Chapter 7

Getting Windows to Lie to You: Discovering How Rootkits Hide

In This Chapter

- Understanding privileges from a rootkit's point of view
- Getting familiar with rootkit hooking tactics
- Checking out alternative, more secretive rootkit technologies
- Looking into the future of rootkit shenanigans

The presence of a rootkit on a computer or network implies a violation of the very framework of the system you have come to know and trust — and that's a security breach of the highest magnitude. No wonder rootkits have to hide to be able to stick around long enough to perform the nefarious functions they were designed for. (There's more about that in Chapter 1.)

How can a rootkit be so all-powerful that it actually gets Windows to lie to you? Well, that power comes from taking advantage of the flexibility and versatility that were built into the Windows architecture — a rootkit just manipulates those features to suit its own needs. That's why this chapter investigates how rootkits trick the operating system into becoming an unwitting accomplice to their deception.

Discovering How Rootkits Hide and Survive

In effect, rootkits are invisible to most traditional malware scanners. They hide by getting the operating system to lie for them or (in the case of scanners) to report falsified results. They intercept and filter system information so the operating system's output excludes any indication of the rootkit.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 152

Whenever a user or program queries the system, the normal assumption is that the information the operating system returns is valid. A rootkit can violate this basic trust — and when that happens, the entire premise of accuracy and reliability that we have come to depend upon gets turned completely upside down. A well-crafted rootkit makes sure the operating system returns only doctored information that doesn't betray its existence. A rootkit hides itself — and all the infected malware components associated with it. It's able to accomplish this by obtaining Administrative access to a system. Once a rootkit gains control of the Administrator account — the Big Kahuna of the Windows operating system — it can then control what the system does (and what it reports).

When a rootkit gets hold of administrative privileges, it can secretly go about changing and hiding vital system components such as files, directories, ports, the Windows Registry, and even the code of the operating system itself. One critical way the rootkit maintains concealment is by *hooking* system function calls (more about those in a minute) — altering the data returned by system utilities and scanners so no evidence of system tampering is revealed.

Some rootkits have what amounts to a self-preservation instinct: The malware associated with such a rootkit (its *payload*) employs a feedback mechanism to ensure the rootkit's survival. When a rootkit command is executed on a compromised system, a component of the malware payload examines other system commands to ensure that the rootkit is fully intact. If it is not, then the rootkit reinfects the system until it's in place and functioning again. In a similar manner, the payload may establish a backdoor and monitor traffic on that port. If the traffic flow doesn't match that of an infected machine, the remote attacker reinfects the host computer (server).

When hackers claim their prize, they sometimes go to extraordinary lengths to stake their claim. Though not associated with a rootkit — yet — one recently discovered threat actually comes with Windows security patches in tow. That's right — once it compromises the system, it plugs the very vulnerability that allowed it to enter so that it can have sole ownership of the computer. Another recently discovered threat that is not known to be rootkit related — again, yet — installs a counterfeit copy of a highly regarded antivirus on the system it has compromised. Once it's on board, it cleans up its new host to prevent other would-be intruders from gaining entry.

Chilling? Creepy? You bet. A hacker will stop at no extreme to own a system, and a rootkit is the ideal tool to allow them to maintain complete and unfettered access. All rootkit modifications — whether at the level of the individual user or the OS kernel — have *one* aim in mind: Conceal the presence of the rootkit and its payload on the infected computer so the payload can proceed with its dirty work without interruption.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 153

Rootkits exist not only on Windows machines but also on other platforms, such as Mac OS, Linux, and UNIX. Because Windows is by far the most popular operating system, most malware writers have chosen to concentrate on Windows so they can get the most bang for their buck.

The remainder of this chapter is a rogue's gallery of rootkit functions — the ways they get access to your system, the different types of rootkits, and just how they hide.

Keys to the Kingdom: Privileges

The concept of *computer privileges* is the key to how rootkits operate — and to how they're classified. Start with a practical fact: Not everybody uses a computer in the same way — especially on a network. So network administrators assign different levels of privilege to different users; the idea is to provide appropriate capabilities (and only the needed ones) for doing all those different jobs. The Administrator account gets to say who can do what.

The most efficient way to imagine computer privileges is as a series of concentric rings around a central point — like a simplified representation of the solar system. Here the "sun" is the kernel of the operating system; and the planetary "orbits" are the rings that represent the different privilege levels of the operating system. The ring closest to the kernel (analogous to the closest orbit around the sun) is referred to as *Ring 0*. Any component that has Ring 0 privileges is operating at kernel level — the highest level of privilege. As you follow the rings or circles farther out, their distance from the center increases — and the farther out they are, the fewer privileges they have.

In reality, Windows systems have fewer rings than the solar system has planets (even if you don't count Pluto). In case you're wondering, Ring 1 and 2 privileges are not used in current Windows platforms; Microsoft used them last in the 1990s, in an operating system called OS2.

If an application wants to access a system resource such as the hard drive or memory, it must communicate with the operating system to do so. How direct or indirect this level of communication is, is determined by the level of privilege a given application has. Rootkits with higher privileges have an easier time compromising your system.

Rootkits need administrative access to infect a computer, so two very basic ways to limit your risk of infection are (1) to log on as a limited user and (2) use strong passwords to protect all your user accounts.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 154

An example of a rootkit snafu

The University of Connecticut (UConn) computer system housed a rootkit for nearly two years before it was noticed, according to a June 27, 2005 article in *eweek*. The sheer scale of possible trouble such a rootkit could have caused was considerable; it was detected on a server that contained identity information for 72,000 people — students, staff, and faculty.

Luckily — *very* luckily — in this particular case, there was no leakage of sensitive information because the rootkit's attempt to install a backdoor failed. Still, the incident serves as a warning to network administrators and IT professionals, and stresses the importance of having adequate recovery and security measures in place. Although the UConn rootkit made the news (basically because they reported it), no doubt there are other servers that have fallen prey to similar attacks but have managed (so far) to escape the public spotlight.

Knowing the Types of Rootkits

Rootkits are classified in two basic ways — as *user-mode* or *kernel-mode* — depending on the scope of the effect and whether they exist only in memory or have written changes to disk that enable them to survive a reboot. Rootkits can reside at the level of user accounts (Ring 3) or at kernel level (Ring 0) — and they can be either persistent (able to survive a reboot) or non-persistent (that is, they disappear at reboot). Table 7-1 provides a basic summary of rootkit types.

Table 7-1 Summary of Rootkit Types

Type	*Privilege Level (Ring)*	*Scope of Action*	*Persistent or Non-persistent*
User-mode rootkit	Low (Ring 3)	Localized effect	Can be either
Kernel-mode rootkit	High (Ring 0)	System-wide (global) effect	Can be either

The following sections give you a closer look at user-mode versus kernel-mode rootkits, and at persistent versus non-persistent rootkits.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 155

User-mode versus kernel-mode rootkits

Rookits are classified by the mechanisms they employ to infect a computer — and by the *scope* of their action (that is, the extent of dirty work they can do). Scope is, in turn, determined by the privilege level a rootkit can obtain for itself. Higher privilege levels mean a greater scope.

Whatever their mode, here's what every rootkit knows: All programs communicate with each other through function calls. Depending on a program's privilege level, it can operate either by making function calls directly to the kernel (if it's at Ring 0) or from a more restricted level of privilege (at Ring 3, the level accorded to user programs). Privilege level determines whether a rootkit operates in user mode or kernel mode (more about that in the next section). Presumably all rootkits would love to reside at Ring 0 — but they'll take whatever privilege level they can get (which really translates into whatever their author is able to code) to operate effectively.

User-level rootkits

User-level programs, orbiting out there at the lower levels of privilege, must use the *application programming interface* (API) to make requests for operating-system resources. These *system calls* go to the kernel indirectly, through user-level dynamic link libraries (DLLs). The DLLs translate user-level API calls into calls the kernel can understand. In effect, user processes must operate through a middleman to talk to the kernel.

Now, a user-level program isn't self-sufficient. Some of its basic needs are provided by the operating-system kernel instead of the program itself — for example, such functions as reading or writing to disk, displaying a window, or printing a document. A user program must make *system calls* to the kernel to request that it perform those actions. For example, if a user clicks an OK button, Windows translates that input into a system call — and then asks the system to act upon it. The language that Windows provides to accomplish this communication between kernel and user program is called the *application program interface* (API). To operate more efficiently, each user program builds its own its unique table — which eventually contains the addresses of all APIs or system functions that it needs the kernel to perform to complete it execution. This table is called the *Import Address Table* (IAT), which makes another (more detailed) appearance later in the chapter. The IAT is part and parcel of a program's executable (EXE) file (or image) that is loaded into memory.

The indirect calls that user-level programs must use limit the effect of user-mode rootkits (which can run only within the confines of another application, or as separate user-level programs). Unable to interact with the kernel directly, limited to what a user program can do, user-mode rootkits exert only a localized effect. They are kept at arm's length away from the kernel, in

13_917106 ch07.qxp 12/21/06 12:04 AM Page 156

user program address space, and must make requests for kernel services through system calls (see the "Hooking to Hide" section later in this chapter), user-mode rootkits can be detected by some conventional security programs, (and nearly all rootkit detectors) that run in kernel mode. Despite that disadvantage, user-mode rootkits are easier to program than their kernel-mode cousins — and are therefore less likely to crash or hang the operating system, so they can operate as long as a kernel-mode rootkit detector isn't looking for them. The trade-off (from the bad guys' point of view) is that a user-mode rootkit is less powerful, and a lot of work has to be done to achieve the desired global effect. To be effective, a user-mode rootkit must find a way to alter the tables of *every* executing user program.

Kernel-level rootkits

The operating system *kernel* is the software equivalent of the central processing unit — the "brain" of the operating system, and its most basic component. The kernel provides the buck-stops-here level of control and functionality for all programs that run on a computer. It maintains and manages many vital system resources and functions — such as memory, security information, and process scheduling — and facilitates communication between software and hardware. The kernel therefore has a global scope — access to the entire operating system, hardware and all applications. All programs must interact with the kernel in some way — and if a rootkit gets control there, it's in the driver's seat.

Unlike user-mode rootkits that operate at Ring 3, kernel-mode rootkits that operate at Ring 0 interact with the kernel more directly — by intercepting *native* (or *kernel-level*) APIs. The kernel has global access to every nook and cranny of the operating system — so inside the kernel is a comfortable place for a rootkit to be. From there, it can access any memory location and any hardware (worse) it can substitute its own code for that of the kernel's or modify the critical data structures that the kernel relies on to keep track of its activities. Kernel-level rookits can, therefore, exert a global impact on the entire system — which makes them potentially a lot more dangerous and insidious.

Kernel-mode rootkits install a device driver to obtain access to kernel-level privileges. Once ensconced, the rootkit driver redirects system function calls so its own code is executed instead of kernel code. (For more about how this process works, see the "Hooking to Hide" section later in this chapter.)

Luckily, it's inherently difficult to implement a kernel-mode rootkit without upsetting the delicate balance of the system kernel. For that reason, kernel-mode rootkits often reveal their presence by causing system instabilities or computer crashes (especially if the rootkit has been sloppily programmed).

13_917106 ch07.qxp 12/21/06 12:04 AM Page 157

Persistent versus non-persistent rootkits

One other essential distinction between rootkits is whether they can survive a reboot. If they can, they're called *persistent*; if they can't, they're *non-persistent.* A closer look at each of these types is the next order of business.

Persistent rootkits

In order to survive a reboot and attain *persistent* status, a rootkit must physically alter the contents of the hard drive. A persistent rootkit does this bit of nasty magic by residing on the disk and adding an autostart entry to the Registry — that way it's loaded into memory and executed automatically every time the computer is started. Although these physical changes to the disk do present opportunities for detection, unfortunately many rootkits are still undetectable to traditional malware scanners and system utilities. What escapes attempted analysis is a range of hidden clues (more about those in a minute) that would alert most dedicated rootkit scanners to a rootkit's presence.

Non-persistent rootkits

Non-persistent rootkits present even fewer detection opportunities to scanners because they exist in memory only — and disappear if there's a reboot. Programs that merely scan physical storage media have no chance of detecting a non-persistent rootkit — after all, it leaves no part of itself behind. True, non-persistent rootkits may seem less threatening to those of us who reboot regularly; restarting the computer should effectively eliminate the threat. Well, it does — but there's a catch: If the infected computer is a network server — connected to hundreds of client computers — rebooting is typically a much rarer event. Memory-resident rootkits take advantage of this fact, and stick around for as long as they can. Consequently a network may harbor an undetected infection for an extended period of time. The very fact that non-persistent rootkits leave no physical clues makes them a lot harder to detect.

Hooking to Hide

One technique a rootkit employs to alter the normal execution path of the operating system is known as *hooking* — intercepting system function calls and adjusting their results to deny the presence of the rootkit. Hooking diverts normal program flow, to rootkit-supplied functions instead of legitimate system functions. The Windows operating system was designed to be

13_917106 ch07.qxp 12/21/06 12:04 AM Page 158

very adaptable and flexible — which probably seemed like a good idea at the time, but as such provides many possibilities for rootkits to "hook up" to system resources.

In the following sections we'll discuss how rootkits hide by attempting to make themselves invisible to the target operating system — and we'll give you the skinny on the various types of hooking.

How hooking works

Both user-mode and kernel-mode rootkits employ hooking techniques to filter the results returned by the operating system and camouflage their existence. In effect, they get Windows to lie for them and perpetuate the illusion that you have a clean system. Rootkit hidden files Registry entries, processes, and ports — will be invisible to most system-analysis and scanning programs. That's because such programs rely on the data provided to them by the operating system to produce their results. For example, if you use (say) the Task Manager to display a list of active processes, the rootkit processes will be excluded from what you see. The same will happen if you try to locate rootkit files and folders using Windows Explorer. Likewise, Regedit won't detect any of the rootkit-installed Registry keys; Netstat won't see any rootkit ports. It's like being in an alternate universe where none of your trusty tools can actually be trusted; Windows blindly becomes a partner to a rootkit's subversion techniques.

The Windows operating system uses many data structures (tables) to store and keep track of critical system information. These tables can be hooked, replaced, and generally corrupted by a rootkit. User-mode and kernel-mode rootkits both use hooking, but what they're permitted to hook — and the tricks they use — are defined by the privileges accorded to them:

- Some rootkits sport a type of built-in enemy radar. They temporarily drop their hooks when they detect a rootkit-scanning program poking around in their vicinity. Then no hooking abnormalities show up in the scan results — and the rootkit remains undetected. They may possess an uncanny intelligence — one of the newer rootkits completely disables all tools capable of detecting it, and even blocks Web access to security sites that provide removal tools.
- User-mode rootkits can only hook data structures (tables) in a user program's address space (the IAT and EAT), so the scope of their effect is limited. They may also insert jump instructions into user-level APIs; the result is to redirect system calls to the rootkit's replacement functions.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 159

- Kernel-mode rootkits hook tables (data structures) used by the kernel — such as the SSST and the IDT — so a kernel-mode rootkit's effect is system-wide or global. (These tables and the various types of hooking are described in the next sections.)

Knowing the types of hooks

Two types of hooks exist, including privileged hooks and unprivileged hooks. User-level rootkits use unprivileged hooks exclusively while kernel-level rootkits use the more efficient privileged hooks. Here's a list that includes the various types of hooking associated with each:

- **Unprivileged hooks:** User-mode rootkits run within the confines of user program address space and use *unprivileged hooks* to redirect program flow. The hooks are called unprivileged because they do not have Ring 0 privileges (they have Ring 3 privileges only); the rootkit exerts its influence by operating within the memory-address space that has been allocated to another program.
- **API hooking:** User-mode rootkits intercept the API (Application Program Interface) calls that user programs use to communicate with the kernel using unprivileged hooks. They hook (modify the addresses of) APIs in the Import address table (IAT) of user processes, so they point to rootkit functions instead of the Windows API functions.

  Most rootkits (both user- and kernel-mode) use API hooking to make sure the operating system returns only filtered results, which omit any indication of the rootkit or its payload.

  User-mode rootkits can only modify tables that belong to user programs. Each user program has its own unique IAT that it assembles to indicate the functions it needs the kernel to perform; to be effective, a user-mode rootkit must find a way to alter the tables of every executing user program — and also monitor for any new programs that start up so it can hook the relevant APIs used by those programs as well. A rootkit doesn't need to hook every API, only those that may expose it. For example, it may hook the APIs that Task Manager uses to display its list of active processes, or the APIs that are used by Windows Explorer to display files and folders. Because it has to hook these same APIs for every active process, user-mode rootkits are much less efficient than kernel-mode rootkits. Kernel-mode rootkits can achieve the same effect by hooking a single structure that all user programs use.
- **DLL injection:** User-mode rootkits use a method called DLL (Dynamic Link Library) injection to implement unprivileged hooks. To get a better handle on what DLL injection is, and how it's used in unprivileged hooking, check out the next section to get a closer look at what a DLL is.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 160

DLLs and the rootkits that love them

A DLL is a type of small program — often it is a building block for other programs — called an *object module*. Some DLLs are application-specific and reside in the application's program folder; other DLLs are provided by Windows itself — and can be shared by all programs. Windows maintains its own collection of small shared program modules used to perform identical system functions for all applications. These interchangeable DLLs reside in the Windows system folder.

A larger program calls a Windows DLL program when it needs to perform the function that the DLL provides. For example, the DLL may provide a print function for a word-processing program, or allow you to write to (or read from) your hard drive. Windows DLL files are not part of the program's executable file; instead, they're linked in at runtime and loaded into RAM as needed. Utilizing shared DLLs this way conserves system resources; executable files can take up less space because they can "borrow" some of their basic functionality from DLLs that aren't loaded into memory until they're needed. DLLs often take the form of drivers — and drivers are a favorite tool of rootkits (as we shall see later in the chapter).

If an application has to use a particular version of a DLL, the application installs that version of the DLL in its own program folder to avoid conflict with the version provided in the Windows system folder. The application then decides which functions it must import from the DLLs that are available to it at runtime.

Static and dynamic DLLs as rootkit targets

Static DLLs, as opposed to dynamic DLLs, are application-specific; because they aren't used by other programs, they can be linked or compiled directly into the application's executable file. Because they are contained within a program's executable file when it is loaded into memory, unlike dynamic DLLs static DLLs are not vulnerable to rootkit modification — unless the rootkit replaces the entire DLL file on disk — but that method is easily detectable and rarely used nowadays.

Windows uses a system called Windows File Protection (WFP) to ensure that the DLL files in its system folder are not overwritten by different versions supplied by application-program DLLs. This means you cannot replace the system version with the user-program version, because Windows will immediately overwrite the new third-party DLL with a bona fide version that it maintains in a system backup folder.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 161

The prize DLLs at kernel and user levels

The `Kernel32.dll` and `User32.dll` files are crucial; they supply system functionality to user programs. For example, `Kernel32.dll` contains APIs that manage memory, handle program interrupts, and take care of input/output tasks. (Lots of potential mischief to be done there.)

You may become aware of the `Kernel32.dll` operation when an `invalid page fault error` or `kernel error` message is generated. The Windows dynamic link libraries are loaded into a reserved area of memory at system startup. Because user programs are not allowed to access the kernel memory space directly; they have to use API calls when they need to perform a function that only the operating system can provide. This difference in access maintains the concept of privilege — user programs have no direct contact with the kernel.

`Kernel32.dll`, `User32.dll`, `Gdi32.dll`, `Ws2_32.dll`, `Advapi32.dll`, `Wininet.dll`, `Rasapi32.dll`, `Urlmon.dll`, and `Netapi32.dll` all contain user-level APIs that can be called to execute specific functions for user-level programs. Because the APIs must be retrieved from user-level DLLs that are not part of the application, these DLLs and their APIs are considered to be "imported." The functions within these imported user DLLs will in turn call the kernel or native APIs required to process a given user-program request. `Ntdll.dll` is a bit of an oddball; it provides a native API interface directly to user-mode programs, even though `Ntdll.dll` still resides in the portion of memory reserved for user programs. User-mode calls to `Ntdll.dll` APIs cause `Ntdll.dll` to execute a SYSENTER (analogous to generating an interrupt in older versions of Windows) which immediately transfers control to the kernel. It is in the kernel that the APIs are processed by executing the appropriate system services — so `Ntdll.dll` basically functions as a user program's gateway to the kernel.

An application maintains a pointer (which is just an address) to each imported user-level API it needs within a special data structure called the Import Address Table (IAT). The IAT enables an application program to locate and execute a system function whenever it needs it, and by handling it this way, Windows makes it operations more streamlined and efficient. The IAT may contain entries that point to any of the DLLs we've listed here. An application assembles its IAT before runtime when it knows the name of the APIs it needs — but it doesn't yet know their addresses. What it does provide are the names of the APIs that it needs, along with the names of the parent DLLs that supply them. Windows uses that information to load the required DLLs, and then it is Windows that computes and fills in the program's IAT with the correct API addresses.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 162

More on DLLs

In reality (say, program executables), each user DLL has an IAT for any APIs it may be importing from other DLLs such as `ntdll.dll`. Windows checks the IAT of each user-level DLL to see if it's importing any functions from any other DLL. If it is, then Windows will load that DLL and overwrite the IAT of the calling DLL with the actual addresses of the APIs in the called DLL, using the EAT of the called DLL. If called DLL is `ntdll.dll`, it will already be loaded into memory.

A rootkit is able to fool Windows by overwriting the address information in a program's IAT with address information that points to its own code, thus mimicking a variation of this last step that Windows has already completed. It accomplishes this feat in the following way: A user-mode rootkit substitutes its own APIs for the normally imported ones by injecting its own DLL into the program's address space. The rootkit can then selectively hook the IAT by replacing legitimate pointers in the IAT with pointers to its own functions. When a user program calls the one of these hooked APIs, execution is redirected to the rootkit's replacement function (located in the rootkit's *injected DLL*, so named because the rootkit injects it into a user programs address space). The rootkit need not hook every API, but only those APIs that are capable of exposing it.

In actual practice, there is another table called the Export Address Table (EAT) that presents another user-mode API hooking opportunity. Unlike the IAT, which is located within a user program's executable file, the EAT is located within the imported DLL itself. An imported DLL's EAT stores pointers to the functions that are exported to user applications already loaded into memory. These entries correspond to those in the IAT of the user program, except for one difference — they contain actual address information for the APIs the application program needs to import. The EAT information in the DLL is used by Windows to fill in the user program's IAT with real addresses (as opposed to just names). The EAT is vulnerable to rootkit hooking, and if the EAT is hooked, the same address information will be duplicated in the user program's IAT because that's what the IAT information is derived from.

To help you pull all this information together we'll first summarize the sequence of actions that take place during *normal* program execution:

1. **When an application program is executed, Windows loads it into memory.**

13_917106 ch07.qxp 12/21/06 12:04 AM Page 163

2. **Windows checks to see if the IAT of the program is calling any APIs.**
3. **If it is, Windows loads the DLLs specified in the program's IAT.**
4. **Windows then computes the real API addresses using the information stored in the EAT of each imported DLL.**
5. **Windows overwrites the program's IAT with the actual API addresses.**

Now we'll look at a practical example (using the FindNextFile API) that shows how a rootkit might use EAT hooking to make sure its files stay hidden:

1. **First, the rootkit scans memory for DLLs that export the FindNextFile API.**
2. **Because FindNextFile is in** `Kernel32.dll`**, this really means it scans memory for all instances of** `Kernel32.dll`**.**
3. **The in-memory** `Kernel32.dll's` **EAT will contain the address of FindNextFile and the rootkit will overwrite that address with the address of a replacement function located inside the rootkit's injected DLL (which is already loaded by the rootkit).**

The influence of a user-mode rootkit's *unprivileged* hooking mechanism is localized. That's because a user-mode rootkit can only modify the data structures available to user programs — and they must inject code into each new process that starts in order to hook any APIs that might reveal its presence. Even though all user-level APIs eventually get converted to native (kernel) APIs, the rootkit can only achieve it goals using level of indirection — and that restricts the scope of unprivileged hooking.

Kernel-mode rootkits, however, overcome this restriction by using *privileged* hooks (more about those a little later in the chapter) to directly modify the data structures that the kernel accesses. The reason is simple: Kernel data structures are used system-wide — so privileged hooking allows kernel-mode rootkits to exert a global effect.

Inline Hooking: A more devious variation on a theme

Beside IAT and EAT hooking, there is another form of hooking used by both user-mode rootkits and kernel-mode rootkits: inline function hooking. User-mode and kernel-mode inline function hooking differ only in regard to what is hooked — user-mode rootkits inline hook APIs imported from user mode DLLs, while kernel-mode rootkits inline hook the native APIs functions that reside in kernel space (see the "Kernel Inline Hooking" section later in this chapter).

13_917106 ch07.qxp 12/21/06 12:04 AM Page 164

An example of DLL injection at work

Here's where we illustrate how a rootkit might use DLL injection and API hooks so it can fake out both the operating system and the user.

Suppose a user invokes a search program to see whether a particular file or folder is present on the local hard drive. The search program uses a couple of Kernel32 APIs — `FindFirstFile` and `FindNextFile` — to conduct its search. Now, suppose the search program is rummaging around in a folder that contains a rootkit file. The rootkit wishes to remain hidden, so it injects a DLL into the address space of the file-search program, and replaces (patches) the `kernel32.dll` addresses with addresses contained within its own injected DLL. That way, when the `FindFirstFile` and `FindNextFile` APIs are called, the execution is diverted to the rootkit-supplied functions in the injected DLL instead of the legitimate `kernel32.dll` functions.

The malicious code lurking in the rootkit DLL for these APIs ensures that the rootkit file is not reported and remains hidden. One of the methods a rootkit can use to accomplish that is by having the replacement function simply call the `FindFirstFile` and `FindNextFile` APIs once again, and then pass control back to `kernel32.dll`. This makes the API skip the rootkit file and jump to the next file, so any search results returned will exclude the rootkit.

Inline user hooks overwrite the actual API function code implemented by imported DLLS (not just a pointer), leaving the IAT and EAT unaltered. Inside the hooked DLL, the first line (usually) of the actual code of an API is overwritten with a jump (JMP) instruction. The JMP jumps or redirects execution to a replacement function inside the rootkit DLL.

Some sneaky rootkit programmers have attempted to insert the JMP instruction further down in the API code, thereby defeating rootkit detectors that only examine the API's first instruction to establish whether an inline hook is present. Inline hooking is much more difficult to detect than API hooking.

A program called API HookCheck by SIG^2 can detect all types of user mode API hooking including inline function hooking. API HookCheck compiles a list of any API addresses from the IAT and EAT that lie outside the memory space of imported DLLs. The program then generates a log composed of these suspicious entries. ApiHookCheck looks for IAT and EAT patching, as well as inline function hooking. You can find more information (and a sample log) at this API Hook Check download link:

`www.security.org.sg/code/apihookcheck.html`

13_917106 ch07.qxp 12/21/06 12:04 AM Page 165

Thread injection

User-mode rootkits can also use a method called *thread injection* to gain entry into a user program's address space. Simply put, a program consists of a series of steps that are executed in sequential order to complete some predetermined goal. A program step may call for a process to be executed. Each individual process is composed of smaller functional units called *threads*. A rootkit can insert its own stowaway thread into a user process to ensure that its code gets executed along with that of the user program. The API used to do this is called CreateRemoteThread. This injection method requires careful monitoring of starting and stopped processes to properly hide the rootkit components.

AppInit_DLLs injection

Malware writers can use a relatively easy technique to inject code into nearly every user-mode process: *AppInit_DLLs injection* — thereby coming pretty close to achieving the scope of a kernel mode rootkit. Most user processes link to `User32.dll`, and when they do, any DLL(s) specified by the AppInit_DLLs Registry value are also loaded. The AppInit_DLLs is defined by the following Registry key:

```
HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Windows\\
          AppInit_DLLs
```

The Appinits_DLLs provides a very convenient method of introducing a malicious DLL that can affect almost every user process. All the bad guys' code has to do is change one Registry key value and create an infective DLL that is then injected into every running process that normally links to `User32.dll`.

A trojan that uses AppInit_DLL injection

Though not a bona fide rootkit, A Cool Web Search (CWS) variant was one of the first infections to utilize AppInit_DLL's injection in early 2003. The particular CWS variant involved was dubbed "the About:Blank" Hijacker; it used this technique to hide its code and continually reinfect users. The infection also installed a Browser Helper Object (BHO) — a small program that plugs into your Web browser and directs its behavior. The name of the hidden CWS DLL file in the AppInit_DLL's value was invisible to Regedit, and could only be "seen" by using an alternative Registry editor called Registrar Lite, or by examining the exported Windows Registry key. The fact that the DLL was difficult to see made it a lot harder to remove. How can you remove a file you don't know the name of? The CWS-hidden DLL also changed its name after several reboots or after botched removal attempts — so even if it was successfully identified, a reboot it might cause it to adopt a completely new name. This practice of using morphing file names has since been adopted by several rootkits, including Apropos, Gromozon, and other stubborn malware including the Vundo trojan (which has a rootkit variant).

13_917106 ch07.qxp 12/21/06 12:04 AM Page 166

Windows did not intend for the AppInit_DLLs feature to be used for malicious purposes. Some very useful programs such as antivirus software and software debuggers use AppInit_DLLs injection constructively. In fact, the AppInit_DLLs value was put to very good use by the security researcher, Ilfak Guilfanov, when he developed the WMF Metafile HotFix, which utilized AppInit_DLLs injection to patch the zero day WMF Metafile vulnerability before the Microsoft patch was released.

Privileged hooks

In Windows NT systems (Windows NT 3.1 and later), an application program can operate in kernel mode by using a kernel-level *device driver* — a small program (implemented by launching a Windows service) that gives user programs unrestricted access to memory, hardware, and CPU privileged instructions. To a rootkit, that kind of access is as irresistible as a pirate's treasure chest — but the only way to get it is to use *privileged hooks* — which operate much like unprivileged hooks (described earlier in this chapter), with one major difference: Privileged hooks alter the data structures or tables used by the kernel itself. Because all programs and processes need to access kernel functions, modifying a kernel data structure affects all running programs.

NT stands for *New Technology,* and the term *NT-based Windows systems* encompasses Microsoft's family of 32-bit multitasking Windows operating systems, the first of which was Windows NT 3.1 introduced in late 1992, Windows NT 3.*x* and 4.0 Workstation and Server versions, Windows 2000, Windows XP, Windows Server 2003, and now Windows Vista.

Installing drivers as rootkits

Installing a driver is a legal route that user programs take to get into the kernel, but it is also one that blackhat rootkit authors have maliciously exploited to infect computers with rootkits.

A driver doesn't have to actually represent a physical device (though many do); to a rootkit, it's a method of obtaining kernel-level privileges rather than a means to operate a specific piece of hardware such as a printer or mouse.

By definition, device drivers operate in kernel mode. They have privileges at Ring 0 — as high on the food chain of privileges as you can get — and that means only a privileged hook can bend the kernel's functions to suit a rootkit's purpose. So a kernel-mode rootkit gains control of the kernel by using a device driver that runs as a Windows service. You're probably familiar with many of the *services* that come preinstalled with the Windows

13_917106 ch07.qxp 12/21/06 12:04 AM Page 167

operating system. These processes run in the background and accomplish many tasks behind the scenes — such as Windows Automatic Updating and Event Logging.

To see a list of Windows services, you can access the Services Console by choosing Start ➪ Run, typing **services.msc** into the Open: field, and pressing Enter. You will notice that there are both Microsoft and application program services listed. Naturally, any fully functional rootkits that might exist in your system won't be listed; they've made sure that the Services Console will ignore them.

Why malware writers use device drivers

Malware writers have determined (accurately) that loading a kernel-mode driver into a system gives them nearly unlimited access. No wonder the preferred rootkit methodology is to load a device driver so the rootkit can intercept and redirect native API calls — and thereby exert a global impact on the entire system. A kernel-mode rootkit can overwrite the basic data structures that the operating system uses to assess the system status — so that those structures fail to report anything that may expose not only the rootkit but also its malware-related processes, files, directories, ports, and Registry entries. Result: The whole infected operation is hidden to a degree that user-mode rootkits rarely match. (Not to mention that user-mode rootkits can easily be ferreted out by rootkit detectors running in kernel mode.)

How installing a device driver allows easy kernel access

The Windows OS treats device drivers as services — and requires verification of digital signatures for device drivers to confirm that a driver is not only compatible with Windows but also hasn't changed since it was last tested. If either one of these conditions isn't met, Windows triggers a user alert — but Windows leaves it up to the users to decide how they want to handle an unsigned driver. For enhanced security, 64-bit versions of Windows Vista depart from this policy and prevents any unsigned device drivers from loading. This is one of the security features implemented in 64-bit Vista to keep rootkits out of the kernel.

Currently, rootkit coders can bypass Windows driver verification by writing a program that relies on low-level API calls to install their driver. This suppresses the alerts that are normally triggered when a program loads an unsigned driver. Pretty sneaky!

Developing a kernel device driver is accomplished by creating a user-mode application to complete that task. Microsoft actually supplies a software-development kit (SDK) to help legitimate programmers who are interested in coding kernel device drivers. Three consistencies make this process easier:

13_917106 ch07.qxp 12/21/06 12:04 AM Page 168

- Kernel device drivers always follow the same naming convention; they all have a `.SYS` file extension. For example, a device driver might have a name like `mydriver.sys`.
- A file with a `.DLL` extension differs from a `.SYS` file in only one respect: the specific DLL files it links to.
- SYS files always link to `ntoskrnl.exe` (short for *NT operating system kernel*) and often link to `hal.dll`; Win32 Portable Executable (PE) DLL and EXE files link to `Kernel32.dll` and `Ntdll.dll`. Win32 PEs may also link to other system drivers depending on the type of support that's required for the Windows platform being used (some examples: `Ndis.sys` provides network driver support and `Wmilib.sys` provides support for Windows Management Instrumentation).

Registering a driver

After a driver is coded, it must be registered as a system service and then started to complete the installation process. Of course, coding the driver source code is the really complicated part. If that's done improperly, it can crash the system — and the infamous *blue screen of death* (BSOD) is a telltale sign of the system that many kernel-mode-rootkit-infected computers experience. It stems from the difficulty involved in coding a kernel device driver.

More about drivers

If you're curious to learn more about the process of developing a kernel device driver, refer to these links:

- Windows NT Kernel Programming — Introduction to Device Drivers:

  `www.catch22.net/tuts/kernel1.asp`

- Instructions on creating DLLs:

  `www.thevbzone.com/l_dll.htm`

Hooking the System Service Descriptor Table (SSDT hooking)

Like user programs, the kernel also maintains its own set of tables that it uses to carry out system functions, interact with hardware devices, and manage its own resources. In fact, every system call from a user program (via a user-level API) is eventually translated into an entry in a data structure that the kernel uses to locate the functions it needs to perform — the System Service Descriptor Table (SSDT). The kernel is where Windows actually performs the work that completes user-program requests; the kernel uses the SSDT like the index in a book to efficiently find the instructions it must perform. Each entry in the SSDT points to a set of instructions that performs a requested user program function. The functions themselves are called system services, and they are located in the NT operating system kernel, aka `ntoskrnl.exe`.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 169

Now, kernel-mode rootkits alter kernel data structures or tables — and most of them hook the SSDT. Because all system calls eventually funnel down to functions that the SSDT points to, modifying the entries in this one kernel data structure enables a rootkit to affect *all* user programs that access it. It's a very powerful and efficient approach; — and if it's done correctly, the user should have no idea that the rootkit is there.

The SSDT stores the entry-level addresses of kernel APIs — and being able to modify the pointers in this table is like winning the lottery to a rootkit. No fiddling around with pointers to user-mode APIs here; kernel-level control is the goal.

A rootkit hooks the SSDT by overwriting the addresses in that table with pointers to rootkit functions, instead of legitimate system services — a process known as *kernel patching* or *native API hooking*. Then, when a call is made to execute a system service that the rootkit has patched, the service dispatcher retrieves and invokes the service that the patched pointer directs it to — a which just happens to be a rootkit function (located within the rootkit driver), of course. Because patching the SSDT affects all user processes that reference that structure, it is commonly referred to as installing a *system-wide hook* or a *global hook*.

Installing a global hook is not only the domain of rootkit programs but many security programs also go about their business by installing global hooks to intercept and monitor key kernel APIs — so it is important to identify the source of the global hook when attempting to decipher security program alerts that notify you of a global hook installation.

Though many kernel-mode rootkits function by hooking the pointers to native APIs in the SSDT, this is an older technique that is pretty much universally detected by anti-rootkit programs. So a more sophisticated technique, known as *direct kernel-object modification* (DKOM) has become the new, preferred methodology for advanced rootkits (more about that in the "Direct kernel-object manipulation" section later in the chapter).

There are over 200 Native APIs in `ntdll.dll` and most of them are undocumented. They all begin with an `Nt` prefix, but when these same APIs are exported, the names of the functions change — an `Nt` prefix is swapped for a `Zw` prefix— so each single `ntdll.dll` function is referred to by two names. For example, the native API in `ntdll.dll` that opens or creates a Registry key is called `NtCreateKey`, but when that same function is exported by `ntdll.dll`, it is called `ZwCreateKey`. The Metasploit (a Web site that provides resources to intrusion-testing researchers) maintains a list of the native API entries indexed by the SSDT (aka the System Call Table). You can find it at the following URL:

```
www.metasploit.com/users/opcode/syscalls.html
```

13_917106 ch07.qxp 12/21/06 12:04 AM Page 170

Kernel inline hooking

Both user-mode and kernel-mode rootkits can actually overwrite user-mode API and kernel-function code, respectively, via a technique called *inline function hooking*. We've already described how user-mode inline hooking works earlier in the chapter. Kernel-mode rootkits also use inline function hooking — but kernel-mode inline hooking involves altering the code that implements system services in the kernel (`Ntoskrnl.exe`). The rootkit hooks a system service by adding a jump instruction to the system service code (pointed to by the SSDT). When the jump instruction is executed, it immediately redirects processing to a rootkit-supplied function. What happens next varies, depending on what the rootkit code says to do — when the rootkit code is executed, it may duplicate the original legitimate kernel code but filter the results — or it may pass control back to the original legitimate kernel function, allow it to complete its work satisfactorily, and then bounce control back to the rootkit function, which then filters the results. Either way, the primary objective is to allow the function to process normally while excluding evidence of the rootkit. Blackhats tend to prefer this technique over hooking the SSDT or IDT (discussed next) because inline function hooking at the kernel level is harder to detect than SSDT hooking — but it's also harder to implement.

ON THE CD

There are a few rootkit detectors that can detect kernel inline function hooking, and two of the programs that are able to do that, namely Gmer and DarkSpy, are included on the CD. (They are discussed in Chapter 9 and in the Appendix.)

SYSENTER hooking

SYSENTER hooking is so named because it hooks the entry point to the SSDT — the crossover from user mode to kernel mode. By hooking this juncture, a rootkit can filter all system calls *without* patching the SSDT. A rootkit does this by overwriting the pointer to the System Service Dispatcher with a pointer to code in its own driver. This way, whenever a user-level API (and subsequent SYSENTER) is called, the rootkit decides whether it should direct execution to a replacement function (to stay hidden) or let the real system service in the kernel handle it. Probably, because so few anti-rootkits programs were able to detect it, SYSENTER hooking was recently used very successfully by a new rootkit dubbed pe386 (more about pe386 in Chapter 12).

Replacing the SSDT

Another interception technique is a bit more drastic: to actually *replace* the SSDT and redirect all system calls to the "new" SSDT, which is actually a copy of the real SSDT with additional pointers to new system services appended. When this approach is used by a rootkit, as you might expect, the new additions would reference rootkit services (in the rootkit driver) rather than legitimate kernel services.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 171

A variation of this technique is used in a beneficial way by some antivirus providers to monitor and intercept actions that could pose a threat to user systems.

Hooking the Interrupt Descriptor Table (IDT)

Kernel-mode rootkits can also modify the Interrupt Descriptor Table to alter the execution of programs. This technique works because an *interrupt* is generated when either a hardware device or program requests CPU time while another program is executing; they have to take turns. For example, if you've scheduled an anti-spyware scan for a specific time, a software interrupt may be issued when that time arrives. A printer may issue a hardware interrupt so it can execute a print job.

Naturally the OS has to keep track of what it's doing and what it's interrupting; the Interrupt Descriptor Table is a system data structure that contains pointers to *Interrupt service routines* (ISRs). Rootkits can replace the pointers to ISR with pointers to their own code.

When an interrupt is triggered, program flow is returned to the operating system, which then executes an appropriate Interrupt service routine — and we're back to services, a known rootkit target (as described earlier in this chapter). If the Interrupt Descriptor Table has been hooked and doctored, then an interrupt triggers and executes a rootkit routine instead of a normal ISR.

Using Even More Insidious Techniques to Hide Rootkits

Technology improves by its nature; rootkits are no exception. Besides hooking, kernel-mode rootkits have newer and stealthier alternative technology with which they can avoid detection — in particular, direct kernel-object manipulation (DKOM). The following sections describe this alternative approach to rootkit technology.

Direct kernel-object manipulation

Since kernel-mode hooking can be detected by several rootkit-detection programs, a newer — and stealthier — alternative to hooking is being used in the most advanced kernel-mode rootkits. This new technique is called *direct kernel-object manipulation* or DKOM. As we've discussed already, obtaining administrative privileges is paramount to a rootkit. Running processes inherit

13_917106 ch07.qxp 12/21/06 12:04 AM Page 172

their privileges from the user accounts they are associated with. DKOM can circumvent limited user privileges by selectively elevating the tokens (security objects that determine privilege) associated with rootkit processes (or with any process, for that matter) so they are equivalent to those of the administrator. That comes in handy if your aim is disabling a firewall, disabling anti-malware scanners, or installing a driver. DKOM even lets a rootkit process evade standard system monitors such as the Event Viewer by changing the group that process is identified with — a rootkit process can inconspicuously blend in by appearing belong to the system. No wonder the most sophisticated kernel rootkits forego hooking altogether and opt to use DKOM — or combine the two — to achieve their stealth.

DKOM enables kernel-mode rootkits to modify kernel structures that exist in memory — such as lists of active processes, of *threads* (individual process components), and of loaded drivers. Normally the kernel uses such lists to keep track of the information it must process — and they present additional opportunities for rootkit deception. Advanced rootkits can use DKOM to manipulate these lists, erasing the traces of untoward activity.

Here are some signature characteristics of DKOM:

- DKOM is different from hooking because the kernel data structures themselves are modified, not just table entries. Many advanced kernel-mode rootkits use this technique to hide active processes and drivers more effectively from rootkit-detection programs.
- DKOM *can* prevent you from seeing a suspicious rootkit process because you won't even know the rootkit process is executing. Processes can also be hidden by hooking APIs (as described earlier in the chapter), but even kernel hooking is a technique that most anti-rootkit programs can detect very easily (with the exception of inline function hooking). DKOM can very successfully hide rootkit processes from many rootkit detectors that are normally capable of detecting them.
- One of the objects manipulated by DKOM, is a data structure the kernel maintains called the *Process List*. The Process List is composed of a linked list of Executive Process Blocks or EPROCESS blocks. Each EPROCESS block represents an active process in memory. An EPROCESS block contains unique identifying information about its respective process, including a pointer to the previous process and a pointer to the next process, in the list. The pointers allow the list to be read out of sequence — by following the pointer to the "next process" in each EPROCESS block. A rootkit using DKOM can manipulate these pointers so a rootkit's process ERPROCESS block is unlinked from the list. This effectively makes Windows skip the rootkit process when assembling the list of active processes. Suddenly the rootkit process "isn't there."

13_917106 ch07.qxp 12/21/06 12:04 AM Page 173

Figure 7-1 illustrates how a rootkit uses DKOM to hide a process from Task Manager or even an anti-rootkit program incapable of detecting DKOM of the Process List. By rearranging the pointers in the Process List, the rootkit EPROCESS link is excluded from the list of active processes.

Figure 7-1: Using DKOM to hide a process.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 174

Now here's a strange twist that works in a rootkit's favor — as long as a process is already active, removing a process from the Process List doesn't prevent it from executing. (Devious, isn't it?) That's because another data structure — the process *thread* list — is the one that is actually consulted for process scheduling. Processes are executed by running the individual components that the processes are composed of — its threads — and a single process can contain multiple threads. However, the Process List is what Task Manager and other Process viewers refer to when they assemble the list of active processes. If a rootkit manipulates the Process List to exclude its own processes, then its processes may appear to be invisible — but they will still be executing behind the scenes.

Klister, a utility developed by Joanna Rutkowska, can detect rootkits that use DKOM to hide their processes from the Process List. Klister can be downloaded here:

```
www.invisiblethings.org/tools/klister-0.4.zip
```

Trojanized utilities

The earliest UNIX rootkits were much simpler than today's super-stealthy versions, and what their authors did was replace system files (utilities) with counterfeit copies of their own. These new (trojanized) utilities were designed to perform their normal functions while secretly hiding the rootkit. To use a parallel example relevant to Windows users, rootkits might replace Task Manager's executable file (`taskmgr.exe`) so that all active processes except those belonging to the rootkit were listed.

Direct replacements of system utilities are easily detectable (as we shall see in Chapter 9) and pretty easy to reverse by installing a new copy of the affected executable. Today, malware writers may try to replace an authentic file with their own, but this is difficult to do to system files because of Windows File Protection, which we discuss earlier in this chapter. Even if Windows File Protection could be bypassed, then an antivirus or anti-trojan should be able to pick up the bogus files in a routine scan. Another obstacle to replacing critical system files with a trojanized version is the difficulty of maintaining the original functionality of the replaced file. Since Windows is not an open-source operating system, all of its code is proprietary. It is not an easy task to write rewrite the code of an essential system file so it incorporates both normal *and* rootkit functionality — doing so might cause a system crash.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 175

Even so, user-mode rootkits may make changes to critical system files by replacing programs with trojanized utilities of their own, or having the files depend on trojan DLLs that run within the context of a system process. A process called *checksum analysis* can help you discover trojanized utilities; see Chapter 9 for more information on checksum analysis.

Looking into the Shady Future of Rootkits

Some rootkits are actually on the side of the white hats — for now, anyway — because they are original laboratory creations, often designed by rootkit researchers for beneficial purposes. They're categorized as non-public or (well, yeah) laboratory-based — which means they've been developed as research tools and used to gain insight into new techniques that future rootkits could possibly use to hide. Such proof-of-concept rootkits are used to develop tools or strategies to combat potential techniques before a malicious rootkit starts using them in the wild. Three non-public rootkits that fall under this classification are Shadow Walker, FUTo, and Subvirt. Here's a look at how these three rootkits incorporate new hiding techniques — and how they've taken rootkit detectors to a whole new level.

Hiding processes by doctoring the PspCidTable

An even newer method of process hiding has been built into the laboratory rootkit called FUTo. FUTo was christened so by its author, Peter Silberman, because it is the successor to the publicly available rootkit FU (or FU II, which became FUTo). Even though most of these laboratory rootkits were developed to challenge and improve anti-rootkit tools, many are available for download on the Web, which means tweaked versions may find their way into distributed malware. Consequently, a version of FU is used in the "Aim Virus" (a misnomer) rootkit, the Fanbot worm, and several spyware/adware programs — proving you don't have to be a kernel programmer to bind FU to a threat.

Besides hiding processes by DKOM of the Process List, FUTo alters the only data structure used by the operating system to keep track of both active processes *and* threads — PspCidTable. If a rootkit hides a process using DKOM of the Process List both IceSword and BlackLight are able to detect it, but neither can successfully detect FUTo's added method of process hiding.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 176

Until FUTo made an appearance, restoring the Process List to its original state was sufficient to reveal hidden rootkit processes. No more; FUTo introduced a new approach to hiding processes by modifying the PspCidTable; it also created a need for a detection utility that can target that technique. That tool is RAIDE, and we discuss it in Chapter 9.

DarkSpy and the GMER rootkit detection programs included on the CD are capable of detecting both FU's and FUTo's method of hiding processes and drivers.

A very simple and quick freeware utility called kproccheck can detect rootkits that hide their processes using both of FUTo's methods. Kproccheck is one of three very effective command line tools developed by the Security Information Integrity Group (SI^G), and we will be talking about another one of their programs (AntiHookExec) in Chapter 9. Kproccheck may be downloaded here:

```
www.security.org.sg/code/kproccheck.html
```

Hooking the virtual memory manager

A new laboratory creature uses virtualization to make itself invisible to any detection strategies employed from within the target computer. *Virtualization* is a general technique used to create a separate simulated environment (complete with operating system) within the host computer. A virtual machine is like a "machine within a machine," and it behaves and functions separately and independently. A program launched within a virtual machine is normally only aware of the virtual environment that surrounds it, but it views that world as the entire computer.

This new rootkit technique uses virtualization to install itself at a level below that of the host operating system. Because it sits at a level that is lower than the host computer's real operating system (effectively right above the hardware), it is able to see not only its own environment, but everything that happens on the computer. However, booting from an external operating system can successfully detect this virtual-machine-based rootkit and its associated malware because an external operating system is able to see the untainted view of the host machine, including the virtual-machine-based rootkit. Shadow Walker is a non-public, in-memory rootkit that works by hooking the Virtual Memory Manager (VMM). ShadowWalker is discussed in detail in Chapter 12.

13_917106 ch07.qxp 12/21/06 12:04 AM Page 177

Virtual-machine-based rootkits

Another advanced rootkit technique developed in the laboratory is the virtual-machine-based rootkit (VMBR). Microsoft and the University of Michigan recently coauthored a VMBR that is essentially undetectable by any method utilized from within the target computer. The project is labeled SubVirt, which is a moniker derived by combining portions of the words *subvert* and *virtual*. The term *subvert* is often used to describe a rootkit's ability to trick the operating system into believing it doesn't exist, which is why Greg Hogland and Jamie Butler decided to call their book *Rootkits: Subverting the Windows Kernel* — and also why we have tried to avoid using that word too often in this book.

SubVirt installs and runs a virtual machine monitor (VMM) in its own area of disk space that is totally undetectable and off-limits to the host operating system. This VMM behaves like a self-sufficient machine (complete with its own operating system) that controls and keeps tabs on what's going on the entire computer — hence the term *monitor*.

Yet, because it effectively operates at a level below the host kernel, SubVirt still has access to all the layers above it — but remains totally inaccessible to the host operating system. The rootkit also places the original host operating system inside a virtual machine. After the host operating system is isolated in this way, its security tools become totally ineffectual against the VMBR. That means any malware that the VMBR installs inside the virtual machine monitor — or any newly spawned Virtual Machines the VMBR creates to further compromise the infected computer — are also invisible to the host.

But even this creature has a weakness: The VMBR must find and exploit a host vulnerability that will enable it to alter the master boot sequence. The VMBR has to load the virtual machine *before* the host operating system loads, so it can sit at a level that underlies the host's operating system. The laboratory VMBR experiment assumes the presence of such host vulnerability (a premise that critics of SubVirt claim is quite difficult to implement).

In the lab, SubVirt successfully installed itself and ran four malicious services that targeted host resources. Since the SubVirt exists at a level in between the hardware and the host operating system, an appropriate countermeasure has been proposed: incorporating detection into a hardware chip that enables the host computer access to the virtual machine files.

An easier disinfectant approach would be to boot from an external operating system contained on an alternate medium (such as a CD-ROM or USB flash drive). The external operating system would offer an uncompromised view

13_917106 ch07.qxp 12/21/06 12:04 AM Page 178

of the *entire* target computer. As such, it would present an effective way to detect and remove malware — using conventional scanners resident on the external medium.

The SubVirt rootkit was successfully installed and run on a Linux platform using VMWare, and on a Windows platform using Microsoft's VirtualPC emulation software.

It is important to emphasize that a VMBR is not about exploiting virtual machine vulnerabilities. Using VMWare or VirtualPC does not make you a more likely target for a VMBR; it actually makes you less vulnerable to VMBR exploits (as discussed in Chapter 6). An operating system that's already running within a user-installed virtual machine would always operate at a level below a VMBR — which would render it less prone to exploit than the normal physical operating system. That's because the lowest layer can see everything above it — while remaining invisible to all the layers above it. Whatever entity establishes itself at the lowest layer — malware or the host operating system — has a distinct advantage and occupies a position of control.

Some critics have described virtual-machine-based rootkits as impractical to implement in the real world. One reason for that is that virtual machines place quite a drain on system resources — particularly memory, which would be difficult for a user to miss. The disk space required is also a consideration, since SubVirt comes with its own operating system in tow. Still, as hardware and memory advance, this may not always be the case. Complexity of installation is another factor that may be prohibitive. Even so, the SubVirt authors contend that their model presents a viable future threat. Without a doubt, virtual-machine-based rootkits would be extremely dangerous, were they to become a reality outside the lab.