The Windows Kernel

Now that we've seen how attackers have their way with the Linux kernel, as well as how we can stop them, we turn our attention to the Windows kernel. Given its widespread popularity on desktops and servers, the Windows operating system and its underlying kernel are a choice target for attack by the bad guys. In this section, we'll start by discussing what the Windows kernel is and going on an adventure looking for kernel artifacts, just like we did for Linux in the last section. After that, we'll see how attackers can invade and manipulate the Windows kernel. For this discussion, we'll focus on the Windows 2000 kernel, the most widely deployed professional version of Windows at the time of this writing. The Windows NT, XP, and 2003 kernels are quite similar to the Windows 2000 kernel, but include minor differences due to the evolution of the kernel over time. I'm very happy to point out that the techniques and tools we'll draw on during our Windows kernel adventure all function on Windows 2000, XP, and 2003. So, as we look at various Windows kernel artifacts, you should be able to follow along with your own machine if you use Windows 2000, XP, or 2003. The innards of Win9x (including Windows Me) differ radically, and won't be our focus in this chapter. So, without further adieu, grab your dusty old cowboy hat and bullwhip as we go on an archeological adventure in the Windows kernel.

Adventures in the Windows Kernel

> *Oh my God! It's full of stars!*
>
> —Dialogue from the movie *2001: A Space Odyssey,* 1968

As you'd certainly expect, the Windows kernel includes numerous components for interacting with and supporting user-mode processes. As we'll see, a lot of the concepts we covered in the Linux kernel have directly analogous ideas in the Windows kernel. After all, they are both operating system kernels, trying to achieve the same goal: servicing user-mode programs by sitting in between these processes and the hardware. The overall Windows kernel architecture is shown in Figure 8.21.

Figure 8.21. An overview of the Windows kernel and its relationship to vital user-mode components.

To get a feel for how all of these layers operate, let's start out at the top: user-mode processes, the programs you run on a day-to-day basis, such as your favorite word processor, a game, or

even an e-mail server. To interact with the operating system, a user-mode process makes function calls into various Win32 subsystem DLLs, roughly analogous to the system libraries we discussed earlier for Linux. When developers create programs to run on Windows, these Win32 function calls are the crucial interface into Windows itself, implementing the API into the Windows operating system. These DLLs include all kinds of capabilities, such as displaying information on the screen, opening files, or running other programs.

To encourage development of applications for Windows, Microsoft has provided a great deal of documentation about the function calls available in the Win32 subsystem DLLs. The Win32 DLLs are grouped into several different files, each with its own lump of code to accomplish certain tasks, including User32.dll, Gdi32.dll, Advapi32.dll, and Kernel32.dll. Yup… That's right. The file named Kernel32.dll is not the kernel. Instead, along with User32.dll, Gdi32.dll, and Advapi32.dll, it runs in user mode and provides an API to various user-mode applications for reading files, writing files, and performing other actions. It's called Kernel32.dll because it provides an API for user-mode programs to send requests to the kernel, but these requests don't go directly to the kernel. Instead, they must pass through Ntdll.dll first.

We should note that Windows supports other groups of subsystem DLLs beyond the Win32 set. Since its inception, Windows NT and its successors include subsystems for programs written for OS/2 (a venerable operating system championed by IBM years ago) and POSIX (a generic UNIX-like environment). The vast majority of Windows programs rely solely on the Win32 APIs, but these other subsystems are available to run older applications or for new programs to be built in those other programming environments.

So, most user processes make function calls directly into the Win32 DLLs. Each function call inside of Win32 can, in turn, do one of three things [19]. First, as shown in element A of Figure 8.22, for relatively simple requests that don't require kernel-level interaction with hardware or other processes, the Win32 function could just handle the request and send a response. An example function of this type is the *GetCurrentProcessId* function, which lets a process get its own process ID number from user space. No deeper level calls are required.

Figure 8.22. Three ways the Win32 DLLs handle requests from user-mode processes.

Another possibility for handling a function call from a user-mode application involves the Win32 DLL needing information from a very special user-mode process that is responsible for keeping the Win32 subsystem running. This type of interaction is illustrated as element B of Figure 8.22. The Csrss.exe process, which is an abbreviation for Client/Server Run-Time Subsystem, keeps the Win32 subsystem operating by invoking user processes and maintaining the state associated with each process. A user-mode process can ask Csrss.exe for information about itself or other processes without calling the kernel.

The third possibility for a Win32 function call is the most interesting for our purposes, and is shown as element C in Figure 8.22. The user-mode application could ask a Win32 DLL to take some action that requires invoking a kernel function. For example, the user-mode process could call the *ReadFile* or *WriteFile* function calls in a Win32 DLL. To interact with the hardware as required by these functions, we are clearly going to need to take a step downward toward the kernel. The highly documented Win32 DLL that developers utilize will map the *ReadFile* and *WriteFile* function calls into another piece of code, called Ntdll.dll, which is an internal and relatively undocumented API. The purpose of Ntdll.dll is to take the highly documented function calls of the Win32 API (like *ReadFile* and *WriteFile*), and convert them into the relatively obscure underlying function calls understood by the kernel (called *NtReadFile* and *NtWriteFile*, respectively).

Once the Ntdll.dll code maps the function calls, we need to make a transition from user mode to kernel mode, jumping through a call gate into the kernel. Using a mechanism we'll explore shortly, the Ntdll.dll code invokes kernel-level functionality called the Executive. The Executive, named for its high and mighty capabilities, serves numerous purposes, including making kernel function calls available to user mode, making various kernel-level data structures available to other kernel-level processing, and managing certain kernel state and global variables. The Executive is implemented inside of a critical file called Ntoskrnl.exe. When the Executive is invoked, it determines which piece of underlying kernel code is needed to handle the request, such as reading or writing a file. After determining which piece of kernel code is required to handle the request, the Executive transitions execution to another component of Ntoskrnl.exe. This bottom piece of Ntoskrnl.exe is called the kernel, even though the Executive itself runs in kernel mode and is implemented in Ntoskrnl.exe as well.

The code in the kernel now needs to interact with the hardware. In our *ReadFile* and *WriteFile* example, the kernel needs to interact with the hard drive. To accomplish this task, the kernel itself relies on yet another level of code, called the Hardware Abstraction Layer (HAL). Implemented in a file called HAL.dll, the purpose of this component is to make various different vendor hardware products look consistent to the kernel itself. By sending messages to HAL, the kernel can read from or write to the file. So, we've traversed the layers of this onion that is the Windows operating system: a user program can make function calls into the documented Win32 DLL, which calls Ntdll.dll, which invokes the Executive, which calls the kernel, which asks the HAL to do something, which interacts with the physical hardware. In the end, a user-mode process can read from or write to a file, or perform other interactions with the hardware.

There's one crucial component of this process that we need to zoom in on: the transition from user mode to kernel mode, that all-important and nifty call gate concept. How does Ntdll.dll make calls into the kernel, invoking the Executive? In a sense, we're doing the equivalent of making a system call in Linux. However, Windows documentation doesn't refer to this concept using the words *system call*. Instead, the Windows terminology for this transition is referred to as *system service dispatching*, a much more high-brow sounding phrase than the simple system call wording of the Linux world. The idea, however, is very much the same, as shown in Figure 8.23, which is really a zoomed-in view of our earlier Figure 8.22.

Figure 8.23. System service dispatching in Windows.

As with Linux, the transition between user mode and kernel mode occurs through the use of a CPU interrupt signal. For Windows, Ntdll.dll triggers interrupt number 0x2E on x86-compatible processors to invoke this transition. At this interrupt, a piece of code inside the Executive, called the system service dispatcher, needs to determine which kind of system service call is required of the kernel to invoke the appropriate underlying kernel code. Based on the information provided in the registers of the CPU at the time of the interrupt, the system service dispatcher looks in a table called the system service dispatch table. This table indicates where the appropriate system service code to handle the request is located in kernel memory. Sounds familiar, right? In essence, the system service dispatch table works a lot like the system call table in Linux. Execution flow is then transitioned to the appropriate kernel code. A good deal of this kernel code for implementing various system service calls is loaded into the kernel from a file called Win32k.sys, which implements much of the kernel mode functionality needed to service the user-mode Win32 API. In fact, about 200 kernel function calls are implemented in Ntoskrnl.exe itself, but more than 500 more function calls are loaded into the kernel during system boot from Win32k.sys. The Ntoskrnl.exe and Win32k.sys functions implement the required system service calls (e.g., reading or writing a file) by relying on even deeper code located in the HAL. All of the kernel data structures and code live at memory addresses starting at 0x80000000 up to 0xC0000000.

So, we've got a high-level view of how Windows user mode and kernel mode fit together. Now, let's see it all in action on a live system. If you'd like to follow along at home, boot your Windows 2000, XP, or 2003 system and log on to the box. As we explore the Windows kernel, it's important to note that Windows includes fewer built in features for looking at the kernel than does Linux. In Linux, all of the tools that we used as kernel archaeologists to look at artifacts were built into the operating system. With a default Windows installation, there aren't nearly as many good built-in tools for kernel analysis. Some people might feel that less information about the bowels of a running Windows kernel helps improve security, as the bad guys cannot as easily find or alter sensitive data structures in the kernel. In essence, this is a security-through-obscurity argument. Unfortunately, security through obscurity isn't a huge hurdle for the bad guys. It might slow them down a bit as they reverse-engineer the system, but it also could lull system administrators into a false sense of the security level they've really achieved. Many gifted reverse engineers (both noble researchers and evil bad guys) are quite adept, and have created all kinds of tools for peering inside the Windows kernel. Just because the operating system doesn't ship with such tools built in, good guys and bad guys alike commonly rely on various tools to analyze the kernel as they develop software on Windows. We'll use some of these tools ourselves shortly.

To analyze Windows kernel artifacts, we'll use some built-in tools and a couple of additional freely downloadable tools on our machines. I'll let you know when we get to the point where you need to install extra software to follow along. Initially, we'll just use the built-in tools that Microsoft provides with Windows.

First, take a look at running processes on your machine. Hit the Ctrl+Alt+Del keys, select Task Manager, and click the Processes tab, as I have done in Figure 8.24. Look at the top few processes, which are all associated with the kernel.

Figure 8.24. The Task Manager Process tab shows running processes.

Image Name	PID	CPU	CPU Time	Mem Usage
System Idle Process	0	98	0:36:54	16 K
System	8	00	0:00:09	212 K
smss.exe	152	00	0:00:00	344 K
csrss.exe	176	00	0:00:06	2,316 K
winlogon.exe	196	00	0:00:01	1,168 K
services.exe	224	00	0:00:01	3,536 K
lsass.exe	236	00	0:00:00	1,048 K
ibmpmsvc.exe	336	00	0:00:00	884 K
svchost.exe	392	00	0:00:00	2,964 K
svchost.exe	444	00	0:00:00	4,220 K
spoolsv.exe	492	00	0:00:00	2,168 K
regsvc.exe	560	00	0:00:00	812 K
MSTask.exe	576	00	0:00:00	1,800 K
vmware-authd.ex	620	00	0:00:00	1,528 K
taskmgr.exe	636	00	0:00:01	2,100 K
vmnetdhcp.exe	720	00	0:00:00	1,212 K
vmnat.exe	732	00	0:00:00	2,176 K
WinMgmt.exe	744	00	0:00:06	168 K
IEXPLORE.EXE	792	00	0:00:18	9,644 K

The first process you see in your listing is the System Idle Process with a process ID (PID) of zero. The System Idle Process, truth be told, isn't really a process at all. Instead, it's a place where the kernel accounts for CPU time that isn't being used by real processes to do work. Next in the list, we see the System process, which always has a PID of 8. Now, this one is very important, as it is used to aggregate information about all of the running threads in kernel mode, whether they are in Ntoskrnl.exe, Win32k.sys, or other kernel-mode code.

Moving down the list, we see the process called Smss.exe, also known as the Session Manager. This crucial item is the first user-mode process that runs on the machine, activated by the kernel during system boot. In a sense, it is analogous to the UNIX init daemon, as the Session Manager's job is to prepare user mode and to activate other user-mode processes while the machine starts up.

Smss.exe, in turn, invokes Csrss.exe (the process that manages the Win32 subsystem) and Winlogon.exe (which lets users log on to the machine). Smss.exe, Csrss.exe, and Winlogon.exe, as well as everything invoked after them, run in user mode. However, although they all run in user mode, all of these processes do invoke numerous system service calls inside the kernel as they run, especially Csrss.exe.

Next, let's get a feel for how often the system runs in kernel mode by looking at the performance view of the Windows Task Manager. Within the Task Manager, click the Performance tab, as I have done in Figure 8.25. Go to the View menu and select Show Kernel Times. The CPU Usage History screen will now display the amount of CPU time devoted to user-mode processes in green. The red line indicates how much CPU time is spent running in kernel mode. Move your mouse around and run an application or two to see how the relative amount of time in user and kernel mode changes as you perform various actions on your system. The Performance view in Windows Task Manager also shows you the number of kilobytes of memory the kernel is using.

Figure 8.25. The Task Manager Performance tab separates user-mode and kernel mode performance data.

So, the kernel is indeed there, and it's burning up some CPU cycles. So far, we've just looked at the kernel usage of the CPU aggregated into a big amorphous blob of kernel time, without regard to which processes are making demands on the kernel, causing it to burn that kernel time. Using the Performance tool built into Windows, we can separate the amount of kernel time burned by individual processes. To accomplish this, bring up the Performance tool, by going to Start ➤ Control Panel ➤ Administrative Tools ➤ Performance.

Inside the Performance tool, click Add (which looks like a plus "+" sign). In the middle of the screen, in the pull-down menu labeled "Performance Object:," select Process. Note that we want to select Process and not Processor. The Process view will let us look at the CPU activity of individual processes, whereas the Processor view lumps everything together. Now, in the "Select counters from list" box, click %Privileged Time and, while holding down the Ctrl key, also select %User Time. Finally, click a process to analyze. We'll start out by looking at the System process. I've illustrated the settings for this view in Figure 8.26.

Figure 8.26. Configuring Performance Monitor to look at individual processes, like the System process.

1. Select the Plus sign.
2. Select the "Process" Performance object.
3. Select the %Privileged Time and the %User Time.
4. Select the System process.

Now, click Add and then Close. The resulting graph is pretty tiny, so you might want to zoom in. To do so, right-click on the graph, select Properties, go to the Graph tab, and enter a Vertical Scale maximum of 5, instead of 100. Now, you'll see the relative amount of CPU time spent for that process on Privileged Time (which means that it's running in kernel mode) and User Time (which is, of course, user mode). To get some action going on the system, run your favorite word processor or a browser, which will burn some CPU cycles and cause system services dispatching to occur. You'll notice, as you might expect, that the System process spends all of its time in kernel mode. As we discussed earlier, that's because the System process is used to aggregate the time for all threads running in the kernel.

After looking at the System process, reconfigure the Performance tool to look at the privileged (i.e., kernel) time and user time associated with the Csrss and Explorer processes. Use the X icon to remove the previous graphs, and the + icon to add new ones. In Figure 8.27, I've shown my Performance tool views of the System, Csrss, and Explorer processes on my box.

Figure 8.27. Performance tool view of privileged (kernel) and user times for the System, Csrss, and Explorer processes.

Note that the Csrss.exe process spends the vast majority of its time in kernel mode, but every once in a while burns a little time in user mode. Although a user-mode process, Csrss.exe invokes kernel functionality through system service dispatching a lot. The Explorer process, as you might recall from Chapter 7, implements the Windows GUI, drawing all of those pretty pictures on your screen. The Explorer's performance view includes a fair amount of time in both user mode and kernel mode. It's important to note that Explorer really is a full user-mode process. However, the Performance tool displays the amount of kernel time that is spent by the kernel handling system service calls on behalf of the Explorer process. Therefore, we can see its normal user-mode time, as well as the time it takes the kernel to handle the requests of the Explorer process.

As we discussed earlier, there are a limited number of tools built into Windows for looking at kernel artifacts. We just looked at a few of them, but to get deeper into the kernel's activities, we need to install some additional tools on our Windows boxes. If you want to continue to follow along on our kernel adventure on your own system, please get a copy of the Process Explorer tool, written by Mark Russinovich, freely available at *www.sysinternals.com/ntw2k/freeware/procexp.shtml*. Additionally, snag yourself a copy of the no-cost Windows Dependency Walker tool, created by Steve P. Miller, at *www.dependencywalker.com/*. To follow along, go ahead and download each tool and install them by simply unzipping their contents into a directory on your hard drive. These tools are for reading information only and not altering it, so they shouldn't have a negative impact on your system.

After installing these tools, double-click Procexp.exe, or invoke it from the command line by just typing "procexp.exe" at a prompt in a directory where the tool resides. Process Explorer shows every running process on the machine, giving details about its status and the DLLs it relies on. It also displays the process hierarchy, showing the relationship of processes to each other by indicating the parent process, grandparent process, and so on for all running processes on the machine.

Based on the indentations you can see in Figure 8.28, the System process (which contains the various kernel threads) started the Smss.exe process (which, as we've discussed, is the first user-mode process that runs). Smss.exe, in turn, invoked the Csrss.exe and Winlogon.exe

processes. I can look for each DLL used by these processes, including Gdi32.dll, Ntdll.dll, and others. By right-clicking a process and selecting Properties, I can even view the kernel time, security parameters, and environment variables associated with each running process.

Figure 8.28. Using Process Explorer to look at kernel and user time, as well as DLLs loaded by every running process.

Process Explorer gets us pretty deep into the guts of the system, looking at the elaborate dance of running processes. However, I'd like to go deeper, getting a glimpse of the function calls made between various components of the system. With that information, we could trace requests through the onion-like layers of Windows. Unlike Process Explorer, which showed us running processes, the Dependency Walker tool opens executable files and DLLs and determines the function calls and other DLLs that glue different EXEs and DLLs together. With Dependency Walker, we're not looking at real-live running processes. Instead, we're checking out the relationships between the function calls and code stored in different executable and DLL files on our systems. One executable might call a given DLL, which, in turn, calls another DLL, which relies on yet other DLLs, right on down into the kernel. This information is tremendously useful in seeing how the kernel operates, as we can trace the relationships of user-mode processes, the various user-mode DLLs, Ntdll.dll, and the kernel itself. If you are following along, go ahead and run Dependency Walker by double-clicking it, or activating it from a command prompt by typing depends.exe in the directory where you unzipped the tool.

After invoking Dependency Walker, we need to select some application for which to analyze dependencies. Let's start out by opening up the simple editor Notepad, which has been built into Windows for years. On the File menu, select Open, and browse to your C:\Winnt\System32 directory (on Windows XP, you should look at C:\Windows\System32). Click Notepad.exe and then Open. You should see the view shown in Figure 8.29, which tells us that the Notepad executable depends on the Comdlg32, Shell32, Msvcrt, Advapi32, Kernel32, Gdi32, User32, and Winspool DLLs. That's quite a list of code, for little old Notepad!

Figure 8.29. Dependency Walker shows the dependencies of Notepad.exe.

Next, expand the Kernel32.dll item under Notepad.exe. As I've shown in Figure 8.29, we can see that Kernel32.dll depends on Ntdll.dll. Additionally, while Kernel32.dll is selected, the upper right-hand component of the window shows what function calls the parent (Notepad.exe) relies on from the selected DLL (Kernel32.dll). The column is labeled PI which stands for Parent Import. In particular, check out how Notepad.exe uses the *WriteFile* function provided by Kernel32.dll. In the middle of the screen, we can see all of the functions that Kernel32.dll offers up, whether Notepad.exe uses them or not. This primary column is titled E, for Export. The bottom of the window shows a laundry list of all DLLs that Notepad.exe relies on, without the nifty hierarchical relationships displayed at the top.

Now, let's take a step deeper down this rabbit hole. Under Kernel32.dll, select Ntdll.dll. Now, as illustrated in Figure 8.30, we can see the *NtWriteFile* function that Kernel32.dll imports from Ntdll.dll. The linkage between the higher level *WriteFile* and lower level *NtWriteFile* is not displayed, however, as such intricacies could only be determined by processing the code inside of Ntdll.dll, an activity beyond Dependency Walker's capabilities.

Figure 8.30. Looking at Ntdll.dll in Dependency Walker to see *NtWriteFile.*

Unfortunately, we cannot jump past Ntdll.dll in Dependency Walker because the transition between user mode and kernel mode doesn't occur by a traditional function call. Instead, the system services dispatcher is invoked by a CPU interrupt, something Dependency Walker just cannot walk across. So, to peek inside of the code that runs in kernel mode, we'll have to open up the Ntoskrnl.exe file itself, located at C:\Winnt\System32\Ntoskrnl.exe on Windows 2000 and C:\Windows\System32\Ntoskrnl.exe on Windows XP. In Figure 8.31, I've done just that.

Figure 8.31. Looking at the Ntoskrnl.exe program's dependencies, and the functions it makes available.

Here, we can see that the Ntoskrnl.exe file, (i.e., the kernel image on the hard drive itself) is dependent on HAL.dll, the HAL, and Bootvid.dll, a piece of code used to interface with the video drivers on the machine. Also, check out how Ntoskrnl.exe exports various functions to its parent (which as we discussed earlier, is Ntdll.dll). In particular, look at the *NtWriteFile* function that the kernel makes available. This is the function that Ntdll.dll will invoke through the system service dispatcher to write to a file.

At this point, we can go deeper into our kernel analysis by using a free tool that implements strace functionality for Windows NT, 2000, and XP. As you might recall from our earlier Linux discussion, strace shows a list of system calls made by a program as it is running. The folks at Bindview Corporation have released a free Windows version of strace that shows all system service dispatch calls made into the Windows kernel, available at *http://razor.bindview.com/tools/desc/strace_readme.html*. Although this Windows strace tool is extremely nifty, I caution you about using it. The Windows strace tool could make your system unstable, so you might want to avoid running it on anything but a test system that you can easily rebuild if it trashes your system. To give you a feel for how the Windows strace tool works, I've run it on my own system, displaying the system services invoked by the familiar Notepad file editor. As you'd expect, the Windows strace tool shows the invocation of the *NTWriteFile* function when I save a file using Notepad, as shown in Figure 8.32.

Figure 8.32. Strace on Windows shows the system services called by Notepad.

Now that we've got some feel for how user-mode code invokes functions inside kernel mode, there's one final area of the kernel we need to look at: device drivers. In Windows, an administrator can alter the functionality of the kernel by adding device drivers, which are chunks of kernel-mode code. Device drivers can add or even replace various system service

calls by altering the system service dispatch table or other kernel structures. In this regard, device drivers operate rather like Linux kernel modules. On Windows 2000, to view the installed device drivers, open your Control Panel, and select Administrative Tools. Now, open up Computer Management ➤ System Information ➤ Software Environment ➤ Drivers, to get the list shown in Figure 8.33. On Windows XP, select Start ➤ Control Panel ➤ Administrative Tools ➤ Computer Management ➤ Device Manager for a list of all devices and their drivers.

Figure 8.33. Looking at installed Windows device drivers on Windows 2000.

These installed device drivers include all kinds of goodies, such as code for extending my networking options with Internet Protocol Security (IPSec) and file system drivers so my system can read and write its hard drive. Each of these device drivers operates in kernel mode, and can access various kernel data structures, including, potentially, the system service dispatch table.

Methods for Manipulating the Windows Kernel

So, the Windows kernel and its associated APIs make up a complex beast, but they function appropriately for millions of users around the globe. Can you imagine anyone wanting to mess with such fine-tuned, complex harmonies? Well, of course, computer attackers want to manipulate the kernel to create kernel-mode RootKits. As you might expect with such complexity, there are numerous options for the bad guys in compromising a Windows kernel. Several kernel-mode RootKit projects are up and running on the Internet, but the most information-rich and prolific site dedicated to Windows RootKits is the *www.rootkit.com* Web site. Created and maintained by Greg Hoglund, *www.rootkit.com* is a virtual watering hole for developers of Windows RootKits to share code and ideas for improving their wares. The site features several discussion lists for different Windows RootKits, and offers up a few choice specimens for free download, including RootKits named Hacker Defender, HE4Hook, NT Rootkit, and GINA Trojan. To download any of the RootKits offered at *www.rootkit.com*, you'll need to register with the site for a free account. After receiving a user ID and password during the online registration process, anyone on the Internet can download and experiment with the user-mode and kernel-mode RootKits available at the site.

Interestingly, all five of the different Linux kernel manipulation tricks we discussed earlier in this chapter have direct analogies in the world of the Windows kernel. Namely, the bad guys could employ evil device drivers, alter a running kernel in memory, overwrite the kernel image on the hard drive, deploy a kernel on a virtual system to trick users, and try to run user-mode code at the kernel level. Now, each of these five elements on Windows machines is a possible avenue of attack, but the first two (employing evil device drivers and altering a running kernel in memory) are by far the most widely used. The other options are possible attack vectors,

which could become more popular in the future. Let's look at each of these attack types in more detail.

Evil Device Drivers

One of the first and most popular techniques for manipulating the Windows kernel involves inserting a malicious device driver into the system, which patches the kernel to alter system service call handling. Just as bad guys exploit Linux kernel modules to load malware inside the Linux kernel, they utilize very similar tricks on Windows. By loading a specialized device driver that alters specific system service calls associated with listing running processes, showing files and directories, and identifying TCP and UDP port usage, an attacker can very effectively alter the kernel to hide a backdoor on the machine, as illustrated in Figure 8.34.

Figure 8.34. Using a device driver to manipulate the Windows kernel.

So, an attacker can inject an evil device driver into the kernel to alter existing functionality and hide backdoor processes. Windows supports digital signatures on device drivers so that an administrator can verify the integrity of all drivers while they are first installed on the machine. However, with administrator privileges on the target machine, an attacker can easily install a device driver even without an appropriate signature. The system will warn the attacker that the device driver isn't signed by a trusted source, but the attacker can easily accept the warning and apply the malware driver.

However, once the device driver containing the attacker's code is inserted into the kernel, how does the attacker coax the Windows kernel into running the attacker's own code, instead of existing Windows kernel code for system service calls? Given the complexity of the Windows kernel, a huge variety of options are available, three of which are illustrated as elements A, B, and C of Figure 8.34. Each of these elements is really a form of API hooking, but this time inside the Windows kernel itself.

In element A, the attacker uses an evil device driver to simply overwrite existing kernel functionality, replacing the code inside the kernel with new code that will hide the attacker's actions by changing system service handling functionality. Alternatively, in element B, the attacker uses a device driver that implements various kernel functions, and then alters the system service dispatch table so that it points to the attacker's code instead of the existing

kernel functionality. Finally, an attacker could employ a technique called *interrupt hooking* to modify how the kernel handles CPU interrupts, as shown in element C. By changing the table associated with interrupt handling in the kernel, the attacker could redirect calls to the system service dispatcher to the attacker's own code, instead of the built-in kernel functionality. Using interrupt hooking, the attacker could grab all calls to the system service dispatcher, and pick and choose which functions to handle with normal kernel processing, and which to deal with using the bad guy's code.

For an example of a popular kernel-mode RootKit for Windows that mixes elements A and B from Figure 8.34, consider the Slanret/Krei tool, which is sometimes referred to as the Ierk8243.sys RootKit based on an embedded string and file name associated with the tool. Originally discovered in early 2003 on Windows 2000 and XP machines, Slanret/Krei actually consists of two pieces: the Slanret device driver and a remote access backdoor tool called Krei [20]. With administrator privileges, an attacker first loads the Slanret device driver onto the victim machine. In a mere 7 kilobytes of code, Slanret modifies the kernel so that it will lie about an attacker's hidden processes, files, registry keys, and TCP and UDP port numbers for any user-mode application that asks about them. What does Slanret hide in particular? It hides Krei, of course. After installing the device driver, the attacker loads the Krei backdoor, a 27-kilobyte user-mode application that listens on TCP port 449 and grants the attacker remote backdoor access to the victim machine. Of course, Slanret and Krei work hand in hand, in that Slanret masks all of Krei's actions.

Slanret is a pretty nasty kernel-mode RootKit, but its developers overlooked one important aspect. The Slanret device driver doesn't hide itself in the list of device drivers. When installed, Slanret will show up in the device driver list under the name IPSEC Helper Services or Virtual Memory Manager. These names sound like reasonable drivers, perhaps fooling a user or administrator into thinking that this driver somehow supports IPSec or the virtual memory system of the machine [21]. Some variations of Slanret call their device driver Ierk8243.sys, a more confusing but less subtle name.

An alternative strain of the Slanret tool uses the same basic code, but listens on a different port and uses a different driver name. The so-called BackDoor-ALI RootKit borrows almost all of Slanret/Krei, but listens on TCP port 961 and uses a driver called P2.SYS PentiumII Processor Driver [22]. With that name, it sounds like a pretty reasonable driver, right? Actually, it's a nasty kernel-mode RootKit.

Although Slanret doesn't do a good job of hiding its own device driver, a more thorough kernel-mode RootKit device driver could certainly hide itself. By using API hooking to grab the system services used by the machine to display active drivers, an attacker could eliminate this piece of evidence to create an even stealthier tool. Be on the lookout for such nasties in the very near future.

Altering a Running Kernel in Memory

Instead of using a device driver, an attacker could directly patch the kernel in the memory of the victim machine, a technique first described in detail by Greg Hoglund [23]. To understand Hoglund's technique, as well as the work of those who built on it, we need to look at how memory is handled in Windows, specifically with regard to the CPU running in Ring 0 and Ring 3. On a Windows machine, the Global Descriptor Table (GDT) contains information about how memory is divided into various segments, allocated to user programs and the kernel itself. As we discussed earlier, all memory locations between 0x80000000 and 0xC0000000 are for use by the kernel, and, under normal circumstances, can't be touched by user-mode processes. The GDT stores data about how various memory segments are carved up, and the CPU ring that is required for a program to touch each part of memory. The segments defined by the GDT can overlap with each other. That is, the same range of memory addresses can simultaneously be in multiple segments in the GDT. As you'd no doubt expect, the default GDT says that to access memory locations between 0x80000000 and 0xC0000000, you need to be running in Ring 0. That's kernel territory.

Here's the rub. By using several tricks to alter memory, an attacker can add a new entry to the GDT, thereby describing a new, attacker-defined segment that maps to a memory range. This new entry won't overwrite existing GDT entries, but will add another entry that refers to the same memory range included in other lines of the GDT. Guess what the new entry says. Yup, the new GDT entry could map out a memory space starting at 0x00000000 and going to 0xFFFFFFFF. On a 32-bit architecture, that's the entire memory space. If you are going to give yourself access to memory, you might as well go for the whole enchilada. Of course, the new GDT entry allows someone running in Ring 3 to read from and write to this new overlapping segment. Bingo! By writing some machine language code that adds an entry to the GDT, the attacker can read and write kernel memory directly, as illustrated in Figure 8.35.

Figure 8.35. Adding an element to the GDT creates a new segment that can be accessed from Ring 3.

Hoglund's paper includes code for altering the GDT in this way, and then exploits this technique to patch the running kernel so that it disables all security checking features of the machine. When any user tries to access a given object, such as a file or registry key, the SeAccessCheck function of the kernel verifies that the user has been granted the rights to touch the given object. By overwriting a mere 4 bytes of the Windows NT kernel in memory, Hoglund's patch bypasses all security checks associated with accessing objects on the victim machine by changing the internal kernel function call to SeAccessCheck. Suddenly, by applying this patch, an attacker can access any file, user account, registry setting, or anything else on the victim machine, without any pesky interference from the kernel and its security controls. This little 4-byte patch demonstrates the power of being able to manipulate the running kernel's memory image. If I can read or write the kernel's memory, I can alter its code to shut off security. Alternatively, I could intercept system service calls and implement code for all of the fancy hiding techniques we saw with evil device drivers.

Building on some of the concepts in Hoglund's paper and introducing additional ideas, a developer named Crazylord released another Windows kernel manipulation paper that delves deeper into the kernel [24]. Crazylord's technique involves utilizing an object in the Windows kernel called \Device\PhysicalMemory. As you might expect with that name, this object contains a representation of all physical memory on the Windows system, both user and kernel memory. Microsoft included this object inside the Windows kernel so that the kernel could track and help control memory use on the box. To look at this interesting object, Mark Russinovich released a free tool called PhysMem, available at *www.sysinternals.com*, that shows the contents of this device. Starting with Hoglund's ideas and the PhysMem tool, Crazylord implemented code that gives an attacker the ability to view, search, and alter any memory on the victim machine, including kernel memory. In essence, Crazylord's project provides a view of Windows kernel memory much like /dev/kmem in Linux. And, of course, Crazylord's project allows for manipulating this memory in much the same way as the read kernel memory (rkm) and write kernel memory (wkm) functions exploited by Sd and Devik on Linux. So, now we have /dev/kmem-like attacks on a Windows machine.

Using these techniques, an attacker can manipulate the kernel and change system service

functionality, thereby hiding files, processes, registry keys, and any other aspect of the system from an administrator. Although Crazylord's article didn't include a RootKit, his Windows /dev/kmem technique offers a starting point for other attackers to create kernel-mode RootKits without the use of device drivers.

Building on his earlier techniques, Hoglund released a tool called NT RootKit. Don't be thrown by the name, however. This tool will run on Windows NT, 2000, and XP. By the time you read this, a Windows 2003-compatible version might have been released. The NT RootKit includes several kernel-mode RootKit features, including file, process, and registry key hiding. It can also perform execution redirection for any user-mode executable process on the machine. Some versions also include a built-in keystroke logger, which records everything typed at the keyboard inside a hidden file for the attacker.

Configuration of the NT RootKit couldn't be much easier. Any file, registry key, or process with a name that starts with _root_ will be automatically hidden. So, the bad guy can just name all of the malicious stuff loaded on to the victim machine appropriately, and it disappears.

The NT RootKit also implements a form of the cone of silence concept we saw earlier with the Linux KIS Tool. If a running process has a name that starts with _root_, it is, of course hidden, but any hidden process is able to see hidden files, processes, and registry keys. Therefore, an attacker could make a copy of the Windows command shell (Cmd.exe), prepending _root_ to its name. Whenever _root_cmd.exe is executed, the resulting command shell will not only be invisible, but it will also have the ability to see any of the hidden items on the machine. Similarly, a version of the Windows Task Manager (Taskmgr.exe) or the Registry Editor (Regedit.exe or Regedt32.exe) with _root_ prepended to its name will be able to see hidden processes and registry keys, respectively.

Patching the Kernel on the Hard Drive

Instead of patching the Windows kernel in memory, an attacker could also alter the kernel image file on the hard drive, replacing functionality inside of Ntoskrnl.exe with modified software that provides a backdoor and hides an attacker's presence on the machine. Now, an attacker cannot alter the Ntoskrnl.exe file by itself, because the integrity of this file is checked each time the system boots. During the boot process, a program called NTLDR verifies the integrity of Ntoskrnl.exe before the kernel is loaded into memory. If the Ntoskrnl.exe file has been altered, the NTLDR program displays a fearsome blue-screen-of-death message, indicating that the kernel itself is corrupt. The system boot never completes, and both the administrator and the attacker are unhappy. Believe me, it's extremely disconcerting to have your system tell you that your kernel is corrupt during a system boot!

To get around this difficulty, the bad guys manipulate both the NTLDR and the Ntoskrnl.exe files. Using a small patch to overwrite a few machine language instructions inside of NTLDR so that it skips its integrity check, the attackers can then freely alter Ntoskrnl.exe at will, as illustrated in Figure 8.36. In Step 1, the modified NTLDR file is copied to memory at the start of system boot. The NTLDR program had been altered to skip the integrity check of the Ntoskrnl.exe file. Therefore, in Step 2, the manipulated Ntoskrnl.exe file is loaded into memory, with all kinds of nasty surprises loaded inside.

Figure 8.36. Modifying NTLDR to skip the Ntoskrnl.exe integrity check, and then modifying Ntoskrnl.exe.

Although this technique hasn't yet been widely used to implement backdoor access and full-fledged RootKits, several viruses have used the technique over the past few years. In particular, the Bolzano and FunLove viruses from 1999 altered NTLDR and Ntoskrnl.exe [25]. Both viruses applied a small patch to the kernel file so that the SeAccessCheck security functionality was disabled, implementing in the kernel file the same basic attack that Hoglund applied to a running kernel's memory. With the security checking functionality disabled, the Bolzano and FunLove viruses could access and alter any objects on the infected machine. Although these viruses targeted just Windows NT and only disabled the SeAccessCheck function, a complete Windows RootKit could be implemented using similar tactics to alter system service calls inside Ntoskrnl.exe and Win32k.sys. To date, no mainstream Windows RootKit has employed such techniques, leaving it relegated to just a handful of rather obscure viruses for the time being.

Creating a Fake System Using a Virtual Machine

Earlier we saw how an attacker could employ UML to create a virtual Linux machine running on top of a compromised Linux system. Administrators and users would think they are logging into the real machine, but are instead logging into a guest operating system built on top of the real system owned by the attacker. A similar approach could be applied against a Windows environment as well. To run a virtual Windows machine, an attacker could install any one of several virtual machine environments that run on Windows, listed in Table 8.4.

Table 8.4. Virtual Machine Tools That Could Be Abused to Trick Users

Tool Name	Commercial/Free	Host Operating Systems Supported	Location
VMWare	Commercial	Linux and Windows	*www.vmware.com*
VirtualPC	Commercial	Windows, MacOS X, and OS/2	*www.connectix.com*
Plex86	Free	Linux	*http://plex86.sourceforge.net*
Bochs	Free	Linux, Windows, and MacOS X	*http://bochs.sourceforge.net*

One significant disadvantage for the attacker of using any of the tools listed in Table 8.4 involves the complexity and lack of transparency in the virtual machine initialization process on

Windows. Sure, the attacker could break into a Windows machine, install a virtual machine tool, build a virtual system that mimics the original machine, and then configure the entire mess to start up appropriately at boot using startup scripts. However, the boot process of the compromised host operating system and the activation of the virtual machine tool would likely be noticed by a system administrator. A similar hurdle is faced by the Linux attackers who employ UML. However, the UML startup script can be disguised so that it doesn't really show any activity to a user watching the boot process on the screen. Fooling an administrator or user sitting at the console of a machine that suddenly starts VMWare, VirtualPC, Plex86, or Bochs is a much more daunting task for the attacker. Each of the virtual machine tools listed in Table 8.4 displays significant amounts of information on the screen as it is activated. Therefore, although still a possibility, this virtual machine approach is less likely to be used on a Windows machine than on Linux.

Kernel Mode Windows? Maybe Someday…Soon

Earlier we discussed how an attacker could use the KML project to run in Ring 0 arbitrary programs designed for user mode, provided that the Linux kernel was built with the appropriate KML hooks. As of this writing, no one has created a full-fledged kernel-mode Windows tool that runs user-mode-style programs inside the Windows kernel. However, there is ongoing work moving in this direction.

In particular, the NT RootKit development team is extending the NT RootKit itself so that it can run any user-mode program inside the kernel. In particular, they are focused on running the Cmd.exe command shell from within kernel mode. That way, an attacker can get a shell prompt that has complete access to any kernel mode data structures, at the same time remaining hidden to all user-mode processes.

Generalizing such a tool beyond a command shell is an arduous task, as the developer has to carefully manage memory access inside the kernel to create a kernel-mode Windows tool. A user-mode program running in kernel mode could easily behave like a bull in a china shop, accidentally smashing critical data structures, rendering the system unstable or even crashing it. Still, in time, I expect to see a generalized kernel-mode Windows implementation that acts as a shield of protection around the bull (i.e., a user-mode program) inside the china shop of the Windows kernel. The shield doesn't protect the bull, mind you, but is instead designed to protect the china shop itself from accidental destruction. Of course, for a bad guy to use such a tool for manipulating the kernel, the shield of protection would need selective holes so the attacker could alter some aspects of the china shop without bringing the whole thing down. Stay tuned for more development on this front.

Defending the Windows Kernel

With Windows kernels exposed to similar types of attacks as the Linux kernel, we must carefully shore up the security of our Windows machines as well. Let's analyze the defenses against Windows kernel manipulation by stepping through the same three categories we discussed for Linux kernel mode attacks: prevention, detection, and response.

Prevention

As with most of the malware we've covered in this book, a crucial element of your defensive plan is to keep the bad guys off of your system by hardening the configuration and applying patches in a timely manner. Such defenses are just as important on Windows as they are on UNIX systems. In addition to these incredibly important base recommendations, though, you might want to consider another class of tools that can help prevent installation of kernel-mode RootKits: intrusion prevention systems (IPSs).

Frankly, I'm not a big fan of the terminology *intrusion prevention system*, as that name is so ambiguous, it could refer to a multitude of products, ranging from firewalls to smart card authentication tokens and more. However, due to various marketing initiatives, the IPS moniker

has stuck to a class of products that are installed on individual end systems to thwart various attacks used to break into the box. I don't like the name IPS, but I am a fan of the functionality offered by these tools. These IPS solutions limit the exposure of your system by locking out functionality often abused by attackers to obtain superuser privileges on a target system. Think of an IPS like a little shield surrounding various critical components of your system, watching and stopping suspicious activity associated with breaking into the box. These activities include some buffer overflow attacks, various race conditions, and suspicious system service calls.

Cisco's Security Agent (formerly known as Okena Storm Watch), Network Associates' Entercept, and Watchguard's ServerLock products are all examples of commercial IPS tools that run on a Windows platform. They offer a variety of protection strategies, but one of the most worthwhile capabilities of these tools involves limiting the system service calls that various applications can make on the machine. As you might recall from earlier in this chapter, the free Systrace tool offers such protection on Linux, FreeBSD, and MacOS X systems. The commercial IPS tools offer similar capabilities on Windows. By configuring the IPS to limit what system calls a given program (e.g., a Web server, mail server, or DNS server application) can make, the bad guys will have a far more difficult time compromising administrator privileges and installing RootKits. Also, some commercial IPS tools support operating systems besides just Windows. In particular, the Cisco Security Agent runs on Windows and Solaris. Entercept is available for Windows, Solaris, and HP-UX. Watchguard focuses on Windows and Solaris systems.

We should note that configuration and maintenance of these IPS tools is no small task in a production environment. You need to install the tool and carefully configure it so that it interoperates appropriately with the application mix on a given machine, allowing the functionality the application needs to run while locking out those functions that aren't required. In a sense, the tool has to be trained regarding normal activity for the machine so that it can spot and stop abnormal behavior. However, after configuring the IPS tool to support the given machine, you've added a significant extra measure of security to the box.

Detection

To detect a kernel-mode RootKit on Windows, many antivirus tools include signatures for dozens of kernel manipulating tools, such as Slanret and the NT RootKit. When an antivirus tool spots a kernel-mode RootKit on the hard drive by matching the contents of the file to one of its signatures, it will quarantine the file so that it cannot be executed and installed. Therefore, a widely deployed and up-to-date antivirus infrastructure, as we first discussed in Chapter 2, supports both the prevention and detection of Windows kernel-mode RootKits.

These antivirus signatures work best before the kernel modifying attack tool is installed, so proactive deployment of antivirus tools is now more important than ever. After the kernel-mode RootKit is installed, the antivirus tool has less of a chance to detect it, because the hiding capabilities of the RootKit could help mask it from the antivirus program. However, many kernel-mode RootKits on Windows can be spotted by an antivirus tool even *after* the RootKit is installed, due to holes in the RootKit's hiding mechanisms. For example, Slanret leaves its device driver name exposed, a telltale sign that can be detected by an antivirus tool even after the kernel manipulation is applied.

Although antivirus solutions offer a significant level of protection from these forms of malware, you should also consider deploying a file integrity checking tool, such as the commercial Tripwire, GFI LANguard System Integrity Monitor, and Ionx Data Sentinel tools. As we discussed in Chapter 7, each of these tools can spot file changes made by a kernel-mode RootKit, if the developer or attacker utilizing the RootKit forgets to disguise such changes. As we noted in the Linux kernel-mode RootKit defenses, it's quite common for attackers to fail to hide all of their file changes with a kernel-mode RootKit. Therefore, looking for these changes with a file integrity checking tool is a sound strategy.

Response

When responding to an attack that employs a kernel-mode RootKit on your Windows machine, make sure you bring a CD-ROM with a fresh copy of your antivirus tool installation and the latest signatures. Many Windows antivirus tools can detect and then uninstall various kernel-mode RootKits, and having this capability in the field for incident response is invaluable. Just install the antivirus program on the victim machine, keep your fingers crossed that it has a signature to find the already-installed RootKit, and then tell the antivirus tool to remove the offending malware.

If the antivirus tool cannot find or remove the malware, you'll need to perform a more detailed analysis of the system without relying on the embedded kernel. Again, the FIRE and Knoppix bootable Linux CD-ROMs come in handy. "How can I use a Linux CD-ROM to analyze my Windows system?" you might ask. Well, although FIRE and Knoppix are bootable Linux images, they include a variety of tools for looking at Windows disk partitions. So, to analyze the system in more detail, you'd configure the system to boot from FIRE or Knoppix, thereby starting a Linux environment. Then, you'd run various Linux tools inside of FIRE or Knoppix to analyze the Windows partition of your machine. FIRE, my favorite tool for performing such analyses, includes a variety of items for analyzing a Windows hard drive, shown in Table 8.5.

Table 8.5. FIRE Tools for Analyzing a Windows CD-ROM

Tool Name	Description
F-prot	A free demo version of the commercial F-prot virus scanner from FRISK Software International. This version can search for Windows and Linux malware, including a variety of kernel-mode RootKits.
Editreg	A Linux command-line tool for searching and altering the registry on a Windows partition.
The Sleuth Kit (formerly called TASK)	A Linux tool for forensics analysis of hard drive images, including various UNIX drive formats, but also Windows FAT and NTFS partitions.

Currently, a bootable Linux CD-ROM is the best way to go, as there aren't any solid bootable forensics CD-ROM images of Windows publicly available at the time of this writing. Microsoft's licensing for Windows prohibits people from creating such a Windows distribution, developing a CD-ROM image of it, and making it available for download on the Internet. Doing that, someone would in essence be giving away Windows, certainly a no-no from a license perspective. Therefore, in our incident-handling operations, we utilize a Linux CD-ROM like FIRE with its built-in tools to support incident handling on our Windows machines.

Armed with these tools on the handy, free FIRE CD-ROM, you'll be able to conduct solid searches of your registry and file system to conduct a detailed forensics analysis of the machine. This book doesn't cover forensics analysis in detail, but I recommend that you grab a copy of *Computer Forensics: Incident Response Essentials* by Warren Kruse and Jay Heiser for an introduction to the craft of computer forensics, or *Incident Response* by Chris Prosise and Keven Mandia for more details on forensics investigations.

PREV < Day Day Up > NEXT