The Linux Kernel

Way back in the heady days of 1991, Linus Torvalds started the project that created the Linux kernel. Today, Torvalds still heads the team that maintains and updates the kernel. Given the Herculean efforts of Torvalds and his team, many people refer to the entire operating system as Linux. However, this terminology, although convenient, is imprecise. If you want to be very particular, the term *Linux* really refers to just the kernel itself, the component of the operating system Torvalds and team crafted and currently maintain.

The rest of the operating system consists of a multitude of different open-source projects, developed by a variety of different groups and collected together in various distributions. For example, the folks over at the GNU Project created the common C language compiler included with most Linux distributions, the GNU C Compiler (gcc). GNU is pronounced "guh-NEW" and is a recursive acronym that stands for GNU's Not UNIX. The GNU project also created a lot of other programs integral to the operating system, including many of the commands utilized every day by administrators and users [5]. Beyond GNU, the GUI-based window system used in most Linux distributions was created by the XFree86 project [6]. Also, there is code from many hundreds of different development teams floating under what we sometimes sloppily refer to as merely Linux. Sure, the Linux kernel is the software that controls and coordinates all of these different parts of the operating system. However, Linux is really just the kernel itself. For this reason, some people refer to Linux-based operating systems as GNU/Linux, a nod to the GNU project and its creation of numerous nonkernel components of the operating system [7].

So, at the heart of a GNU/Linux system, we find the Linux kernel, a very juicy target for the bad guys. The Linux kernel is really just a large piece of complex code that includes a huge number of features running in Ring 0 on x86 hardware. Before we analyze how bad guys attack this target, let's look at the Linux kernel in a little more detail. In the next section, we'll go on a brief adventure through the Linux kernel.

Adventures in the Linux Kernel

> *All your life has been spent in pursuit of archeological relics. Inside the Ark are treasures beyond your wildest aspirations.*
>
> —Dialogue from the movie *Raiders of the Lost Ark,* 1981

For our Linux kernel adventure, please feel free to boot up your own Linux machine and follow along with our discussion by typing commands on your own box. Or, if you don't like hands-on analysis, you can simply read this section and tuck the ideas away for some other time. Our goal here is to demystify the kernel and explore some of its fundamental structures so that we can later understand how attackers manipulate them. In a sense, we'll be acting like archaeologists on a dig of our system for juicy tidbits associated with the kernel. Just as an archaeologist analyzes artifacts left over from ancient civilizations to determine facts about their culture and activities, so too will we be analyzing artifacts created by and associated with our kernel to get a feel for its activities. When you boot a Linux system and log in to it, you are typically staring at a GUI or terminal that exists in user mode. For our adventure, we'd like to peer inside the kernel to see what it's up to. So, how can our user-mode processes get information about the kernel? Fortunately, Linux offers an amazingly simple and intuitive way to view various kernel-mode data structures so we can see what's going on underneath the sheets.

On most Linux systems, the kernel creates a very special directory called /proc, which is pronounced "slash proc." Unlike most directories on the Linux file system, /proc isn't really a set of bits on your hard drive. It's virtual, living only in memory, appearing nowhere on your disk. The kernel creates /proc as a nifty abstraction of itself so that administrators and running

programs can view the kernel's status and other aspects of the running system. In other words, /proc is the kernel's elegant way of giving you a portal to view the innards of your operating system. To make viewing these data structures easy, this portal appears as a piece of your file system, with virtual directories and files that contain vital statistics about your machine.

But /proc is more than a mere portal for you to peek into. Indeed, a lot of commands that you run on a Linux system just grab data from /proc and format it nicely for you. For example, when you run the `netstat` command to get a list of listening TCP and UDP ports, the command just grabs data from the directory /proc/net, where information about the network status is made available to all commands running on the box. In fact, you can think of `netstat` and many other commands as merely nice user interfaces that gather information from /proc and format it for your viewing pleasure.

Most of /proc is read only. However, some parts of it can be written to. Writing to various select places in the /proc directory can be used to alter the configuration of the kernel in real time. For example, by changing the value of some of the settings inside of /proc/net, an administrator can configure the machine to forward packets (making it behave like a simple router) or adjust its firewall rules. Typically, these changes are made with a configuration tool that tweaks stuff inside of /proc. However, they can be applied to a running kernel more directly by editing some of the values of /proc.

So, /proc is very powerful. To get a better feel for its capabilities, let's take a look inside of /proc. Log in to your machine and use the `cd /proc` command to change directories into /proc. Note that on most Linux systems, we don't even need root-level access to look at /proc, so you can log in with any user ID you choose. You won't be able to see everything if you are a nonroot user, but you'll still be able to get a solid idea of the kernel and its status. As we explore /proc, I advise you to just look around, using the `cd`, `ls`, and `less` commands, which only let you view items and not change them. The `cd` command is used to change directories, `ls` shows a directory listing, and `less` displays the contents of a file. Hit the q key to get out of `less` when you are finished viewing a file. I advise you not to change anything in /proc, as such alterations could make your system unstable. If you just use `cd`, `ls`, and `less`, you'll be safe, as these commands only let you navigate and view the contents of directories and files, without altering any data. Once inside /proc, run the `ls` command to get a listing of the /proc virtual directory, as I've done in Figure 8.3.

Figure 8.3. Peering inside /proc to look at kernel information.

In /proc, a bunch of directories have the names of various integers, starting at 1 and increasing. These directories contain information about each running user-mode process on the machine, with the directory name being set to the process ID number (e.g., 1, 1012, 1147, etc.). You can change into one of these directories, look at components of the process using the `ls` command, and use the `less` command to view various details of any running user-mode process on the system. In a sense, /proc lets you look into the soul of each running user-mode process. We can view the command-line invocation that was typed to start the process (cmdline), the process's current working directory (cwd), its environment variables (environ),

an image of the binary executable (exe), and other elements of the process. In Figure 8.4, I've changed into the directory of process ID number 1 which is the init daemon, the master user-mode process that started all other user-mode processes on my machine during system boot. Init always gets assigned a process ID of 1 because it's the first user-mode process to exist on the box, created by the kernel at boot time. I ran the `ls` command to view various elements of the init daemon process. To view many of these elements, I need root privileges on the box. However, I can view the status of the process by running the command `less status`. The status shows information about the name, process ID, user ID, and virtual memory associated with the running process.

Figure 8.4. Looking inside a process ID in /proc to view its status.

So, looking inside the soul of running processes can be fun and informative. It sure is nice of the kernel to create this detailed view of all running processes. However, we're here to look at the kernel itself, not user-mode processes. So, let's go back into /proc and look at the kernel-specific information presented there. Inside of /proc, the kernel provides a variety of useful tidbits about itself, including those files described in Table 8.1.

Table 8.1. A Sampling of Interesting Components of /proc

File or Directory	Purpose
`/proc/cpuinfo`	This file contains information about the system's CPU, including its speed, cache size, and other parameters.
`/proc/devices`	This file contains a list of various devices on the machine, such as hard drives and terminals.
`/proc/kmsg`	This file holds log messages from the kernel, which can be read using the `dmesg` command.
`/proc/ksyms`	This file includes a list of all variables and functions that are exported via loadable kernel modules on the machine.
`/proc/modules`	This extremely important file holds a list of loadable kernel modules that have been inserted into the kernel to extend or alter its base functionality.
`/proc/net/`	This directory contains information about the current network configuration and status of the machine.
`/proc/stat`	This file includes statistics about the kernel itself, such as data about the CPU, virtual memory, and hard drive usage.
`/proc/sys/`	This directory includes a variety of subdirectories and files that show kernel variables. These variables can be used to view or even tweak the configuration of the kernel.
`/proc/version`	This file indicates the version of the kernel that is currently running on the machine.

Table 8.1 gives only a sampling of some of the more important elements included in /proc. Feel free to explore these items, as well as others in your /proc directory. For each of these files or directories, you can safely use the `cd` and `less` commands to view their contents on your machine.

Inside of /proc, the loadable kernel module information in /proc /ksyms and /proc/modules is of particular interest, because loadable kernel modules allow for the extension of the kernel. By altering the kernel so that it can support new features, loadable kernel modules let Linux more easily adapt to new hardware types or additional software functionality. For example, you could add a module that functions like a device driver for some unusual fancy new hard drive that a stock kernel just doesn't know how to handle. In the olden days of the Linux kernel, you had to recompile your kernel to extend its abilities. Now, you can just insert additional modules. These kernel modules are dynamically loaded into a running kernel and don't even require a reboot of the machine to take effect. What's more, these loadable kernel modules are actually part of the kernel itself, running in Ring 0, with full access to all kernel code and data. The modules referred to in the directory /lib /modules are automatically applied to the system during boot. Additionally, any root-level user can add a loadable kernel module at any time using the `insmod` command. These kernel modules are very important, especially as we start to talk about ways to attack the kernel.

Outside of /proc, another very interesting artifact in your file system associated with the kernel is /dev/kmem. As you might recall from Chapter 7, the /dev directory contains pointers to various devices included on your system, such as components of your hard drive, the mouse, and terminals. As with most things kernel-related, /dev/kmem is special, in that it contains an image of the running kernel's memory. A related file, /dev/mem, contains an image of all of the memory of the system, not just kernel memory. The /dev/kmem and /dev/mem files were constructed by and for the kernel to read and use, not humans, so they're not designed to be easily read by the human eye. Even if you could directly read them, /dev/kmem and /dev/mem would be incomprehensible gibberish without the appropriate tools to parse, display, and search them. However, even though we cannot directly view or edit it under normal circumstances, /dev/kmem is yet another potential target for kernel-altering bad guys, as we shall soon see.

Now that we've gotten a high-level tour of what the kernel wants to show us with /proc and /dev/kmem, let's look at how user-mode processes interact with the kernel. Whenever you run most programs, the kernel creates a process, which includes memory space for the program's code and data, as well as threads of execution running through the memory space. As they run, most processes usually need to tell the kernel to do something. If a process wants to interact with any of the hardware, such as reading or writing from the hard drive or network interface, it'll have to somehow interact with the kernel to get such tasks done. Or, if it wants to run another program to do some other activity, it'll have to ask the kernel to execute that other program.

How do processes make these requests of the kernel? To interact with the kernel, user-mode processes rely on a concept termed *system calls*. The Linux kernel supports a variety of different system calls to do all kinds of activities, including opening files, reading files, and executing programs. These system calls represent a transition from user mode to kernel mode, as the user-mode process asks the kernel to do something by invoking a system call. To get a feel for which system calls your machine supports, you can look at the header file included in your system for building software (including the kernel itself) that utilizes system calls. This file is typically located in /usr/include/sys/syscall.h, /usr/include/bits/syscall, or /usr/include/asm/unistd.h. Although these locations are pretty common for these files, the particular location of these files does sometimes vary between different Linux distributions, so you might have to hunt for them. More than 100 different system calls are supported in a modern Linux kernel, but a few of the most important ones are shown in Table 8.2. The maximum number of system calls that can currently be supported by Linux is 256.

Table 8.2. A Small List of Some Important System Calls

System Call Name	Function
SYS_open	**Opens a file**
SYS_read	Reads a file from the file system
SYS_write	Writes to the file system
SYS_execve	Executes a program
SYS_setuid	Sets the permissions of a running program
SYS_get_kernel_syms	Accesses the system table
SYS_query_module	Helps insert a loadable kernel module into the kernel

Now, most user-mode processes don't activate these system calls directly. Instead, the operating system includes a system library full of code that actually invokes the system call when it is required. These standard system libraries, which are typically just a group of shared C language routines, are built into the Linux operating system. So, a running user-mode process calls a system library to take some action. The system library, in turn, activates a system call in the kernel. To activate a system call, the system library sends an *interrupt* to the CPU, essentially tapping the CPU on the shoulder, telling it that it needs to change to Ring 0 and handle a system call using kernel-mode code. To initiate a system call, the user-mode program or system library runs a machine-language instruction that triggers CPU interrupt number 0x80, a hexadecimal number that tells the Linux kernel to use its system call handling code.

To determine which kernel code to run to handle the system call, the system relies on an absolutely critical data structure in the kernel known as the *system call table*. The system call table is really an array maintained by the kernel that maps individual system call names and numbers into the corresponding code inside the kernel needed to handle each system call. In other words, the system call table is just a collection of pointers to various chunks of the kernel that implement the actual system calls. The system call table is not the same thing as the syscall.h header file we discussed earlier. That file is just used for compiling software and the kernel. The system call table is a live data structure stored in kernel memory mapping various system calls to kernel code. The relationships among user mode processes, system libraries, the system call table, and the kernel code that implements system calls are illustrated in Figure 8.5.

Figure 8.5. Processes call libraries, which invoke system calls using the system call table.

To look at various system calls supported by your machine, you can view the file System.map, which is located in /boot/System.map, /System.map, or /usr/src/linux/System.map. Whereas the syscall.h file is just used for compiling software, the System.map file was created when your kernel was originally built and reflects far more specific information about your kernel. In particular, the System.map file contains a listing of various symbols used by the kernel. These symbols are nothing more than a bunch of data structures associated with the kernel, including global variables, tables, and system calls. Keep in mind that System.map doesn't hold your current running system call table for the machine; instead, it holds information about the original system call table that was created when your kernel was originally compiled. Even if you didn't compile the kernel yourself, this file was created when your kernel was originally compiled, and it came as part of your installation. The symbol information in System.map is listed by memory address location and symbol name. This memory address is the place inside of kernel memory where that particular structure is located. In Figure 8.6, I've shown the contents of my System.map file using the command `less /boot/System.map`. Note that there are a lot more elements in here than just the system calls. There are a huge number of other symbols in addition to the system call information, such as other variables and signals associated with the kernel.

Figure 8.6. Looking at System.map to see the execve system call information.

```
skoudis@fred:/boot
c01059c0 T __switch_to
c0105aa0 T sys_fork
c0105ac0 T sys_clone
c0105af0 T sys_vfork
c0105b10 T sys_execve
c0105b70 T get_wchan
c0105be0 T __up
c0105c00 T __down
c0105ca0 T __down_interruptible
c0105d60 T __down_trylock
c0105d98 T __down_failed
c0105da4 T __down_failed_interruptible
c0105db0 T __down_failed_trylock
c0105dbc T __up_wakeup
c0105dd0 T copy_siginfo_to_user
c0105e80 T sigsuspend1
c0105f20 T sys_sigsuspend
:
```

Here is the execve system call and its address.

In Figure 8.6, I have paged down to the point where I can see the SYS_execve system call, which is the system call used to execute programs. When one program, such as a command shell, needs to execute another program, such as a command, it calls the SYS_execve system call to ask the kernel to start the other program. Note that the memory address associated with SYS_execve (c0105b10), as well as all other items inside of System.map, start with a hexadecimal number c. That's because, when referenced from a user-mode process on a system with a 32-bit processor, all kernel memory structures are located in memory locations ranging from 0xC0000000 to 0xFFFFFFFF [8].

Linux includes a nice tool named `strace` for watching various system calls made by a running user-mode process. You can use `strace` to invoke any program, and `strace` will display all system calls, the arguments passed to those system calls, and the return values from the system calls as the program runs. In Figure 8.7, I used the `strace` tool to run the command `ls` so we could see all of the system calls made by `ls` as it lists the contents of a directory. I could have straced any other program, but I chose `ls` because it is a familiar program to most Linux users.

Figure 8.7. Using strace to analyze the system calls invoked when running the `ls` command.

I used strace to invoke the `ls` command

Here is the execve system call to run the /bin/ls program.

Here, the machine invokes the open system call to utilize some shared libraries.

```
skoudis@fred:~
[skoudis@fred home]$ strace ls

execve("/bin/ls", ["ls"], [/* 21 vars */]) = 0
uname({sys="Linux", node="fred", ...})  = 0
brk(0)                                  = 0x80541e4
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=42988, ...}) = 0
old_mmap(NULL, 42988, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/libtermcap.so.2", O_RDONLY)  = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\260\r\0"..., 1024) = 10
24
fstat64(3, {st_mode=S_IFREG|0755, st_size=11832, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
x40022000
old_mmap(NULL, 14932, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40023000
mprotect(0x40026000, 2644, PROT_NONE)   = 0
old_mmap(0x40026000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x200
0) = 0x40026000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
```

As you can see, as the `ls` command runs, the execve system call is invoked to run the /bin/ls program, and the open system call is utilized to access various shared libraries. Other system calls that are invoked by `ls` include fstat (which checks a file's status, including its permissions and owner) and mprotect (which limits access to a region of memory while a given program uses that memory). Using `strace`, we are witnessing various transitions from user mode to

kernel mode, as the program uses system calls to ask the kernel to perform various operations. Therefore, we can get a feel for the relative importance of various system calls by watching which ones common commands on the system rely on. Additionally, we can start to see which system calls attackers might want to alter as they attack the kernel.

Methods for Manipulating the Linux Kernel

> *Our methods have not differed as much as you pretend. I am but a shadowy reflection of you. It would take only a nudge to make you like me… to push you out of the light.*
>
> —Dialogue from *Raiders of the Lost Ark,* 1981

With our whirlwind tour of the Linux kernel complete, let's turn our attention to how attackers manipulate the kernel to achieve their dastardly deeds. Keep in mind that the goal of each of these kernel manipulation tactics is still the same main objective of all RootKits: to provide backdoor access, while hiding the attacker's presence on the system. In particular, these kernel manipulation tactics provide methods for implementing backdoors and then hiding those backdoors on the machine.

With that goal in mind, there are at least five different methods for implementing a kernel-mode RootKit in Linux. Additional possibilities might also exist, currently tucked away in a researcher's or attacker's lab waiting to be unveiled on an unsuspecting victim. Yet these five possibilities represent the most common methods today for implementing kernel-mode RootKits on Linux machines. These kernel attacks include applying evil loadable kernel modules, altering /dev/kmem, patching the kernel image on the hard drive, creating a fake view of the system with User Mode Linux, and altering the kernel using Kernel Mode Linux. Let's analyze each method in more detail.

Evil Loadable Kernel Modules

A primary method for invading the Linux kernel to implement a kernel-mode RootKit involves creating an evil loadable kernel module that manipulates the existing kernel. This technique first emerged publicly in approximately 1997, and grew in popularity over subsequent years, with a huge variety of different evil module variations now available [9]. Today, it remains the most popular technique for implementing kernel-mode RootKits on Linux systems.

Remember, loadable kernel modules are a legitimate feature of the Linux kernel, sometimes used to add support for new hardware or otherwise insert code into the kernel to support new features. Loadable kernel modules run in kernel mode, and can augment or even replace existing kernel features, all without a system reboot. Because of the convenience of this feature for injecting new code into the kernel, it's one of the easiest methods for implementing kernel-mode RootKits on systems that support kernel modules (e.g., Linux and Solaris). To abuse this capability for implementing RootKits, some malicious loadable kernel modules change the way that various system calls are handled by the kernel, as illustrated in Figure 8.8.

Figure 8.8. Some loadable kernel module RootKits alter the system call table to execute the attacker's module code instead of the legitimate system call code.

To launch this kind of attack, the bad guy utilizes a loadable kernel module that includes two components, identified as elements A and B in Figure 8.8. The attacker inserts this module into the kernel, jumping the gap between Ring 3 and Ring 0 by using the `insmod` command to put the module's code inside of the kernel. Once inserted, the attacker's loadable kernel module, shown as element A in the figure, includes code that operates quite similarly to the original system call code within the kernel. In our example, the bad guy has created a loadable kernel module that implements the SYS_execve system call, used to execute programs, but the bad guy throws in a little twist. When the new, malicious SYS_execve system call is invoked, it will check to see which program it has been asked to execute. If the execution request is for a program that the attacker configured the system to redirect, the evil kernel module will actually execute a different program instead. Otherwise, if the execution request is for some program the attacker isn't interested in redirecting, the normal program will be run. The new SYS_execve system call includes intelligence to decide what to execute outright and what to redirect. That's the twist.

This is all nice, but how does the attacker's malicious SYS_execve get run in the first place? That's where element B from Figure 8.8 comes into play. The attacker's loadable kernel module will alter the system call table so that it no longer points to the normal SYS_execve call in the kernel. Instead, the entry in the system call table associated with SYS_execve will now point to the attacker's own code. The legitimate SYS_execve system call will remain unused on the system, lying dormant. What the attacker is doing here is playing bait and switch with system calls to redirect execution of selected user-mode programs.

Instead of implementing all of this functionality from scratch, the attacker could just wrap the existing SYS_execve system call code with the attacker's own code that includes intelligence to determine whether to pass the execution request through to the real SYS_execve or to execute some other program instead. This system call wrapping option, which requires less custom code from the attacker and is therefore more efficient, is illustrated in Figure 8.9. The system call table is still manipulated, but now points to the attacker's wrapper code. When the SYS_execve call occurs, the attacker's wrapper is activated, which checks to see if the execution request is for a program that the attacker wants to redirect. If so, it'll pass the request off to the real SYS_execve code to execute the alternate program. Otherwise, the wrapper will just pass in a request to execute the actual program requested in the system call. Using either alternative

(creating entirely new system call code or wrapping an existing system call's software), the end result is the same: The SYS_execve call inside the kernel will include execution redirection.

Figure 8.9. Some loadable kernel module RootKits wrap existing kernel code for system calls.

This technique of rewriting a pointer in the system call table so that it executes the attacker's code is really another form of the API hooking technique we discussed in Chapter 7. On a Windows machine, DLL injection involves inserting DLL code into a running process. API hooking redirects various function calls into the DLL code injected by the attacker. In Chapter 7, we discussed this concept in the context of injecting Windows DLL code into Windows user-mode processes. Of course, the Linux kernel doesn't run Windows DLLs. Here, the attacker is inserting code, in the form of loadable kernel modules, into the Linux kernel. Then, the attacker performs API hooking by overwriting various memory addresses in the system call table so that they point to the loadable kernel module. It's code injection and API hooking all right, but this time in the Linux kernel.

Of course, using this technique against the SYS_execve system call, the attacker has modified only the execution associated with some user-mode programs, and not any system calls associated with reading those programs' binary executable files. The resulting execution redirection is very powerful, because the technique can defeat the file integrity checking tools we discussed in Chapter 7. As you recall, file integrity checking tools are programs that look for alterations to various system files, such as the login routine or sshd, which are used for accessing the system. By reading these files and comparing cryptographically strong hashes of them against known trusted fingerprints for the files, the file integrity checker can detect a user-mode RootKit, which would replace the login or sshd binary executable files with backdoor versions.

With kernel-mode RootKits, everything changes in favor of the attacker. Now, the bad guy will use execution redirection in a loadable kernel module RootKit to map the execution of the login and sshd binary executable files to some other programs that include backdoors, such as programs named alt_login and alt_sshd, where alt stands for "alternative." These alternatives

include some backdoor password the bad guy can use to remotely access the machine. Now, when a file integrity checking tool comes along and compares the hashes of the login and sshd files to their previous values, they will remain exactly the same. That's because the attacker doesn't modify the login or sshd files. The file integrity checker uses the SYS_open and SYS_read system calls to look at login and sshd, and they appear completely intact, because they *are* intact. However, when the system tries to execute the login or sshd programs for a new user logging in, the evil SYS_execve system call will kick in. The evil kernel module will run the backdoor versions of these programs, alt_login or alt_sshd.

So far, we've just discussed kernel manipulation in the context of the SYS_execve system call. An attacker could likewise modify the SYS_open, SYS_read, and any other system call using this technique. By modifying these and other system calls, the attacker could hide files, TCP and UDP ports, and running processes on the system. When any user-mode program makes a system call, the attacker's code will check to see if the user's program is asking questions about some hidden item in the system. If the user program is looking for a hidden item, the kernel will lie and say that the item is not on the machine. A single evil kernel module could do all of this work, remapping or wrapping an arbitrary number of system calls, all with the same piece of code. In fact, most real-world kernel-mode RootKits alter half a dozen or more system calls to hide various nefarious activities of the attacker.

For example, suppose an attacker breaks into a machine and installs a backdoor shell listener, such as the Netcat tool we discussed in Chapter 5. Running the backdoor shell listener creates several items on the machine an administrator could look for: the executable binary file associated with Netcat, the running backdoor process, and a TCP or UDP port on which the process is listening. An administrator might look for the file using the `ls` or `find` commands, the process using the `ps` or `top` commands, and the network ports using `netstat` or `lsof`. By installing a kernel-mode RootKit to alter various system calls, the bad guy can hide the file, process, and network ports. The kernel will fib about any of these traces associated with the backdoor, regardless of the program that comes asking about it, whether it's `ls`, `find`, `ps`, `top`, `netstat`, or `lsof`. That, dear reader, is the power of a kernel-mode RootKit in action.

At this point, we should note that installing multiple kernel-mode RootKits on a single system could have very mixed results. If each RootKit manipulates different system calls, the two could coexist on the same machine, happily unaware that the other kernel-mode RootKit has been inserted. Two attackers could coexist on the box, without even knowing or seeing the activities of each other. However, in all likelihood, the kernel-mode RootKits will go after the same set of system calls, such as the popular and powerful SYS_execve and SYS_open calls. In this case, the features associated with the last kernel-mode RootKit installed on the box would override any features of previously installed RootKits. In other words, the last one in wins the game.

So, we've seen how the attacker can hide files, processes, and network usage with loadable kernel modules, but the attacker has a problem. There's still the issue of the module itself. If anyone uses the `insmod` command to insert a module, under normal circumstances, that module will show up in the output of the `lsmod` command, as well as inside of the /proc/modules file. An administrator could check the list of modules and look for something fishy. Of course, that's only under normal circumstances, which kernel-mode RootKits deviously work to change. To avoid detection by `lsmod`, an attacker could add another system call modification to the kernel-mode RootKit that hides the kernel module itself. Any requests to list all kernel modules will be intercepted by the attacker's code, which will only list those modules the attacker wants the victim to know about. That list, of course, won't include the evil kernel module. Furthermore, the /proc/ksyms file displays symbols implemented by loadable kernel modules. However, a kernel module can choose whether or not to export its symbols into /proc/ksyms with a single line of code. Therefore, looking for evil loadable kernel modules inside of /proc/ksyms or using the `ksyms` command (which just reads /proc/ksyms and displays its contents) is usually futile.

There is another problem for the bad guy with using loadable kernel modules to implement this type of attack. Loadable kernel modules don't survive across a system reboot. Both legitimate

and evil kernel modules are flushed out when the system is shut down and have to be reloaded into the kernel during each and every boot sequence. Of course, the attacker wants to make sure that the evil loadable kernel module sticks to the machine across reboots, without tipping off an administrator about the attacker's presence.

One common technique to get around this problem is to alter some program included in the boot process so that it reloads the evil kernel module when the system starts up. The most popular choice for a carrier of the evil kernel module is the init daemon, the first process that runs on the box, as illustrated in Figure 8.10. When you boot your system, the kernel is loaded into memory, as shown in Step 1. Then, in Step 2, the kernel starts the init daemon, which in turn activates all other user-mode processes on the machine. Attackers often add code to the init daemon so that, as soon as it starts running, it inserts the evil kernel module, which is illustrated in Step 3. By using the executable binding techniques we discussed in Chapter 6, the code to insert the modules is just prepended to the normal init daemon code, resulting in a single binary executable file for init.

Figure 8.10. Modifying the init daemon to reload an evil kernel module during the boot sequence.

Of course, once inserted, the loadable kernel module itself masks any changes to the init file on the hard drive. If any program, such as a file integrity checker, tries to open the init program file to look at its contents, as shown in Step 4, the kernel module will respond with a lie (in Step 5), saying that the init daemon file looks perfectly intact! Therefore, a file integrity checker won't be able to detect the subterfuge, as shown in Step 6. Because the init daemon runs before any other user-mode process on the box, it poisons the kernel before any detection mechanisms can be executed. Of course, in lieu of the init daemon, an attacker can alter any other startup script or binary executable on the system to load the evil kernel module, using any of the startup techniques we discussed in Chapter 5.

Now that we've analyzed the general methods used by most evil loadable kernel modules, let's focus on two rather popular specific implementations of these ideas. In the next two sections, we'll look at Adore and the Kernel Intrusion System (KIS), both of which implement all of the ideas we've discussed so far.

Example Loadable Kernel Module RootKit: Adore

Adore is the most popular Linux kernel-mode RootKit in widespread use today. Perhaps that's where it gets its name: Attacker's "adore" it. On some Web sites in the computer underground, the tool is even referred to as "mighty Adore," no doubt because of its solid feature set, the simplicity of its use, and the power it gives an attacker. Written by a developer named Stealth, Adore targets Linux 2.2 and 2.4 kernels, allowing an attacker to hide on the system by remapping and wrapping various system calls using a single loadable kernel module. In addition to Linux, a programmer calling himself bind has ported Adore to FreeBSD. Once installed on a victim Linux or FreeBSD machine, Adore lets the attacker do the following:

- Hide or unhide files.
- Make a given process ID visible or invisible.
- Make a process ID invisible permanently, so that even Adore cannot make it visible again.
- Execute any program as root, regardless of the actual permissions of the user invoking the program.
- Hide the promiscuous mode status of the user interface to disguise a sniffer.
- Hide the Adore loadable kernel module itself.

To accomplish these tasks, Adore consists of two components: a loadable kernel module (called Adore) and a program the attacker uses to interact with the kernel module (named Ava). Think of Ava as the user interface for Adore. After installing the Adore module using the `insmod` command, the attacker must configure it by running Ava on the same system where the module resides. Ava doesn't work across a network; it must be used to configure Adore on the local system. Ava presents a simple menu-driven interface, as shown in Figure 8.11.

Figure 8.11. Ava, the Adore user interface.

```
root@fred:/home/tools/adore
[root@fred adore]# ./ava
Usage: ./ava {h,u,r,R,i,v,U} [file, PID or dummy (for U)]

        h hide file
        u unhide file
        r execute as root
        R remove PID forever
        U uninstall adore
        i make PID invisible
        v make PID visible

[root@fred adore]#
```

For remote access of the victim machine, Adore also includes a backdoor root shell listener on a port configurable by the attacker. The attacker can use Netcat in client mode to connect to this backdoor listener from across the network and get direct command shell access to the machine. The command shell process, of course, is also hidden by the kernel module.

Adore also hides TCP and UDP port numbers configured by the attacker. That way, other network-listening processes created by the attacker will be disguised.

Although Adore does have many features, it does have a significant shortcoming from a

capability perspective. The tool does not include execution redirection capabilities; its focus is solely on hiding files, processes, TCP and UDP ports, and promiscuous mode. Interestingly, execution redirection was available in an earlier version of Adore (version 0.32), but was inexplicably removed in subsequent releases (versions from 0.39b to 0.42 lack the feature).

Example Loadable Kernel Module RootKit: The Kernel Intrusion System

Although Adore might be the most popular kernel-mode RootKit on Linux, there are more powerful tools available. KIS, written by Optyx, actually includes more features, and is one of the most powerful kernel-mode RootKits released to date. Implemented as a loadable kernel module, KIS targets the Linux 2.4 kernel. It offers a standard complement of kernel-mode RootKit functionality, including the hiding of files and directories, processes, network ports, and promiscuous mode. KIS also offers execution redirection capabilities.

You might shrug your shoulders and say, "We've seen that before, so what's the big deal?" Well, the big deal associated with KIS is its incredible ease of use, manifested in two forms: a slick GUI and an interface centered around hidden processes. First, let's look at its user interface, shown in Figure 8.12. Using a series of helper screens, the attacker can configure the KIS kernel module and attach it to any binary executable on the file system, such as the init daemon, to get KIS restarted at system boot. Once the kernel module has been loaded, the GUI lets the attacker remotely control the kernel module using the same GUI. The attacker configures various settings in the GUI, and encrypted commands are carried across the network to the victim machine, where the KIS kernel module executes them. The KIS user interface is highly reminiscent of earlier application-level Trojan horse programs, such as the Back Orifice 2000 and Sub Seven tools that we referenced in Chapter 5. However, the KIS GUI controls a kernel-mode RootKit, not a mere application-level Trojan horse backdoor.

Figure 8.12. The KIS user interface.

As a bonus feature, for its communication across the network, KIS even implements a nonpromiscuous sniffing backdoor to receive commands on the network without listening on a port. As we discussed in Chapter 5, this type of backdoor listens for commands from an

attacker by sniffing them off of the line, thereby avoiding a listening port and throwing off the investigation team. So, embedded inside of KIS, we have a kernel-mode nonpromiscuous sniffing backdoor. What a nasty combination!

The KIS GUI is certainly a major step forward in the evolution of ease of use in kernel-mode RootKits, endearing it to legions of script kiddie fans around the globe. However, the GUI is not the most significant innovation introduced by KIS. The real paradigm shift introduced by KIS is its use of hidden processes as the conceptual model for interacting with the kernel module.

To understand why the KIS fixation on hidden processes is so important, let's take a step back to other kernel-mode RootKits, such as Adore, for a moment. Suppose an attacker breaks into a machine and creates a backdoor listener on the box. After creating the backdoor, the attacker has to load the evil kernel module and then configure it to hide the backdoor's file, process, and TCP or UDP ports. Implementing all of this hiding can take valuable time away from the attacker. Making matters worse, once all of these items are hidden, the attacker cannot see them any more either! With most kernel-mode RootKits, the kernel lies about the presence of hidden items to all users of the machine, administrators and attackers alike. Often, when I personally use a kernel-mode RootKit in my lab, I forget where I put all of my hidden stuff on the machine. Attackers sometimes do this as well. They'll hide a backdoor, leave for a few days, and then return, only to grope around trying to find the files or process they hid earlier. Some attackers even jot down notes on paper to help remember where they put all of their hidden items on a conquered target. If law enforcement officers seize the attacker's notes, they'll be able to find all of the hidden elements recorded in those notes.

In a sense, most kernel-mode RootKits go too far in hiding various items, confusing some attackers in the process. KIS doesn't have this problem. By using hidden processes as the central mental model for interacting with the tool, KIS is far easier to use. With KIS, anything created by a hidden process is itself hidden, so an attacker can break into a machine and create a hidden process. From this hidden process, the attacker can install a backdoor. All aspects of the backdoor, which likely consists of a file, a running process, and some TCP or UDP port, will automatically be hidden because they were created by the original hidden process. Similarly, if an attacker runs a sniffer from within a hidden process, the resulting promiscuous mode status is automatically hidden. The attacker doesn't have to remember to go back and hide each element, because they are already hidden. That saves the attacker time.

However, the hidden process model goes even deeper. You see, a hidden process can view all hidden items on the machine. Outside of a hidden process, all hidden items are, of course, hidden. So, an attacker doesn't have to jot down paper notes about where various hidden elements are located. Instead, the bad guy can just fire up a hidden process and then use it to view all hidden files, processes, and port usage on the machine. However, a system administrator, who logs into the machine without a hidden process, will not be able to see all of the attacker's subterfuge. In this way, as illustrated in Figure 8.13, the attacker uses KIS to create a cone of silence, carving user mode into two worlds: a visible environment and a cloaked environment. From inside the cone of silence, where the attacker lives, everything on the system is viewable, hidden items and visible items alike. Outside the cone of silence, where users and administrators dwell, all hidden items are completely invisible. The KIS kernel module keeps the two worlds separate by carefully manipulating the system call table to hide things from visible processes, yet allowing invisible processes to see. That's a highly effective paradigm for interacting with a kernel-mode RootKit. Sadly, the very powerful ideas originally introduced by KIS are starting to trickle down into other kernel-mode RootKits.

Figure 8.13. Using KIS, the attacker creates a cone of silence, dividing user mode into a visible world and a hidden world.

With all of these innovations, you might be wondering why Adore remains the more popular choice for attackers over KIS. This phenomenon is likely due to the fact that Adore is far easier to compile and install than KIS, so the script kiddies often migrate to Adore. Once it's installed, however, KIS is easier to use and more powerful.

Who Needs Loadable Kernel Modules? Attacking /dev/kmem Instead

> */dev/kmem is our friend.*
>
> —Kernel-mode RootKit developers Sd and Devik, 2001

Although modifying a running kernel using loadable kernel modules is a widespread and effective technique, it's not the only game in town for implementing kernel-mode RootKits. Suppose the target machine was built without kernel module support. When compiling a custom kernel for a Linux machine, an administrator can choose whether to add loadable kernel module support or omit it from the resulting kernel. Without module support in the kernel, the administrator will have to build all kernel-level functionality right into the core kernel itself. Such kernels cannot be abused with evil loadable kernel modules, as the hooks necessary for loading such modules into the kernel (stuff like the /proc /ksyms file) are left out. For information about building a kernel that doesn't require or support modules, you can refer to various free Internet guides [10]. Alternatively, you could use Bill Stearns' wonderful kernel-building package (called, appropriately enough, buildkernel), at *www.stearns.org/buildkernel/*, which includes an option for creating a kernel that doesn't support modules.

So, if you build a kernel that lacks module support, are you safe from kernel-mode RootKits? Sadly, the answer is no. Various kernel-mode RootKit developers have honed their wares so they can now invade the kernel even without using any loadable kernel modules. To accomplish this, they utilize the facilities of /dev/kmem, that interesting file that holds an image of the kernel's own memory space where the running kernel code lives. By carefully patching the kernel in memory through /dev/kmem, an attacker can implement all of the attacks we discussed in the loadable kernel module part of this chapter, but without using any modules at

all.

"But wait a minute," you might be thinking, "earlier in the chapter you said that /dev/kmem was incomprehensible gibberish for humans." Yes, that's true. However, with the appropriate parsing tools, /dev /kmem can be read from and written to by a root-level user. In fact, some hard-core system administrators utilize debuggers and custom code to interact directly with /dev/kmem when troubleshooting systems. However, the concept of using /dev/kmem for implementing kernel-mode RootKits was originally introduced publicly in a detailed technical discussion and political manifesto written by Silvio Cesare in November 1998 [11]. The ideas were further refined and simplified by two kernel-mode RootKit developers named Sd and Devik in their white paper devoted to the topic in late 2001 [12].

In their white paper, Sd and Devik released code that searches /dev/kmem, looking for the system call table. When it finds the system call table, their software searches the table for various system call entries, such as those associated with SYS_open, SYS_read, and SYS_execve. Then, things get very interesting. The code released by Sd and Devik includes a variety of functions, but of most interest are the functions rkm (an abbreviation for read kernel memory) and wkm (which stands for write kernel memory). Using rkm, the attacker can read various useful items inside of kernel space. With wkm, the bad guy can insert code directly into kernel memory space. With rkm and wkm, in a sense, these developers have used /dev/kmem instead of modules to jump the divide between user mode and kernel mode.

Using this technique for altering /dev/kmem in a live kernel, an attacker can implement any of the ideas we discussed in the loadable kernel module section, without the use of any loadable kernel modules at all. For example, the attacker can use rkm and wkm to insert alternative code for the SYS_open, SYS_read, and SYS_execve system calls. Additionally, the attacker can modify or even replace the system call table inside the kernel so that it points to the attacker's code and not the legitimate kernel code. With these capabilities, shown in Figure 8.14, the attacker has complete control over the system and can implement file, process, network port, and promiscuous mode hiding that we saw in earlier kernel-mode RootKits. Additionally, as before, an attacker can tweak the kernel so that it performs execution redirection.

Figure 8.14. Altering a running kernel by reading and writing to /dev/kmem.

As with loadable kernel module RootKits, these changes to a live kernel through /dev/kmem do not survive across a reboot. Therefore, most attackers apply the same trick with the init daemon or other start-up program or script to get the /dev/kmem alteration applied to the kernel while the system is booting.

In addition to providing the useful parsing tools for searching, reading, and writing /dev/kmem, Sd and Devik also released a sample kernel-mode RootKit built on these ideas. They gave their tool the very elegant name SucKIT, which is an acronym for Super User Control Kit. From a functionality and usability perspective, the SucKIT kernel-mode RootKit is very similar to Adore, offering file and process hiding, as well as a password-protected backdoor shell listener. The biggest difference with SucKIT, of course, is that no kernel module is included, and module support isn't required on the target machine. By simply running SucKIT at the command line while logged in as root, the program automatically locates the system call table in memory, allocates space in the kernel to use, injects code into kernel memory, and alters the system call table to point to the new code. Although there is no GUI, installation couldn't be much simpler than that. All of the hard work of reading, searching, and altering /dev/kmem is done by the software itself. The attacker just runs a single command line to completely take over the system.

Patching the Kernel Image on a Hard Drive

You know, having to do that little dance of reloading kernel alterations, whether loadable kernel modules or /dev/kmem manipulation, every time the system reboots can be complex. Unnecessary complexity could lead to failures, either crashing the system or breaking the kernel-mode RootKit. There is in fact a simpler way to manipulate the kernel. With root-level permissions on the box, the attacker could just replace or patch the kernel image file on the hard drive itself. That way, on the next reboot of the system, the attacker's evil kernel would be reloaded into the system instead of the original wholesome kernel. Because the kernel image on the hard drive is just a file (readable and writable by root-level accounts), there's no need for the attacker to jump from user mode to kernel mode to make changes to this file. User mode to kernel mode transitions (e.g., those that occur through system calls, `insmod`, and /dev/kmem) are only required to interact with a running kernel, but aren't necessary to change the kernel image file on the hard drive. By just exercising rootly privileges, the attacker can overwrite the

kernel image file on the hard drive and get the new, evil kernel loaded into memory at the next reboot, as illustrated in Figure 8.15.

Figure 8.15. Replacing the kernel image on the hard drive.

In the Linux file system, the kernel image is stored in a file called vmlinuz, typically located in the /boot directory. To minimize storage requirements for boot devices, most of this kernel image file is compressed. During system boot, the first portion of vmlinuz gets loaded into memory and executed. This first portion of vmlinuz then decompresses the rest of the vmlinuz file and loads the entire uncompressed kernel image into memory. Sometimes, if you build your own kernel, you'll find a file called vmlinux, with a trailing x instead of a z. A vmlinux kernel image isn't compressed, and must first be compressed to prepare it for booting, converting it to vmlinuz. When replacing an original kernel with an evil version, the attacker must create the alternative kernel image, compress it, and overwrite the existing /boot /vmlinuz file with the evil replacement.

Replacing the entire kernel image file with a nasty variant is rather easy. An attacker could build a custom kernel on his or her own machine, and deploy this evil kernel on the victim's machine. Because Linux is an open-source operating system, the bad guy can modify the kernel source code to create a custom kernel that provides the attacker with backdoor access and hides nefarious activities on the machine. For example, with a dozen or so tweaks to some system calls in the kernel source code, an attacker can create a kernel image that would hide files with certain names, mask specific TCP and UDP ports, render processes with some names invisible, and implement execution redirection. Rather than monkeying with the system call table, the attacker can just sprinkle some new code right into the existing system call functions. In other words, the entire new kernel would be the RootKit, replacing the old kernel outright. The attacker could even program the new evil kernel so that it looks like the original kernel. For example, the evil kernel can be configured so that if anyone opens the altered /boot/vmlinuz file, the kernel will return the old, unmodified kernel image file, which it has squirreled away on the hard drive, instead of the modified version. In this way, an attacker can foil any file integrity checks against the kernel image file by altering system call code associated with opening and reading files.

There is a bit of a problem for the bad guys with the wholesale replacement of the kernel, though. Perhaps the victim machine has very specific kernel options, tricked out with custom code created by a system administrator who dabbles in specialized kernel development. Or, perhaps the existing kernel has some very special hardware support compiled in it that the attacker doesn't know about. If the attacker creates a brand new kernel and swaps it in place of the customized kernel, the administrator might quickly notice the attack or some hardware might become inaccessible. To avoid this situation, the attacker can simply edit the existing vmlinuz file instead of replacing it. By applying patches to the kernel image file on the hard drive instead of replacing it entirely, most of the existing functionality of the custom kernel will be preserved. The attacker's options will just be grafted into the existing kernel image file, as pictured in Figure 8.16.

Figure 8.16. Applying patches directly into the kernel image on the hard

drive.

In 2002, someone called Jbtzhm released a white paper and some code that allows an attacker to open, uncompress, parse, and apply patches directly to a vmlinuz file [13]. Jbtzhm's technique lets the attacker append new code to the end of the kernel image file, and then modify pointers within the existing code to point to the new functionality. Jbtzhm designed his software so that it would insert the code from a loadable kernel module right into the kernel image file, rather than having to load modules the old-fashioned way—after system boot. Loadable kernel modules, after all, are nice little chunks of kernel-mode code, ready to be applied into the kernel. Jbtzhm's technique just inserts the bundles of code from loadable kernel modules into the kernel image file to simplify the implementation of code to be grafted into the kernel. Therefore, using this technique, an attacker could patch a kernel image file with the Adore or KIS loadable kernel module RootKits, and have them automatically applied from the vmlinuz file itself during system boot.

The three methods for altering kernels that we've discussed so far (loadable kernel modules, altering /dev/kmem, and altering kernel images on the hard drive) are by far the most popular ways to implement kernel-mode RootKits on Linux today. However, there are two other methods that attackers have discussed at public conferences for implementing kernel-level attacks. These other methods for implementing kernel manipulation involve tools called *User Mode Linux* and *Kernel Mode Linux*, which we'll discuss in the next two sections. Although they haven't yet been widely used in attacks, these two additional methods could be utilized increasingly in the near future.

Faking Out Users with the User Mode Linux Project

> *Do you think that's air you're breathing now?*
>
> —Dialogue from *The Matrix,* 1999

The substitute or patched kernel idea from the last section could be extended even further, employing an amazing tool called User Mode Linux (UML), a project originally created and currently headed by Jeff Dike. Freely available at *http://user-mode-linux.sourceforge.net/*, UML lets its user run an entire Linux kernel inside of a normal user-mode process. It's called User Mode Linux because it runs an entire Linux system, with its own kernel, applications, and so forth, inside a user-mode process on a host Linux system. So, with UML, I can take my Linux machine, with its normal kernel intact and running just fine, and create multiple UML instances running as user-mode processes on my existing system. Each of these additional UML instances has its own kernel mode and user mode inside.

With UML, my underlying operating system acts as a host, with all of my UML instances as guest operating systems running on top of the host. These guest operating systems are entire Linux installations, each with its own kernel, network options, file system, and applications, all wrapped up inside of a standard Linux user-mode process. Each UML instance is independent of the others, running whatever programs it requires inside its own user-mode space. I can

therefore create virtual Linux machines that run on top of my real system, right alongside of normal user processes, as illustrated in Figure 8.17.

Figure 8.17. Legitimate uses of User Mode Linux involve creating multiple virtual Linux Machines on a single Linux system.

Perhaps you're familiar with VMWare or VirtualPC, two tools that let users create guest operating systems running on top of a host operating system. UML can also be used to implement guest operating systems on a host, but it differs from VMWare and VirtualPC in two important ways. First, UML is free and open source. Second, VMWare and VirtualPC implement a virtual x86-compatible processor, so almost any x86-compatible operating system (e.g., Linux, Windows, BSD, etc.) can be installed as a guest on them. UML, on the other hand, doesn't emulate an x86 processor. Instead, it acts as a proxy for making Linux system calls, creating the abstraction of Linux guest kernels living on top of a Linux host operating system. The current iteration of the project is Linux-centric. Still, despite this difference, UML is quite useful.

Please keep in mind that UML wasn't designed as an attack tool. It can be employed in all sorts of positive roles. For hard-core programmers working on changes to their kernel or writing new applications, UML provides a nifty little sandbox to run experiments inside. If the kernel modifications or new application completely crash the UML instance, the developer can simply restart that UML instance without rebooting the entire host system. Therefore, UML provides a great deal of convenience for developers and experimenters. Additionally, service providers could utilize UML to provide virtual Linux hosting services to clients. Each client could rent (or be given) a UML instance on the service provider's single Linux machine. The UML instances are independent of each other, so, to users, it would appear that they are logging into and utilizing their own separate Linux machine. In fact, as of this writing, there are numerous commercial UML hosting service providers available on the Internet [14].

How could an attacker apply the otherwise virtuous UML in a subversive role, undermining the existing kernel on the machine? Consider the attack shown in Figure 8.18. The bad guy could break into the machine with root privileges, make a copy of the existing file system, including the kernel, all applications, and user data, and load them into a guest UML instance on the machine. Then, after starting this UML guest containing a copy of the original system, the attacker could install a new evil kernel on the underlying host system. All users and administrators logging into the machine would be unwittingly accessing the UML instance, and not the "real" underlying operating system, controlled by the attacker. The attacker, meanwhile, could run all sorts of nasty processes on the host operating system, which the users inside of the UML instance would not be able to notice. In essence, this attack works like a reverse honeypot. Instead of trapping attackers inside a jail without their knowing it, which normal honeypots do, this type of attack traps system administrators and users in a jail.

Figure 8.18. Employing User Mode Linux to confine legitimate users

inside a prison.

To successfully implement this subterfuge, the attacker would need to ensure that the UML instance with the image of the real system is restarted at each and every reboot of the overall host operating system. This isn't a major problem, as the various scripts and programs associated with running UML can be set as startup scripts on the host operating system. Of course, the rather complex process of booting up the actual (but evil) kernel, followed by initiating a UML session with its own virtuous kernel wrapped inside, might get noticed by a suspicious system administrator watching messages from the startup scripts during the boot process. However, the attacker could carefully disguise the actual boot-up messages and the UML initiation messages so that the system appears to be normal during the boot-up phase.

By deploying UML on a victim machine, attackers turn the whole system into their playground, confining normal users and administrators into a small UML prison tucked away in a corner of the system. The real concern here, of course, is that the users and administrators have no idea that they are in a prison. UML becomes a cone of silence wrapped around legitimate users and administrators. With UML going about its business, the system looks normal to them. Their normal kernel is running, all of their files are still on the hard drive, and programs run the same way they did before the attack occurred. The victims are blissfully ignorant of their UML-induced cage.

The Kernel Mode Linux Project

With UML, we've just seen the power of running an entire Linux kernel inside a user-mode process. There's another technique that sort of reverses this concept, which can again be exploited in a kernel-level attack. Instead of running an entire Linux kernel inside a user-mode process, how about simply running a user-mode process in kernel mode itself? That is, we could run a user-mode process, but have it execute in Ring 0 of the CPU, giving it full access to all kernel data structures. As with UML, there's even an open-source project devoted to this concept, called, appropriately enough, Kernel Mode Linux (KML).

KML is the brainchild of Toshiyuki Maeda, and is freely available at *http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/*. To deploy KML, an administrator (or attacker) must compile a special kernel with KML support. Implementing KML isn't a major feat of coding, however. The KML implementer just needs to download Maeda's code, and answer "Y" in the kernel-build script when prompted whether to insert KML functionality. Then, once the KML-capable kernel is installed on a system, a special directory called /trusted is created. Any binary executable located in /trusted will run in kernel mode on the machine. So, for example, if you want to run the `ls` command inside of kernel mode, you'd just copy `ls` into /trusted, and then

execute /trusted/ls. The `ls` command now runs, but this time in kernel mode. Actually, the `ls` command, while executing, is a separate process, not grafted into the kernel memory. However, it runs with all of the permissions of the kernel, existing in Ring 0, not Ring 3. Because `ls` is fairly well-behaved, it won't hurt the system. However, we've just employed KML to cross the Rubicon from Ring 3 to Ring 0, as shown in Figure 8.19.

Figure 8.19. Using KML to run a process in kernel mode.

Like UML, KML wasn't created with evil intentions. It was designed so that a software developer or administrator could run well-behaved programs in the kernel mode to improve efficiency and performance. On a normal (non-KML) Linux system, whenever a user-mode process makes a system call (which happens all the time), a major context switch occurs. When the flow of execution transitions from Ring 3 to Ring 0, several user-mode data structures have to be saved in memory, and new kernel-mode data needs to be loaded. This transition takes time and CPU cycles. Maeda created KML for applications with very high performance demands to avoid the context switch.

Of course, running programs designed to execute as user-mode processes in kernel mode can be very dangerous. The process could accidentally (or purposely) alter data structures inside the kernel, making the system highly unstable, or instantly crashing it. Therefore, KML isn't for the faint of heart, nor is it appropriate in the vast majority of production environments. Still, for experimental systems and playing with running kernels, KML is a fascinating project.

Of course, an attacker could use KML in a kernel-level attack. Suppose a bad guy takes over your machine. The attacker could replace your kernel or patch it so that it now supports KML. Then, the attacker could write a malicious program that runs a process in kernel mode, utilizing KML to make the jump from Ring 3 to Ring 0. Once running, the malicious process would search for and alter the system call table and system call code to replace them with the attacker's own software. The attacker's software would implement a kernel-mode RootKit, with all of the hiding and execution redirection tricks we've seen with other forms of kernel-level malware. This type of attack is illustrated in Figure 8.20. Although this type of attack hasn't yet been reported in the wild, it is certainly possible.

Figure 8.20. Using KML to attack the kernel, altering the system call table and system call code.

Defending the Linux Kernel

So, as we've seen, there are a myriad of possibilities for attacking the Linux kernel, all of which result in complete domination of the victim machine by a nefarious attacker. How can you defend against such attacks? Well, as with the user-mode RootKits we discussed in Chapter 7, the defenses fall into three different categories: Prevention, Detection, and Response. Let's explore the defenses available in each of these categories.

Kernel Mode RootKit Prevention on Linux

> *An ounce of prevention is worth a pound of cure.*
>
> —Anonymous

Just like the user-mode RootKits we discussed in the last chapter, all of the kernel manipulation attacks we've discussed in this chapter require the bad guy to obtain root-level permissions on the victim machine first, before installing any kernel-manipulation code. Therefore, you can stop would-be kernel-altering attackers in their tracks by preventing them from getting superuser privileges on your machines in the first place. Vigorously apply the defenses we've discussed throughout this book. Use tools like Bastille Linux, which we discussed in more detail in Chapter 7, to harden your system configuration. Disable unneeded services and make sure you rapidly deploy patches to your sensitive systems. Older versions of the Linux kernel are particularly susceptible to kernel attacks, and they have widely known vulnerabilities that an attacker could exploit, such as the ptrace flaw that plagued Linux kernel version 2.4 in 2002 and 2003 [15]. By keeping your system, and especially the kernel, patched and up to date, you won't have such vulnerabilities acting as entry points for the bad guys. Furthermore, educate users about the need to secure their systems and not run untrusted code. With kernel-mode RootKits on the loose, it's more important now than ever to run a tight ship when configuring and maintaining your systems.

In addition to configuring your systems securely and patching them, you might want to consider deploying Linux kernels that do not support loadable kernel modules on your most sensitive systems, such as your publicly accessible Web, e-mail, DNS, and firewall systems. You likely

don't need kernel module support on such machines, as patching the kernel on a live production system with a module is very dangerous and could crash the system. When was the last time you inserted modules into your critical production Web, e-mail, DNS, or firewall servers? Probably never. Following directions readily available on the Internet [10], or using Bill Stearns' kernel building script [16], you can easily create a custom Linux kernel that has all the functionality that you require built in, without supporting kernel modules.

Of course, as we saw earlier in the chapter, bad guys could go after /dev/kmem directly and poison your kernel even if module support isn't available. Still, by just getting rid of loadable kernel modules, you've raised the bar against the rank-and-file script kiddies who rely solely on loadable kernel modules for their attacks. Instead of allowing an attacker to completely hose your kernel with a simple `insmod` command, you've increased security so that your adversaries will have to work somewhat harder to undermine your kernel. We should note that some people use the term *monolithic* to refer to a kernel without module support, although hard-core kernel developers blanche at using this word for this concept. They call such kernels *non-modular*, reserving the word *monolithic* to indicate a kernel that supports numerous features in kernel mode, instead of pushing almost all capabilities into user space [17].

A related approach is to utilize a kernel that was specifically modified to prohibit a module's ability to alter the system call table. In particular, some versions of the Linux kernel do not export the system call table [18]. Exporting of the system call table allows modules to read and even update this crucial data structure in the first place. Without this export, loadable kernel modules cannot alter the system call table, foiling some kernel-mode RootKits. In particular, RedHat grafted this feature into the version of the kernel included in RedHat 8.0 and 9.0, and Linus Torvalds built it into the development kernel version 2.5.41. For this reason, the stock version of Adore and most other module-based kernel-mode RootKits will not work on RedHat 8.0 and 9.0. That's pretty nice, as Adore is the most popular kernel-mode RootKit in use today. Outside of recent RedHat versions and experimental kernels, though, this feature hasn't been widely included in other kernel versions as of this writing. Also, it's important to note that, even with this feature, the /dev/kmem-style RootKits, like SucKIT, will still function appropriately. To make Adore or KIS work on these systems, an attacker would have to modify the RootKit code to take advantage of /dev/kmem, or add the system call table export feature that RedHat removed back into the kernel. As you'd no doubt guess, there is even freely available code for re-adding the system call table export, called addsyms, available at *http://xenion.antifork.org/files.html*.

After hardening your machine and removing kernel module support, you might want to turn to some freely available tools to help limit attackers' access to your systems. One noteworthy free tool for identifying and controlling the flow of action between user mode and kernel mode is Systrace by Niels Provos, available at *www.citi.umich.edu/u/provos/systrace/*. Don't get confused by the name Systrace. Earlier in this chapter, we ran a tool called strace, which merely shows the system calls made by an application. Systrace goes far beyond simple strace. Once installed on Linux, FreeBSD, and Mac OS X machines, Systrace tracks and limits the system calls that individual applications can make.

So, using Systrace, you can run an application under normal, controlled circumstances and record which system calls it makes. For example, you could run your Web server on a test machine and log all of its system call activity. You now have a known set of system calls required by the intact Web server. Now, you can use Systrace to limit that application so that it cannot make any other system calls on the machine. In a sense, you've locked the application so that it can only access the normal set of kernel functionality that it requires to do its job. If it tries to make other system calls, such as those calls associated with inserting a module into the kernel, Systrace will stop the activity and return a failure notice for that system call. In this way, you can isolate various programs inside of little cages, where they can only execute the system calls they normally require. If Systrace observes an application trying to run other system calls, it'll alert you about a misbehaving application, possibly due to an attacker's undermining that program.

In addition to Systrace, you could also turn to security-enhancing loadable kernel modules. Just

as the bad guys employ evil kernel modules to undermine the Linux kernel, system administrators and security personnel can utilize wholesome modules to buttress the overall security of a Linux system. Of course, if you've removed module support from your kernels, you'll have to compile in any code offered by a security-related kernel module directly into your kernel. One worthwhile project that focuses on increasing the overall security of Linux, starting with the kernel, is the Linux Security Module (LSM) initiative, described in detail at *http://lsm.immunix.org*. It's important to note that LSM doesn't stop evil kernel modules directly. Instead, LSM technology makes the overall system more secure, closing various avenues that attackers typically employ to break into root. By denying them root access, LSM improves security so that the bad guys cannot modify the kernel or otherwise compromise the machine.

Let's look at the origins of LSM to get a feel for its design goals. Back in March 2001, the U.S. National Security Agency (NSA) delivered a presentation on its Security Enhanced Linux (SELinux) project in front of the Linux Kernel Summit, an annual gathering of hard-core kernel developers. Prior to the presentation, the NSA publicly released a version of the Linux kernel that includes far more detailed security controls, applying mandatory access controls to critical system components and functionality. "Normal" Linux is built around discretionary access controls, which allow users and administrators to apply permissions to various system files at their own discretion. Under this paradigm, a user or administrator can purposely or accidentally weaken the security of a system by changing the read, write, and execute permissions on various critical files. With mandatory access controls, such as those implemented in SELinux, access to certain critical system components, including data structures and files associated with the kernel, is controlled by default and cannot be altered by a user or administrator. That's why these controls are mandatory and not discretionary. In a sense, many security settings, like the read, write, and execute permission of some critical files, are hard coded into the machine. Therefore, if the mandatory access controls are implemented properly, the kernel and other pieces of the operating system are less exposed to manipulation by a bad guy. Based on the NSA presentation at the 2001 Linux Kernel Summit, Linus Torvalds and other kernel developers began to discuss how to incorporate some of the SELinux ideas into the overall Linux kernel, and the LSM project was born.

Mandatory access controls are just one possible security feature that could be implemented via LSM, but other options are certainly available. In fact, LSM is an architectural framework for plugging all kinds of security features into the Linux kernel. The LSM project is currently spearheaded by Immunix, a company that creates a commercialized hardened version of Linux. In essence, LSM adds security hooks to the Linux 2.4 and 2.5 kernels. These hooks allow a loadable kernel module to make security decisions about what should and shouldn't be allowed. LSM doesn't specify what these security decisions should be. It just provides an interface for connecting the decision-making security logic with the kernel itself. Whereas evil loadable kernel module RootKits undermine the kernel, LSM lets modules be applied to enhance the security of the overall system, thereby preventing manipulation by the bad guys.

In plain old vanilla Linux, a base set of security controls is built into the kernel itself. However, these controls are a one-size-fits-all approach that Linux inherited from UNIX systems of decades ago. These default controls focus on access to files, specifying who can read, write, and execute each file on the file system. With LSM, a kernel module can specify all kinds of different or additional access controls, specifying, for example, files that should be strictly off limits or even data structures in the kernel that shouldn't be altered.

LSM provides the overall framework and interface for writing these security modules. A variety of different groups have created LSM-compatible modules that increase the built-in security of Linux. After all, a security specification is nice, but only implementations make it real and usable. Table 8.3 includes a variety of free, open-source LSM implementations that improve the overall security of a Linux machine. Each of these modules can boost the underlying security of Linux to prevent a bad guy from getting root and mounting a kernel-mode RootKit attack. It's crucial to note, however, that use of any of these modules fundamentally changes the security controls of your Linux system. Therefore, it's possible that applications installed on a Linux box will break if you install LSM without first carefully configuring and testing the system. Also,

because it changes the underlying access control rules in Linux, an LSM module could complicate administration of the machine. A system administrator fully versed in "normal" Linux could be completely confounded by the security controls introduced by an LSM. Therefore, system administrators and security personnel must gain experience on the specific security features implemented in an LSM before rolling it into production.

Table 8.3. Various LSM Implementations

LSM Name	Location	Purpose
SELinux	*www.nsa.gov/selinux*	This LSM implements a security architecture based on SELinux, created by the NSA. It includes mandatory access controls, as well as role-based access controls, which assign users to different roles and determine their privileges based on their assignments.
Domain and Type Enforcement	*www.cs.wm.edu/~hallyn/dte/*	This module groups processes together into a set of domains. Various files are then assigned an attribute called a *type*. Then, various domains are given controlled and explicit access to specific types.
Openwall LSM	*www.openwall.com/linux*	This module implements several security restrictions, including limits on user access of the /proc file system and nonexecutable process stacks to prevent a variety of buffer overflow attacks.
LIDS	*www.lids.org*	The Linux Intrusion Detection System (LIDS) provides a variety of security features, including: • File protection, locking files so that they cannot be altered, even with root permission • Process protection, to prevent access to critical processes • Fine-grained access control lists • Security alerts for attacks against the kernel • Kernel-level port scanning detection • Restrictions on processes from listening on network ports

Kernel Mode RootKit Detection on Linux

Even with the best defenses, an attacker still might find a hole in your armor and install a kernel-mode RootKit. Once a kernel-mode RootKit is installed, we cannot fully trust any results from our system. It all comes down to how thoroughly the kernel-mode RootKit software hides itself and how carefully the attacker configures it. Although detection can be a major challenge, we do have numerous mechanisms at our disposal to discover traces of kernel-mode RootKits on our systems.

First, look for suspicious network activity coming from a system. Even though local activity is hidden from system administrators, a network-based IDS can observe attack packets coming from a machine infected with a kernel-mode RootKit as the attacker tries to take over other systems across the network. Furthermore, if the attacker plants a backdoor listening on a TCP or UDP port, a port scanner such as Fyodor's Nmap (which is free at *www.insecure.org*) can remotely detect the listening ports, even though they are hidden from all local users and administrators. Also, look for unexpected reboots of your systems. Although loadable kernel module and /dev/kmem alterations don't require a reboot, the other methods of kernel manipulation we've discussed (overwriting the kernel image, using UML, and installing KML) do require the attacker to reboot the system. Although an unexpected reboot is no guarantee that an attacker has taken over your box and installed one of these nasties, it is an indication that something might be out of the ordinary. You should take a deeper look, using the response tools we'll discuss in this section, if your system reboots itself from time to time.

Additionally, you should still use file integrity checking tools, such as Tripwire, AIDE, and the related programs that we discussed in Chapter 7. A thorough bad guy will configure the manipulated kernel with execution redirection and other alterations that lie to the file integrity checker about all file changes on the system. If the attackers very carefully cover all of their tracks, they can fool a file integrity checker. However, a less careful attacker might forget to configure the kernel-mode RootKit to hide alterations to one or two sensitive system files. Even a single mistake in the file-hiding configuration of the kernel-mode RootKit by the bad guys could expose them to detection by your file integrity checker. Therefore, file integrity checking tools remain very valuable, even though a kernel-mode RootKit can foil them if the attacker is super careful. I'd rather not depend solely on the attackers' making mistakes to discover their treachery, but you better believe I'll be sure to take thorough advantage of their errors. Deploying file integrity checking tools on all of my sensitive systems lets me prepare for such circumstances.

Another tool that we discussed in Chapter 7 can be useful in detecting these kernel-mode attacks, namely chkrootkit. By looking for various system anomalies introduced by kernel-mode RootKits, the free chkrootkit tool can detect Adore, SucKIT, and several other kernel-mode RootKits. For you fans of *The Matrix*, chkrootkit is really looking for glitches in the Matrix. In the movie, glitches in the Matrix occur when the bad guys start changing things, creating a déjà vu. Similarly, with a kernel-mode RootKit, an inconsistency in the system's appearance could be an indication that something foul has been installed. The scripts included in chkrootkit perform tests that can be used to catch the kernel in a lie about the existence of certain files and directories, network interface promiscuous mode, and other issues that kernel-mode RootKits generally fib about.

One of the ways that chkrootkit finds kernel-mode RootKits is by looking for inconsistencies in the directory structure when a file or directory is hidden. Each directory in the file system has a link count, which indicates the number of other directories and files that a given directory is connected to in the file system structure. For each directory, this link count should be two more than the number of files in the directory. That way, the directory would have one link for each file, plus one for the parent directory (..) and one for itself (.). Many kernel-mode RootKits, such as Adore, hide files and directories without manipulating the link count of the parent directory. Chkrootkit combs through the entire directory structure, counting the number of files and directories that it can see inside each directory and comparing it to the link count. If it finds a discrepancy, chkrootkit prints a message indicating that there might very well be directories that are hidden by a kernel-mode RootKit. Unfortunately, as of this writing, the current version of chkrootkit cannot detect KIS, which manipulates even the link count associated with hidden files and directories. KIS is smart enough not to introduce that glitch into the Matrix.

Beyond general RootKit detectors like file integrity checkers and chkrootkit, there are also tools that specialize in detecting the behavior most often associated with kernel-mode RootKits, such as altering the system call table or loading modules. In particular, a tool called KSTAT (an awkward acronym that stands for Kernel Security Therapy Anti-Trolls) is freely downloadable from *www.s0ftpj.org/en/tools.html*. On Linux 2.4 kernels, KSTAT helps find and uninstall kernel-mode RootKits. For detection, KSTAT looks for changes to the system call table. It'll even

scan /dev/kmem to look for the memory locations associated with all system calls, and compare these results with the information in the System.map file. If it finds a discrepancy, KSTAT warns a system administrator that someone has altered the system call table. Just as the bad guys look through /dev/kmem to break our systems with tools like SucKIT, we can use KSTAT to look through /dev/kmem to find their attacks.

Additionally, like Systrace, the KSTAT tool can also create a list of fingerprints for the system calls used by various critical programs, such as a Web or mail server program. If any of these system calls are altered, or additional system calls are invoked by these programs, KSTAT can warn an administrator that something foul might be occurring.

In addition to KSTAT, another free project that looks for manipulation of the system call table on Linux is called Syscall Sentry, written by Keith J. Jones. Syscall Sentry is a loadable kernel module that is typically inserted during system startup. If an attacker inserts a module that alters the system call table, the Syscall Sentry module detects the alteration, logs the event, and alerts the system administrator about this anomalous activity.

Beyond Linux, other tools provide system call table monitoring for other varieties of UNIX. In particular, a tool named KSEC provides such services on FreeBSD and OpenBSD, available at *www.s0ftpj.org/tools/ksec.tgz*. On Solaris systems, you can use a tool called Listsyscalls by Bruce M. Simpson, available at *www.packetstormsecurity.org*. Both KSEC and Listsyscalls provide very similar functionality to that offered to Linux users through KSTAT and Syscall Sentry.

Kernel Mode RootKit Response on Linux

Now, suppose these detection mechanisms or even your intuition tells you that some dastardly attacker has installed a kernel-mode RootKit on your machine. When you investigate to determine what is really happening on your system, you cannot trust anything that comes out of the kernel. Any analysis tool that you run on the system might be fooled by the existing kernel, and therefore cannot be trusted. You are in a fantasy world of the attacker's making, but you need answers about the real state of your system. So how can you cope?

Again, I refer you to the tools we discussed in Chapter 7. Do you remember how we said that to respond to a RootKit attack, you should use a bootable CD-ROM that includes a Linux operating system? We even discussed using William Salusky's FIRE and Karl Knopper's Knoppix distributions, which include specific customizations for security and computer forensics investigations. Well, back in Chapter 7, I specifically included the word *bootable* in our description of FIRE and Knoppix because that very characteristic would become helpful in this chapter. An investigator can insert the FIRE or Knoppix CD-ROM in a potentially compromised machine, and boot from the CD-ROM. As the system shuts down, the potentially evil, deceiving kernel will stop running. When the system reboots, the trusted kernel from FIRE or Knoppix will be loaded into memory. Because this new kernel is grabbed from the CD-ROM, an investigator can use it to read the victim machine's file system with more trustworthy results than one can get from an evil kernel. Therefore, after booting from the CD-ROM, the investigator can run a file integrity checker (built into the CD-ROM, of course) to look for changes to critical files on the hard drive.