UNIX User-Mode RootKits

Girl, you know it's true!

—From the 1989 hit song "Girl You Know It's True" by Milli Vanilli, the pop music duo who revealed that they did not actually sing any of their hit songs and instead lip-synched their way to the top of the charts

RootKits were originally created for UNIX systems. UNIX environments are very well-suited to RootKit attacks, given their reliance on the root account. The root account is sometimes called the superuser account, given that it has all power on a typical UNIX system. From a root-level account, an attacker can completely reconfigure the box, overwrite existing applications, change logs, and view any data stored unencrypted on the target machine. Additionally, UNIX administrators rely very heavily on a handful of command-line programs to determine the status of their systems. With root-level access, an attacker has all of the permissions required to replace these command-line programs, altering the system to suit the attacker's needs. Given the power of root and reliance on individual command-line tools, UNIX is very fertile ground for RootKits.

The first very powerful UNIX RootKits were discovered in the early 1990s, replacing a few executables on victim UNIX boxes. They primarily targeted SunOS, but were rapidly ported to other UNIX systems popular then, including DEC Ultrix, HP-UX, and others. Given their inherent usefulness for the bad guys, these vintage 1990 RootKits were shared only among a handful of the most elite attackers. To prevent system administrators from deploying defenses against these RootKits, the attackers kept them very close to the vest in the early years. They were distributed among bulletin board systems, Internet Relay Chat, and a handful of esoteric FTP sites on the Internet.

Today, however, anyone can download a very powerful RootKit from a variety of freely available Web sites we'll discuss throughout this chapter. Also, today's RootKits are even more powerful than the RootKits of yesteryear, transforming numerous programs on a system to custom tailor the machine for the attacker. The tools bundled together in most user-mode RootKits on UNIX can be broken into five different areas:

- *Binary replacements that provide backdoor access.* These tools are the heart of the user-mode UNIX RootKit. By overwriting various programs and services used to access the machine, an attacker uses these replacements to log in to the system through various backdoors. When the backdoors are used, the attacker is immediately granted root privileges on the target system.

- *Binary replacements to hide the attacker.* These tools overwrite existing binaries on the system, replacing them with Trojan horse versions that let an attacker hide. These new binaries lie to users and administrators about the attacker's files, processes, and network usage on the victim machine.

- *Other tools for hiding that don't replace binary programs.* These programs let attackers alter the system to hide their nefarious activities, although they don't replace commands. Instead, they support the RootKit by including features such as altering the last modification time of a program to disguise the alterations caused by installing the RootKit. Others even remove evidence of particular account usage on the box. Still others let the attacker edit logs.

- *Additional odds and ends.* Many UNIX RootKits also include various other tools useful to an attacker on the target system. Some RootKits come with a built-in sniffer, for

gathering traffic from the LAN, which might include valuable clear-text user IDs and passwords. Backdoor shell listeners, like the tools we covered in Chapter 5, are another popular option bundled with RootKits.

- *Installation script.* This program opens up the other bundled RootKit tools, compiles them if necessary, and moves them to the appropriate location. Rather than manually pushing every binary in place and handcrafting it to fit properly in the system, automated RootKit installation scripts run through the entire installation process, which usually requires a mere 10 seconds or less. After the replacement programs are loaded in the proper places, this script resets the last modification date and might even compress or pad portions of the binary replacements so that they are all the same length as the original programs.

If you think about each of these different categories all bundled together in a single package, you can see that RootKits really are kits, handy collections of tools used to transform a system at the attacker's whim. An attacker wielding a user-mode UNIX RootKit is kind of like a doctor making a house call. When doctors show up at a house call, they carry a little black bag with a variety of tools they'll need to alter their patient's bodies. It's impractical and unnecessary for the doctor to bring an entire operating room, when a single black bag can hold everything needed by the doctor. When breaking into a system, the attacker brings along a RootKit, which includes a whole host of useful individual tools for manipulating the system. The attacker doesn't need to rebuild the entire operating system, when only a few select tools nicely bundled together in a RootKit will accomplish the goal. Of course, this analogy does break down, in that the doctor's goal is to improve the health of a patient, whereas the attacker's goal for the target computer is quite the opposite.

The computer underground has created a huge variety of different types of RootKits for all flavors of UNIX systems, including Linux, BSD, Solaris, HP-UX, AIX, and others. These RootKits have a variety of quirky and exotic names, including LRK, URK, T0rnkit, Illogic, SK, ZK, and even Aquatica. Although each RootKit varies in the particulars of what it replaces and how it is configured, all user-mode UNIX RootKits follow the same general themes and methodologies. Therefore, we can learn a lot about how to defend against such attacks by studying a handful of the more powerful and widely used RootKits. To get a better feel for how user-mode UNIX RootKits alter a target system, let's look at a few specimens in more detail, namely the Linux RootKit (LRK) family, the Universal RootKit (URK), and some particularly interesting RootKit-like tools called RunEFS and the Defiler's Toolkit.

LRK Family

One of the most widely employed user-mode RootKits today, and indeed over the past several years, is the Linux RootKit family of tools. I refer to LRK as a family, because it includes several generations of RootKits, each based on continuous improvement over previous incarnations. The firstborn of the family, named LRK1, was released in early 1996 by someone named Ira. A variety of other developers picked up the LRK mantle by adding new features to the kit or improving the capabilities already built in. The development of the LRK family, shown in detail in Table 7.1, is a classic example of software refinement over time, just as we see with legitimate commercial software tools. Based on actual experience gained by using RootKits to attack real-world environments, various software developers with names like Cybernetic and Lord Somer constantly improved the tool, releasing LRK2 through LRK5. There are even reports of an LRK6 release, although it is not yet widely available as of this writing.

Table 7.1. Development of the Linux RootKit (LRK) Family through Successive Releases

RootKit Tool Category	RootKit Component	Purpose of Program	Linux RootKit				
			1	2	3	4	5

Binary replacements with backdoor	`login`	Authenticate users and log them in	X	X	X	X	X
	`rshd`	Allow remote shell access		X	X	X	X
	`chfn`	Alter a user's full name or phone number in the GECOS field		X	X	X	X
	`chsh`	Change a users' default shell		X	X	X	X
	`inetd`	Listen on the network for services such as Telnet and FTP		X	X	X	X
	`passwd`	Change a password		X	X	X	X
	`tcpd`	Filter connections for certain applications using a TCP wrapper			X	X	X
	`sshd`	Access the machine using an encrypted session					X
	`su`	Change user accounts					X
Binary replacements that hide attacker	`netstat`	Look at network statistics	X	X	X	X	X
	`ps`	Look at running processes	X	X	X	X	X
	`top`	Look at the top running processes consuming the most CPU cycles	X	X	X	X	X
	`ls`	List files		X	X	X	X
	`du`	Look at disk usage		X	X	X	X
	`ifconfig`	Look at network interface configuration		X	X	X	X
	`syslogd`	Record system logs		X	X	X	X
	`killall`	Terminate processes given a process name				X	X
	`crontab`	Schedule programs to run				X	X
	`pidof`	Find the process ID of a running program				X	X
	`find`	Locate a file				X	X
Other tools for hiding (these support the RootKit, but do not replace existing commands)	`fix`	Pad a file and change file access and update dates	X	X	X	X	X
	`zap2`	Delete accounting data		X	X	X	X
	`wted`	Edit accounting data		X	X	X	X
	`lled`	Edit the last login information	X	X			
Other odds and ends (these	`bindshell`	Grant backdoor shell		X	X	X	X

support the RootKit, but do not replace existing commands)		access					
	linsniffer	Sniff data from the network		X	X	X	X
	sniffit	Sniff data from the network		X			
	sniffchk	Verify that sniffer is running				X	X
Installation script	makefile	Install the RootKit	X	X	X	X	X

Table 7.1 highlights three important aspects of the LRK family. First, LRK1 was quite powerful right out of the gate, including replacements for several important Linux programs like `login`, `netstat`, `ps`, and `top`. With the full source code of all Linux commands publicly available, attackers were able to easily graft RootKit functionality directly into the operating system, without having to reverse-engineer any functionality. Implementing a RootKit is far easier with access to the source code, because the attacker can reuse a great deal of the existing program code and just sprinkle in some RootKit features. Furthermore, numerous developers have added functionality to the baseline tool over the years, morphing it considerably. Finally, the family's continuous improvement over time has made it even more formidable. As a result of this continual evolution, the LRK family is perhaps the most full-featured user-mode RootKit available today. To get a better feel for its capabilities, let's analyze the various components built into the latest versions of LRK.

LRK Binary Replacements That Provide Backdoor Access

The LRK family includes a variety of executables that replace existing programs associated with logging in and using accounts to implement backdoor access to the machine. Some of these backdoors provide remote root-level access across the network. Others require an attacker to log in to a nonroot account first, and then let that attacker escalate privileges to root level by running some local command and providing a backdoor password. Each of the RootKit backdoor components included in LRK is illustrated in Figure 7.2.

Figure 7.2. LRK binary replacements providing backdoor access.

One of the most fundamental of these backdoor replacements is the familiar `login` program, built into LRK since the heady days of the original LRK1. A normal `login` program asks users for their user IDs and passwords when they log in at the system console or via Telnet. The LRK Trojan `login` program replacement acts the same way, but with an added bonus. If someone types in a special backdoor password, that user is automatically given root-level control of the

system. With the magic password, the attacker can log in directly as root. Because this backdoor password is built into the executable file, the attacker can use it again and again, even if an administrator changes the real root password. The backdoor root password remains the same.

The LRK `login` substitute also defeats security controls and logging on the victim machine. As a security precaution, many system administrators configure their UNIX systems to prevent direct log in by users as root. On such machines, administrators need to log in to the system first as a nonroot user, and then change to a root-level account, using the `su` command. By preventing direct login as root, an attacker would not be able to remotely guess password after password, attempting to log in as root. Forcing users to rely on `su` to get root access also fosters accountability for administrators, as their actions can be tracked to individual user accounts that have employed the `su` command. However, such functionality is mighty inconvenient for attackers. Therefore, the LRK version of the `login` program lets attackers log in directly as root by using the account name "rewt". Note that the account rewt and the backdoor password are not stored in the normal account and password files on Linux machines (/etc/passwd and /etc/shadow). Instead, the account and password are built directly into the executable programs themselves. The password is configured when the RootKit is compiled, but several defaults are often used. Early versions of the LRK family use a default password of *lrkr0x*, which apparently stands for Linux RootKit Rocks. Other versions use the word *satori* as their default password. Of course, most attackers alter the defaults, providing their own password.

The rshd and sshd backdoor replacements included with LRK work in a similar fashion to the login program. When a user is prompted for a password for remote shell (RSH) or secure shell (SSH), the attacker provides the backdoor password to gain remote root-level access. The sshd backdoor also includes another feature very useful to an attacker: All shell traffic sent across the network is encrypted. That way, if a suspicious system administrator tries to monitor the connection with a sniffer, the attacker's commands will be invisible inside the encrypted session. Rounding out the remote access backdoors in LRK, the inetd and tcpd replacements include a backdoor listener that provides a remote shell on any TCP or UDP port of the attacker's choosing. By default, this LRK backdoor listens on TCP port 5002.

Beyond these remote access backdoors, LRK also includes a variety of local backdoors that allow an attacker logged into the box with a nonroot account to jump instantly to root privileges. When logged into any account, the attacker can invoke the change finger command, `chfn`, which is normally used to alter a user's name or phone number stored in the so-called GECOS field of the /etc/passwd file. With the LRK version of `chfn`, the attacker can provide the backdoor password instead of new user information for instant root access. Likewise, LRK includes a new version of the change shell command, normally used to change which command-line shell a user is assigned when logging in. By typing the LRK backdoor password in place of a shell name, the attacker gets root. Also, a replacement for the `passwd` program accepts the backdoor password, in addition to its normal function of allowing users to change their passwords. Finally, the `su` command also includes backdoor password functionality. Normally, `su` lets users change their login privileges to those of another user, if they know that user's password or they are operating as root. By providing the RootKit version of `su` with the backdoor password, the attacker is immediately given root access. Whew! That's a lot of backdoors. All told, LRK includes at least nine of them, lacing the system with openings for the bad guy to access.

LRK Binary Replacements That Hide the Attacker

In addition to backdoors, LRK replaces several programs that system administrators typically use to determine the status of their systems, including tools for managing running programs, network settings, the file system, and system logs. These replacements are illustrated in Figure 7.3. In essence, the attacker alters each of these commands so that they lie to the system administrator. These commands act as the eyes and ears of the system administrator. With altered eyes and ears, the administrator cannot determine the true state of the system. To understand how these different hiding mechanisms work, think about what the bad guys need

from a RootKit. After taking over a target system and installing a RootKit, the attacker will likely run some programs on that machine. These programs could be additional backdoor listeners, other attack tools used to scan for more vulnerable systems, or individual exploits used to take over more targets. Beyond the RootKit itself, these additional attacker programs on the system will require the attacker to:

- *Create running processes.* The attacker's tools will create processes on the system, which could be detected or even killed by a nosy system administrator.
- *Use the network.* The attacker might run a sniffer to capture user IDs and passwords, as well as a backdoor port listener to provide remote shell or GUI access. Unless the sniffer is hidden, administrators could discover that the interface is in promiscuous mode, tipping them off that a sniffer is in use. Likewise, unusual local port listeners could trigger an investigation.
- *Create directories and files.* Attackers usually write various program and configuration files to the victim machine's file system. Also, bad guys often store stolen information, such as password files, pirated software, confidential documents, and pornography on the victim machine. If they are not hidden, these files could reveal the attacker's presence.
- *Generate logs.* As the attacker manipulates the system, normal logging will show several incriminating events. To remain stealthy, the attacker needs to make sure these events never show up in the system logs.

Figure 7.3. LRK binary replacements that hide an attacker on a system.

Without the attacker's intervention, a diligent system administrator might notice each of these activities. To address this situation, LRK comes to the attacker's rescue by including replacements for various tools used by system administrators to find these anomalies.

First, LRK includes several replacements that hide running processes on the machine. To use this capability, the attacker must include the name of the process to be hidden in the file /dev/ptyp. On a stock Linux system, there are files called /dev/ptyp0, /dev/ptyp1, /dev/ptyp2, and so on using hexadecimal notation up to /dev/ptypf, but there aren't typically any real files named simply /dev/ptyp. Depending on the configuration of this RootKit file, various commands on the system can hide processes based on their full name, substrings of the process name, the user terminal (known as the tty) that the process is attached to, or even all root-level processes. Then, LRK replaces the `ps`, `top`, and `pidof` commands, all of which are used to determine which processes are actively running on a system. Furthermore, LRK overwrites the `killall` command so that the attacker's hidden processes cannot be killed using the command. That way, even if the administrator is

miraculously able to discover it, the attacker's running process cannot be stopped using the `killall` command. It's important to note, that although the attacker's processes are hidden by the `ps`, `top`, and `pidof` commands, they will still be visible inside of the /proc directory, a component of the file system created by the kernel to show the status of all running processes and the kernel itself. In Chapter 8, we'll explore how kernel-mode RootKits hide even the evidence shown in /proc.

LRK includes a modified version of crontab, which is used to start various programs at specific times. By default, the altered version of crontab automatically activates the program names stored by the attacker in the file /dev/hda02. Again, Linux systems normally include files called /dev/hda1, /dev/hda2, and so on to indicate portions of the hard drive, but on a stock Linux machine, there are no files called /dev/hda02. The zero makes it different. Whereas the normal crontab's configuration is available for the system administrator to see, this alternative crontab uses this additional hidden configuration file.

Beyond process-related hiding, LRK also supports hiding network usage. On some older Linux systems, the `ifconfig` command shows whether the network interface is in promiscuous mode, gathering all traffic from the LAN. LRK replaces `ifconfig` so that it never shows promiscuous mode, thereby disguising sniffers. Additionally, administrators frequently use the `netstat` command to show which TCP and UDP ports are listening for traffic. The LRK version of `netstat` shows all port usage, except those ports configured by the attacker in the file /dev/ptyq. As with the /dev/ptyp file, /dev/ptyq isn't normally included on a system.
Only /dev/ptyq0, /dev/ptyq1, and so on up to /dev/ptyqf should be present. By default, the LRK `netstat` hides TCP and UDP port 31337, although the attacker can configure the system to hide any other additional ports.

LRK really shines in its ability to hide files in the file system. The attacker creates the file /dev/ptyr, which contains a list of files to be hidden. The `ls` command, normally used to show the listing of a directory, will omit from its output any files that are hidden. Similarly, the `find` command, used to search for files, won't be able to find any of the hidden entries. Finally, the `du` command, which shows the disk usage of the hard drive, will omit the space taken up by the attacker's hidden files. With each of these replacements, finding the attacker's tools on the system could prove quite difficult for a system administrator. It's important to note, however, that by default, the `ls` command included in LRK will show all files, including the hidden ones, if it is invoked with the "minus slash" flag, as in "`ls -/`". Attackers can turn off this default "`ls -/`" behavior, but many of them leave it on so that they can find their own files hidden on a machine. There are few things worse for attackers than taking over a system, installing a bunch of backdoors, and then cluelessly groping around, trying to guess the location of all of the stuff they've just hidden. Hiding can be a two-edge sword, confusing the attackers too. The minus slash option eliminates the need for the attacker to guess where all of the hidden files are located.

Finally, LRK replaces the syslog daemon (syslogd), the program that is used to record all logs on the system. The LRK version of syslogd will not record any log entries that contain a string that matches the contents of the attacker's configuration file, /dev/ptys. Attackers might enter their own source IP address in that file, so that all log events related to their source machine will be omitted from the file. Likewise, specific types of events could be omitted, simply by including an identifying string associated with each type of event in the /dev/ptys file.

Now, take a step back and consider the configuration files associated with each of these types of hiding: /dev/ptyp, /dev/hda02, /dev/ptyq, /dev/ptyr, /dev/ptys. They look like a bunch of gobbledygook that you might expect to be in the innards of your Linux system, right? That's what the attackers want: RootKit configuration files that blend in with the machine. Also, notice that these files are all located in the /dev directory. Normally, this directory contains a comprehensive list of all devices associated with your system, including various components of the hard drive, the CD-ROM drive, user terminals, audio devices, the mouse, and others. For a typical Linux machine, there are an enormous number of rather esoteric names in this directory. On my own Linux system, there are exactly 5,052 entries listed in the /dev directory.

You might have more or less, depending on the configuration of your system. Still, that's a lot of files for an administrator to inspect looking for a few unusual entries.

Furthermore, this directory normally contains several devices with the name pty followed by a character or two. Normally, these devices are associated with open terminals on your system, such as a console or Telnet login. By plopping a few LRK configuration files in the /dev directory and giving them names that match closely with the terminal devices normally included in /dev, the LRK configuration files are nicely camouflaged. Furthermore, after installing the RootKit, the attacker could edit these configuration files so that they themselves are hidden, simply by loading each configuration file name into /dev/ptyr, the list of hidden files.

Suppose, however, the attacker is in a hurry, and forgets to hide these tell-tale configuration files. In this case, you might be able to spot them in your /dev directory, discovering the attacker on your system. Is this a sure-fire way to find LRK in all cases? Sadly, the answer is no. Keep in mind that the LRK family source code is fully available on the Internet. Therefore, even a rushed attacker with very limited programming skills could easily change the location of any of these configuration files by simply editing one line of code per file to make LRK look in a different location for its configuration. Replacing /dev with /bin in the source code requires less than a dozen keystrokes, and would totally relocate the LRK configuration files. Alternatively, an attacker could alter the LRK source code to make it automatically hide these configuration files, wherever they might be located. By simply changing the code so that it automatically hides the configuration files, the attacker won't have to remember to hide them. It's all taken care of in the software itself. So, if files such as /dev/hda02 and /dev/ptys show up on your system, you should certainly investigate the box in more detail. You might have LRK installed by a sloppy attacker. However, you cannot rely solely on this mechanism to identify LRK-infected systems.

Other LRK Hiding Tools

LRK's subterfuge goes beyond just file replacements. The kit also includes a variety of additional tools to hide the attacker's presence, shown in Figure 7.4. As we've seen, when LRK is installed, it changes over a dozen different files located all over the victim machine's file system. On most standard Linux file system implementations, each and every file includes three time-related fields: an indication of the last time the file was accessed (known as the *a_time*), a time when the file was last modified (called the *m_time*), and the creation time of the file (of course, that's the *c_time*). These time-related fields are used to determine the exact date and time when each event occurred. As it overwrites existing files with Trojan horse replacements, the LRK installation process modifies each of these time values. If you ever notice any of these time values mysteriously changing for critical system files on your machine, two things could have occurred. First, it's possible that a system administrator just patched the system, which would update the time fields for each updated file as well. Alternatively, you might just have a RootKit infestation.

Figure 7.4. Other LRK hiding tools.

System with LRK installed

As you might expect, the authors of LRK want to avoid detection by suspicious administrators looking at file creation and update times. To achieve this goal, the authors of LRK included a tool called *fix* in their RootKit. The fix tool resets the a_time, m_time, and c_time for each file replaced by the RootKit to its original, pre-RootKit value. Therefore, an administrator won't be able to detect any changes in these times. It doesn't stop there, though. The fix tool goes even further in disguising changes. Suppose someone uses the noncryptographic checksum algorithm cyclic redundancy check to look for changes to critical system files. For example, an administrator could use the Linux cksum program to determine this value, which uses an algorithm to combine all of the bits in a program together to create the checksum. In addition to modifying the various time fields associated with a file, LRK's fix tool also pads programs so that their noncryptographic checksum matches the original value as well. That's pretty devious! It's important to note, however, that this padding mechanism built into the fix tool only works for the CRC algorithm included in the UNIX cksum command. As we'll see later, the fix tool won't work for cryptographically strong hashing algorithms, such as MD5 or SHA-1.

LRK's other hiding tools let the attacker mask the use of accounts on the target system. On a UNIX system, several files record information about who has logged into the system, including:

- *utmp.* This file stores information about who is currently logged into the machine. The contents of this file are consulted when a user runs the `who` command.
- *wtmp.* This file contains information about every user who has ever logged into the machine.
- *btmp.* This file contains information about bad logins, such as when a user mistypes a password or when an account gets locked out.
- *lastlog.* This file lists the last login date and time for each user, as well as the source address of that login. For some UNIX services, this data is displayed when the user logs in. For example, when you log into a UNIX machine, it might say, "Your last login was at 2:38 AM on May 1, from *www.counterhack.net.*" All of this data is retrieved from the lastlog file.

These files are not stored in plain, old ASCII, and therefore cannot be edited using a standard file editing tool. Instead, attackers require a specialized tool to parse these files and edit them to cover their tracks. Of course, LRK includes just such a tool, called *Zap2*, to edit each of these files. Zap2 blanks out all utmp, wtmp, btmp, and lastlog information for a given user ID selected by the attacker. The user ID is still listed in these files; it's just that the login date and time information for that user is blanked out. Another LRK tool, called *wted* (short for wtmp

editor) goes even further. Rather than just blanking out information associated with users, this tool lets an attacker completely eradicate any information in utmp, wtmp, and btmp associated with a given user or machine on the network. Zap2 removes all login information about the user, but leaves the user's name. Wted can remove every indication that the user has ever logged into the box. In older LRK iterations, a similar tool called *lled* implemented similar capabilities for the lastlog file. However, this helpful command was removed from later packages.

Other LRK Odds and Ends

Although LRK's main focus is on replacing various programs built into the system, it also includes several additional new programs not originally included in the operating system. These miscellaneous tools round out the kit by giving the attacker additional access and information about the system. One of these programs is the very properly named *bindshell*, which creates a backdoor shell listener on a TCP port specified by the attacker. This tool is roughly equivalent to the backdoor listeners we discussed in Chapter 5. An attacker activates the LRK bindshell program, which can listen only on TCP ports. Then, from across the network, the attacker uses Netcat in client mode to connect to the appropriate port where bindshell silently waits.

LRK also includes a sniffer so the attacker can gather sensitive information transmitted in clear text across the local network. The so-called *linsniffer* tool built into LRK automatically grabs user IDs and passwords for FTP and Telnet connections. Linsniffer is very simple, only grabbing account information and dumping it to a file. However, if you boil down what attackers really want from a sniffer, simple little linsniffer addresses their most pressing need. Older members of the LRK family included a more powerful sniffer, called *sniffit*, which includes filtering capabilities for a variety of different services. However, because it was more complex to configure, sniffit was omitted in later releases in favor of the far simpler but more limited linsniffer.

Finally, LRK also includes a program called *sniffchk*. This simple script just tells the attacker whether the sniffer is in fact still running. Remember, the attacker cannot use the `ifconfig` command to detect the sniffer, as `ifconfig` has been altered to disguise promiscuous mode. Furthermore, the sniffer process is usually hidden by the `ps` command. So, if the attacker is concerned that the sniffer might have crashed, or worse yet, been discovered by a pesky system administrator, the sniffchk program comes to the rescue.

LRK Installation Script

So, modern releases of LRK include more than two dozen programs and scripts designed to transform a system to the attacker's specifications. However, compiling, installing, and applying the fix program to each of the components of LRK by hand would likely require hours of work. To avoid this drudgery and speed up the process, LRK includes an easy-to-use installation script, in the form of a makefile. A makefile is merely a recipe for compiling and installing software. The LRK makefile tells the system which ingredients are required for each program, how to compile those ingredients to create the executables, where to put those executables in the file system, and how to disguise them using the fix tool. Of course, for the makefile to work, a compiler needs to be installed on the victim machine. Alternatively, the attacker could compile the RootKit in advance on a similar system, and deploy the precompiled RootKit on a victim machine. Depending on the processor speed and how heavily loaded the system is, the entire installation process could take between 10 seconds and a few minutes. Still, given all the power and complexity included in LRK, that's a short time to total domination of the machine. The really sad part is that there's no real need for the attacker to understand how any of this stuff works! The makefile does all of the installation work, so the attacker can just sit back and use the RootKit itself.

The Universal RootKit (URK)

> *One ring to bring them all and in the darkness bind them.*

—*Lord of the Rings*, J.R.R. Tolkien, published in 1954

As we've seen, LRK replaces some of the guts of a Linux system to bend the machine to the attacker's will, but Linux isn't the only target of RootKit-wielding bad guys. Other UNIX variations succumb to user-mode RootKits all the time. In fact, a casual stroll over to the Packet Storm Security Web site's RootKit folder (at *http://packetstormsecurity.nl/UNIX/penetration/rootkits/*) reveals user-mode RootKits for numerous different UNIX flavors, including BSD, OpenBSD, FreeBSD, Solaris, SunOS, HP-UX, AIX, IRIX, and several other operating system types.

Now, imagine for a moment that you are a bad guy. You are very busy, hacking into dozens of system around the planet each week, with a variety of different operating systems. It's a tough life of hacking toil, but you get by somehow. Now, suppose you conquer a bunch of different versions of UNIX systems in your exploits. Today, you grabbed a lot of Solaris boxes, yesterday was FreeBSD day, and the day before you focused on HP-UX. You could stock a bunch of different RootKits in your tool belt, one for each type of UNIX machine that you conquer. However, it would require a lot of work to sort out all of your different RootKits, as well as master all of the different commands and features of each different RootKit tool. Surely, unless you were extremely careful, you'd occasionally make a mistake and try to install the wrong RootKit on the wrong type of UNIX system, possibly disabling all remote access or even crashing the box. If only there were some way to use a single RootKit on a bunch of different UNIX variations, your life as an attacker would be far simpler.

Well, such general-purpose RootKits aren't in the category of "if only" any longer. A developer named K2 released the Universal RootKit (abbreviated URK and usually pronounced "U–R–K" not "urk") to meet just this need. URK functions on a variety of different UNIX variations, including Linux, Solaris, BSDI, FreeBSD, IRIX, HP-UX, and OSF/1, all rolled up into one single convenient RootKit package. In the words of the README file included with URK, K2's stated goal for the tool was to create one RootKit that would "Run on most every UNIX you may encounter."

Like other user-mode RootKits, URK includes a variety of replacement programs that implement backdoors and hide the attacker, as well as various helper tools, listed in detail in Table 7.2. Note that URK includes a subset of tools built into operating-system-specific RootKits such as LRK. Even though it doesn't include every single knick-knack built into LRK, URK still packs a strong punch, and its cross-platform capabilities make it especially useful for attackers.

Table 7.2. Components of the Universal RootKit (URK)

RootKit Components	Function
`login`	The familiar `login` program lets users log in to a system. The URK `login` program includes a backdoor password that is located in the urk.conf file.
sshd	This sshd backdoor is not included in all releases of URK. For those versions that include it, the backdoor `sshd` supports remote encrypted backdoor access by the attacker.
`ping`	Normally the `ping` command is used to send an Internet Control Message Protocol (ICMP) Echo Request packet to another system to see if it is alive. The ping program built into URK, on the other hand, also includes a local backdoor. By typing the `ping` command, followed by the backdoor password locally on the system from a low-privileged account, an attacker will be escalated to root privileges at the command prompt.
`passwd`	This program, typically used to set a user's password, is another local backdoor that works like the `ping` backdoor just described. By typing `passwd` [backdoor_password], the attacker will get root privileges.

`su`	The `su` command, which normally is used to alter a user's current login identity, includes a backdoor that functions just like the `ping` and `passwd` backdoors.
`pidentd`	This process offers a remote command shell backdoor, listening on TCP port 113. If the attacker connects to this port, types the characters 23, 113, and then the backdoor password, the system will respond with a remote root-level command shell.
`ps`	The `ps` program is used to show a list of running processes. This URK version filters out any processes that the attacker wants to hide on the system.
`top`	Normally, `top` shows a continuously updated list of running programs on the machine. Like the URK version of `ps`, this program also filters out hidden processes.
`find`	The URK alters the `find` command, typically used to search for files, so that it filters the attacker's files from its output.
`ls`	The `ls` command included with URK filters an attacker's files from its output.
`du`	This command, which shows the disk usage of files, has been modified to lie about any space the attacker's files occupy.
`netstat`	The URK version of `netstat` shows all listening TCP and UDP ports, except those in use by the attacker.
`sniffer`	The `sniffer` program built into URK gathers network traffic destined for various services that use clear-text authentication, such as Telnet and FTP.

Most of the binary replacements in URK have a particularly interesting twist. To give URK universal appeal, K2 didn't implement the binary replacements as brand new pieces of code, as was done in most user-mode RootKits such as LRK. Instead, in URK, most of the binary replacements are actually just wrapper programs that call a hidden version of the real program and then give backdoor access or filter the real program's output to hide the attacker's presence. Figure 7.5 illustrates the process for the familiar `ps` command, which is frequently used to generate a list of processes running on a system.

Figure 7.5. The `ps` replacement program included in the Universal RootKit is really just a wrapper and filter of the original `ps`.

System with Universal RootKit installed

Let's look at this wrapping and filtering process in more detail. First, URK moves the real `ps` command to an obscure directory on the machine, such as /usr/man/man1/, where system documentation in the form of "man" pages are typically stored. Then, the URK version of `ps` is

written where the real `ps` was formerly located. Now, whenever a user or system administrator runs `ps`, the URK version of the command will be invoked. This fake `ps` first runs the hidden, real version of the `ps` command and grabs its output before it is displayed on the screen. The fake `ps` then filters the output, removing any references to processes that the attacker wants to be hidden. The URK versions of `ps`, `top`, `find`, `du`, and `ls` all use this filtering wrapper method.

In a similar fashion, the `passwd`, `su`, and `ping` backdoors are all designed as wrapper programs. If the wrapper is invoked with the special backdoor password, the attacker is given a root-level shell prompt. Otherwise, the wrapper program activates the normal command hidden away somewhere inside the file system. So, instead of having to write brand new replacement binaries for each of these programs on a whole bunch of UNIX flavors, a single set of general-purpose filtering wrapper programs will suffice. That's a pretty efficient method of creating a RootKit with universal applicability.

This standard complement of RootKit programs implemented as filtering wrappers is certainly useful for the bad guys, but URK would be nothing without its installation program, the makefile. When building and installing URK, the attacker activates the makefile with a single argument: the flavor of UNIX that the resulting RootKit should be compiled for. Then, the makefile contains the intelligence to grab the appropriate pieces of code to create a RootKit tailored to that kind of UNIX. After compiling the appropriate code, the makefile inserts it into the appropriate places on that type of target operating system, thereby RootKitting the machine.

URK is initially configured using two files: urk.h and urk.conf. The first of these files is used while the RootKit is compiled, identifying the location of the original versions of various wrapped programs, as well as the password to be used for the backdoors. By default, the URK backdoor password is set to h4x0r, a variation of the word *hacker*. The urk.conf file specifies where the individual wrapper configuration files are stored. Each of these configuration files in turn includes a list of process names, port numbers, and file names that will be hidden by URK. Of course, URK modifies the system so that the urk.conf file is itself hidden. With all of these capabilities and its ability to run on the vast majority of UNIX flavors, URK is certainly a formidable user-mode RootKit tool.

File System Manipulation with RunEFS and the Defiler's Toolkit

So far, most of the tools we've seen in this chapter have focused on replacing critical system binary executables so an attacker can gain backdoor access or hide on a system. However, several tools go beyond diddling with binaries and instead focus on manipulating the underlying file system structure of the victim machine. As you probably recall from our LRK discussion, the fix tool lets an attacker tamper with the creation, modification, and last access time of individual files. Although certainly useful for a bad guy, the fix tool was but a foretaste of even more powerful tools that allow a bad guy to manipulate the file system. RunEFS and the Defiler's Toolkit are two related tools written by someone called "the grugq" that accomplish even more powerful attacks. Although they aren't RootKits by themselves, RunEFS and the Defiler's Toolkit could certainly be added to user-mode RootKits (or even the kernel-mode RootKits we'll discuss in Chapter 8) to make even more subtle, yet still devastating, attacks.

Computer Forensics Meets Antiforensics

RunEFS and the Defiler's Toolkit, available at *www.phrack.org/show.php?p=59&a=6*, attempt to foil computer forensics techniques. Over the past several years, the relatively new field of computer forensics has blossomed into a complete discipline and valuable resource in the information security community. Computer forensics experts fight computer crime by gathering and analyzing evidence, including log files, hard-drive images, and memory dumps from compromised systems. In particular, hard-drive images are among the most useful forms of evidence to the forensics specialist, as they contain a copy of the files and directories of the victim machine. Hard-drive images are the closest thing we've got to a crime scene in many cyberattacks. For this reason, most forensics specialists quickly snag a backup of a victim

system early on in the incident-handling process, before any evidence is tainted.

Because bad guys don't want to get caught, they have developed a variety of techniques to frustrate computer forensics analysis, especially as that analysis applies to the highly important evidence on hard-drive images. These techniques are known collectively as *antiforensics*. In his very detailed paper describing RunEFS and the Defiler's Toolkit, the grugq defines antiforensics as "the removal or hiding of evidence in an attempt to mitigate the effectiveness of a forensics investigation"[1].

A Brief Overview of the ext2 File System

To see how RunEFS and the Defiler's Toolkit manipulate evidence, we need to explore the structure of file systems. Of course, the file system itself is merely the arrangement used by the operating system to store files and organize them into directories on the system's hard drive. Without a file system, your hard drive is just one vast ocean of undecipherable and completely unusable bits. The file system tames this ocean of bits, applying a coherent structure so we can navigate the system and access files. To a normal user, the file system looks like a bunch of files allocated in various directories. However, the file system itself works hard to mask its own underlying complexity and the physical details of the hard drive. Numerous different file system types are in use today, including the ufs file system used by several flavors of UNIX, the NTFS file system used by Windows NT/2000/XP/2003, and the ext2 file system used by many versions of Linux. RunEFS and the Defiler's Toolkit attack the ext2 file system. Because of that, we'll focus on the details of ext2, although similar high-level concepts and related attacks apply across all of these file system types.

One of the most fundamental components of the ext2 file system is the data block, where file content is written. During formatting, the hard drive is carved up into a series of these blocks, with a typical ext2 block being 4,096 bytes in length (although other sizes are supported). A file is nothing more than a collection of blocks that are related to each other. The blocks making up a single file are likely not even contiguous on the hard drive.

"But," you might ask, "how does the file system relate a bunch of noncontiguous blocks together into a file?" This grouping of blocks is accomplished through the magic of the *inode*. Each and every file on the file system has one inode, which is a data structure storing critical information about that file, including a list of blocks that hold the associated file's contents. Each inode has a unique number, called the inode number, used to identify that inode. The relationship of an inode to a series of blocks making up a single file is shown in Figure 7.6.

Figure 7.6. The relationship between a file's inode and the blocks that store that file's data.

An inode contains a bunch of pointers to the blocks that make up a file. Some of these blocks are pointed to directly by the inode and are therefore referred to as direct blocks. Ext2 supports up to 12 direct blocks. For larger files that have more blocks, there aren't enough slots in the inode to point directly to all of the blocks, so an extra level of indirection is used. With indirect blocks, the inode itself points to another block that holds an array that points to other blocks holding the file's contents. If the file is still too large to be represented with direct and indirect blocks, yet another level of indirection supports double indirect blocks. And, yes, for gargantuan files, ext2 even supports trebly indirect blocks. Trebly is just a fancy way of saying triply, by the way.

In addition to a list of blocks that make up the file, the inode also contains information about the file's permissions, the file's owner, and the size of the file. The inode also holds the access, creation, and modification times associated with the file (that's the a_time, c_time, and m_time that we discussed earlier in this chapter). To get an idea of how your hard drive is organized, think of an inode like a set of instructions in a scavenger hunt. The scavenger hunt instructions tell all the players where to find various goofy objects required to win the game. Similarly, a file's inode tells you where to pick up all of the different blocks on the file system so you can reassemble that file when someone wants to use it.

The blocks making up a single file are scattered all around your hard drive, but the inodes themselves are grouped together on your file system. That way, the operating system can easily find the inodes and use them to figure out how to get at the files. To spell out how many inodes are available on the system, ext2 uses something called a *super block*, which is a master data structure that defines the overall shape of the file system. In the olden days of UNIX file systems, there was a single super block at the beginning of the hard drive that laid out the inode structure. The super block was followed by all of the inodes, which were in turn followed by the blocks themselves. However, there was a fundamental flaw in this overall strategy. If the super block got corrupted, the entire hard drive was hosed, as the operating system couldn't even figure out how the inodes themselves were constructed without an intact super block. To deal with this problem, in modern UNIX file systems, copies of the super block are located in several places on the hard drive, as shown in Figure 7.7. This overall structure containing the super block, inodes, and data blocks is repeated at several locations on the hard drive, with each iteration carving up that piece of the drive.

Figure 7.7. The organizing scheme of the ext2 file system brings together the super block, inode and block bitmaps, the inodes themselves, and data blocks.

The attentive reader will also note that Figure 7.7 contains two additional elements we haven't yet discussed. After the super block, ext2 contains a bitmap of all inodes. This bitmap specifies which inodes are in use and which are free to be used for new files. This inode bitmap is not a bitmap in the sense that it contains some graphical image. It just has a bunch of bits that can either be on or off, depending on whether an inode is in use or not. After the inode bitmap, the system stores a block bitmap that shows which blocks are in use and which are free. Using these two bitmaps, the file system can get the status of each and every inode and data block in that file system to determine if it is allocated. When a new file is created, the system consults these bitmaps to find an unused inode for the file as well as unused blocks to store the file's contents. These bitmaps are then updated to reflect the presence of the brand new file.

There's one special inode we need to zoom in on. The first real inode on the system (and I'm not referring to the super block) contains a list of *bad blocks*. These bad blocks haven't been naughty; they're simply unusable. When you conduct a full format of your hard drive, the formatting program will discover a few blocks on the hard drive that cannot properly store data. These so-called bad blocks might be the result of a pimple or scratch on the physical hard-drive media, some bad magnetic resonance, or another flaw. However, just because there are a few bad blocks, we don't want to toss the whole drive in the garbage. Indeed, most drives have a handful of bad blocks scattered here and there. The formatting program generates a list of bad blocks and writes that list in the first inode. By consulting this bad blocks inode when storing new data in unused blocks, the file system can make sure it only uses good blocks. The file system then avoids bad blocks like the plague.

We've now seen how files themselves are constructed, but we've left off a crucial element. Note that we've said nothing yet about the file's name or the overall directory structure. Where are these items located? Are they in the inode? Nope. The underlying file system structure deals solely in inodes and blocks, and doesn't assign names or organize the directory hierarchy. Instead, the directory is implemented pretty much like a standard file itself in terms of an inode containing pointers to data blocks. In the case of the directory, however, the data blocks contain a data structure with the names of all files in the directory as well as their relationship to each other in the directory hierarchy. The directory contains a set of entries, one for each file. A directory entry holds the file's name and the inode number the file corresponds to, as well as a record length, which describes how long that directory entry is.

Messing with Ext2 Using RunEFS

Now that we've completed our whirlwind tour of the ext2 file system, let's see how RunEFS and the Defiler's Toolkit mess with all of this beauty to foil forensics analysis. We'll first address the RunEFS tool, which allows an attacker to hide data such as a backdoor program or sniffed passwords on the file system. RunEFS takes advantage of the fact that many forensics investigators and the tools they use don't look inside of bad blocks. These investigators and the authors of these tools make the unfortunate (but seemingly reasonable) assumption that no real data could be stored in bad blocks. If the blocks are referred to in the bad blocks inode, they can't possibly have the attacker's data stored in them, right? Wrong!

RunEFS carves out a portion of the hard drive and labels the associated blocks as bad by writing their block numbers into the bad blocks inode. RunEFS does all of this on the fly, without reformatting the hard drive or corrupting any data. An attacker can use RunEFS to label a set of blocks as bad on your system, put data in those blocks, and take data out of the fake bad blocks. An attacker could store any type of digital information in these fake bad blocks,

including computer attack programs, backdoors, password lists, stolen information, supersecret macaroni and cheese recipes, pirated software, conspiracy theories, or pornography. To actually run a program using the current version of RunEFS, however, the attacker would have to copy it from bad blocks into good blocks before executing it. However, while running, the program is copied into memory. The attacker can then delete the program file from the good blocks on the hard drive.

As an example, suppose you have a 40 GB hard drive. I hack into your system and upload a backdoor listener program, a sniffer, and various other tools. I don't want you to find my dastardly tools, so I'll use RunEFS to carve out a 100 MB area of the hard drive, and label all of those blocks as bad. They're really good blocks, where I can write data using the RunEFS tool. However, I'll modify the bad blocks inode to lie and say that these blocks are bad. When these blocks are labeled bad, though, your hard drive will suddenly look slightly smaller: 40 GB less 100 MB is 39.9 GB of good space left for you. Yet, few people would notice such a small change in the size of their hard drive. This operation is shown in Figure 7.8.

Figure 7.8. Using RunEFS to label good blocks as bad, making the drive appear somewhat smaller.

If, per chance, you somehow detected my sniffer program running or found the port used by my backdoor, you'd likely make a backup of the victim machine for forensics analysis. Unless you snag a copy of all of the data blocks for analysis, including the bad blocks, you won't see all of my dastardly tools. You'll only see the elements I don't stash away in blocks labeled as bad. Even if you do get a complete bit-by-bit copy of the entire drive, including bad blocks, some forensics analysis tools have a bug that prevents them from looking at bad blocks. In particular, the venerable yet still widely used tool, The Coroner's Toolkit (TCT) by Wietse Venema and Dan Farmer, does not analyze any blocks referred to in the bad blocks inode. Similarly, tools derived from TCT, including older versions of The @Stake Sleuth Kit (TASK) by Brian Carrier, don't look at the bad blocks inode or any blocks referenced by it. Carrier fixed this issue in newer releases of TASK, which is now officially called the Sleuth Kit. The Sleuth Kit is a wonderful forensics analysis tool for delving deep into a hard drive's structure, and is available for free at *www.atstake.com*. However, if you are using TCT or older versions of TASK, I'd still fly under the radar screen of your forensics analyst by using RunEFS.

Messing with Ext2 Using the Defiler's Toolkit

Whereas RunEFS carves out a hidden area on the hard drive for the attacker, the Defiler's Toolkit is focused on securely deleting forensically useful data from the file system. First off, check out that name: the Defiler's Toolkit. It reminds you of The Coroner's Toolkit, the forensics tool, doesn't it? Here we see an antiforensics tool specifically designed to foil a forensics tool. To see how, note that whenever you create a file on an ext2 file system, information about that file is stored in three places: the data blocks where the file contents are written, the inode that

points to those blocks associated with the file, and a directory entry describing the file's name. Suppose you delete the file. When you do a run-of-the-mill file deletion using the Linux `rm` command, the blocks where the file is written are freed up for future use, but not cleared. Similarly, the inode and directory entry are freed up, but they still contain the original data.

Under normal circumstances, this data will only be overwritten if the data blocks, inode, and location of the directory entry are allocated to another file. Before another file taking up these resources is created, all of this data is still available on the hard drive. During an investigation, a forensics examiner could use an undelete program to recover such a deleted file. Numerous undelete programs are available, on both a free and commercial basis, such as the commercial R-Undelete tool from R-Tools Technology, Inc. Available at *http://r-undelete.com*, R-Undelete supports both Linux and Windows file systems. Forensics tools use similar techniques to recover deleted evidence from a hard drive.

An attacker can create various files on the system, and might later need to remove those files without leaving any tracks for a forensics analyst. These files could contain, for example, the installation programs for a RootKit that can be safely discarded once the RootKit is in place. Even if the attacker's file is overwritten once or twice, a forensics analyst could use fancy hardware to read the previous bits based on residual magnetic resonance of these bits [2]. So, instead of using the `rm` command, which leaves data around for the forensics investigator, the attacker often wipes the contents of the file using a tool that overwrites all of the data blocks, thus blanking out their contents. For example, Linux includes the `shred` command to wipe out the contents of a file. By default, `shred` overwrites the file's blocks 25 times using a pattern of bits designed to make sure every last bit of evidence in the data blocks is destroyed. With `shred`, the bits cannot be recovered even with specialized hardware that can view earlier recent values of a given bit on the drive even though it has been overwritten.

However, note that `shred` and related tools are focused solely on the data blocks holding the file contents. A forensics investigator could still analyze the inode and directory entries to learn about the attacker's activities. Sure, the investigator won't be able to get a copy of the dastardly tools, but their names and sizes could certainly be useful. Also, from a forensics perspective, the creation, modification, and access times are hugely important in understanding the attacker's activities and building a court case. All of this data is left intact by the `shred` command and most other file wiping tools. TCT, TASK, and other forensics tools look for just this metadata during a forensics analysis.

As an antiforensics measure, the Defiler's Toolkit destroys inode and directory entry information associated with deleted files to make sure that forensics tools cannot retrieve it. To accomplish this goal, the Defiler's Toolkit includes two programs: Necrofile and Klismafile. Necrofile scrubs inodes clean, removing any information about the blocks that were assigned to the inode and file. All inode information is cleared, including its owner, permissions, and any time information. The attacker activates Necrofile, specifying the criteria to use in selecting inodes to clean, such as inode number, user ID, or times referred to in the inode.

Klismafile focuses on the directory structure, overwriting the directory entries associated with deleted files. This tool even supports using regular expressions to formulate very flexible searches for the name of a file or directory to be cleaned. However, Klismafile cannot completely remove every scrap of evidence from the directory. You see, in the ext2 file system, directory entries are written one right after another, and they have variable length, as illustrated in Figure 7.9. These directory entries are variable in length because file names have variable size, up to 255 bytes. Therefore, when Klismafile cleans out a directory entry, there is a big blank gap left in the directory structure. Klismafile tries to cover up this gap by making the preceding directory entry's record length larger. A superskilled forensics analyst combing through the directory structure in mind-numbing detail might notice this little discrepancy. Some modern forensics tools can automatically highlight the unexpected directory entry size difference. The directory entry will be too big for the name loaded into the entry, kind of like an overstretched sock that's hanging on a skinny foot. However, the only information the forensics analyst can glean will be the size of the deleted file's name. It's actual name, contents, inode,

and anything else will be long gone, deleted by the Defiler's Toolkit. Klismafile could be modified to rewrite the directory structure entirely, in a sense defragmenting it. However, this rewrite would take some time, and would involve significant disk usage as the directory structure is rebuilt to resize the removed entries.

Figure 7.9. Removing directory entries using Klismafile leaves larger record lengths in remaining directory entries.

UNIX RootKit Defenses

> *We have learned of evil, though not as the Evil One wished us to learn.*
>
> —C.S. Lewis, *Perelandra*, Book Two of his Space Trilogy, 1943

As we've seen, user-mode RootKits on UNIX are not something to be trifled with. To stop their use on your systems, you need to plan your defenses carefully. As we analyze the various defensive strategies available for dealing with these tools, keep the following analogy in mind: In some sci-fi television programs, a standard plot device involves evil aliens kidnapping the captain of a starship. After nabbing the real captain and locking him in a dungeon, the aliens replace him with a counterfeit captain, just because they're evil. This fake captain takes control of the starship, with the real crew unaware of the malicious switch. At some later time, the real captain escapes his captivity and confronts his counterfeit face to face. Inevitably, some crew member on the ship is required to choose which is the real captain and which is the counterfeit. Of course, this crew member must carefully decide and quickly vaporize the imposter with a ray gun. The correct choice will save the captain, the ship, and all of humanity from the grip of the evil aliens. The wrong choice will lead to certain doom.

The defenses from this classic sci-fi dilemma map quite well to the problems we face with defending against user-mode RootKits. How can you thwart evil attackers who play switching games with your critical operating system programs? Instead of swapping the captain of a starship, our evil attackers swap operating system components with user-mode RootKit counterfeits. The defenses against such attacks fall into three categories: prevention, detection,

and response.

User-Mode RootKit Prevention on UNIX

To prevent attackers from installing user-mode RootKits on your systems, you must carefully harden your systems and apply patches. Remember, to install a RootKit, the attacker must first conquer root-level permissions on the machine, via guessing passwords, exploiting a weak system configuration, or finding an unpatched flaw on the machine. If attackers cannot break into your system with root-level permissions, they cannot use a RootKit on the machine. In our sci-fi captain-switching analogy, this step is the equivalent of preventing the attackers from kidnapping the captain in the first place.

To harden your system, you should shut off and remove unnecessary services and functionality on the machine. Look at all available network services. Which do you really require for the system to perform its required business purpose? After shutting down all unneeded network services, look at various local packages installed on the machine. Are they all required? Any local program with a security flaw could offer a malicious user the ability to escalate permissions to root level. To help you walk through this process, consider using one of the large varieties of solid system hardening programs or guides available for UNIX operating systems.

For the Linux, HP-UX, and Macintosh OS X flavors of UNIX, you should consider Bastille, written by Jay Beale. This wonderful free tool, available from *www.bastille-linux.org*, is an automated script that walks you through system hardening. Although it was originally crafted for use on Linux, Bastille now has solid support on HP-UX and Macintosh OS X as well as Linux. According to the eminent Jay Beale, the primary goal of Bastille is to "provide the most secure, yet usable, system possible." Although a daunting task, Bastille delivers amazingly well on this steep objective.

As Bastille runs, it prompts the system administrator, asking whether it should complete each hardening step. This prompting, shown in Figure 7.10, serves two very useful purposes. First, it allows an administrator to custom-fit the functionality of the hardened box for the particular environment in which it will live, just by answering a series of questions in a nice GUI. Second, by running Bastille, an administrator can learn the various steps needed to secure a box. The screen in Figure 7.10 illustrates this educational component, as it describes the change Bastille will make to the Linux kernel configuration to prevent some types of buffer overflow vulnerabilities. This valuable knowledge comes in handy time and time again, as the administrator uses Bastille on other machines and even on machines without Bastille support. If you are system administrator for Linux, HP-UX, or Macintosh OS X, you should try Bastille. It'll make your life easier.

Figure 7.10. Bastille hardens systems, while helping educate system administrators.

Besides Bastille, the SANS Institute offers various Step-by-Step Guides for hardening individual operating systems, including Linux and Solaris, available for a reasonable fee at *http://store.sans.org/store_item.php?item=83* and *http://store.sans.org/store_item.php?item=21*, respectively. Furthermore, a wonderful hardening guide and checklist for Solaris is available from the System Administrator's Guild (SAGE) at *http://sageweb.sage.org/resources/online/solaris/index.html*.

In addition to hardening your machines, you need to keep them patched. To keep your system patched, implement a comprehensive patching process, based on the tips we discussed in Chapter 3. The timely application of patches can be tedious hard work, but it is essential for maintaining a secure system and keeping the bad guys from installing user-mode RootKits.

User-Mode RootKit Detection on UNIX

Although prevention goes a long way in stopping user-mode RootKits, we must go further to minimize the chance of falling victim to these foul tools. If an attacker installs a user-mode RootKit on your system, you need to be able to quickly detect the attack. Back to our counterfeit starship captain analogy, your crew should be able to identify the imposter quickly, before he steers your spaceship into dangerous territory. There are two types of tools that can detect user-mode RootKits: file integrity checkers and RootKit-specific identification tools.

We've run across file integrity checking tools before, specifically in Chapters 2 and 6. We gave a high-level overview of such tools then, but we need to get into more detail now, simply because these tools really shine in defending against user-mode RootKits.

As you might recall, when they are installed, file integrity checking tools create a database of cryptographic hashes of critical system files, including configuration files and sensitive binaries. These hashes act as fingerprints for the known good files on the machine, and are usually stored on a write-protected medium such as a write-once CD-ROM or write-protected floppy disk. These tools rely on cryptographically strong one-way hashing or digital signature algorithms, such as MD5 or SHA-1. Seriously smart cryptographers designed these algorithms so that an attacker couldn't determine a counterfeit that has the same resulting hash as a legitimate program. Devising a counterfeit with a hash that matches the original would require gobs of processing power over eons of time, ranging from decades to the widely accepted age of the universe, depending on the particulars of the algorithm in use. Therefore, although RootKit tools like fix can pad a replacement program to match the noncryptographic checksum, attackers simply cannot make the replacement's cryptographically strong hash match the hash of the original program. Using these tools, we get cryptographically strong file protection. Back to our starship captain analogy, a file integrity checker acts like a DNA analysis tool, looking for

discrepancies between the real and fake captains at the microscopic level.

After creating an initial database of hashes of critical system files, a system administrator schedules the file integrity checking tool to run on a periodic basis, such as once per day or even once per hour on sensitive systems. While the file integrity checking tool runs, it recalculates the hash of each critical file and compares the hash to the database of known good hashes. If there is a discrepancy between the hashes, someone altered the file. It is up to the system administrator to determine whether the file was altered by routine system administrator tasks or by an evil attacker who has compromised the system. This reconciliation step could be a significant amount of work! Many patches alter the system significantly, causing a file integrity checking program to fire off all kinds of warning messages.

On my own systems, I run a file integrity checker once each day, as well as just before I apply a system patch. By running the tool just before I install the patch, I can make sure my system is in a known good state prior to the patch. After reconciling any prepatch changes that I discover, I install the patch. Quickly after installing the patch, I run the file integrity checking tool again, instructing it to re-create its database of file hashes. That way, not only will I know my system was in good shape before the patch, but I'll also have a fresh postpatch reference of hashes to check it against in the future. I've illustrated this process in Figure 7.11. While I install patches and reconcile changes with a file integrity checker, this process keeps me from ripping the hair out of my head—an increasingly important need as I grow older.

Figure 7.11. Using file integrity checking tools before and after patch installation.

Each file integrity checking tool comes with a list of critical system files that are often altered by attackers. These lists vary slightly, but they all include the standard complement of programs that are frequently altered by attackers, including sshd, `login`, `netstat`, `ps`, `ls`, and all of the other RootKit replacements we've discussed throughout this chapter.

File integrity checking tools have been available for many years. Tripwire, originally by Gene Kim and Gene Spafford, was the first very powerful tool in this category. Tripwire remains one of the strongest and most widely used solutions in this space. It's available for free at *www.tripwire.org*, or on a commercial basis at *www.tripwire.com*. You've got to like those easy-to-remember URLs. The commercial version includes vendor support and enhancements for centrally managing Tripwire across an enterprise. Tripwire has wide platform support, running on the vast majority of UNIX operating system flavors (including Linux, Solaris, HP-UX, IRIX, AIX, and Tru64). There's also a Windows version, which we'll discuss in more detail later in the chapter.

Tripwire isn't the only tool in this field. The Advanced Intrusion Detection Engine (AIDE) is a free, open-source alternative to Tripwire, written by Rahmi Lehti and Pablo Virolainen. Available at *www.cs.tut.fi/~rammer/aide.html*, AIDE supports Linux, Solaris, various BSD incarnations, Unixware, AIX, and Tru64. Other tools in this genre include Osiris (at *http://osiris.shmoo.com/*) and Samhain (at *http://la-samhna.de/samhain/*).

Although some form of file integrity checking tool is a must for any security-minded system administrator, other tools complement their capabilities by specifically identifying RootKits. The very aptly named *chkrootkit* is one of the most popular tools in this category, available at no charge at *www.chkrootkit.org*. Written by Nelson Murilo and Klaus Steding-Jessen, chkrootkit is well versed in the specific changes various RootKits make to a target system, and can look for those changes to detect more than 45 different RootKit strains. Using our starship captain swapping analogy, chkrootkit acts like crew members who can ask the captain very carefully selected questions to differentiate between the real captain and the counterfeit. For example, the crew could test the captain by asking him about some very esoteric event in his past, such as the name of an exam the captain aced while at Starfleet Academy. Alternatively, they could ask the captain about some ethical dilemma, knowing that the real captain would respond differently from the imposter. If the given "captain" cannot answer the questions properly, the crew will know that he's the counterfeit and vaporize him.

Chkrootkit runs locally on Linux, Solaris, FreeBSD, OpenBSD, NetBSD, HP-UX, and Tru64 operating systems and asks questions of the local software in an attempt to determine if a RootKit is installed. So, what kind of questions does chkrootkit ask? It asks several dozen questions based on the most widely used RootKits available today. Each question asked of the operating system is in the form of a specific test automatically run by chkrootkit. For example, chkrootkit looks for the names of various configuration files used by many RootKits, such as the /dev/ptyp and other related files created by LRK. Furthermore, chkrootkit checks for signs of deleted entries from the lastlog and wtmp files, an indication that the wted or Zap2 tools might have been used. It also attempts to see if the network interface is in promiscuous mode, a likely sign of a sniffer (although, sadly, this sniffer check isn't reliable on most recent Linux distributions using Linux kernel 2.4). To help detect backdoor shell listeners associated with various RootKits, chkrootkit looks at the various TCP and UDP ports in use on the machine to see if any match the well-known ports associated with popular RootKits. It even looks for small snippets of code from some of the more popular RootKits in use today, including LRK.

Although all of those individual tests are quite useful, chkrootkit's best feature is probably its ability to analyze a select set of individual binary programs to determine if they have been modified. The tool looks for anomalies in all of the following commands:

```
amd basename biff chfn chsh cron date du dirname echo

egrep env find fingerd gpm grep hdparm su ifconfig inetd

inetdconf init identd killall ldsopreload login ls lsof

mail mingetty netstat named passwd pidof pop2 pop3 ps

pstree rpcinfo rlogind rshd slogin sendmail sshd syslogd

tar tcpd tcpdump top telnetd timed traceroute w write
```

Wow! That's quite a list, and covers nearly everything I've ever seen a user-mode RootKit try to replace. It's important to note that a user running chkrootkit doesn't have to manually perform each of these checks. Every last one of them is built into the chkrootkit program. When run, the tool conducts all of these complex tests and spits out a single answer: INFECTED or not infected. And yes, if you have a RootKit installed, the tool actually displays the infected message in all capitals, which seems quite fitting. If the system has indeed been infected, chkrootkit will also specify which test detected the anomaly, so you can determine which binary was replaced.

There is a small chance of a false positive from chkrootkit, especially in its promiscuous mode check, which is unreliable on Linux systems. Still, besides the promiscuous check, I very seldom get a false positive when using the tool. False negatives, where the tool doesn't detect the real presence of a RootKit on an infected machine, are also conceivable with chkrootkit. If the attacker creates a custom RootKit that hides the attacker's presence in novel ways, chkrootkit might not be able to detect it. Again, for most uses, chkrootkit performs admirably in minimizing false positives and negatives.

One beautiful aspect of chkrootkit is that its anomaly checks do not require establishing a database of hashes in advance. Unlike the file integrity checking tools we discussed earlier, chkrootkit doesn't need such a database. Its tests are not based on comparing hashes; all of its checks are built into the logic of the tool itself. Please don't misunderstand my point here: I'm not saying that chkrootkit should replace your file integrity checking tool—not by any means! Chkrootkit is a perfect complement to a file integrity checking tool. By using both types of tools for RootKit detection, you'll have a much better shot at discovering an attacker.

User-Mode RootKit Response on UNIX

Now suppose you discover that an attacker has installed a user-mode RootKit on your system. How should you respond to this fact? Uttering a few choice curse words under your breath is a reasonable and popular way to start. Beyond that, to investigate a system that potentially has a RootKit installed, you can't really rely on the software on the system. You shouldn't just log in to the machine and start running the existing command programs on the box for your investigation. The RootKit likely modified these programs so that they lie to you.

To address this situation, I've seen some system administrators create copies of their critical binaries with a different name that an attacker wouldn't know. For example, an administrator could name a copy of the `ls` command `new_ls` or the `ps` command `admin_ps`, and then use those programs instead of the normal ones installed on a system if a RootKit is suspected. This technique might help a small bit, but I'm not crazy about it as a general-purpose solution. An attacker could hunt down and alter `new_ls` or `admin_ps` without too much difficulty, especially if they are loaded in the administrator's path. Because of this concern, I don't use this technique on my own systems.

For a better approach, if you want more trustworthy results, you'll have to bring your own programs to the system to conduct your investigation, instead of relying on commands loaded on the machine. If you handle computer investigations, you should create a bootable CD-ROM that includes all of the binary executables you require for analyzing a system, such as `ls`, `lsof`, `ps`, `du`, and `netstat`. When investigating a potential RootKit incident, you can run these tools from your CD-ROM. Please make sure these command programs on the CD are statically linked. That is, compile them (or download precompiled versions) so that they do not rely on any libraries on the system's hard drive. Statically linked executables are self-contained binary programs that don't need any libraries to run. The binary executable makes calls directly into the operating system kernel to investigate what's going on. If a bad guy alters one of the library files on your system, the statically linked binary will bypass this alteration as you look through your system. The answers will be more trustworthy than the results you'd get from the programs already loaded on the machine.

There are several free CD-ROM images with collections of trusted, statically linked binary programs available on the Internet. I'm quite fond of Bill Stearns' static tools for Linux at *www.stearns.org/staticiso*. My favorites, though, are FIRE (created by William Salusky and available at *http://fire.dmzs.com*) and Knoppix (developed by Klaus Knopper and available at *www.knoppix.org*). You can download each of these bootable Linux distributions and burn them onto a CD-ROM to carry with you on your investigations. These packages are full of useful investigation tools, including standard UNIX command programs as well as specific forensics tools.

After you investigate the RootKit-infected system using a tool like FIRE or Knoppix, you'll need

to rebuild your system. Your best bet is to reinstall the operating system itself, reload all critical applications, and apply all appropriate patches. Sorry, but you cannot just replace the single malicious program detected by the file system integrity checker or chkrootkit. If attackers modified one piece of your operating system, they likely modified many other components as well, including the applications installed on the machine. You could reinstall the operating system from the original media, or use a trusted backup. However, you must make sure your backups really are trustworthy. Rebuilding your system from a backup that includes a RootKit will get you nowhere. Either use the original installation media and patches, or run an integrity check against your backups before using them. On a starship, if evil aliens replace the captain with a malicious interloper, who's to say that they haven't also planted a substitute for the first mate, the chief science officer, or the communications expert? If the captain has been compromised, the whole crew is suspect and should be replaced with fresh, trusted personnel. We should do the same thing in a RootKit attack by restoring our operating system from the original media or a trusted backup.

Once the system is restored, monitor it carefully using network- and host-based intrusion detection tools. Bad guys frequently return to the scene of the crime, and try to log in to the system again. If you are monitoring the system looking for their return, you will be much more likely to protect the systems and possibly even catch the attackers.

Also, in defending against user-mode RootKits, don't forget about how RunEFS and the Defiler's Toolkit manipulate the underlying file system, especially their tricks for falsely labeling blocks as bad. If you conduct forensics investigations, you need to be aware of these attack tools. Also, make sure you utilize forensics analysis tools that can analyze the bad blocks inode as well as blocks that are labeled as bad. Recent releases of the free Sleuth Kit tool, as well as most commercial forensics tools, have this ability. Finally, when you create a system backup for forensics analysis, make sure to get a bit-by-bit copy of the entire file system, including blocks that are labeled as bad.

PREV < Day Day Up > NEXT