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]

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.

---

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).

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.

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]

---

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

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 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.

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.

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. For example, perhaps the target computer is controlling an anti-aircraft system. In this case, stealth is not a concern—just crashing the system is enough to achieve the objective. In most cases, a computer crash will be obvious (and disturbing) to the victim. If this is the kind of attack you want to learn more about, this book will not help you.

Now that you have a basic understanding of attackers' motives, we'll spend the rest of this chapter discussing rootkits in general, including some background on the subject as well as how rootkits work.

---

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

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.

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.

```
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.
```

Figure 1–1 Menu for a kernel rootkit.

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]

Having detected some of these operations, the US planted bogus plans, software, and materials into the collection channel. In one reported incident, malicious modifications to software (so-called "extra ingredients") were credited for a Siberian gas pipeline explosion.[8] The explosion was photographed by satellites and was described as "the most monumental non-nuclear explosion and fire ever seen from space."[9]

Legitimate Uses of Rootkits

As we alluded to already, rootkits can be used for legitimate purposes. For instance, they can be used by law-enforcement agencies to collect evidence, in an advanced bugging operation. This would apply to any crime in which a computer is used, such as computer trespass, creating or distributing child pornography, software or music piracy, and DMCA[10] violations.

Rootkits can also be used to fight wars. Nations and their militaries rely heavily on computing machinery. If these computers fail, the enemy's decision cycle and operations can be affected. The benefits of using a computer (versus conventional) attack include that it costs less, it keeps soldiers out of danger, it causes little collateral damage, and in most cases it does not cause permanent damage. For instance, if a nation bombs all the power plants in a country, then those power plants will need to be

---

7. G. Weiss, "The Farewell Dossier," in *Studies in Intelligence* (Washington: Central Intelligence Agency, Center for the Study of Intelligence, 1996), available from www.cia.gov/csi/studies/96unclass/farewell.htm.

8. This implies that the explosion was caused by some sort of software subversion.

9. D. Hoffman, "Cold War hotted up when sabotaged Soviet pipeline went off with a bang," *Sydney Morning Herald,* 28 February 2004.

10. The Digital Millenium Copyright Act of 1998, PL 105-304, 17 USC § 101 et seq.

rebuilt at great expense. But if a software worm infects the power control network and disables it, the target country still loses use of the power plants' output, but the damage is neither permanent nor as expensive.

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.

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

---

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).

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.

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).

---

12. www.tripwire.org

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.

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]

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!

---

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

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

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 source-control servers and the source-code distribution sites? There are several examples of attackers gaining access to source code. A major example of this type of compromise took place when the root FTP servers for the GNU Project (gnu.org), source of the Linux-based GNU operating system, were compromised in 2003.[16] Modifications to source code can end up in hundreds of program distributions and are extremely difficult to locate. Even the sources of the very tools used by security professionals have been hacked in this way.[17]

The Legality of Software Modification

Some forms of software modification are illegal. For example, if you use a program to modify another program in a way that removes copyright mechanisms, you may be in violation of the law (depending on your jurisdiction). This applies to any "cracking" software that can commonly be found on the Internet. For example, you can download an evaluation copy of a program that "times out" and stops functioning after 15 days, then download and apply a "crack," after which the software will run as if it had been registered. Such a direct modification of the code and logic of a program would be illegal.

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*.

---

15. Linus Torvalds is the father of Linux.

16. CERT Advisory CA-2003-21, available from www.cert.org/advisories/CA-2003-21. html.

17. For example, D. Song's monkey.org site was compromised in May, 2002, and the Dsniff, Fragroute and Fragrouter tools hosted there were contaminated. *See* "Download Sites Hacked, Source Code Backdoored," SecurityFocus, available at www.securityfocus. com/news/462.

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.

The Virus Problem

Even though a rootkit is not a virus, the techniques used by a rootkit can easily be employed by a virus. When a rootkit is combined with a virus, a very dangerous technology is born.

The world has seen what viruses can do. Some virus programs have spread through millions of computers in only a few hours.

The most common operating system, Microsoft Windows, has historically been plagued with software bugs that allow viruses to infect computers over the Internet. Most malicious hackers will not reveal software bugs to the vendor. In other words, if a malicious hacker were to find an exploitable bug in Microsoft Windows, she would not reveal this to Microsoft. An exploitable bug that affects the default installation of most Windows computers is like a "key to the kingdom"; telling the vendor about it would be giving away the key.

Understanding rootkit technology is very important for defending against viruses. Virus programmers have been using rootkit technology for many years to "heat up" their viruses. This is a dangerous trend. Algorithms

---

18. A zero-day exploit is brand new, and no software patch exists yet to fix it.

have been published for virus propagation[19] that can penetrate hundreds of thousands of machines in an hour. Techniques exist for destroying computer systems and hardware. And, remotely exploitable holes in Microsoft Windows are not going away. Viruses that use rootkit technology are going to be harder to detect and prevent.

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])

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]

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).

---

19. N. Weaver, "Warhol Worms: The Potential for Very Fast Internet Plagues," available from www.cs.berkeley.edu/~nweaver/warhol.html.

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

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.

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.

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

24. www.eEye.com

Type-Safe Languages

Programming languages that are *type-safe* are more secure from certain exploits, such as buffer overflows.

Without type safety, program data is just a big ocean of bits. The program can grab any arbitrary handful of bits and interpret it in limitless ways—regardless of the original purpose of the data. For example, if the string "GARY" were placed into memory, it could later be used not as text, but as a 32-bit integer, 0x47415259 (or, in decimal, 1,195,463,257—a rather large number indeed!). When data supplied by an external user can be misinterpreted, software exploits can be employed.

Conversely, programs written in a type-safe language (like Java or C#[25]) would never convert "GARY" to a number; the string would always be treated as text and nothing else.

Why Exploits Are Still a Problem

The need for software security has been known for a long time, yet software exploits continue to be a problem. The root of the problem lies within the software itself. Bluntly stated, most software is not secure. Companies like Microsoft are making huge strides in designing better security for the future, but current operating-system code is written in C or C++, computer languages that by their *very nature* introduce severe security holes. These languages give rise to a problem known as *buffer-overflow exploits*. The buffer-overflow bug is the most significant weakness in software today. It has been the enabler for thousands of software exploits. And, it's a bug—an accident that can be fixed.[26]

Buffer-overflow exploits will eventually go away, but not in the near future. Although a disciplined programmer can write code that does not

---

25. C# (pronounced "see sharp") is not the same language as "C" ("see") or C++ ("see plus plus").

26. Although buffer-overflow bugs are not confined to C and C++ code, the C and C++ programming languages make it difficult to ensure safe coding practices. The languages are not type-safe (discussed later in this chapter), use built-in functions that can overflow buffers, and are difficult to debug.

have buffer-overflow bugs (this is regardless of language; even a program written by hand in Assembly can be secure), most programmers are not that diligent. The current trend is to enforce safe coding practices and follow this up with automated code-scanning tools to catch mistakes. Microsoft uses a set of internal tools for this purpose.[27]

Automated code-scanning tools can catch some bugs, but not all of them. Most computer programs are very complex, and it can be difficult to test them thoroughly in an automated fashion. Some programs may have too many states to possibly evaluate.[28] In fact, it is possible for a computer program to have more potential states than there are particles in the universe.[29] Given this potential complexity, it can be very hard to make any determination about the security of a computer program.

The adoption of type-safe languages (such as Java and C#) would nearly eliminate the risk of buffer overflows. Although a type-safe language is not guaranteed to be secure, it significantly reduces the risks of buffer overflows, sign-conversion bugs, and integer overflows *(see sidebar on page 15)*. Unfortunately, these languages cannot match the performance of C or C++, and most of Microsoft Windows—even the latest and greatest version—still runs old C and C++ code. Developers of embedded systems have begun to adopt type-safe languages, but even this uptake is slow—and the millions of legacy systems out there will not be replaced any time soon. What this means is that old-fashioned software exploits will be around for awhile.

---

27. For example, PREfix and PREfast were developed and deployed by Jon Pincus, Microsoft Research. *See* http://research.microsoft.com/users/jpincus/

28. A "state" is like an internal configuration within the software. Every time the software does something, the state will change. Thus, most software has a huge number of potential states.

29. To understand this, consider the theoretical bounds for the number of permutations of a string of binary bits. For example, imagine a 160MB software application that uses 16MB (10% of its total size) of memory to store state. That program could, in theory, have up to $2^{16,777,216}$ different operational states, which is far, far larger than the number of particles in the universe (variously estimated at around 10^{80}). [Thanks to Aaron Bornstein for this clarifying example.]

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]

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 and attempts to detect rootkits. Active offenses can also be used to render system-administration tools useless for detecting an attack. A complex offense could render any security software tool ineffective. For example, an active offense could locate a virus scanner and disable it.

Passive offenses are obfuscations in data storage and transfer. For example, encrypting data before storing it in the file system is a passive offense. A more advanced offense would be to store the decryption key in non-volatile hardware memory (such as flash RAM or EEPROM) instead of in the file system. Another form of passive offense is the use of covert channels for exfiltration of data out of the network.

Finally, a rootkit should not be detected by a virus scanner. Virus scanners not only operate at runtime, they can also be used to scan a file system "offline." For example, a hard drive on a lab bench can be forensically analyzed for viruses. To avoid detection in such cases, a rootkit must hide itself in the file system so that it cannot be detected by the scanner.

Bypassing Forensic Tools

Ideally, a rootkit should never be detected by forensic scanning. But the problem is hard to solve. Powerful tools exist to scan hard drives. Some

---

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.

tools, such as Encase,[31] "look for the bad" and are used when a system is suspected of an infection. Other tools, such as Tripwire, "look for the good" and are used to ensure that a system remains uninfected.

A practitioner using a tool like Encase will scan the drive for byte patterns. This tool can look at the entire drive, not just regular files. Slack space and deleted files will be scanned. To avoid detection in this case, the rootkit should not have easily identifiable patterns. The use of steganography can be powerful in this area. Encryption can also be used, but tools used to measure the randomness of data may locate encrypted blocks of data. If encryption is used, the part of the rootkit responsible for decryption would need to stay un-encrypted (of course). Polymorphic techniques can be used to mutate the decryptor code for further protection. Remember that the tool is only as good as the forensic technicians who drive it. If you think of some way to hide that they have not, you might escape detection.

Tools that perform cryptographic hashing against the file system, such as Tripwire, require a database of hashes to be made from a clean system. In theory, if a copy of a clean system (that is, a copy of the hard drive) is made before the rootkit infection takes place, an offline analysis can be performed that compares the new drive image to the old one. Any differences on the drive image will be noted. The rootkit will certainly be one difference, but there will be others as well. Any running system will change over time. To avoid detection, a rootkit can hide in the regular noise of the file system. Additionally, these tools only look at files, and, they may only look at *some* files—maybe just files considered important. They don't address data stored in non-conventional ways (for example, in bad sectors on a drive). Furthermore, temporary data files are likely to be ignored. This leaves many potential places to hide that will not be checked.

If an attacker is really worried that the system administrator has all things hashed and the rootkit will be detected, she could avoid the file system altogether—perhaps installing a rootkit into memory and never using the drive. One drawback, of course, is that a rootkit stored in volatile memory will vanish if the system reboots.

To take things to an extreme, perhaps a rootkit can install itself into firmware present in the BIOS or a flash RAM chip somewhere.

---

31. www.encase.com

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.

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!

---

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.