give it to a lazy person; he will find an easier way to do it.

THLADE'S LAW

Developers engineer clever solutions to avoid work. In fact, this laziness drives many innovations in code. The ability to layer drivers is one such innovation. Using layers, a developer can chain multiple drivers together. In this way, a developer can modify the behavior of an existing driver without coding a whole new driver from scratch.

Think about it: What if you want to encrypt the contents of a hard drive? Would you like to write an NTFS driver from scratch that supports not only the exact hardware of the drive mechanism, but also its NTFS protocol and encryption routines? Using layered drivers, this is not necessary. You simply intercept the data as it travels to the pre-existing NTFS driver and modify it with encryption. More importantly, the details of the NTFS protocol can be decoupled from the hardware details of the drive mechanism. This elegant idea applies to most drivers in the Windows environment.

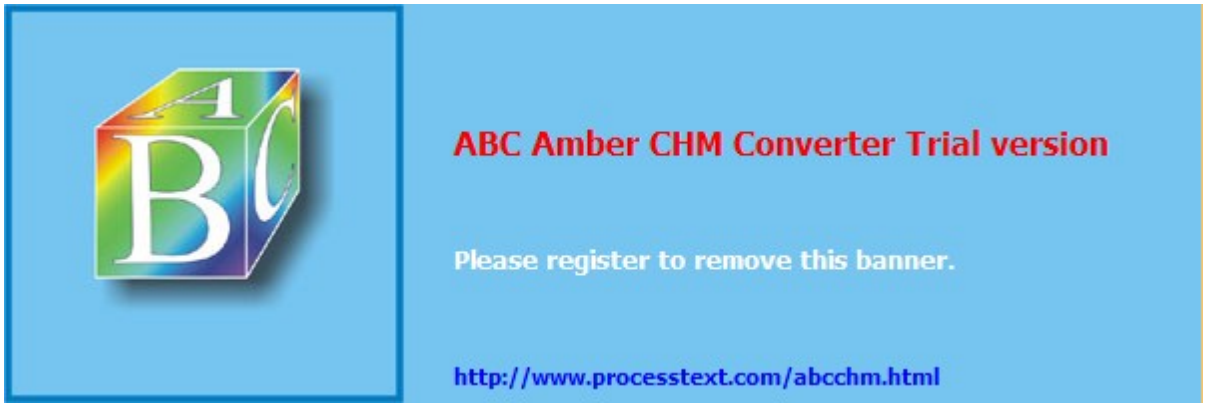
Driver chains exist for almost all hardware devices. The lowest-level driver deals with direct access to the bus and the hardware device, and higher-level drivers deal with data formatting, error codes, and the conversion of high-level requests into the smaller, more pointed details of hardware manipulation.

Layering is an important concept for rootkits, because layered drivers are involved in the movement of data in and out of lower-level hardware. Layered drivers not only intercept data; they can also modify this data before passing it on. In other words, they are *perfect* for rootkit developers.

Almost every device on the system can be intercepted in this way. And, using layering, we can be lazy and intercept only the data we are interested in. Best of all, we can avoid dealing with complicated hardware. If we want to sniff keystrokes, for example, we just layer our interception over the already existing keyboard driver.

In this chapter, you will learn how to use layering techniques to intercept and modify data in a system. We will start by discussing how the Windows kernel handles drivers, and take you through a detailed walk-through of a sample keyboard filter driver for sniffing keystrokes. We will end the chapter with a discussion of file filter-drivers.

By the time you finish reading this chapter, you should be able to intercept everything a user types, and to hide the file or directory where you are storing the data.



A Keyboard Sniffer

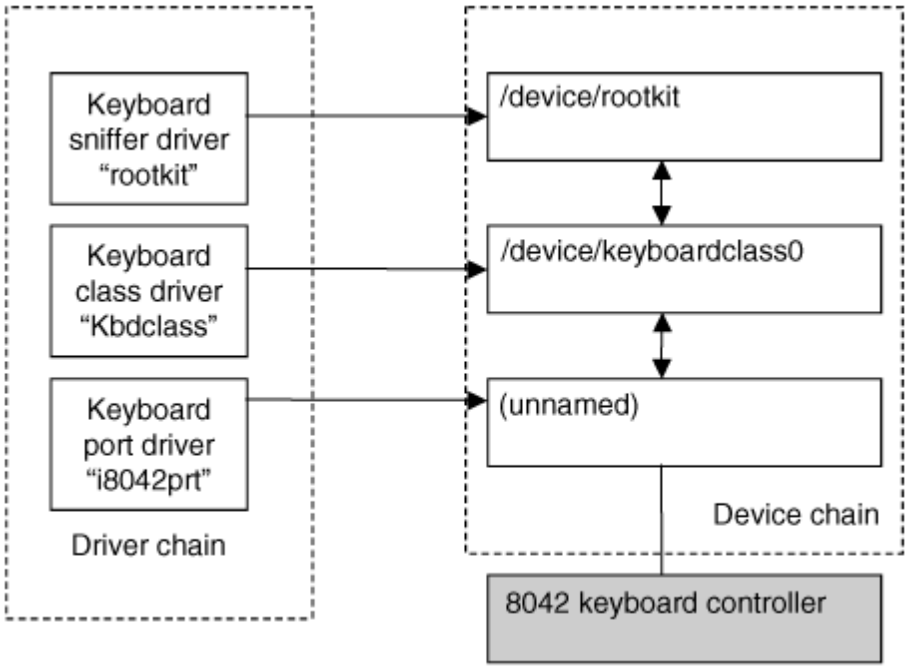
Layering a driver requires some firsthand knowledge about how the Windows kernel handles drivers. This is best learned by example. In this chapter, we will walk you through creating a "hello layers" keyboard-sniffer rootkit. The keyboard sniffer will use a layered filter driver to intercept keystrokes.

The layered keyboard sniffer operates at a much higher level than that of the keyboard hardware. As it turns out, even working with hardware as simple as a keyboard controller can be very problematic. (See Chapter 8, Hardware Manipulation, for an example that directly accesses the keyboard hardware.)

With a layered driver, at the point at which we intercept keystrokes the hardware device drivers have already converted the keystrokes into I/O request packets (IRPs). These IRPs are passed up and down a "chain" of drivers. To intercept keystrokes, our rootkit simply needs to insert itself into this chain.

A driver adds itself to the chain of drivers by first creating a device, and then inserting the device into the group of devices. The distinction between device and driver is important, and is illustrated in Figure 6-1.

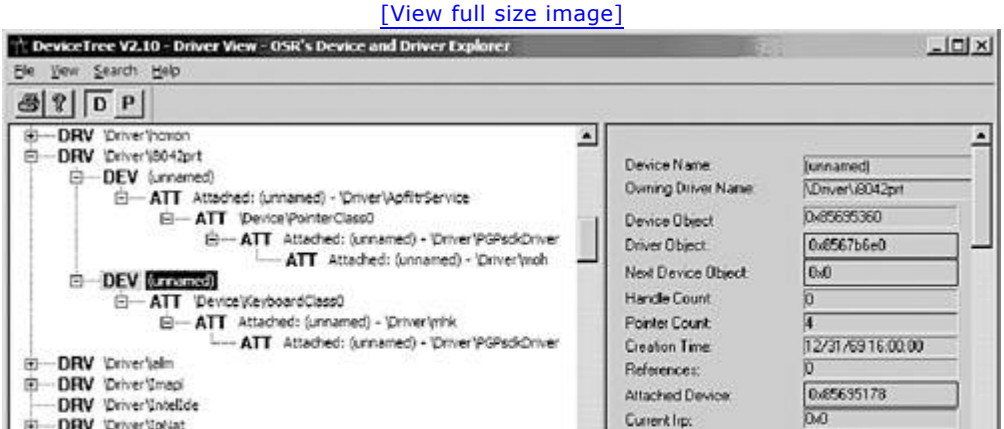
Figure 6-1. Illustration of the relationship between a driver and a device.

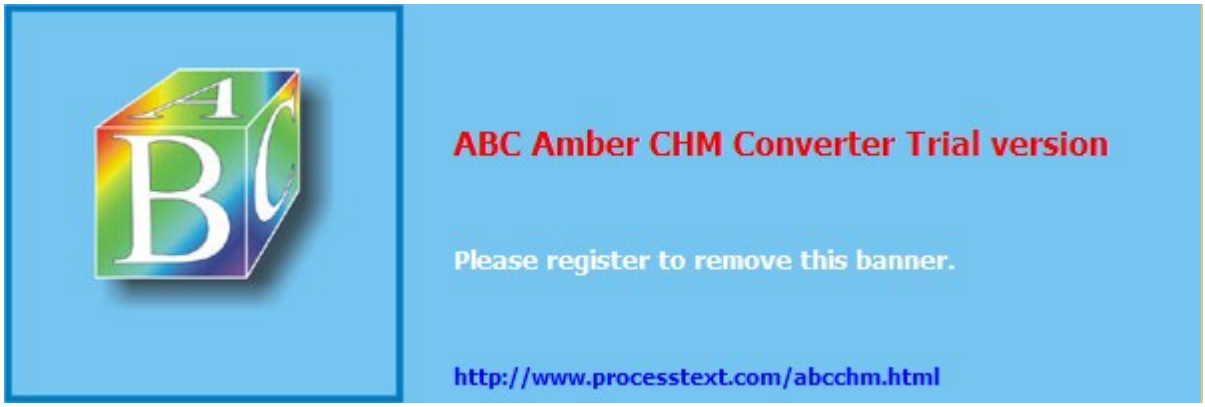


Many devices can attach to the device chain for legitimate purposes. As an example, Figure 6-2 shows a computer having two encryption packages, BestCrypt and PGP, both of which use filter drivers to intercept keystrokes and mouse activity.

Figure 6-2. DeviceTree utility[1] showing multiple filter devices attached to the keyboard and mouse.

[View full size image]





The KLOG Rootkit: A Walk-through

Our example keyboard sniffer, called KLOG, was written by Clandestiny and is published at www.rootkit.com.[3] What follows is a walk-through of her code.

[3] A popular example of a keyboard layered filter driver is available at www.sysinternals.com. It is called ctrl2cap. KLOG is based on the ctrl2cap code.

Rootkit.com

The KLOG rootkit is described at:
www.rootkit.com/newsread.php?newsid=187
It may be downloaded from Clandestiny's vault at rootkit.com.

Note that the KLOG example supports the US keyboard layout. Because each keystroke is transmitted as a scancode, and not the actual letter of the key pressed, a step is required to convert the scancode back to the letter key. This mapping will be different depending on which keyboard layout is being used.

First, DriverEntry is called:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING RegistryPath )
{
  NTSTATUS Status = {0};
```

Next, in the DriverEntry function, a pass-through dispatch routine called DispatchPassDown is set up:

```
for(int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
   pDriverObject->MajorFunction[i] = DispatchPassDown;
```

Next, a routine is set up to be used specifically for keyboard read requests. KLOG's function is called DispatchRead:

```
// Explicitly fill in the IRP handlers we want to hook.
  pDriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;
```

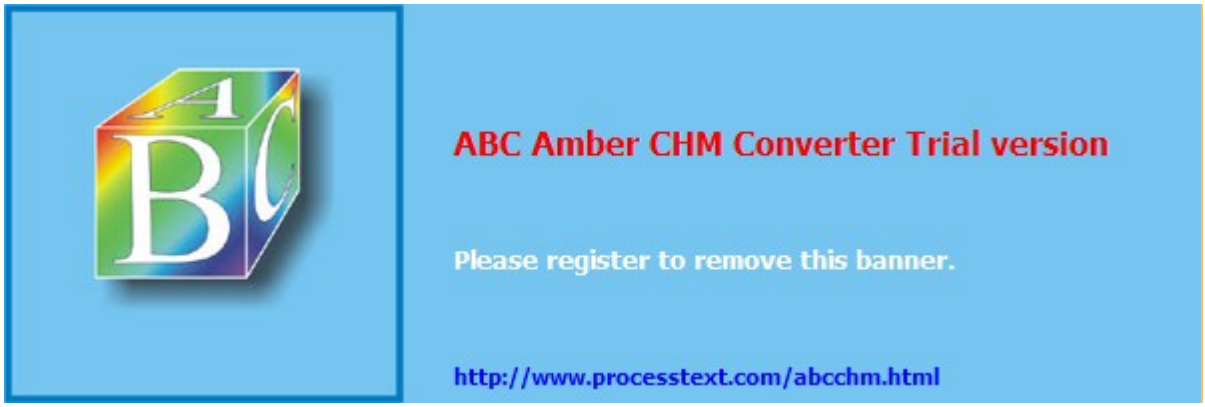
The driver object has now been set up, but it still needs to be connected to the keyboard-device chain. This is done in the HookKeyboard function:

```
// Hook the keyboard now.
  HookKeyboard(pDriverObject);
```

Taking a closer look at the HookKeyboard function, we find the following:

```
NTSTATUS HookKeyboard(IN PDRIVER_OBJECT pDriverObject)
{
// the filter device object
  PDEVICE_OBJECT pKeyboardDeviceObject;
```

IoCreateDevice is used to create a device object. Note that the device object has no name, and that it's of type FILE_DEVICE_KEYBOARD. Also note that the DEVICE_EXTENSION size is passed. This is a user-defined structure.



File Filter Drivers

Layered drivers can be applied to many targets, not the least of which is the file system. A layered driver for the file system is actually quite complex, mostly because the file-system mechanisms offered by Windows are fairly robust.

The file system is of special interest to rootkits for stealth reasons. Many rootkits need to store files in the file system, and these must remain hidden. We can use hooks like those covered in Chapter 4 to hide files, but that technique is easy to detect. Also, hooking the System Service Descriptor Table (SSDT) will not hide files or directories if they are mounted over an SMB share. Here we'll discuss a better approach, a layered driver that can hide files.[4]

[4] We discuss the approach in theory here. The source code is not available for download.

We'll start by taking a look at the DriverEntry routine:

```
NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT DriverObject,
  IN PUNICODE_STRING RegistryPath
  )
{

…
    for( i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ )
   {
      DriverObject->MajorFunction[i] = OurDispatch;
    }
 DriverObject->FastIoDispatch = &OurFastIOHook;
```

Within the DriverEntry routine, we set up the MajorFunction array to point to our dispatch routine. In addition, we set up a FastIo dispatch table. Here we see something unique to file-system drivers. FastIo is another method by which file-system drivers can communicate.

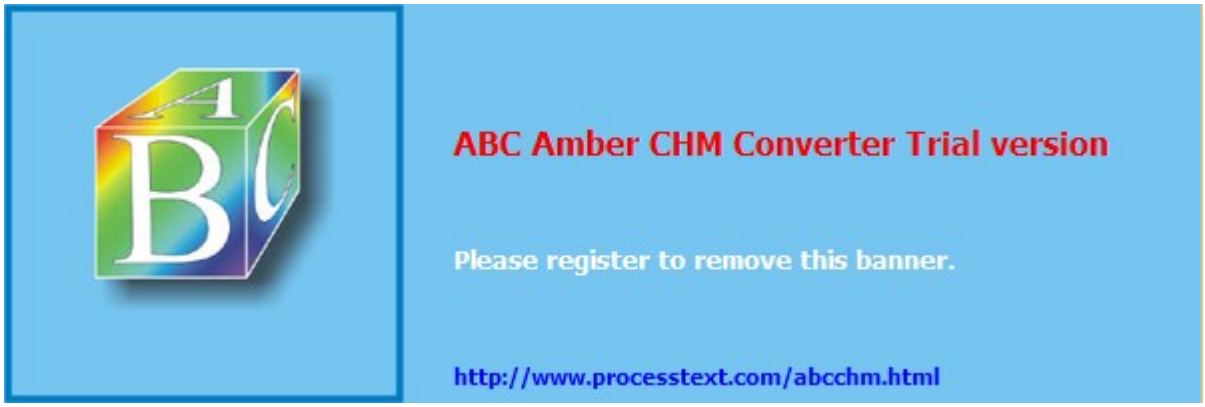
Once the dispatch table is in place, we then must hook the drives. We call a function, HookDriveSet,[5] to install hooks on all available drive letters:

[5] The HookDrive and HookDriveSet functions were originally adapted from the released source code of filemon, a tool available at www.sysinternals.com. This code was modified a great deal, and runs totally in the kernel. The source code for Filemon is no longer available for download from Sysinternals.

```
  DWORD d_hDrives = 0;
  // Initialize the drives we will hook.
  for (i = 0; i < 26; i++)
  DriveHookDevices[i] = NULL;
  DrivesToHook = 0;
  ntStatus = GetDrivesToHook(&d_hDrives);
  if(!NT_SUCCESS(ntStatus))
    return ntStatus;
  HookDriveSet(d_hDrives, DriverObject);
```

Here is the code to get the list of drives to hook:

```
NTSTATUS GetDrivesToHook(DWORD *d_hookDrives)
{
  NTSTATUS ntstatus;
```



Conclusion

Layering is a reliable and robust way to intercept and modify data in the system. It can be used not only for stealth, but also for data collection and modification. Adventurous readers and would-be rootkit developers can expand on the examples in this chapter to intercept or modify network data, create covert channels, intercept or create video signals, and even create an audio bug.

PREV < Day Day Up > NEXT

Chapter 7. Direct Kernel Object Manipulation

Generally in war the best policy is to take a state intact; to ruin it is inferior to this.

?SUN TZU

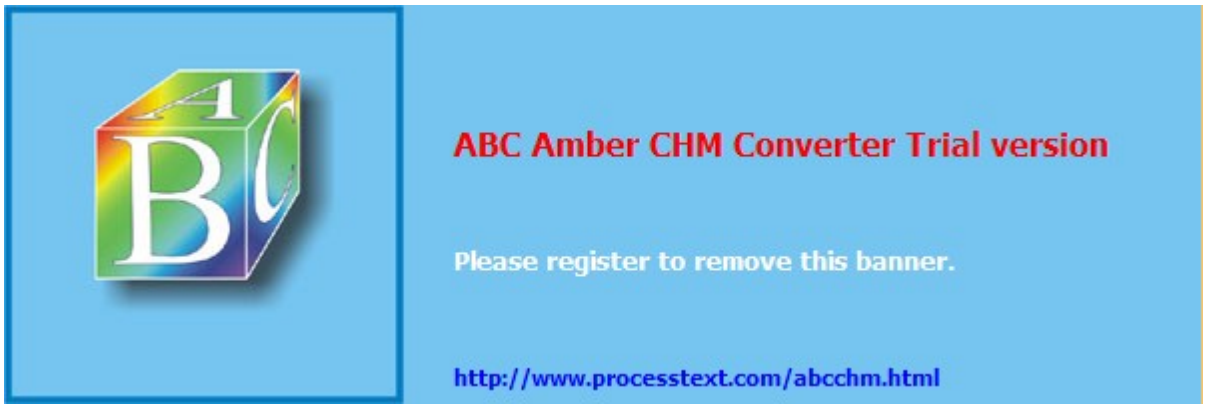
In the preceding chapters, we covered a great deal about hooking techniques. Hooking the operating system is a very effective process, especially since you cannot compile your rootkit into the manufacturer's distribution. In certain instances, hooking is the only method available to a rootkit programmer.

However, as we saw in earlier chapters, hooking has its drawbacks. If someone knows where to look, a hook can usually be detected. In fact, it is relatively easy to detect hooking. In Chapter 10, Rootkit Detection, we will cover how to detect hooks, and you will learn about a tool called VICE that does just that. Also, kernel-protection mechanisms, such as making certain memory pages read only, either today or in the future may make the hooking approach unusable.

In this chapter we discuss another technique that may serve your purposes: Direct Kernel Object Manipulation (DKOM). Specifically, you will learn how to modify some of the objects the kernel relies upon for its bookkeeping and reporting. By the time you have finished this chapter, you should be able to hide processes and drivers without installing any hooks.

You will also learn how to modify any process's token in order to gain System or Administrator privileges without making a single call to any of the process or token APIs. Preventing this type of attack is very difficult.

(Note: In discussing DKOM, the term *object* can be used interchangeably with the more familiar term *structure*. *Object* is the term Microsoft uses in reference to the kernel structures.)



DKOM Benefits and Drawbacks

Before we get into the nitty-gritty of learning how to use DKOM techniques, it is important to understand DKOM's benefits and its drawbacks. On the positive side, DKOM is extremely hard to detect. Under normal circumstances, altering kernel objects such as processes or tokens requires going through the Object Manager in the kernel. The Object Manager is the central point of access to kernel objects. It provides functionality common to all objects, such as creation, deletion, and protection. Direct Kernel Object Manipulation bypasses the Object Manager, thereby bypassing all access checks on the object.

However, DKOM has its own set of problems, one of which is that it is extremely fragile. Because of this fragility, before altering a kernel object a programmer must understand several things about the object:

- What does the object look like, or what are the members of the structure? This can sometimes be the most difficult question to answer. When most of the research began for this book, the only way to answer this question was to spend a lot of time working within Compuware's SoftIce or another debugger. Recently, Microsoft made this job a little easier. Using WinDbg, which is free for download from Microsoft's Web site, you can display the object members by typing `dt nt!_Object_Name`. For example, to list all the members of the EPROCESS structure, type `dt nt!_EPROCESS`. Figuring out what Microsoft calls the object is still a problem, and not all objects are "documented" in WinDbg.

- How does the kernel use the object? You will not understand how or why to modify the object until you understand how it is used by the kernel. Without a thorough understanding of how it is used, you will undoubtedly make a lot of incorrect assumptions about the object.

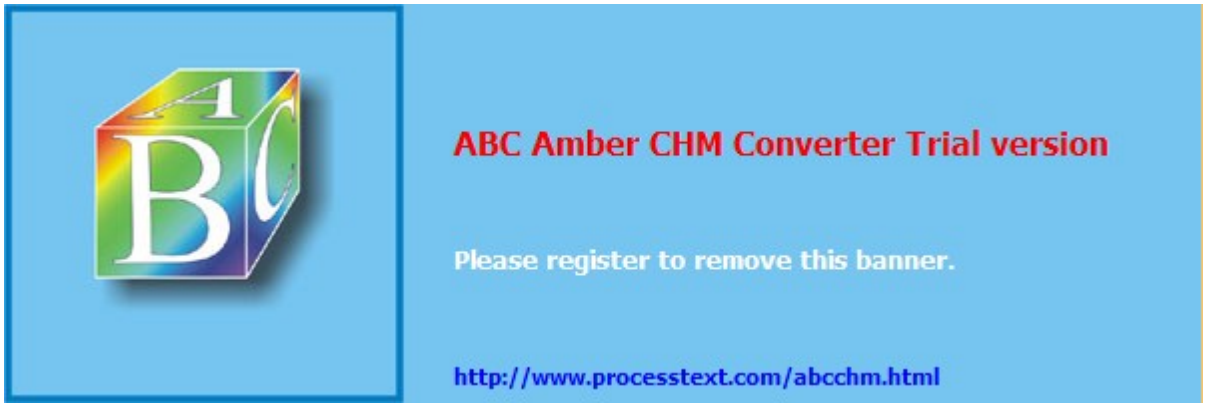
- Does the object change between major versions of the operating system (such as Windows 2000 and Windows XP), or between minor service-pack releases? Many of the objects you will use with DKOM change between versions of the operating system. The objects are designed to be opaque to the programmer, but since you will be modifying them directly, you must understand any such changes and take them into account. Since you will not be working through any function call to modify the objects, backward compatibility is not guaranteed.

- When is the object used? We do not mean *when* in the temporal sense of the word, but rather, the state of the operating system or machine when the object is used. This is important because certain areas of memory and certain functions are not available at different Interrupt Request Levels (IRQLs). For example, if a thread is running at the DISPATCH_LEVEL IRQL, it cannot access any memory that would cause a page fault in the kernel.

Another limitation of DKOM is that you cannot use it to accomplish all of a rootkit's purposes. Only the objects that the kernel keeps in memory and uses for accounting purposes can be manipulated. For example, the operating system keeps a list of all the processes running on the system. As we will see in this chapter, these can be manipulated to hide processes. On the other hand, there is no object in memory representing all the files on the file system. Therefore, DKOM cannot be used to hide files. More-traditional methods, such as hooking or using a layered file filter driver, must be used to hide files. (These techniques are covered in Chapters 4 and 6, respectively).

Despite these limitations, DKOM can be used to successfully accomplish the following:

- Hide processes
- Hide device drivers
- Hide ports
- Elevate a thread's, and hence a process's, privilege level
- Skew forensics



Determining the Version of the Operating System

Since kernel structures change between major versions of the operating system and, in rare cases, between service packs, a rootkit developer must be aware of the system version on which the rootkit will run. The authors of this book believe it is poor form to use hard-coded addresses, or even offsets. Instead, your code should adapt to its surroundings. The goal: Compile once, or at most twice, but run everywhere!

If your rootkit has a user-mode portion, you can determine the operating system version in a userland process using the Win32 APIs. Alternatively, you can determine the system version in the kernel. Obviously, the former is much easier than the latter.

User-Mode Self-Determination

With the Win32 API, it is very easy to determine what version of the operating system your rootkit is installed upon. The structure used to retrieve this information is called OSVERSIONINFO or OSVERSIONINFOEX. It contains information about the major and minor versions of the operating system. The EX version also specifies the major and minor versions of service-pack level.

OSVERSIONINFO vs. OSVERSIONINFOEX

When planning to use either OSVERSIONINFO or OSVERSIONINFOEX to identify the operating-system version, keep in mind that certain versions of Windows are not able to process the EX version of the OSVERSIONINFO structure. The size member of the OSVERSIONINFO structure indicates which version of the structure you are using. You can make the same call to the GetVersionEx function in either case. In the case of OSVERSIONINFO, you must parse the szCSDVersion element of the structure to determine the service-pack level.

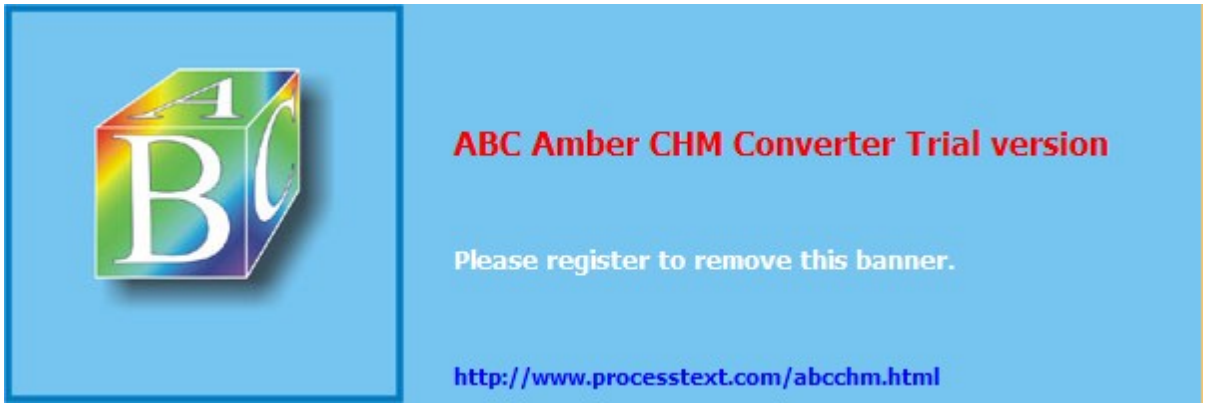
The definition of the OSVERSIONINFOEX structure follows:

```
typedef struct _OSVERSIONINFOEX {
 DWORD dwOSVersionInfoSize;
 DWORD dwMajorVersion;
 DWORD dwMinorVersion;
 DWORD dwBuildNumber;
 DWORD dwPlatformId;
 TCHAR szCSDVersion[128];
 WORD wServicePackMajor;
 WORD wServicePackMinor;
 WORD wSuiteMask;
 BYTE wProductType;
 BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX, *LPOSVERSIONINFOEX;
```

Declare a structure of this type in your code and pass a pointer to this structure when you call the GetVersionEx function. Here is the function prototype for GetVersionEx:

```
BOOL GetVersionEx( LPOSVERSIONINFO lpVersionInfo );
```


After you have made this call, you should have identified the version of the operating system executing



Communicating with the Device Driver from Userland

If you are using a userland process to pass command and control information or initialization data to a rootkit that is structured as a device driver, you will need to use I/O Control Codes (IOCTLs). These control codes are carried in I/O request packets (IRPs) if the IRP code is IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL.

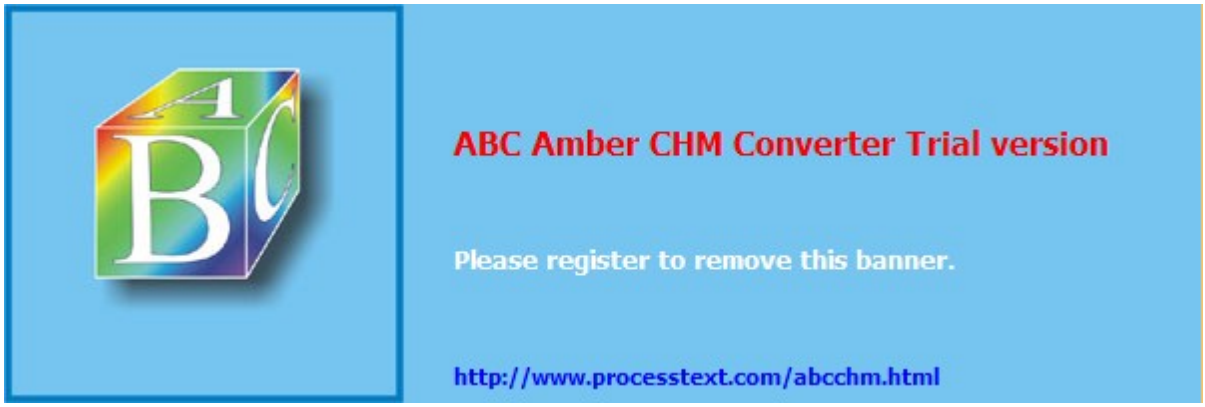
Both your userland process and the driver must agree upon what the IOCTLs are. This is typically accomplished with a shared .h file. The .h file would look something like this:

```
// Filename ioctlcmd.h used by a userland process
// and a driver to agree upon the IOCTLs. The user
// code and the driver code would import this .h file.
#define FILE_DEV_DRV      0x00002a7b
////////////////////////////////////////////////////////////
/////
// These are the IOCTLs agreed upon between the driver and the
// userland program. The userland program sends the IOCTLs down
to the driver
// using DeviceIoControl()
#define IOCTL_DRV_INIT (ULONG) CTL_CODE(FILE_DEV_DRV,0x01,
                                        METHOD_BUFFERED,
                                        FILE_WRITE_ACCESS)
#define IOCTL_DRV_VER  (ULONG) CTL_CODE(FILE_DEV_DRV,0x02,
                                        METHOD_BUFFERED,
                                        FILE_WRITE_ACCESS)
#define IOCTL_TRANSFER_TYPE(_iocontrol) (_iocontrol & 0x3)
```

In this example, there are two IOCTLs: IOCTL_DRV_INIT and IOCTL_DRV_VER. Both use the I/O passing method called METHOD_BUFFERED. With this method, the I/O manager copies data from the user stack into the kernel stack. By referring to the .h file, the user program can use the DeviceIoControl function to talk to the driver. The program requires an open handle to the driver, and the correct IOCTL code to use. Before you can compile the user program, you must include winioctl.h before your own custom .h containing your IOCTLs.

An example is provided in the following code, representing the userland portion of the rootkit. It includes winioctl.h as well as the .h file holding the definitions of the IOCTLs, ioctlcmd.h. Once a handle to the driver is opened, the user code passes down an IOCTL for the initialization function.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <winioctl.h>
#include "fu.h"
#include "..\SYS\ioctlcmd.h"
int main(void)
{
   gh_Device = INVALID_HANDLE_VALUE; // Handle to rootkit
driver
   // Open a handle to the driver here. See Chapter 2 for
details.
   if(!DeviceIoControl(gh_Device,
                       IOCTL_DRV_INIT,
                       NULL,
                       0,
                       NULL,
```



Hiding with DKOM

All operating systems store accounting information in memory, usually in the form of structures or objects. When a userland process requests of the operating system information such as a list of processes, threads, or device drivers, these objects are reported back to the user. Since these objects are in memory, you can alter them directly; it is not necessary to hook the API call and to filter the answer.

Process Hiding

The Windows NT/2000/XP/2003 operating system stores executive objects describing processes and threads. These objects are referenced by Taskmgr.exe and other reporting tools to list the running processes on the machine. ZwQuerySystemInformation uses these objects to list the running processes. By understanding and modifying these objects, you can hide processes, elevate their privilege levels, and perform other modifications.

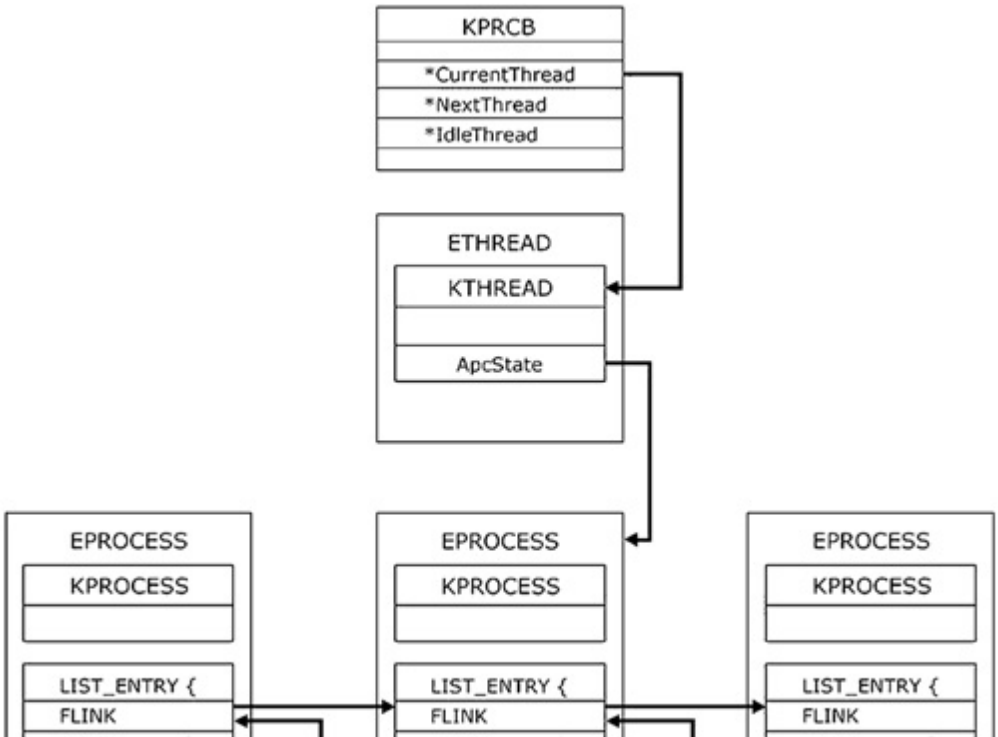
The Windows operating system's list of active processes is obtained by traversing a doubly linked list referenced in the EPROCESS structure of each process. Specifically, a process's EPROCESS structure contains a LIST_ENTRY structure that has the members FLINK and BLINK. FLINK and BLINK are pointers to the processes in front of and behind the current process descriptor.

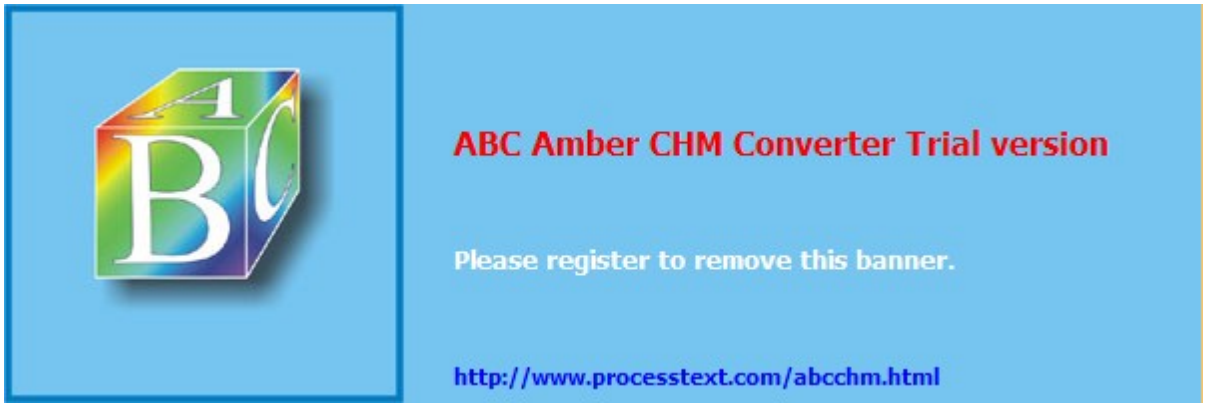
To hide a process, you must understand the EPROCESS structure, but first you must find one in memory. The EPROCESS structure changes in almost every release of the operating system, but you can always find a pointer to the current running process, and hence its EPROCESS, by calling PsGetCurrentProcess. This function is actually an alias for IoGetCurrentProcess. If you disassemble this function, you will see that it is just two moves and a return:

```
mov eax, fs:0x00000124;
mov eax, [eax + 0x44];
ret
```

Why does this code work? Windows has what it calls the *Kernel's Processor Control Block* (KPRCB), which is unique and is located at 0xffdff120 in kernel space. The Assembly code for IoGetCurrentProcess goes to the offset 0x124 from the fs register. This is the pointer to the current ETHREAD. From the ETHREAD block, we follow the pointer in the KTHREAD structure to the EPROCESS block of the current process. We then traverse the doubly linked list of EPROCESS blocks until we locate the process we wish to hide (see Figure 7-1).

Figure 7-1. Path from KPRCB to the linked list of processes.





Token Privilege and Group Elevation with DKOM

A process's token is all-important when it comes to determining what the process is allowed and not allowed to do. A process's token is derived from the log-on session of the user that spawned the process. Every thread within a process can have its own token; however, most threads use their default process token.

One important goal of a rootkit writer is to gain elevated access. This section covers gaining elevated privilege for a nor process once your rootkit has already been installed. This is useful because you want to exploit only once, install your rootkit, and then return under more-normal circumstances so that your original vector of entry is not discovered.

The code in this section will deal only with a process's token; however, it could easily be applied to a thread's token. Th only difference is how you would locate the token in question. All the rest of the techniques and code remain the same.

Modifying a Process Token

To modify a process token, the Win32 API provides several functions, including OpenProcessToken(), AdjustTokenPrivileges(), and AdjustTokenGroups(). All of these functions, and the others that modify process tokens, require certain privileges, such as TOKEN_ADJUST_GROUPS and TOKEN_ADJUST_PRIVILEGES. This section covers a way to add privileges and groups to a process's token without any special privileged access to the process's token. Once your rootkit is installed, DKOM is the only "privilege" you need to understand.

Finding the Process Token

Using the FindProcessEPROC function from the Process Hiding subsection earlier in this chapter to find the address of EPROCESS structure of the process whose token your rootkit will modify, add the token offset to it. The result will be the location within the EPROCESS containing the address of the token. Use the information in Table 7-2 as a guide.

Table 7-2. Offsets to token pointer within the EPROCESS block.

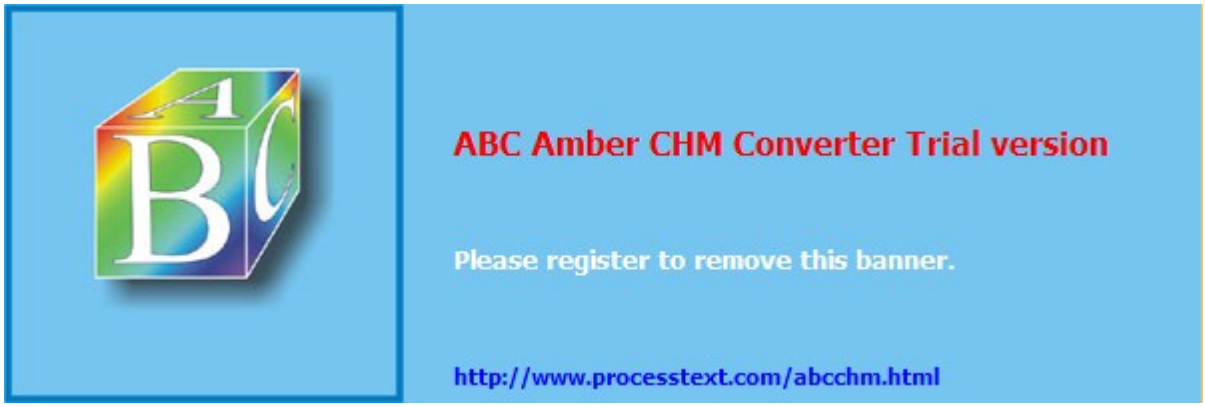
	Windows NT	Windows 2000	Windows XP	Windows XP SP 2	Windows 2003
Token Offset	0x108	0x12c	0xc8	0xc8	0xc8

The member of the EPROCESS structure containing the address of the token was changed between Windows 2000 (a prior versions) and the newer Windows XP (and later versions). It is now an _EX_FAST_REF structure, which is defi as follows:

```
typedef struct _EX_FAST_REF {
    union {
        PVOID Object;
        ULONG RefCnt : 3;
        ULONG Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

To find the process token, use the following FindProcessToken function:

```
DWORD FindProcessToken (DWORD eproc)
{
   DWORD token;
   __asm {
      mov eax, eproc;
      add eax, TOKENOFFSET; // offset of token pointer in EPROCESS
      mov eax, [eax];
```



Conclusion

In this chapter, you learned how to modify some of the very objects the kernel relies upon for its bookkeeping and reporting. Your rootkit can now hide a process and modify its access privileges so that when you return you have all the power of System. These DKOM tricks are very difficult to detect and extremely powerful! However, they also provide ample opportunity to crash the whole machine.

DKOM is not limited to just the uses presented here. You could also use DKOM to hide network ports by modifying the tables of open ports maintained by TCPIP.SYS for bookkeeping, to name just one example.

When seeking to modify kernel objects and reverse engineer where they are used, SoftIce, WinDbg, IDA Pro, and the Microsoft Symbol Server are invaluable tools.

PREV < Day Day Up > NEXT

Chapter 8. Hardware Manipulation

Throughout your life, advance daily, becoming more skillful than yesterday, more skillful than today. This is never-ending.

—HAGAKURE

A scenario:

The intruder slips along the wall toward a janitor cart resting at the end of the hall. His eyes are on a set of keys. A quick look around the corner; good, the janitor is down the hall cleaning a doctor's office. The intruder gently lifts the key chain and dashes back into the dark hallway. Around a corner, stopping at a door, he tries the lock. This doesn't take long. Once the door is open, he sneaks back to the cart and replaces the keys.

The office is dark except for a computer terminal in the back. After moving the monitor and keyboard to the floor, he sits in the crook of the desk. This is a good spot; his actions are not visible to anyone in the hall.

The login screen is locked, but it doesn't matter. The intruder removes a CD-ROM from his jacket, inserts it into the machine, and hard-reboots the workstation. The machine promptly reboots and displays: Press any key to boot from CD. . . ." The intruder taps the spacebar. The rootkit that's on the CD infects the BIOS of this workstation, and also modifies the Ethernet card. It's nothing fancy this time, just a password sniffer. But it will stay here for a long time, even if the "oh-so-intelligent" IT staff re-installs Windows. The intruder smiles: This workstation is "owned."

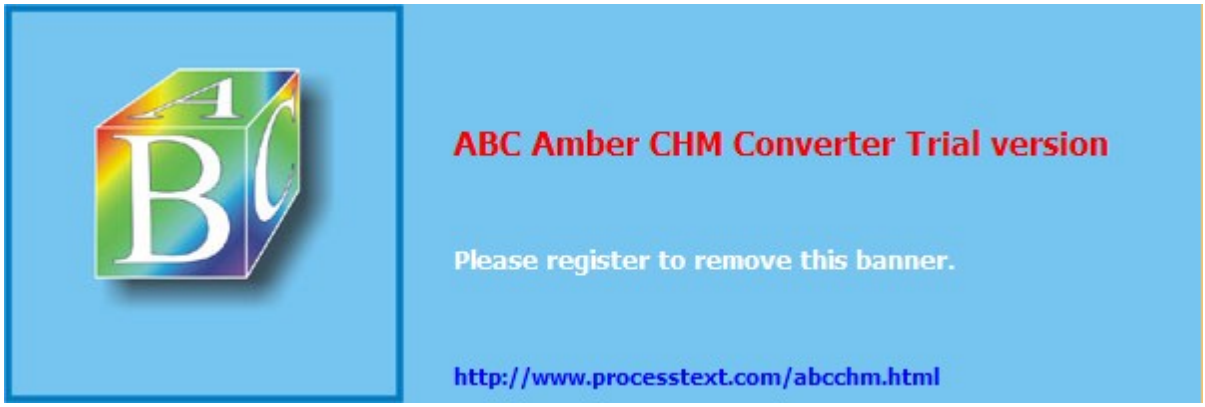
About 30 minutes later, everything is back where it was and the computer is freshly rebooted into Windows. The victim will not notice that the machine has been rebooted. This workstation is a plain-vanilla "Wintel" box, like millions of others in the world. The motherboard is a standard Intel motherboard and the Ethernet card is a 3Com card with on-board processor. What makes this workstation important is that it sits on the same switched network as a pair of Sun E10K servers down the hall—servers that manage hundreds of gigabytes of protein research. The data is worth millions of dollars.

To capture passwords in the real world, this scenario would likely require in-memory kernel modifications in addition to hardware specifics. If *only* the network card were modified, passwords and/or password hashes might be sniffed. This type of rootkit is there for the long term; if the IT staff were to install a newer version of Windows, or even a service pack, the rootkit should keep working. However, if any sort of kernel-level modifications were made in addition to the firmware modifications, an OS or service-pack installation could break everything.

Using the BIOS and direct firmware modification is risky business and is very specific to the target platform. However, the flip side is that with careful planning, such a rootkit would be very difficult to detect. Modifications to the firmware in a "smart" Ethernet card are a very advanced concept, requiring very detailed information about the card. This kind of information might be obtained via reverse engineering, documentation, or insider information. Such modifications don't necessarily need to be made in place, at the user's work location. They can also be made on intercepted computer shipments.

Dealing with a system at such a low level might seem unnecessary. In many cases, this is true. When dealing with a personal computer, you will have access to a lot of software—software that is already on board and running. Much of this software can itself deal with low-level hardware, so you don't have to. It makes sense to use what is already there.

But not all computers are "personal computers" as we know them, abounding with numerous software programs. Many computers are tiny embedded systems that perform small and specific tasks. These systems are everywhere around us—and for the most part, we don't notice them.



Why Hardware?

Hardware manipulation is a double-edged sword. On the one hand, it puts your rootkit at a layer below all other things. This means your rootkit has more control and more stealth (it's about as stealthy as you can get). Your options include direct access to peripheral hardware, disk controllers, USB keys, processors, and firmware memory. On the other hand, hardware is more difficult to work with, and is inherently very platform-specific. Your rootkit must be specifically designed for a given piece of hardware. In other words, the rootkit won't be very portable. The decision to use such technology in a rootkit should not be made lightly.

If you're going to incorporate hardware access into your rootkit, it's important for you to understand that *firmware* is just very specialized software—ultimately, we are still dealing with a software rootkit. Also consider that hardware tends to be cranky—it wants things done in very specific ways.

Even two devices with the same model number may differ "under the hood." The model number is a marketing label. Only the serial numbers can really be relied upon when determining which version of the device you're dealing with. Serial numbers can be traced back to production runs, and small fixes or modifications are made between runs.

So, before you dive in, ask yourself why you need hardware access in your rootkit. Is your goal simple or complex? Simple goals, like making a copy of a packet or flipping a bit here and there, are better for hardware. A good example is a hardware mod that waits until it sees a specific byte sequence in a packet before it crashes the computer. Complex back-door programs and user shells should be written in higher-level software (for instance, in kernel or user mode), and should employ hardware tricks sparingly if at all.

Assuming you've determined that you do need hardware access in your rootkit, read on. We will cover firmware modification, how to address the hardware, timing problems, and other topics. We will also craft an example rootkit that can interface with the keyboard controller chip.

Modifying the Firmware

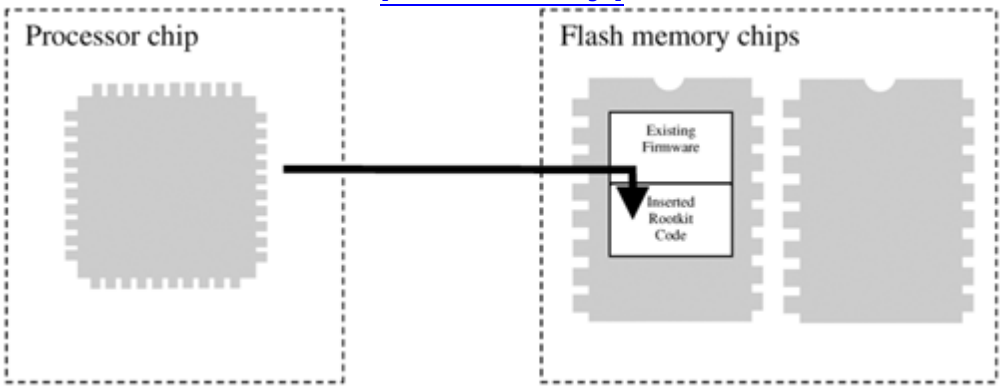
By design, a processor will begin functioning by executing a program stored in memory chips. For example, a PC executes the BIOS when booted. Hardware systems vary widely, but they all share a common fact: *somewhere, somehow, bootstrap code must be activated*. This bootstrap code is sometimes called *firmware;* it is always *non-volatile* (that is, it does not get erased when the system is shut down). If you don't know where to start, go to the boot code.

Considering that firmware is very important for the system operation, a rootkit should not remove existing firmware features. Instead, a rootkit should add new features to the existing code (see Figure 8-1). This can be simple if you reverse-engineer the firmware in a program like IDA-Pro[1] and you find a decent place to patch the execution path. The size of firmware memory is restricted, so if a rootkit is not small enough to fit in the limited amount of unused space, it may need to overwrite some existing firmware code. If this is the case, it is hoped there are some features that are never used, or some data sections that can be overwritten.

[1] www.datarescue.com

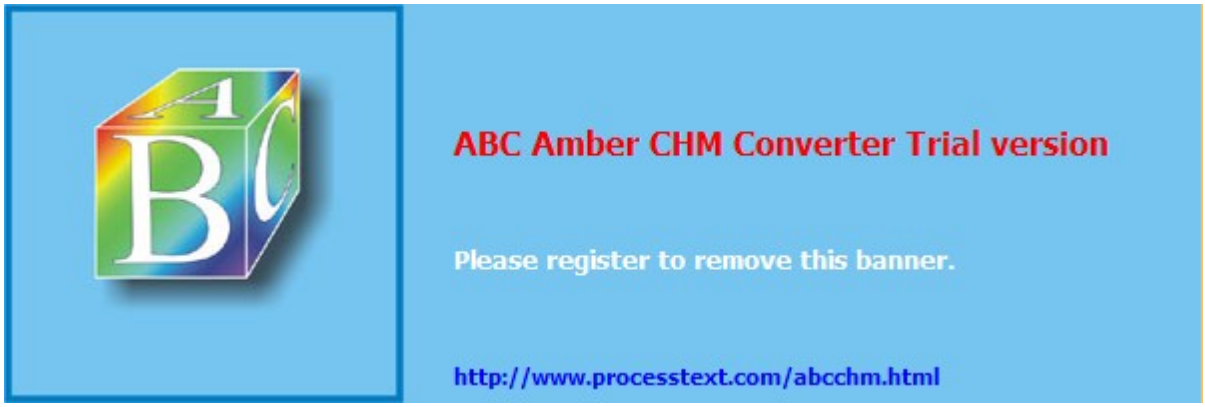
Figure 8-1. A rootkit adds new features to existing firmware.

[View full size image]



To place the rootkit into firmware requires writing to the memory chips. (For a PC, the most obvious place to modify is in the BIOS.) This can be done with an external device, or with on-board software. An external device requires physical access to the target. The software approach requires a loader program. The software loader approach is most commonly applicable to PCs. A software exploit or Trojan can be used to deliver the loader program. The loader program can then alter the firmware.

If the target device is a router or an embedded system, a loader program may be difficult to use. Many hardware devices are not designed to run third-party software and don't have mechanisms for starting multiple processes. Sometimes the best you can hope for is a firmware-upgrade feature that allows code to be uploaded.



Accessing the Hardware

Other than being a glorified calculator, software does one thing very well: It moves data from one place to another. In fact, moving data is sometimes more important than calculating data. No self-respecting power user would ignore the speed at which data can move: bus speeds, drive speeds, CPU speeds. It's all about moving data as quickly as possible.

Most of the hardware on the computer can be controlled with software via moving data and instructions to and from a microchip. Most hardware devices have a microchip that can be addressed somewhere.

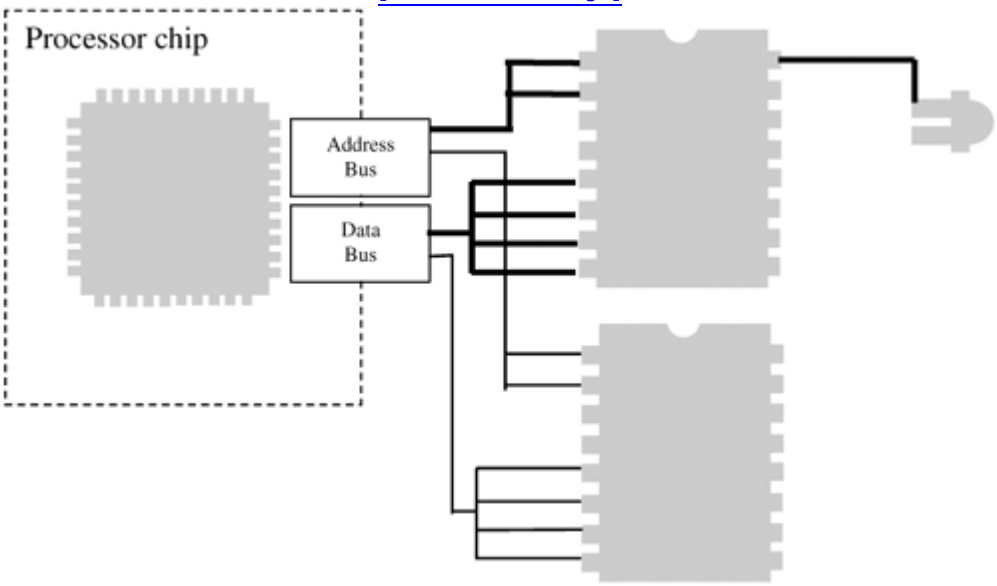
Hardware Addresses

To move data to and from a microchip requires an address. Typically these addresses are known ahead of time and are hard-wired into the system. The address bus consists of many small wires, some of which are wired to each microchip. So, by specifying an address to write to in memory, you are really selecting a microchip.

Once selected, the microchip reads data from the data bus. This microchip then controls the hardware in question. Figure 8-2 illustrates how a microchip is selected by the address bus, and data is then read from the data bus.

Figure 8-2. The address bus selects a hardware controller chip; data is then read.

[View full size image]



Most hardware has some sort of controller chip that exposes an addressable memory location, sometimes called a *port*. Reading and writing to a port may require special opcode instructions: Some processors have special instruction sets that must be used for communicating with ports.

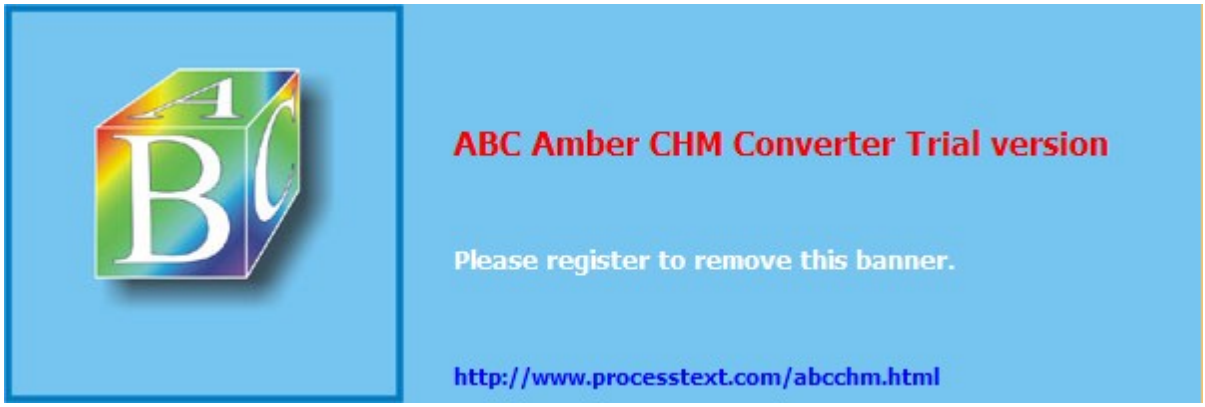
On the x86 architecture, ports are accessed using the `in` and `out` instructions (to read from and write to the port, respectively). However, some chips are memory-mapped, and can be accessed using the more common move instructions (`mov` on the x86).

Regardless of the instruction used, an address will be required. This is how the motherboard will know where to route your data.

Addressing hardware can be complex. Just knowing an address is not enough. The following sections explain some of the challenges.

Accessing Hardware Is Not Like Accessing RAM

Hardware can behave strangely in that it doesn't operate like normal RAM. If you write to an address



Example: Accessing the Keyboard Controller

Now that you know the ins and outs of addressing hardware, let's put that knowledge to use and access some hardware. In our example, we'll access the keyboard controller.

The keyboard is the main hardware interface between a user and the machine. Look at all those keys: It's one of the most complex interfaces ever devised. The keyboard is the source of many secrets?not the least of which is the coveted password. But even beyond passwords, all online communication?including e-mail and instant messaging?must pass through the keyboard. As the source of nearly all user-provided information, the keyboard is something many people want to "sniff." There are many ways to do this, but the subject of this chapter is hardware, so let's figure out how to do it using the keyboard controller chip.

The 8259 Keyboard Controller

It's very simple to control a chip, assuming you know its address; usually, the process is as simple as using the `in` and `out` assembly instructions. The 8259 keyboard controller on most PCs is addressable at addresses 0x60 and 0x64. These locations are sometimes called *ports*, as each provides a portal into the hardware chip.

When using the DDK, you should have a few macros available to read and write to these ports:

```
READ_PORT_UCHAR( ... );
WRITE_PORT_UCHAR( ... );
```

Alternatively, you could use the direct assembly instructions:

```
in
out
```

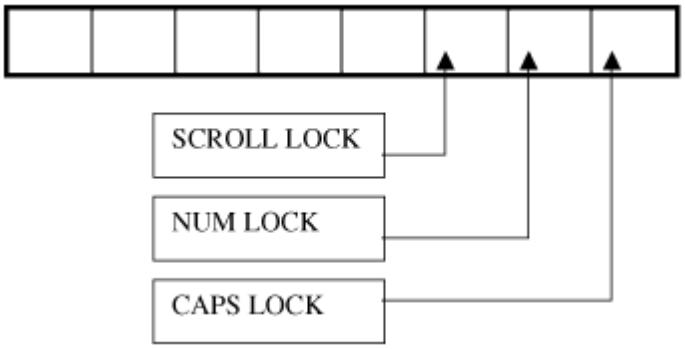
So, what can you do with the keyboard port? Most obviously, you can read the keystroke! Also, you can place a keystroke into the keyboard buffer. You can also change the settings of the LED indicators on the keyboard. By playing around with the keyboard indicators, you can see instant results of your work.

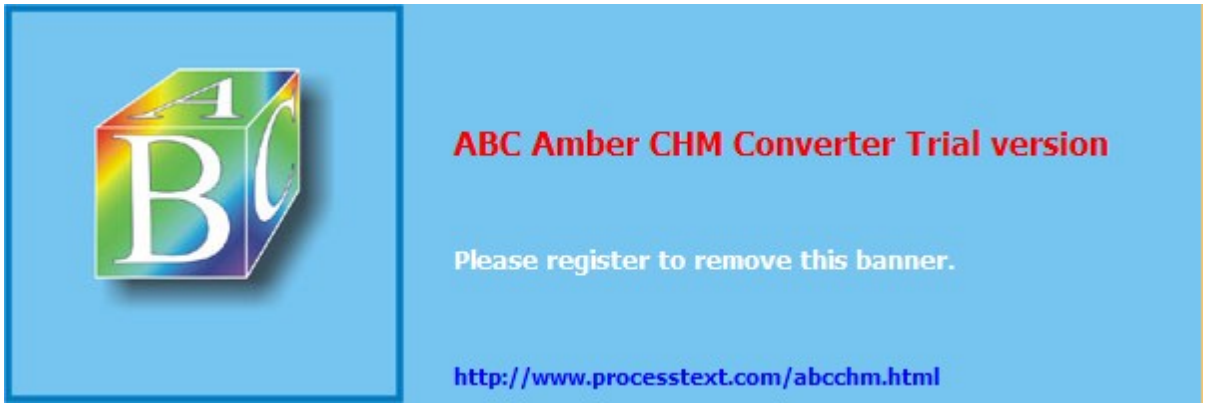
Changing the LED Indicators

The command to set the LEDs is 0xED. The 0xED byte must first be sent to the keyboard controller before we can blink the LED lights. This command is sent to port 0x60, followed immediately by another byte to indicate which LEDs to set. The second byte indicates which LEDs to set in the lower 3 bits of the value.

Figure 8-6 shows the data byte that is used with the 0xED command.

Figure 8-6. The data byte used with the 0xED command.





How Low Can You Go? Microcode Update

Modern processors from Intel and AMD[7] include a feature known as a *microcode update*. It allows special code to be uploaded to the processor that can alter the way the hardware works. That is, the processor chip can be internally modified. How it actually works under the hood remains somewhat of a mystery. When we were writing this book, the public documentation was sparse.

[7] AMD's U.S. Patent No. 6438664.

Microcode update wasn't designed for hacking; it is intended to allow bug fixes to be applied to the processor. If something is wrong with the processor, a microcode update can fix it. This prevents the need to recall computers (a very expensive process). Internally, the microcode allows new "micro-opcodes" to be added or altered. This can alter the way existing instructions are executed, or disable features on the chip.

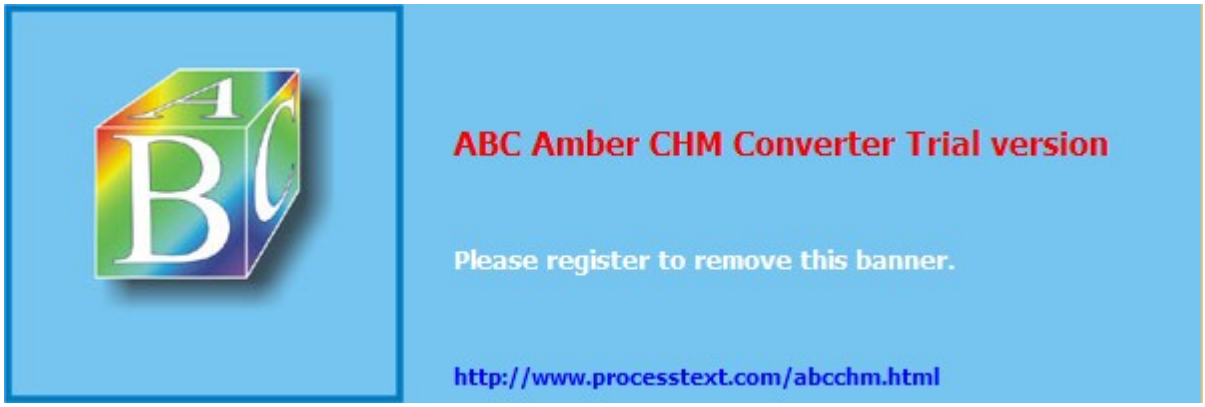
In theory, if a hacker were to supply or replace microcode in the processor, she could add subversive instructions. It seems that the biggest hurdle is understanding the microcode update mechanism itself. If it is understood, it might be possible to craft additional back-door op-codes. An obvious example would be an instruction that can bypass the restriction between Ring Zero and Ring Three. A GORINGZERO instruction, for example, could put the chip into supervisor mode without a security check.

The microcode update is stored as a data block and must be uploaded to the processor every time it is booted. The update takes place using special control registers on the chip. Typically, the microcode update block would be stored in the system BIOS (a flash chip) and applied by the system BIOS upon startup. If used by a hacker, the microcode could be altered in the startup BIOS, or it could be applied "on the fly." No reboot is required?the new microcode is utilized immediately.

Intel processors protect their microcode update blocks with strong encryption. In order to correctly modify the update block, the crypto would need to be broken. AMD chips do not use encryption, so they are easier to work with. For Linux there exists an update driver that can upload new microcode to the AMD or Intel processor. To find it, search for "AMD K8 microcode update driver" or "IA32 microcode driver" on the Internet.

Although many people are currently "playing around" with microcode updates in efforts to reverse engineer them, it should be noted that modifications made to the microcode update blocks could, in theory, damage the microchip.[8]

[8] If the processor includes FPGA-like gates that can be reconfigured, it might be possible to alter the physical configuration of gates in a way that permanently damages the hardware.



Conclusion

Although our coverage of hardware has been sparse, this chapter has introduced the concept. We hope it will inspire you to perform your own research.

We have introduced the basic instructions needed to read from and write to hardware, and some of the "gotchas" to watch out for. Technical manuals are available that cover the bus in excruciating detail, and you should obtain one of these manuals if you want to explore the system.[9] We hinted at the potential of hardware exploitation with BIOS modification and microcode updates. We illustrated a useful rootkit feature called *keystroke monitoring*. And, as always, we would like to drive home the point that it's possible to defeat most rootkit-detection schemes by simply getting as low as possible in the system.

[9] *See,* for example, the "PC System Architecture Series" books, authored by Don Anderson and Tom Shanley (with others), published by Addison-Wesley.

PREV < Day Day Up > NEXT

Chapter 9. Covert Channels

"We are what we pretend to be, so we must be careful what we pretend to be."

—MOTHER NIGHT, KURT VONNEGUT, JR.

A *covert channel* is a secret communication pathway. *Covert* means hidden, so the communication must be concealed. The term originates from the design of highly secure, compartmentalized computer systems—the ones found in military installations that handle classified information.

These systems are supposed to keep one process from communicating with another process. As it turns out, that is very hard to do. No matter how minor, any detectable signal that can be influenced by two parties may become a conduit of communication between them.

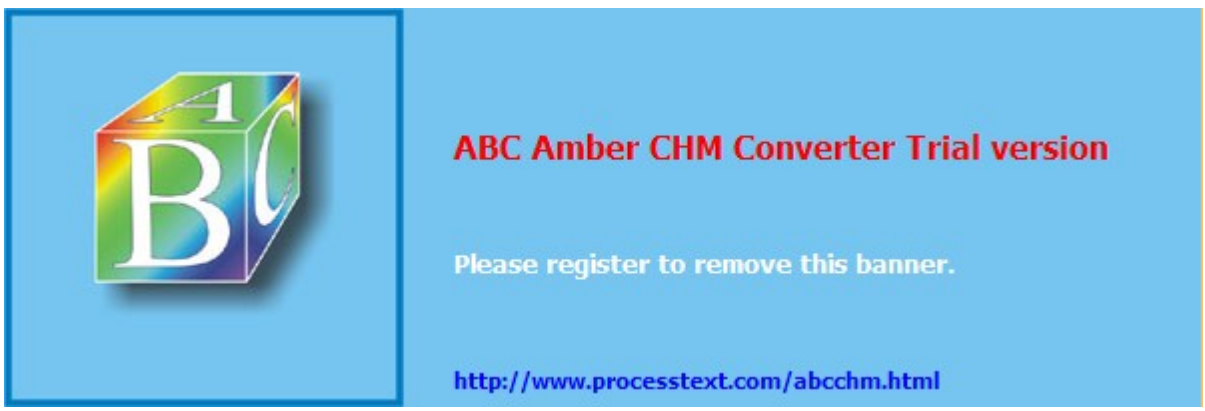
A covert channel doesn't have to be fancy or meet academic standards of stealthiness; it just needs to be unanticipated—so that it slips by unnoticed.

For a rootkit, a covert channel typically means a communication path that breaks through firewalls undetected (by sniffers, IDS systems, or other security mechanisms). The channel must be robust enough to support exfiltrating data from the computer and allow command and control messages. Such capacity enables an attacker to communicate with a rootkit, steal data, and remain undetected while doing it.

Covert channels must be designed. They cannot be known protocols or software designs. A covert channel is usually some form of extension upon an existing protocol or software communication process created in order to move hidden data.

A class of data hiding known as *steganography* forms the basis of many covert channels. Basically, steganography is about "hiding in plain sight." This has been popularized in movies and the press with such concepts as hiding secret messages inside digital photographs.

In this chapter, we begin our discussion of covert channels by explaining the concepts of remote command, control, and data exfiltration. Next, we launch into the topics of disguised TCP/IP protocols, kernel TCP/IP support for your rootkit, and raw network manipulation. We introduce NDIS and TDI mechanisms you can use to send and receive network data to and from a Windows kernel driver. Armed with this knowledge, you should be able to create a rootkit that lets you move in and out of data networks without being detected.



Remote Command, Control, and Exfiltration of Data

As you know, a rootkit is installed to gain remote access to a computer. This serves two primary purposes: to control computer software operation, and to copy data from the system. Examples of such *command and control* include shutting a computer down, enabling or disabling features, and manipulating the kernel. Taking data from a system is typically called *exfiltration*, or *exfil* for short. Exfiltration may take such arcane forms as data transmissions over electromagnetic emissions, via extra data inserted into network protocols, and in the form of time delays.

Where remote access is required, the rootkit must be able to communicate over a network. For a TCP/IP network, this could mean via a TCP connection. Once a connection has been established, commands can be issued and data can be exfiltrated.

In the hacker underground, a typical generic solution to the problem of exfil is the *remote shell*. A remote shell is simply a TCP session connected to the native command interpreter on the system. The command interpreter is supplied with the operating system. On an MS-Windows machine, this would be `cmd.exe`, and on a UNIX system it may be `/bin/sh` or `/bin/bash`.

These command interpreters are actually software programs themselves. Since the command interpreters are already installed on the system *before* the hacker arrives, the attack program just connects the command interpreter to a network port. In other words, the hacker borrows the existing program when she attacks.

For the most part, hackers are just lazy; they don't want to write their own shell programs. There are, however, cases where hackers have created complex remote-control software. Back Orifice 2000[1] is one example of a full remote-control system, with file access, screen capture, and even audio bugging.

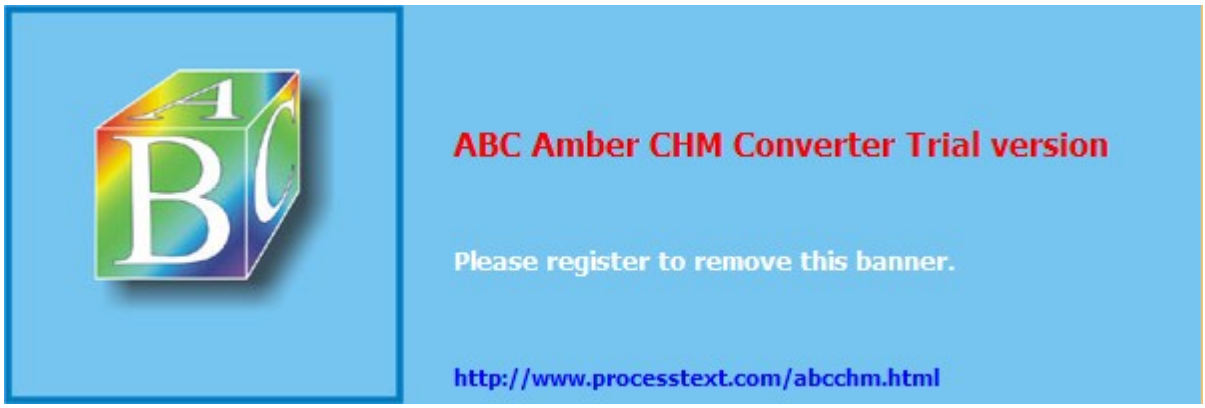
[1] "Back Orifice" is a play on "BackOffice," the name of a product offered by Microsoft.

Large, full-featured back-door programs have a few drawbacks. First, they are overkill for most needs. Second, every virus scanner on the planet will detect them. Third, and perhaps most importantly, they are written by people you don't know.

When engaging in an activity as sensitive as remote penetration, you should be concerned about risk of exposure before anything else. Two concepts that are key to avoiding exposure are *minimal footprint* and *unique structure*.

- Minimal footprint: The tools used for remote penetration should affect as little as possible on the remote system. (This is a good reason to design a rootkit that never uses the file system.) This minimizes the chance of detection. Also, fewer lines of code means less complex code, and less complex code means less chance of failure.

- Unique structure: The tools used for remote penetration should have structures and methods that are unique. Virus-detection solutions are always looking for the known. In virus-detection development, a publicly known virus is analyzed for general patterns, and these patterns are then applied to finding unknown viruses. If you attempt to download a rootkit from www.rootkit.com, for example, your virus scanner will likely quarantine the file. If they do not contain patterns found in known infections, then your tools will slip by undetected.



Disguised TCP/IP Protocols

A rootkit's activities should be covert—undetectable. Communication over a TCP socket can easily be detected, both on the network and in the kernel. Opening a TCP socket is a very noisy event that creates a SYN packet, followed by completion of the famous three-way handshake.[2] Any packet sniffer will report it. Intrusion-detection systems will almost always log the event, and may even create an outright alarm. Finally, TCP ports can usually be mapped back to the software process that created them. These are all really bad for a rootkit. More-subtle measures must be used.

[2] TCP protocol dictates that three packets are used to set up a new connection; this is known as the "three-way handshake," and is detailed in many documents available in the public domain.

In a noisy environment like a network, intrusion-detection systems look for activities that stand out—that are *different*. One approach to good covert-channel design is to use a protocol that is in constant use on the network (such as DNS, Domain Name Service). In using DNS as a covert channel, a rootkit will use a modification to the protocol to place extra data into a packet. The goal is to make the packet "look and smell" just like legitimate traffic (so that nobody will notice it). Even if you don't make your packets look exactly like the real thing, sometimes they still won't be noticed.

The rule is simple: *Hide in traffic that is already there.*

If you don't want to get into protocol specifics, just start by using a source and destination port of a common protocol. For DNS, this is port 53 (UDP or TCP). In many cases, DNS is even allowed over a firewall. For the Web protocol, the port is TCP port 80, or 443 for encrypted Web. If you choose port 443 and encrypt everything, you can be sure no one will take a look *inside* your packets. One word of warning, though: Technology exists to unencrypt SSL Web sessions.[3] This technology can be used by IDS equipment (but usually isn't).

[3] Ettercap (http://ettercap.sourceforge.net) is a tool for this purpose.

"Hiding in plain sight" can be harder than you might expect. In the following sections, we detail many challenges you will face, and we make some creative suggestions for your covert-channel designs.

Beware of Traffic Patterns

Hiding data in a known protocol is just a first step in creating covert communications. You must also use conservative traffic patterns. A covert channel should not create an excessive amount of traffic: To avoid being noticed, you must not spike above normal usage.

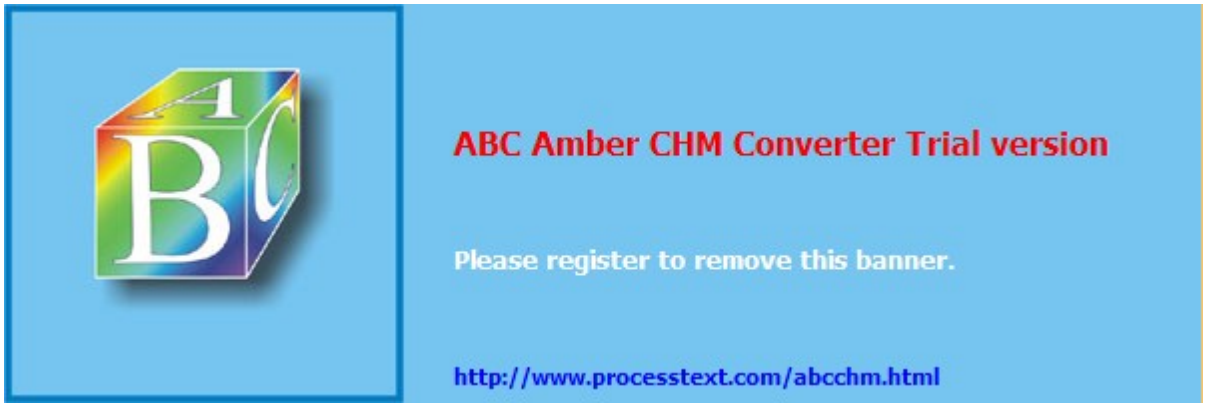
If your rootkit is creating solid green bars on the MRTG[4] graph, someone is bound to notice. If the network is quiet, and suddenly, at 3 a.m., a big traffic spike occurs, an administrator's first thought will be that someone is engaged in a high-traffic activity, such as sharing an "iso" of Quake III on some file share. If the administrator investigates, the traffic spike will lead her right to your infected machine. That's bad on all counts.

[4] Multi Router Traffic Grapher (www.mrtg.org).

Don't Send Data "in the Clear"

This is a fine point, but even if you use a known protocol and don't create traffic spikes, you should still hide your data so that it doesn't look malicious. Hide your data inside of other, innocuous-looking data. If you put unencrypted password files in the payload of the packet, for example, someone is going to notice. If some admin examines this packet, big alarm bells will go off. Furthermore, some IDS systems do blanket searches of all packets for suspicious strings, like "etc/passwd." The payload should be obfuscated at the very least. Even better, you should use encryption[5] or steganography.

[5] Sometimes using encryption increases the chance that something will look suspicious. If the protocol typically uses easy-to-read text, and you're transmitting garbled bytes or high-entropy data (read: encryption), the packets will stand out like a sore thumb.



Kernel TCP/IP Support for Your Rootkit Using TDI

All this talk about TCP/IP naturally leads us to some code. In a Microsoft Windows environment, you basically have two modes in which to write networking code: *user mode* and *kernel mode*. The advantage of user mode is that it's easier, but a downside is that it's more visible. With kernel mode, the advantage is more stealth, but the downside is complexity. In the kernel, you don't have as many built-in functions available to you and you must do more stuff "from scratch." In this section, we focus primarily on the kernel-mode approach.

In a kernel-mode approach, the two major interfaces are TDI and NDIS. TDI has the advantage of using the existing TCP/IP stack on the machine. This makes using TCP/IP easier, because you don't have to write your own stack.

On the other hand, a desktop firewall can detect a TCP/IP-embedded communication. With NDIS, you can read and write raw packets to the network and can bypass some firewalls, but on the downside you will need to implement your own TCP/IP stack if you want to use the protocol.

Build the Address Structure

Your rootkit lives in a networked world, so naturally, it should be able to communicate with the network. Unfortunately, the kernel doesn't offer easy-to-use TCP/IP sockets. Libraries are available, but these are commercial packages that cost money. They might also be traceable. You don't need these expensive packages to use TCP/IP in the kernel, of course, but they may be the easiest solutions.

For the do-it-yourself programmer, there is a kernel library that supports TCP/IP functionality, and you can work with it from a kernel-mode device driver. Device drivers can call functions in other drivers; this how you can use TCP/IP from your rootkit.

The TCP/IP services are available from a driver which exposes several devices that have names like `/device/tcp` and `/device/udp`. Sound interesting? It is if you need a sockets-like interface from kernel mode.

The *Transport Data Interface* (TDI) is a specification for talking to a TDI-compliant driver. We are concerned with the TDI-compliant driver in the Windows kernel that exposes TCP/IP functionality. Unfortunately, as of this writing there is no decent example code or documentation you can download to illustrate how to use this TCP/IP functionality. One problem with TDI is that it's so flexible and generic that most documentation on the subject is broad and confusing.

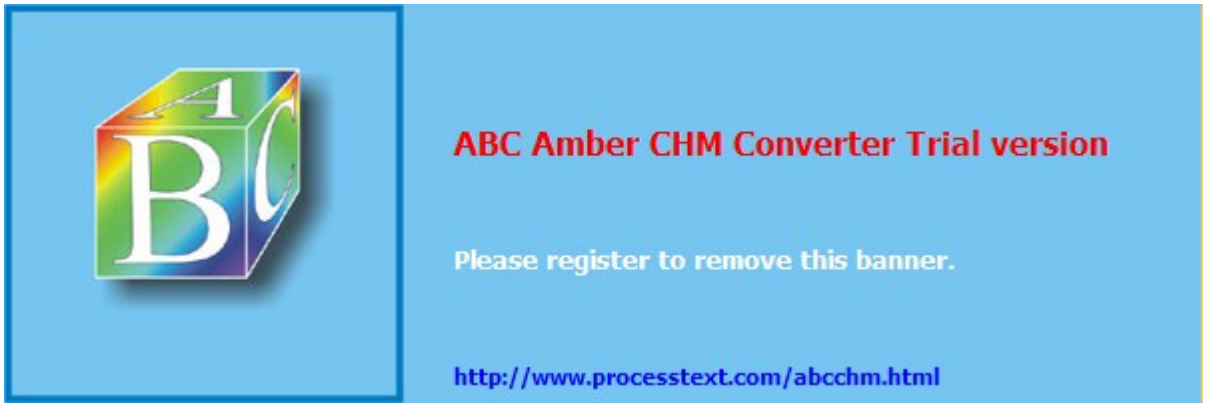
In our discussion focusing on TCP/IP, we have created an example that will ease you into TDI programming.

The first step in programming a TDI client is to build an address structure. The address structure is very much like the structures used in user-mode socket programming. In our example, we make a request to the TDI driver to build this structure for us. If the request is successful, we are returned a handle to the structure. This technique is very common in the driver world: Instead of allocating the structure ourselves, we make a request to another driver, which then builds the structure for us and returns a handle (pointer) to the structure.

To build an address structure, we open a file handle to `/device/tcp`, and we pass some special parameters to it in the open call. The kernel function we use is called ZwCreateFile. The most important argument to this call is the *extended attributes* (EA).[12] Within the extended attributes, we pass important and unique information to the driver (see Figure 9-2).

[12] Extended attributes are used mostly by file-system drivers.

Figure 9-2. Driver A makes request to Driver B via the ZwCreateFile call. The extended attributes structure contains the details of the request. The returned file handle is actually a handle to an object built by the lower-level driver.



Raw Network Manipulation

When using a kernel rootkit, you will typically have access to the device drivers that control the network card. This means you can read and write raw frames from and to the network. With a raw frame, you can control all parts of the protocol?in other words, the parts of the communication that control routing and identification. For example, with raw frames you can control your Ethernet address (MAC address), TCP source port, and source IP address. With raw frames, you are not dependent on the infected host's TCP/IP stack. This can be useful, enabling you to better hide the source of communication. More importantly, it can allow you to bypass firewalls and IDS systems.

To get started, we cover raw packet manipulation from a user-mode program. Although this book is about kernel rootkits, we felt it would be easier for you to learn about and practice with raw packets and protocol manipulation in a user-mode program. We cover raw packet manipulation in the kernel later in the chapter.

Implementing Raw Sockets on Windows XP

For a long time, Microsoft didn't offer a raw sockets interface. This forced developers to use driver-level technology to do anything "cute" (for example, spoofing packets) with the TCP/IP stack. Now that raw sockets have been made available in Windows, rootkit authors can forge packets from user mode.

If a system is running XP service pack 2 (SP2), the functionality of raw sockets is limited. Probably in response to Internet worms, Microsoft chose to limit the power of raw sockets with SP2. If SP2 is installed, you cannot craft raw TCP frames (for example, you cannot run a SYN Scan). You can write raw UDP frames, but you cannot spoof the source address. And, SP2 makes it difficult to create a port scanner: If you attempt a full TCP-connection scan, you will be rate-limited.

Raw sockets are opened the same way ordinary sockets are?they just function a bit differently. As with all sockets programs for Windows, the first step is to initialize Winsock using WSAStartup():

```
WSAData wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
   {
      printf("WSAStartup() failed.\n");
      exit(-1);
   }
```

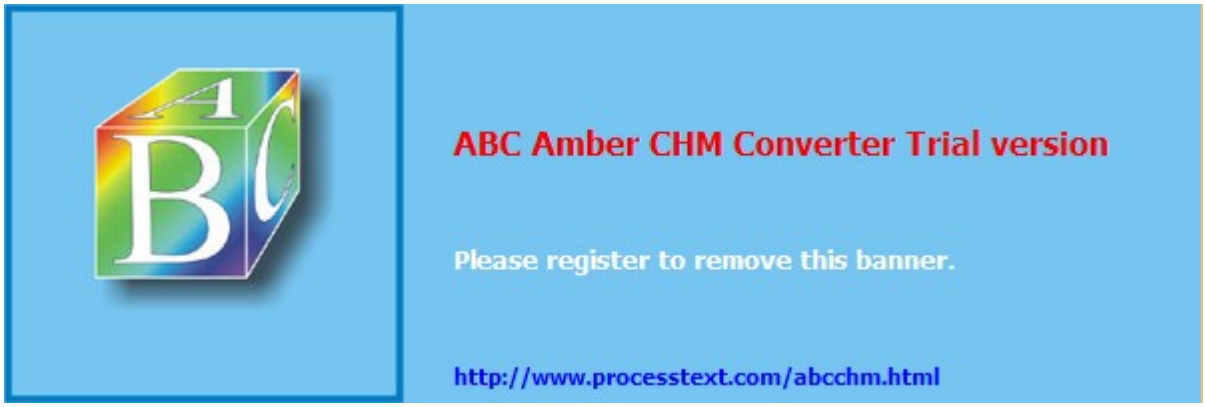
Next, you must open a socket using the socket() function. Note the use of the constant, SOCK_RAW. If this succeeds, you will now have a raw socket you can use to sniff packets and send raw packets.

```
SOCKET mySocket = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
   if (mySocket == INVALID_SOCKET)
   {
      printf("socket() failed.\n");
      exit(-1);
   }
```

Binding to an Interface

A raw socket is not operational until it has been bound to an interface. To bind, you must specify the IP address of the local interface you wish to bind to. In most cases you will want to determine the local IP address dynamically. The following code obtains the local IP address and stores it within the in_addr structure:


```
// Discover Hostname/IP.
```



Kernel TCP/IP Support for Your Rootkit Using NDIS

So far, we have shown only how to craft raw packets from a user-mode program. This is fine for experiments, but when it comes to creating a real-world rootkit, you must be able to send and receive raw packets from the kernel.

Using the NDIS interface allows a driver access to raw packets. While NDIS is best used to sniff packets, you can also send raw packets using an NDIS driver.

Our example is an NDIS protocol driver. It allows forging as well as sniffing of raw packets. Our protocol driver does *not* filter packets; we cannot control packets going to and from the host (our rootkit is not a packet firewall). We get a *copy* of each packet to sniff, not the original.

To start sniffing, we must first register a protocol, and then define callback functions that will handle events.

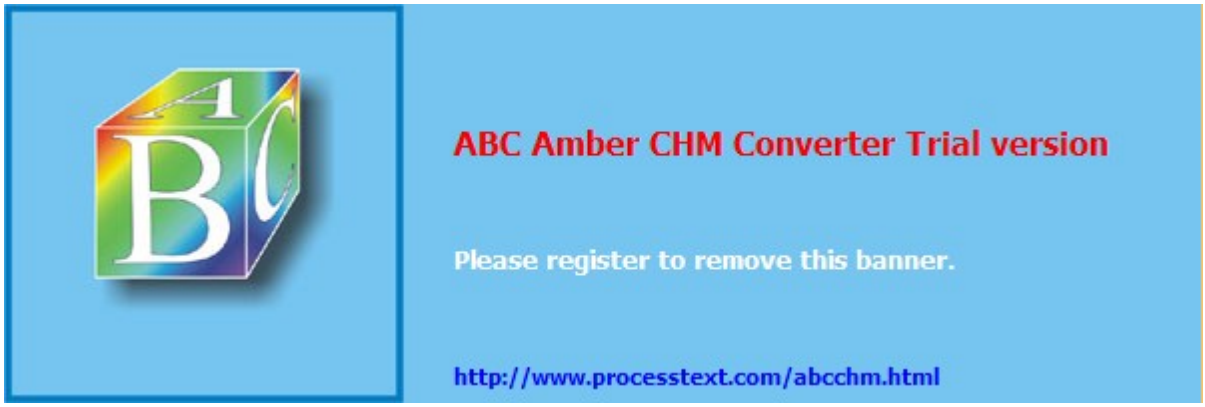
Registering the Protocol

In order to begin sniffing packets, you must register a protocol-characteristics structure with the system. This requires a linkage argument that specifies which interface (Ethernet interface, wireless card, etc.) you will be working with. The interface is sometimes called the *MAC*. In our example, we hard-code this argument, and we give our protocol the name ROOTKIT_NET.

```
#include "ntddk.h"

// Important! Place this before ndis.h.
#define NDIS40   1

#include "ndis.h"
#include "stdio.h"
struct UserStruct
{
   ULONG   mData;
} gUserStruct;
// handle to the open network adapter
NDIS_HANDLE      gAdapterHandle;
NDIS_HANDLE      gNdisProtocolHandle;
NDIS_EVENT       gCloseWaitEvent;

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN
PUNICODE_STRING
theRegistryPath )
{
   UINT             aMediumIndex = 0;
   NDIS_STATUS      aStatus, anErrorStatus;
   // We try only 802.3.
   NDIS_MEDIUM      aMediumArray=NdisMedium802_3;
   UNICODE_STRING   anAdapterName;
   NDIS_PROTOCOL_CHARACTERISTICS   aProtocolChar;
   NDIS_STRING   aProtoName = NDIS_STRING_CONST("ROOTKIT_NET");

   DbgPrint("ROOTKIT Loading...");
```

You can obtain the list of potential interfaces from either of the following registry keys[17]:

[17] Code to get TCP bindings can be found at: www.winpcap.polito.it/docs/man/html/Packet_8c-source.html.



Host Emulation

Using the NDIS protocol driver, we now can emulate a new host on the network. This means our rootkit will have its own IP address on the network. Rather than using the existing host IP stack, you can specify a new IP address. In fact, you can also specify your own MAC address! The combination of IP and MAC addresses is usually unique to each physical computer. If someone is sniffing the network, your new IP-MAC combination will appear to be a stand-alone machine on the network. This might divert attention away from the actual physical machine that is infected. It may also be used to bypass filters.

Creating Your MAC Address

The first step we need to take to emulate a new host on the network is to create our own MAC address. The MAC address is associated with the network card being used. Usually, this is hard-coded at the factory, and it is not meant to be changed. However, by crafting raw packets, it's possible to have any MAC of your choosing.

A MAC consists of 48 bits of data, including a vendor code. When you craft a new MAC address, you can select the vendor code to use. Most sniffer programs resolve the vendor code.

Some switches can be configured to allow only one MAC address per port. In fact, they can be configured to allow only a *specific* MAC address on a given port. If a switch is configured this way, the actual host MAC and your new MAC will conflict. This usually results in your new IP-MAC combination not working, or the entire port getting shut down.

Handling ARP

Forging raw network frames is not without its complications. If you are forging a source IP address and an Ethernet MAC address, you are required to handle the ARP (address resolution) protocol. If you don't provide for ARP, no packets will be routed to your network. The ARP protocol tells the router that your source IP is available, and more importantly, which Ethernet address it should be routed to.

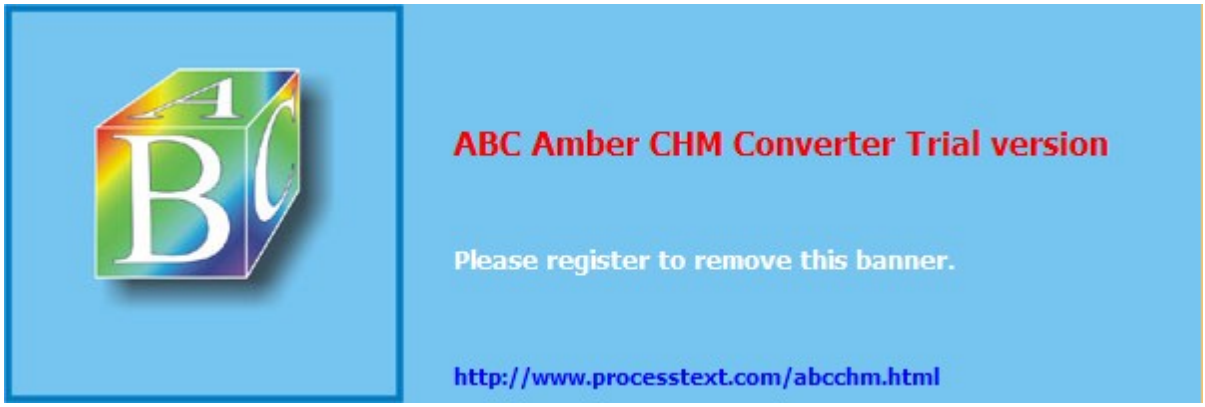
This is also important for switches. A good switch will know which Ethernet address is using which ports. If your rootkit doesn't handle the Ethernet address properly, then the switch may not send packets down the right wire. It should also be noted that some switches allow only a single Ethernet address per port. If your rootkit tries to use an alternate MAC address, the switch might throw an alarm and block communication on your wire. This has a tendency to make a system administrator put down her doughnut, grab a crimper, and start "debuggering." That is the last event you want your rootkit to initiate.

What follows is example code from a rootkit that responds to an ARP request. This code was taken from a publicly available rootkit, rk_044, which can be downloaded from rootkit.com.

Rootkit.com

The source code for the entire rootkit excerpted here may be found at:
www.rootkit.com/vault/hoglund/rk_044.zip

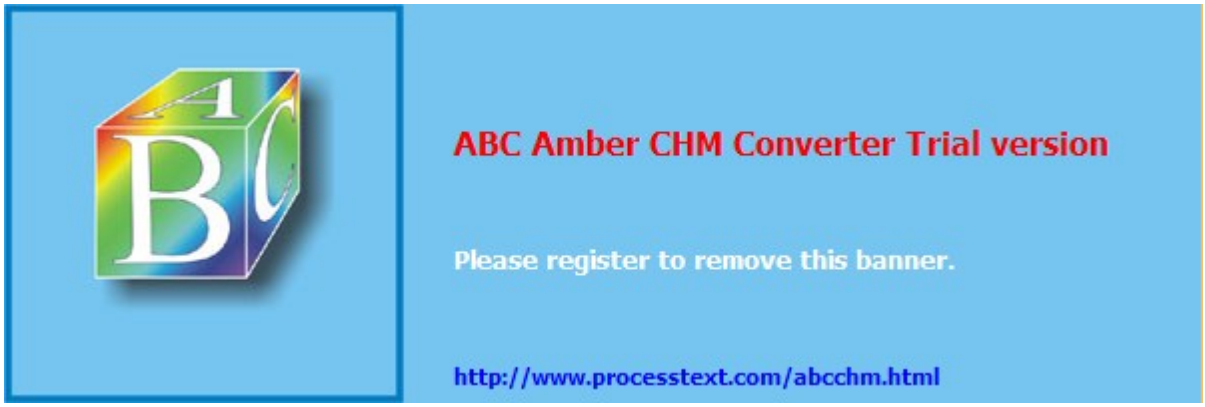
```
#define ETH_P_ARP   0x0806    // Address Resolution Packet
#define ETH_ALEN      6      // octets in one ethernet addr
#define ARPOP_REQUEST   0x01
#define ARPOP_REPLY   0x02

// Ethernet Header
struct ether_header
{
  unsigned char    h_dest[ETH_ALEN];  /* destination eth addr
*/
```



Conclusion

Data hiding is an old topic applied to new technologies. Even Hollywood and popular fiction have sensationalized the idea. In this chapter, we touched upon the essential concept of "hiding in plain sight," and introduced NDIS and TDI mechanisms that can be used to send and receive network data from a Microsoft Windows kernel driver.

Using the available technology, systems can be crafted to move data into and out of networks without detection. That may seem to be a lofty claim, but most networks are busy, overtaxed, and lack robust intrusion detection architectures. For the most part, the network admins just do their best to keep everything running, and a little trickle of covert data will simply be overlooked.



Chapter 10. Rootkit Detection

I know not whether my native land be a grazing ground for wild beasts or yet my home!

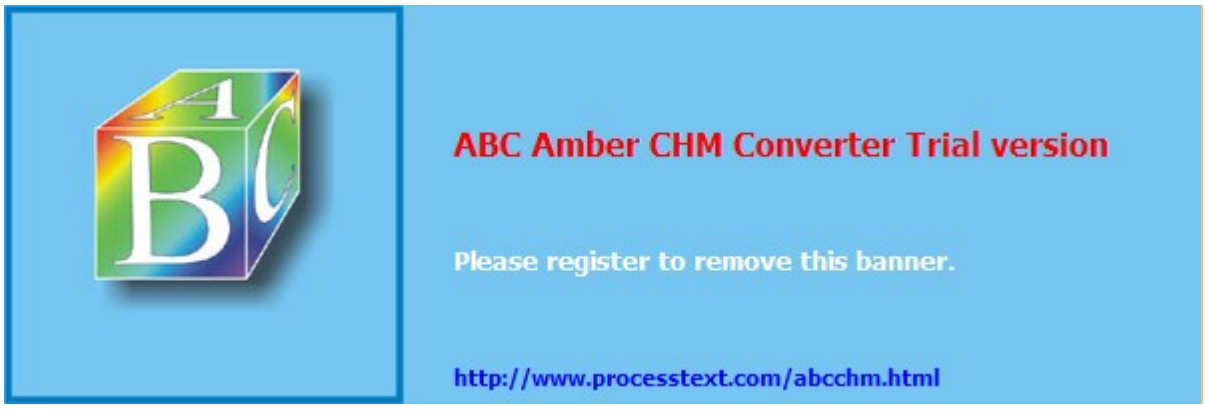
?ANONYMOUS POET OF MA'ARRA

As we have shown throughout this book, rootkits can be difficult to detect, especially when they operate in the kernel. This is because a kernel rootkit can alter functions used by all software, including those needed by security software.

The same powers available to infection-prevention software are also available to a rootkit. Whatever avenues can be blocked to prevent rootkit intrusion can simply be unblocked. A rootkit can prevent detection or prevention software from running or working properly. In the end, it comes down to an arms race between the attacker and the defender, with a large advantage going to whichever one loads into the kernel and executes first.

That is not to say all is lost for the defender, but you should be aware what works today may not detect the rootkit of tomorrow. As rootkit developers learn what detection software is doing, better rootkits will evolve. The reverse is also true: Defenders will constantly update detection software as new rootkit techniques emerge.

In this chapter, we take a look at the two basic approaches to rootkit detection: detecting the rootkit itself, and detecting the behavior of a rootkit. Once you become familiar with these approaches, you will be in a better position to defend yourself.



Detecting Presence

Many techniques can be used to detect the presence of the rootkit. In the past, software such as Tripwire [1] looked for an image on the file system. This approach is still used by most anti-virus vendors, and can be applied to rootkit detection.

[1] www.tripwire.org

The assumption behind such an approach is that a rootkit will use the file system. Obviously, this will not work if the rootkit runs only from memory or is located on a piece of hardware. In addition, if anti-rootkit programs are run on a live system that has already been infected, they may be defeated.[2] A rootkit that is hiding files by hooking system calls or by using a layered file filter driver will subvert this mode of detection.

[2] For best results, file integrity checking software should be run offline against a copy of the drive image.

Because software such as Tripwire has limitations, other methods of detecting rootkit presence have evolved. In the following sections, we will cover some of these methods, used to find a rootkit in memory or detect proof of the rootkit's presence.

Guarding the Doors

All software must "live" in memory somewhere. Thus, to discover a rootkit, you can look in memory.

This technique takes two forms. The first seeks to detect the rootkit as it loads into memory. This is a "guarding-the-doors" approach, detecting what comes into the computer (processes, device drivers, and so forth). A rootkit can use many different operating-system functions to load itself into memory. By watching these ingress points, detection software can sometimes spot the rootkit. However, there are many such points to watch; if the detection software misses any of the loading methods, all bets are off.

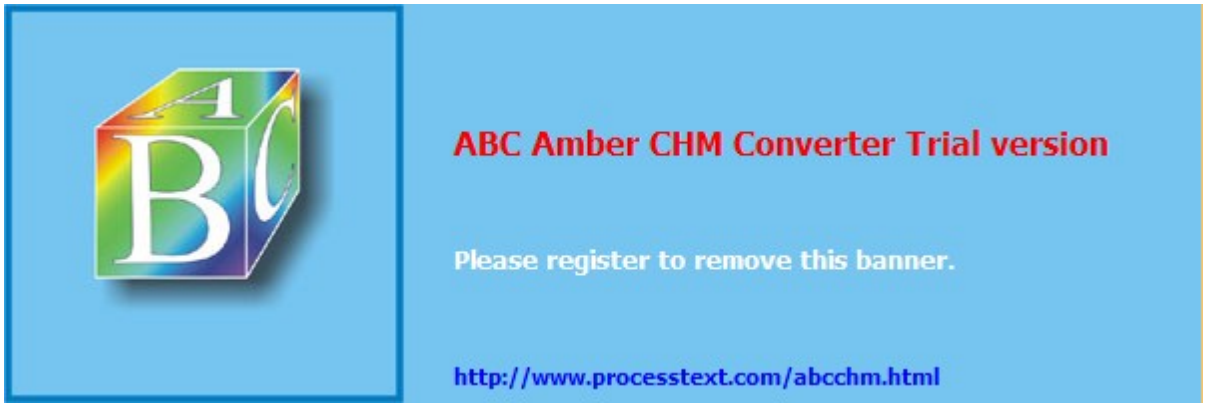
This was the problem with Pedestal Software's Integrity Protection Driver (IPD)[3]. IPD began by hooking kernel functions in the SSDT such as NtLoadDriver and NtOpenSection. One of your authors, Hoglund, found that one could load a module into kernel memory by calling ZwSetSystemInformation, which IPD was not filtering. After IPD was fixed to take this fact into account, in 2002, Crazylord published a paper that detailed using a symbolic link for \\DEVICE\\PHYSICALMEMORY to bypass IPD's protection.[4] IPD had to continually evolve to guard against the latest ways to bypass the protection software.

[3] It appears Pedestal (www.pedestalsoftware.com) no longer offers this product.

[4] Crazylord, "Playing with Windows /dev/(k)mem," *Phrack* no. 59, Article 16 (28 June 2002), available at: www.phrack.org/phrack/59/p59-0x10.txt

The latest IPD version hooks these functions:

- ZwOpenKey
- ZwCreateKey
- ZwSetValueKey
- ZwCreateFile
- ZwOpenFile
- ZwOpenSection
- ZwCreateLinkObject
- ZwSetSystemInformation



Detecting Behavior

Detecting behavior is a promising new area in rootkit detection. It is perhaps the most powerful. The goal of this technique is to catch the operating system in a "lie." If you find an API that returns values you know to be false, not only have you identified the presence of a rootkit, but you have also identified what the rootkit is trying to hide. The behavior you are looking for is the lie. A caveat to this is that you must be able to determine what the "truth" is without relying upon the API you are checking.

Detecting Hidden Files and Registry Keys

Mark Russinovich and Bryce Cogswell have released a tool called Rootkit-Revealer.[7] It can detect hidden Registry entries as well as hidden files. To determine what the "truth" is, RootkitRevealer parses the files that correspond to the different Registry hives without the aide of the standard Win32 API calls, such as RegOpenKeyEx and RegQueryValueEx. It also parses the file system at a very low level, avoiding the typical API calls. RootkitRevealer then calls the highest level APIs to compare the result with what it knows to be true. If a discrepancy is found, the behavior of the rootkit (and, hence, what it is hiding) is identified. This technique is fairly straightforward, yet very powerful.

[7] B. Cogswell and M. Russinovich, *RootkitRevealer,* available at: www.sysinternals.com/ntw2k/freeware/rootkitreveal.shtml

Detecting Hidden Processes

Hidden processes and files are some of the most common threats you will face. A hidden process is particularly threatening because it represents code running on your system that you are completely unaware of. In this section, you will learn different ways to detect processes the attacker does not want you to see.

Hooking SwapContext

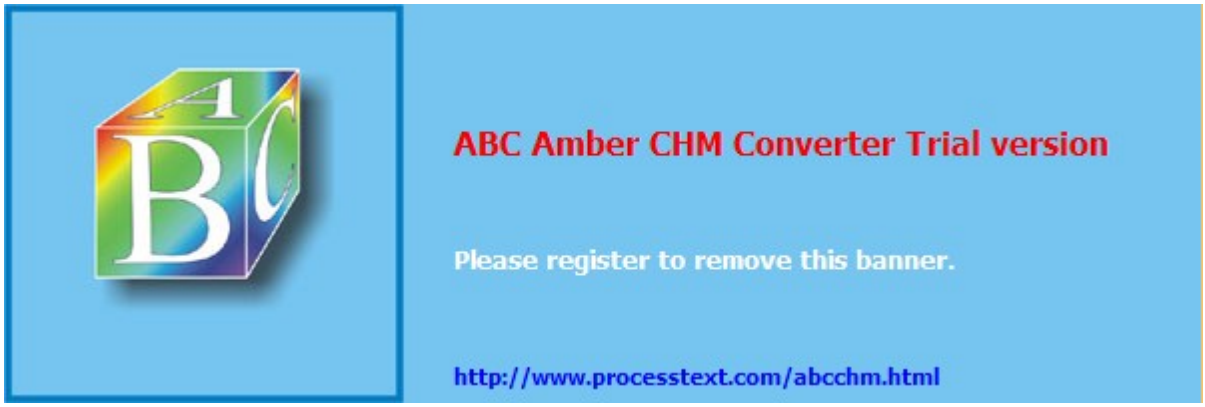
Hooking functions is useful during detection. The SwapContext function in ntoskrnl.exe is called to swap the currently running thread's context with the thread's context that is resuming execution. When SwapContext has been called, the value contained in the EDI register is a pointer to the next thread to be swapped in, and the value contained in the ESI register is a pointer to the current thread, which is about to be swapped out. For this detection method, replace the preamble of SwapContext with a five-byte unconditional jump to your detour function. Your detour function should verify that the KTHREAD of the thread to be swapped in (referenced by the EDI register) points to an EPROCESS block that is appropriately linked to the doubly linked list of EPROCESS blocks. With this information, you can find a process that was hidden using the DKOM tricks outlined in Chapter 7, Direct Kernel Object Manipulation. The reason this works is that scheduling in the kernel is done on a thread basis, as you will recall, and all threads are linked to their parent processes. This detection technique was first documented by James Butler et. al.[8]

[8] J. Butler et al., "Hidden Processes: The Implication for Intrusion Detection," *Proceedings of the IEEE Workshop on Information Assurance* (United States Military Academy, West Point, NY), June 2003.

Alternatively, you could use this method to detect processes hidden by hooking. By hooking SwapContext, you get the true list of processes. You can then compare this data with that returned by the APIs used to list processes, such as the NtQuerySystemInformation function that was hooked in the section Hooking the System Service Descriptor Table in Chapter 4.

Different Sources of Process Listings

There are ways to list the processes on the system other than going through the ZwQuerySystemInformation function. DKOM and hooking tricks will fool this API. However, a simple alternative like listing the ports with netstat.exe may reveal a hidden process, because it has a handle to a port open. We discuss using netstat.exe in Chapter 4.



Conclusion

This chapter has shown you many different ways to detect rootkits. We have covered practical implementations, and discussed the theory behind other techniques.

Most of the methods in this chapter have focused on detecting hooks and hidden processes. Whole books could be written on file-system detection, or on detecting covert communication channels. By identifying hooks, though, you will be well on your way to detecting most public rootkits.

No detection algorithm is complete or foolproof. The art of detection is just that?an art. As the attacker advances, the detection methods will evolve.

One drawback of spelling out both rootkit and detection methodologies is that this discussion favors the attacker. As methods to detect an attacker are explained, the attacker will alter her methodology. However, the mere fact that a particular subversion technique has not been written up in a book or presented at a conference does not make anyone any safer. The level of sophistication in the attacks presented in this book is beyond the reach of the majority of so-called "hackers," who are basically script-kiddies. We hope the techniques discussed in this publication will become the first methods that security companies and operating system creators begin to defend against.

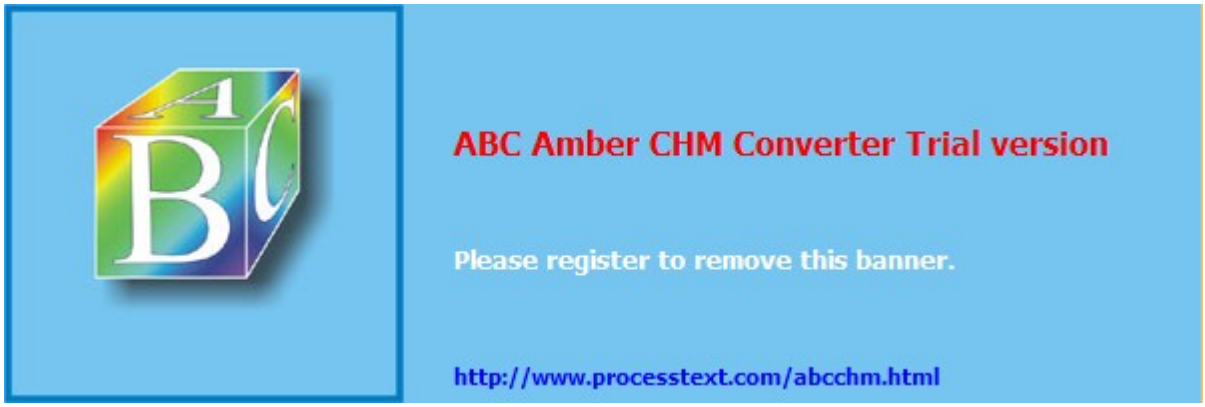
More-advanced rootkit techniques and their detection are being developed as you read these words. Currently, we are aware of several efforts to cloak rootkits in memory so that even memory scanning is corrupted. Other groups are moving to hardware with embedded processors in order to scan kernel memory without relying upon the operating system.[11] Obviously these two groups will be at odds. Since neither implementation is available for public scrutiny, it is hard to say which one has the upper hand. We are sure that each one will have its own limitations and weaknesses.

[11] N. Petroni, J. Molina, T. Fraser, and W. Arbaugh (University of Maryland, College Park, Md.), "Copilot: A Coprocessor Based Kernel Runtime Integrity Monitor," paper presented at Usenix Security Symposium 2004, available at: www.usenix.org/events/sec04/tech/petroni.html

The rootkits and detection software mentioned in the previous paragraph represent the extremes. Before you begin to worry about these new tools, you need to address the most common threats. This book has shown you what they are, and where the attacker is likely to go.

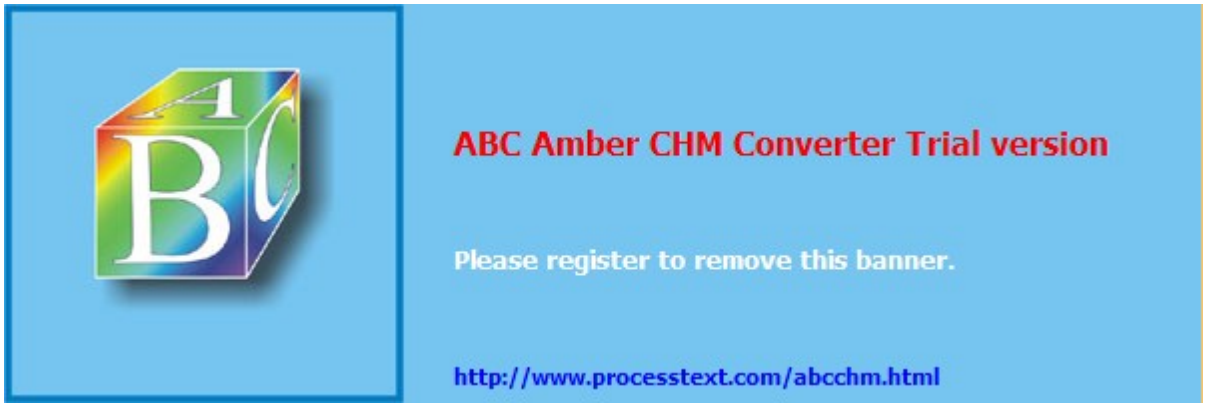
Recently we have seen companies showing their first signs of interest in rootkit detection. We hope this trend will continue. Having more-informed consumers will cause protection software to advance. The same can be said for having more-informed attackers.

As we stated in Chapter 1, corporations are not motivated to protect against a potential attack until there is an attack. You are now that motivation!



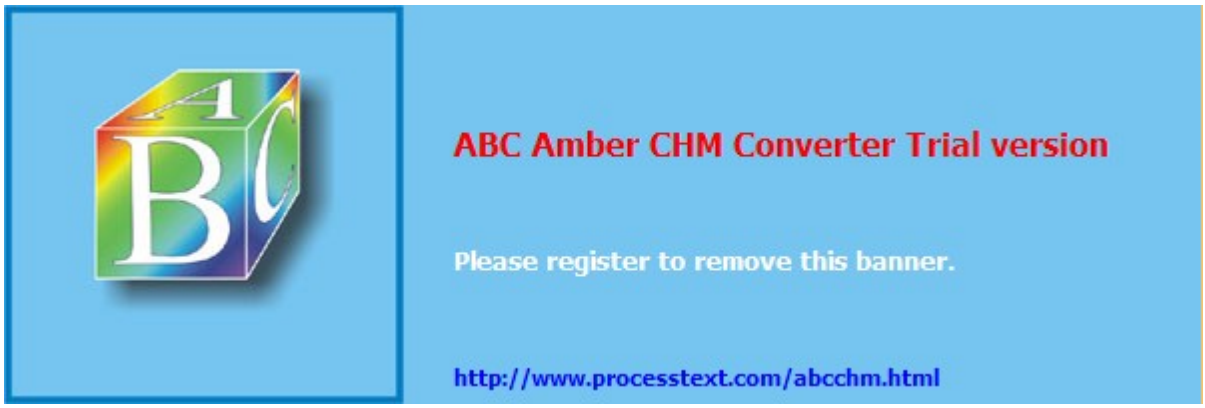
Index

PREV < Day Day Up >



Index

PREV < Day Day Up >



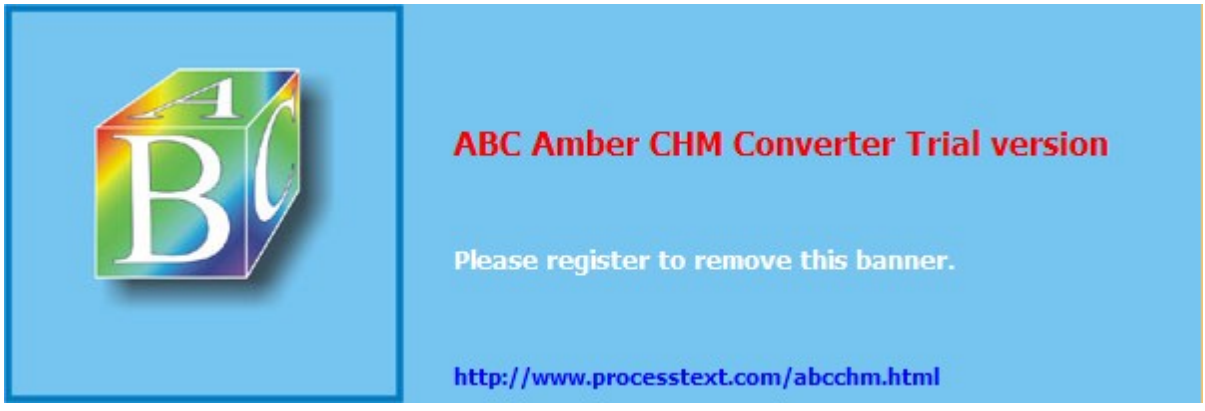
Index

PREV < Day Day Up >

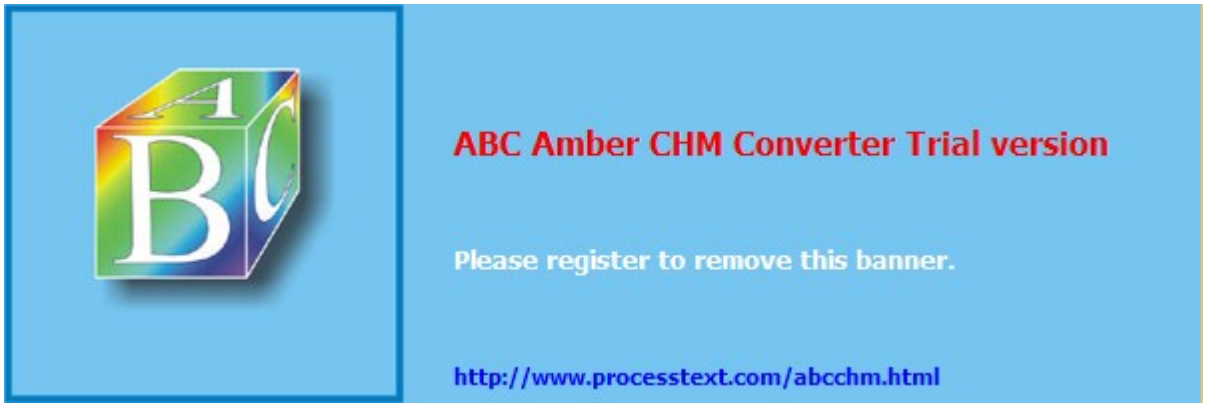
Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [Z]

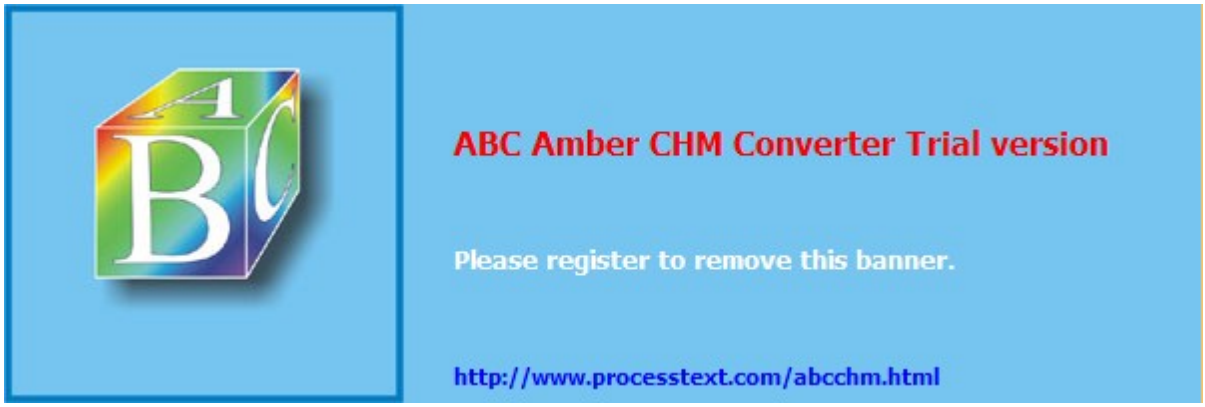
PREV < Day Day Up >



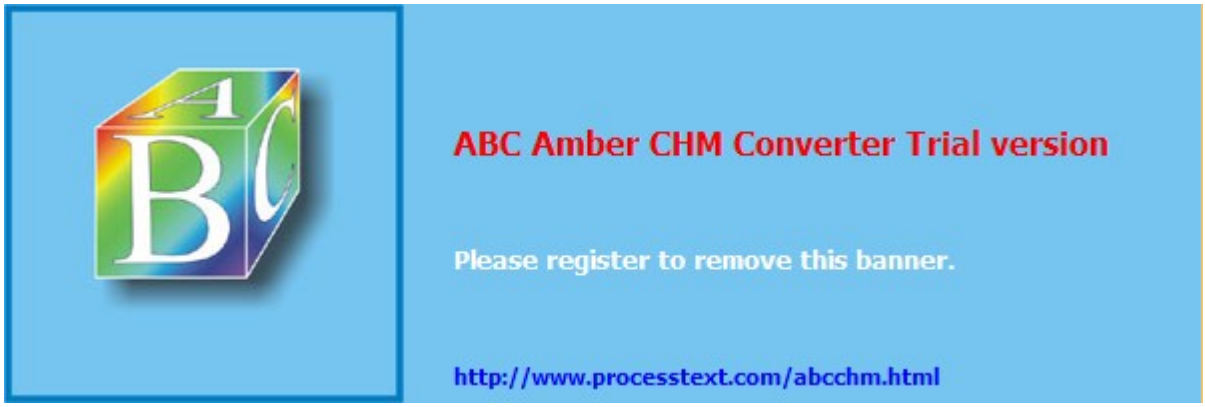
Index



Index

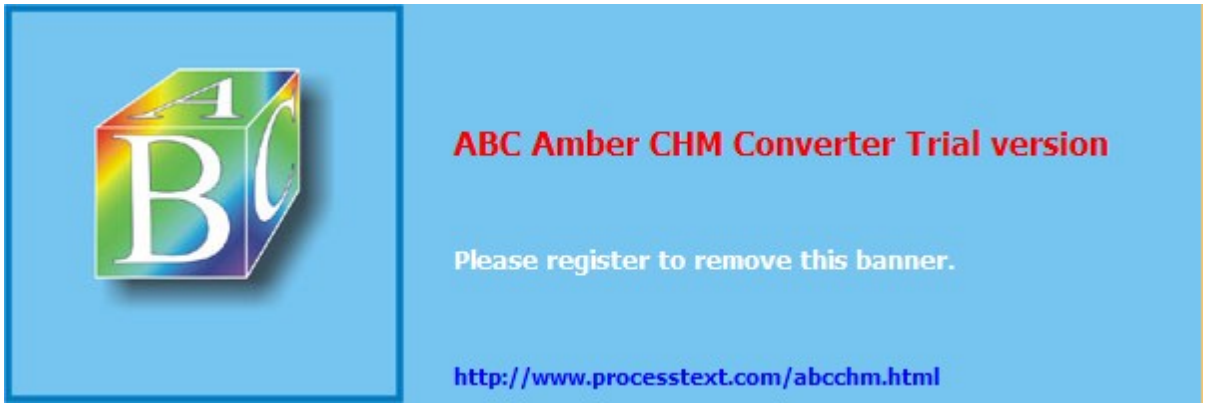


Index



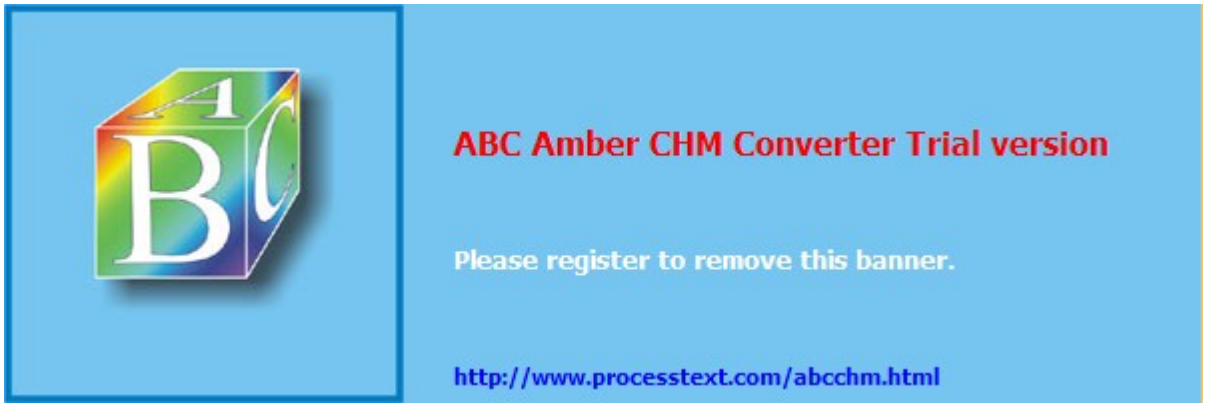
Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [Z]



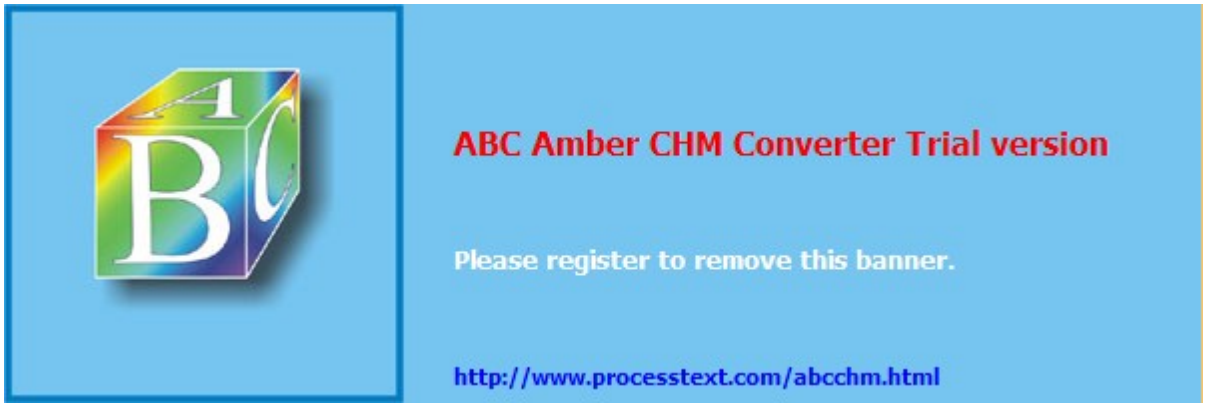
Index

Index



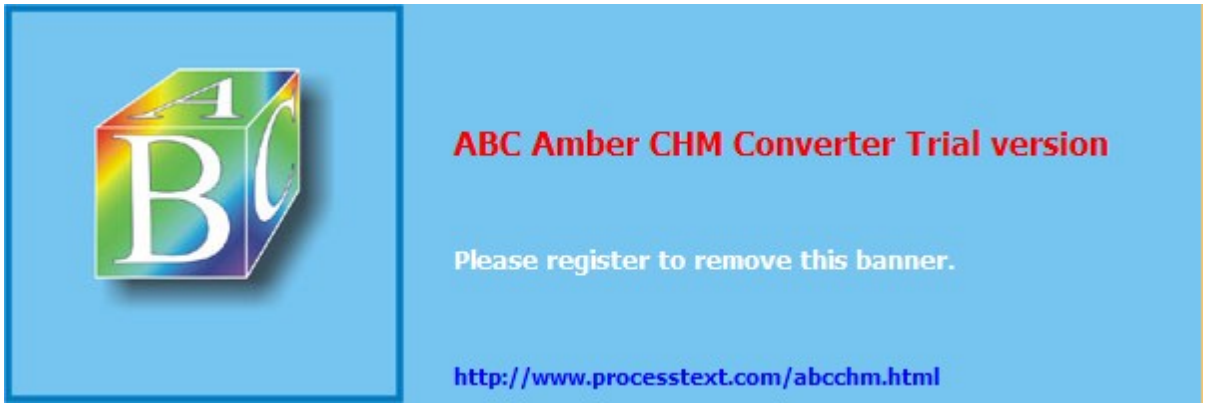
Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [Z]

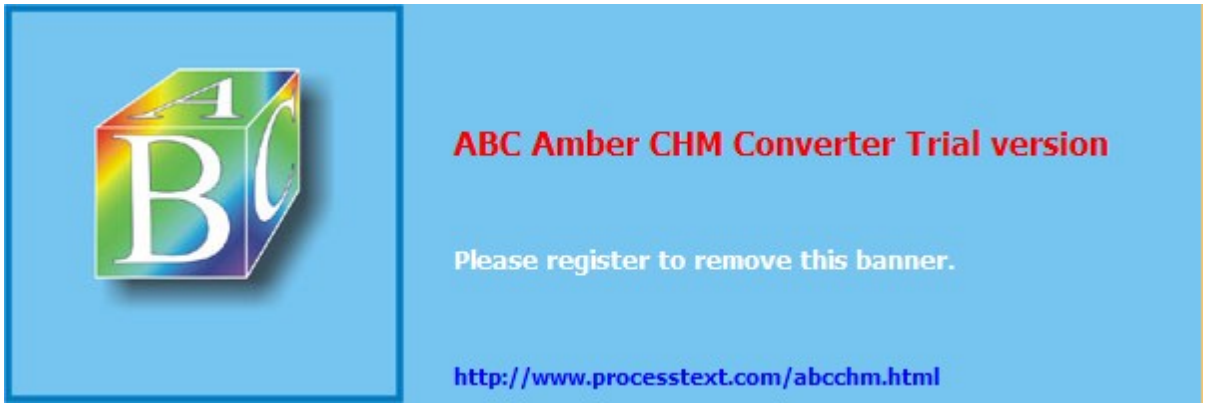


Index

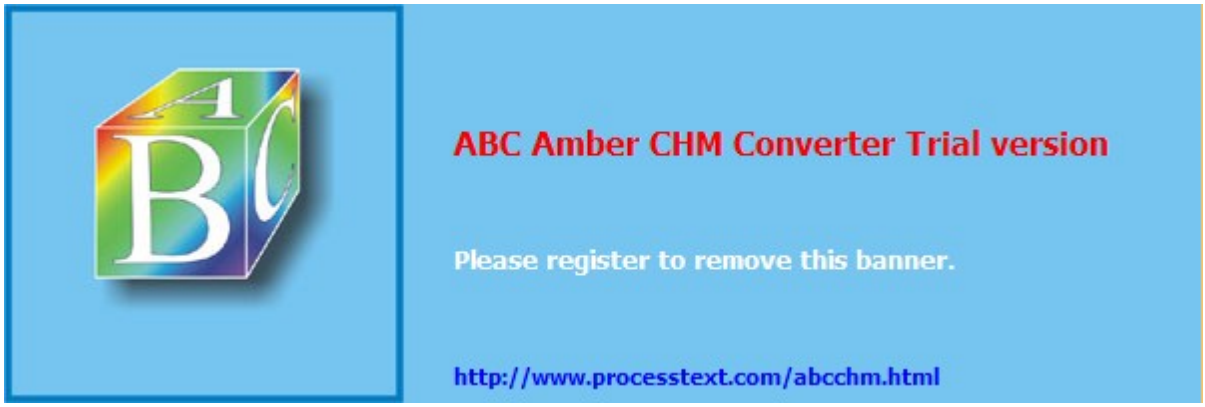
PREV < Day Day Up >



Index

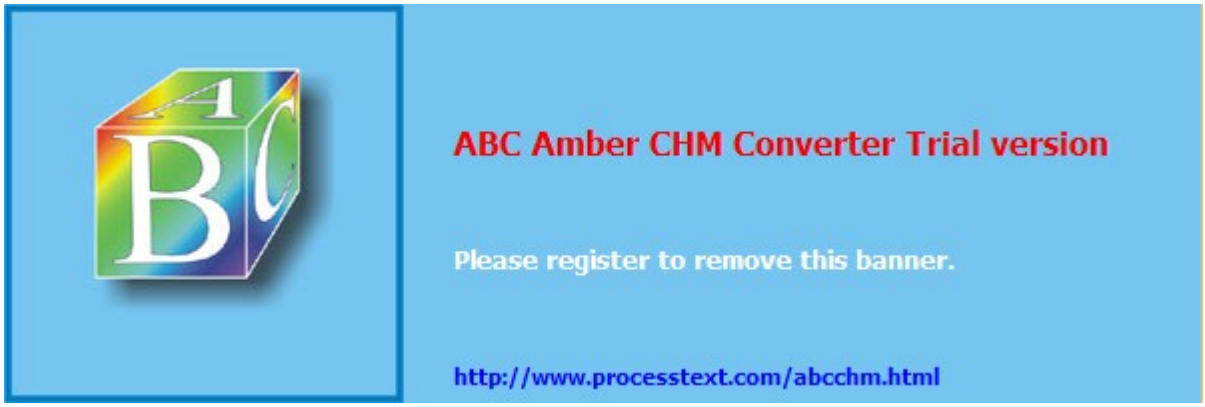


Index



Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [**M**] [N] [O] [P] [R] [S] [T] [U] [V] [W] [Z]

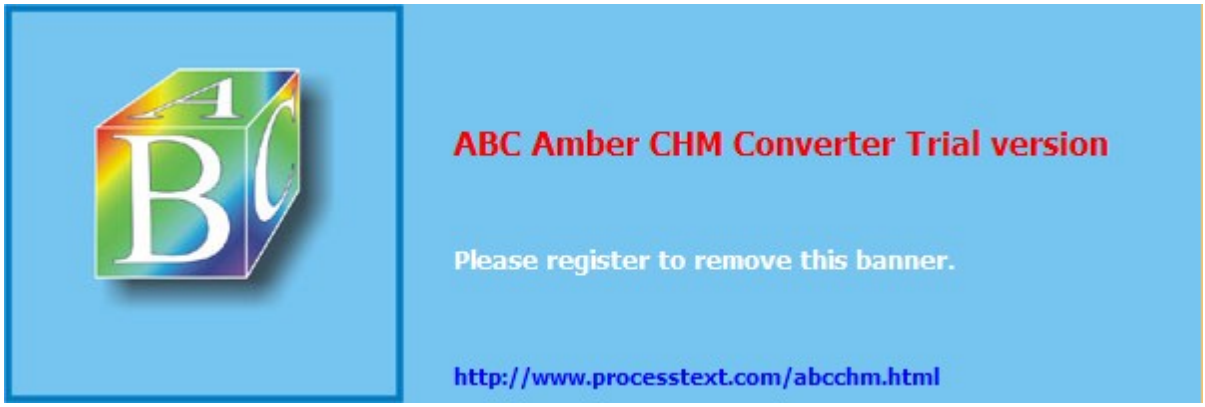


Index

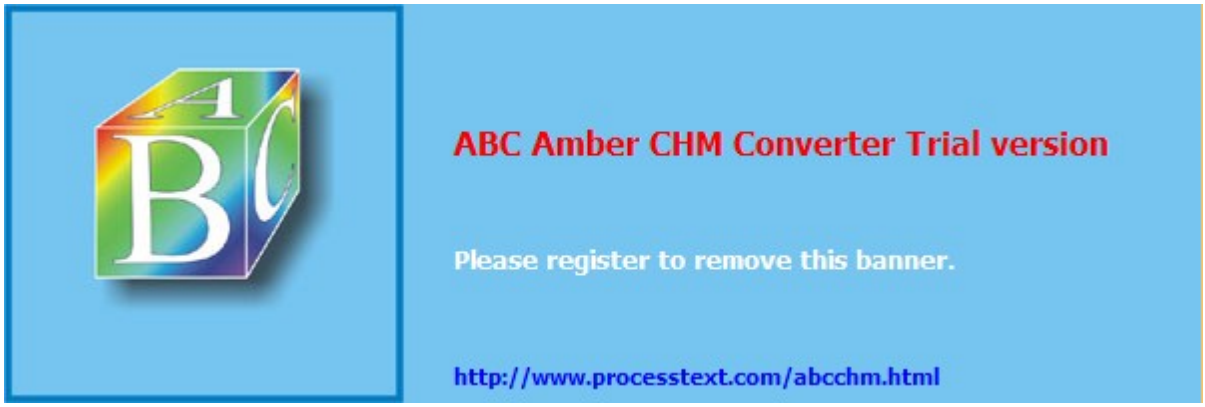
PREV < Day Day Up >

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [Z]

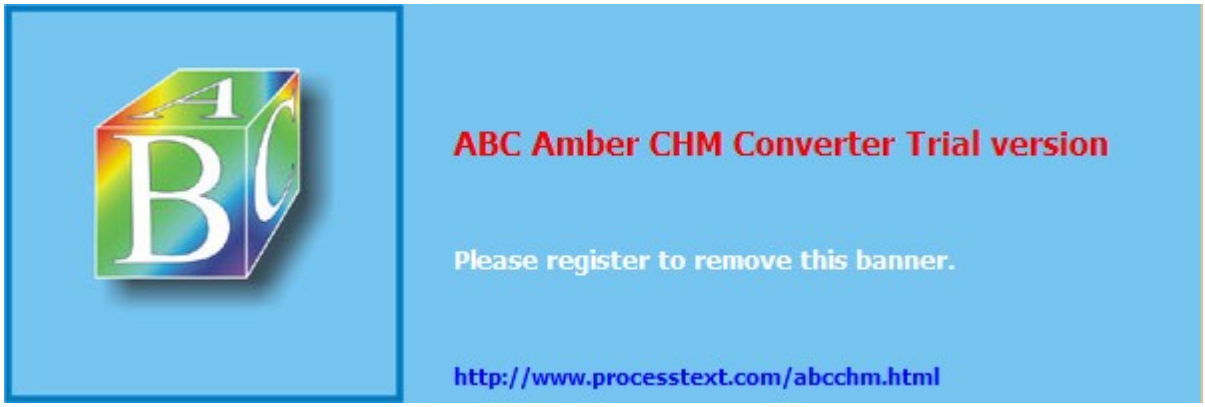


Index

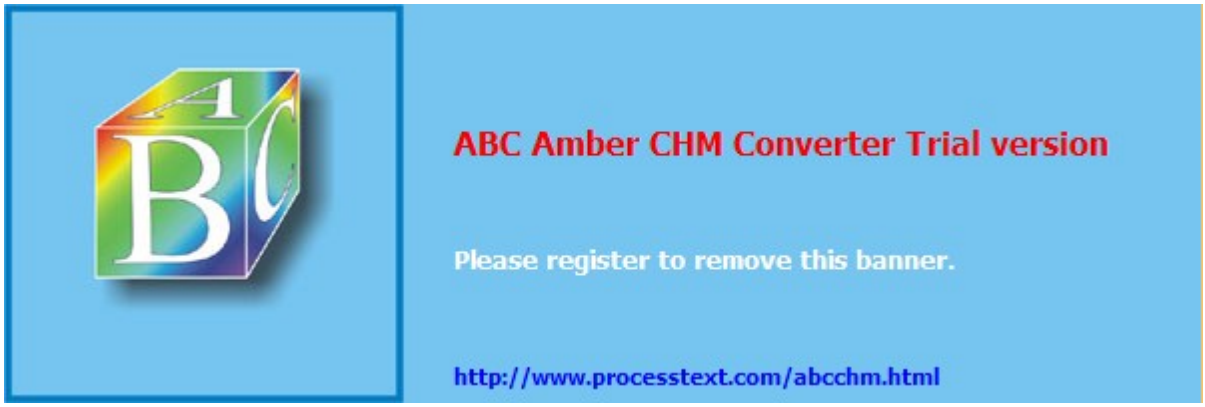


Index

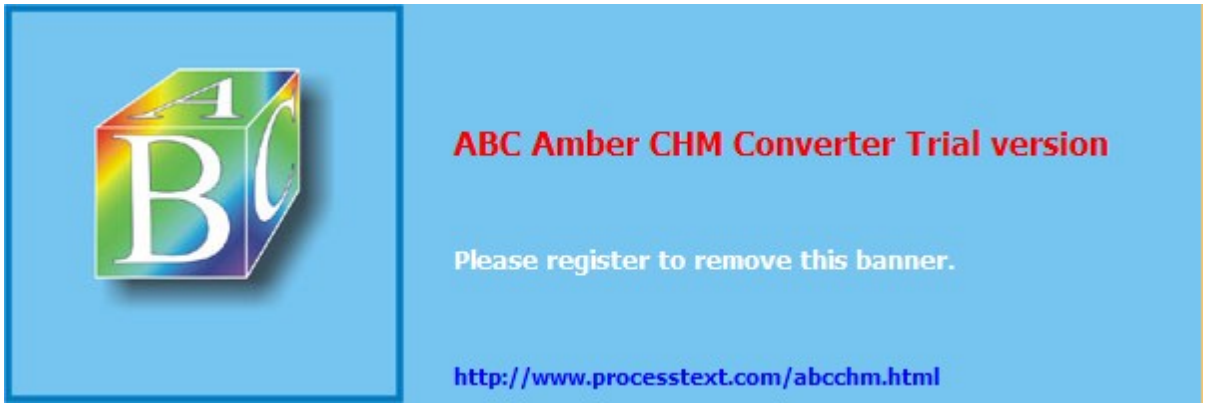
[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [Z]



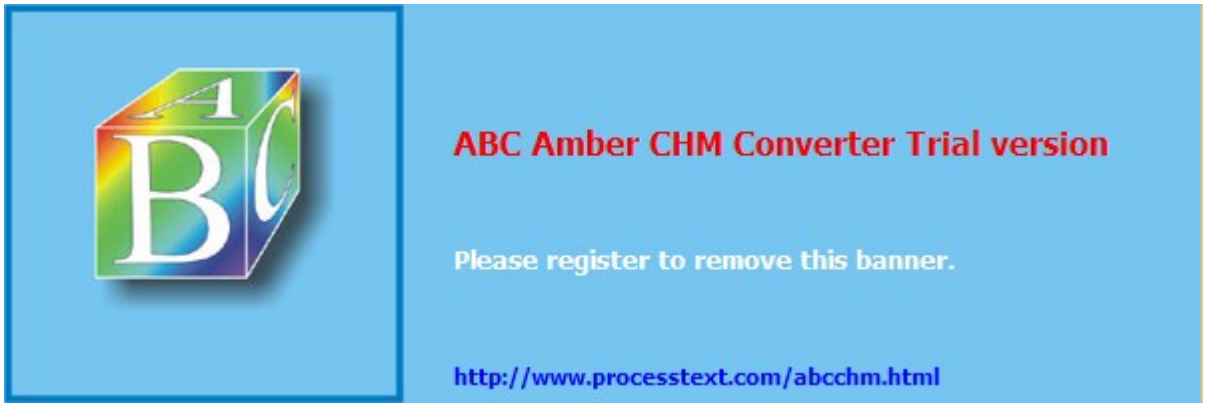
Index



Index

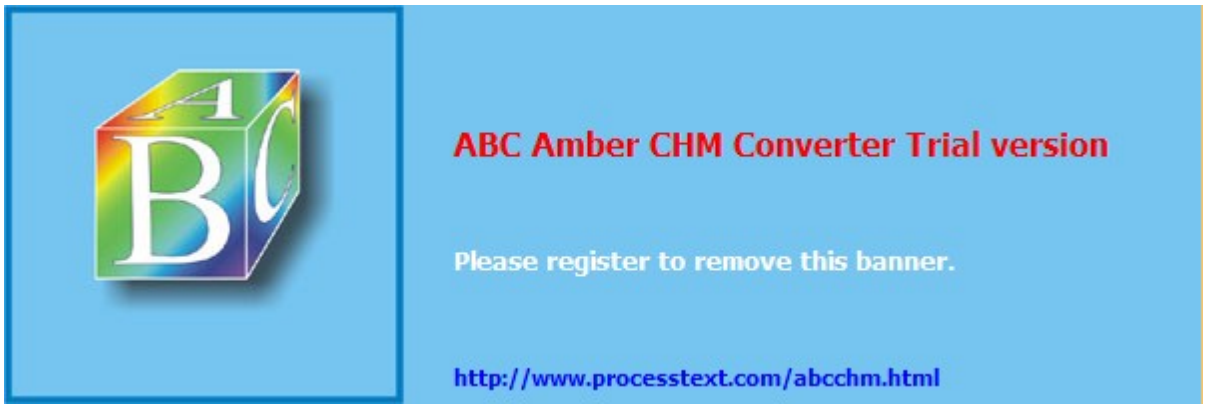


Index

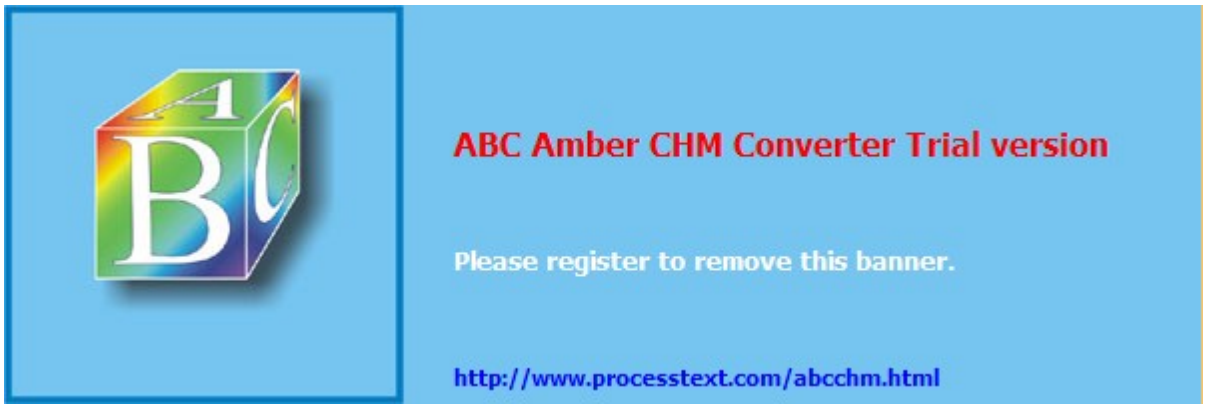


Index

PREV < Day Day Up >



Index



Index

PREV < Day Day Up >