UNIX Security

By Rajib K. Mitra

Introduction

UNIX has been around for decades. Many corporate giants have come along with their dazzling new proprietary operating systems in an attempt to replace the defacto standard for corporate servers, and some have done quite well, but UNIX remains dominant.

There are many reasons for the popularity of UNIX. It is standardized, portable, and proven. Most importantly, however, UNIX is versatile. Most technological advances can be supported by the fundamental design of a UNIX system.

Herein lies the problem which we will address in this booklet. A versatile operating system allows people to do many different things-- even if the system administrator doesn't want them to. When that happens, security is breached. In today's increasingly electronic society, security breaches have the potential of being very serious.

Most UNIX system administrators today fail to take reasonable steps to ensure the security of their UNIX systems. I know. I am currently incarcerated in the Waukesha County Wisconsin Jail for unauthorized access to computer systems. Does that make me the enemy? Former enemy, perhaps. I've also administered UNIX systems both professionally and as a hobby.

I've wanted to write a book for a while now. I've read several other books on UNIX security and they all have the complex, dry, and tedious style of a college textbook. Most system administrators I know are very busy people so they have neither the time nor the will to undertake an in-depth study of security for their system. And unless they're responsible for sensitive information, they don't. But it's the simple fact that if you're a busy system administrator that makes security concerns worth a little of your time. Because when you have to unexpectedly interrupt your work and spend hundreds or thousands of man-hours to track down a suspected breach, even a relatively unimportant UNIX system can stir up a lot of trouble.

Computer use is growing at a phenomenal rate. You may have a firewall or an unlisted phone number, but with a billion users and abusers out there, you just can't hide. Most of my attacks were on systems I found at random, and there are increasing numbers of people like me. When a builder builds a home, he takes a little time to make sure doors and windows are lockable and can't be forced open. Doesn't it make sense to do the same for your UNIX system?

People say that no computer can be 100% secure. But can anything? The goal is to find a reasonable level of security, without too much effort or too much neglect. A little attention can prevent a lot of problems. I know. I want you to know too.

Section 1 - Common Sense Security

You've heard them so many times that you could recite them in your sleep. Rules like, "Don't write down your password." and, "Remember to log out." Do I need to repeat them? Yes. Security isn't about rules at all. Hackers don't sit down and follow a flowchart, so neither should you. Security is a way of thinking, a simple "What if?" that should accompany everything you do. Once you get used to it, it'll be easy.

Passwords Programmers would like people to believe that if a potential intruder doesn't know a password there's nothing he can do. And banks would like you to believe that if a potential thief doesn't have your mother's maiden name, there's nothing he can do. Four nights ago I slept in a bunk above a man who robbed a bank with a broken, unloaded pellet gun. UNIX security, like any security, has many issues and scenarios to consider. So while the password isn't everything, it's a good place to start.

A compromised password, even without access, is a security breach in itself. It's a technique used by beginners, but still kept in mind by the pros. There are a lot of ways to find someone's password, and be assured that clever troublemakers find new ways each day.

A password is a secret kept between the user's memory and the computer. In fact, a UNIX system needs not, should not, and most often does not, know a user's password.

A brief explanation of DES - On UNIX systems, passwords are stored using an algorithm called the **Data Encryption Standard** or DES. There's no need for me to explain the mathematical theory behind it, but the idea is that the password code on a UNIX system can only be used to verify a password, and, supposedly, (countless brilliant researchers have tried to show otherwise without success) cannot be directly reversed into the user's password. Perfect? Hardly. Computers are so fast that they can take the password code and attempt to verify millions (with that number increasing daily due to faster computers) of possible passwords in a single second. That's why people say, "Don't use a name or word." and, "Use non-alphanumeric characters." But still, you don't want any user on your system to have infinite guesses at another user's password, so the password code was made secret.

Shadowed password files - In the beginning, password codes were stored in the file */etc/passwd*. That file, however, is also used to associate user names and UIDs (user numbers), "real names", home directories, and default shells. So when people started attacking password codes, */etc/passwd* couldn't simply be hidden from normal users. Instead, vendors replaced the password code field with a character or two that is meaningless, and put password codes in a separate file that can only be read by programs and users with authority. The name, location, and format of the separate file varies depending on the type of UNIX system.

```
      Is your password file shadowed? Issue this command:
            grep root: /etc/passwd
If the field following the third column in the first line is longer than
five characters, your password file may not be shadowed. Contact your
vendor immediately.
```

The trial-and-error process of cracking the DES code is well known, and many programs have been written to do it. Some system administrators use these cracking programs to test their own users' passwords. This gives them a false sense of security. You can bet that a hacker has a better password guessing system than you. But let's think of password security in a more general sense. If the password is a secret between the user's memory and the computer, it should never be written down, spoken aloud, nor displayed on a screen. (CRT monitors emit radiation from all sides which can be received, even through walls, and reconstructed into an image. Oh yeah, people can look over shoulders, too.) I'll refrain from telling you about the

laser device that can hear a conversation through glass from a mile away. This is supposed to be practical security, right? My point is this: whether it's the janitor walking by or a multi-million dollar spy satellite, common sense every step of the way still helps.

There's always the issue of new accounts. Ideally, you'd like to hand the user the keyboard and have him or her enter a new password. Unfortunately, that's not always possible. Never send a password in unencrypted e-mail. E-mail travels through and is stored on multiple computers before reaching its destination, and you can't be sure that all those computers are secure. Criminal hackers almost always search a compromised computer's e-mail for passwords and other secrets. It's just one way that a criminal hacker can use one compromised computer to easily get into another. If you must snail (physical) mail a password, contact the recipient for confirmation of receipt and to ensure that it was the intended recipient who used it. Also, be sure to have the system force the user to change the initial password immediately. That way, an old piece of paper won't turn up and cause trouble down the line. Some systems use the *passwd* command to do this (often called "password expire now"). See *man passwd*. It's also a good idea to have passwords expire on a regular basis to prevent lasting threat, but if users have to change their passwords so often that they start forgetting them, it can be more of a detriment than a benefit to security.

UNIX has the ability for accounts to have no password (or, on some variants, a blank password). Never use it. Sometimes it's tempting to think, "It'll only be there for a few minutes, why bother?" Once into a UNIX system, I can usually ensure my future ability to access it within seconds. And what if you get distracted and forget about the account? I've seen it happen several times.

If you're setting up a restricted public access account, you may wish to give out the password or have no password. There is always a more secure way. Consider a separate connection daemon.

Whom to trust Admittedly, I'm bitter and cynical about the topic of trust. But again and again I have found systems that give user access-- or worse, superuser access-- to too many people. This creates many complications. When you give superuser access to anyone else, even someone who would never betray you, you are almost doubling your risk of an attack. Nothing is 100% secure, so it's simple probability. If your superuser account is 99% secure, and you make another, your level of security is 99% of 99% secure, or 98.01% secure. And that's assuming that the new superuser is as careful and security-minded as you, which is sometimes far from true. I've seen systems that create accounts, both ones with superuser access and without, that they forget about until some criminal hacker finds and uses them. I suggest that you separate all users on a large UNIX system into groupings that are not interdependent and put them on separate machines, especially if you're burdened to the extent that you feel you need multiple superusers. Some people might argue that this multiple system approach is inefficient. I agree, for small UNIX systems. But when you start to reach a large size, separating into smaller systems can improve reliability and security, and reduce hardware expenses. Think ahead- when you start with a small UNIX system, classify users and tasks to make it easier to separate in case the system grows.

Physical security When you think about computer security, you probably don't think about chains and padlocks. It is frequently overlooked, but I can't thoroughly address UNIX system security without mentioning it. If someone can physically access your computer, he or she can obtain superuser access. What's worse, he or she can probably even walk away with the computer. I'm not an expert on this aspect of computer security, but I have seen mistakes that you should avoid. If your computer is in a locked machine room, you have an advantage over a workstation arrangement. Regardless of your arrangement, keep this in mind: Locks can be picked, cables can be cut, windows can be broken, and doors can be left open. I've seen all four happen. I'm sure that fellow inmates here have done such things to obtain less than the value of a typical UNIX computer. Most physical security devices are not designed for unsupervised use. The idea is to give someone a chance to see or hear the attempt and stop it. Never assume that an unlocked building or room won't have trespassers.

In a way, common sense security is all you need to keep your UNIX system secure. The

rest of this book will show you how to apply common sense to more technical aspects of UNIX system administration.

Section 2 - File Permissions

Even small UNIX systems have thousands of files, and each file has associated with it a certain set of permissions.

The nature of permissions, chmod, ls -l Using the familiar command *ls -l* to list files, here is a typical file:

```
-rwxr-xr-x root  bin   32767 Wed Oct 8 13:30  test
```

From left to right, we have: permissions, owner, group, size, date, and name. Every position in the 10-character permissions field is significant. The first indicates the type of file. This example is of type '**-**', indicating that it is a normal file. We will examine other types later. The next three indicate the permissions that the owner has. '**r**', '**w**', and '**x**' are all present, meaning that the owner (in this case, **root**) can read, modify, and execute the file. The next three characters are for the group (**bin**). Members of the group (as specified by referencing the fourth field in */etc/passwd* lines with entries in */etc/group*) can read and execute the file, but not modify it (because there is a '**-**' where there would be a '**w**'). The next three characters work the same way, and apply to users who neither own the file, nor are members of the file's group.

Directories are slightly different. Consider this:

```
drwxrwx---  root  daemon     1024  Wed Oct 8 13:30 syslog
```

Note the '**d**' for directory in the first position. The next three groups of three characters apply to the owner, group, and others, just like a normal file. However, the meanings of '**r**', '**w**', and '**x**' are slightly different. '**r**' denotes permission to list files in the directory. '**w**' denotes permission to add or remove files in the directory. '**x**' denotes permission to change to the directory, which is needed to access specified files or subdirectories in the directory.

The *chmod* command is used to change the permissions of a file or directory. To change the permissions of a file, you must have write access to its directory and you must own the file or be superuser. I'll spare you the lesson in octal number systems and briefly explain the simpler syntax of *chmod*:

chmod u+x **filename**

'**u+x**' means that you are granting ('**+**' to grant, '**-**' to remove) access of '**x**' (execute for a file, change to directory for a directory-- can also be '**r**' or '**w**') to the owner ('**u**' is owner, '**g**' is group, and '**o**' is others).

evil, evil suid A particularly dangerous feature of the UNIX filesystem is the **suid** bit. This allows an executed program to have a different level of access than the user who executes it. For example:

```
-rwsr-sr-x root  security   32767 Oct 8 14:05 /bin/passwd
```

To change a user's password, the *passwd* program needs to have root authority. The first '**s**' (where an '**x**' would normally be) allows the program to have a level of access equivalent to the file owner, **root**. The second '**s**' allows it to have access to the group named **security**. Of course, *passwd* also needs to know who is running it so it changes the correct user's password. Suid programs can determine who is really running it and act appropriately if correctly written. Obviously, if a program is not made to have suid, the results can be chaotic. One quick, easy way for a criminal hacker who has gained root access to keep it is to make a program which allows the user to run other programs (like a shell) be suid root. This program can then be hidden anywhere on your filesystem for later use. You might want to validate all suid files on your system on a regular basis.

Unfortunately, even programs written intended for suid aren't written securely enough.

One example was a popular mail program which inadvertently allowed crafty abusers to read and write any file on the system. Unless you write all your own programs, you probably can't do much about a program with an undiscovered flaw. My advice is that you always have the latest versions of suid programs on your system, because programmers usually fix the problems as soon as they are discovered. I'll address related dangers in Section 5, "Stack Smashing".

The *ls -lR* command can be used to recursively list files and their permissions, and makes it easy for both hackers and system administrators to automatically search for permissions and suids that may compromise a system's security. Be sure you do it before they do.

the PATH... just what are you running? All major command shells have some notion of a path- a listing of directories to search for issued commands and includes */bin* and */usr/bin*. The problem here is that the path is searched in order, with only the first match being executed. If a malicious user has write permission to a directory contained in the path, he or she can place trojans-- programs which look like something you want to run, but really aren't-- in the path for other users to stumble upon. A hacker could rename a common program, like *who*, to '**who.old**', and make a *who* program which did bad things before running '**who.old**' so the victim would never see what really happened. Simply put, giving write access to a directory in a user's path is like giving out that user's password. Check your system.

Of course, write access to a file that a user normally executes is just as bad. Generally, executable files should only have write permissions for the owner. If a user writes a program which is then shared with other users, the system administrator should use *chown root* **filename** to prevent that user-- or someone using that user's account-- from modifying the program after others learn to trust it.

filesystems without permissions Some UNIX systems have the ability to mount a filesystem that does not use file permissions. Typically, a mount option specifies what permissions apply to all files on that filesystem. Sometimes, UNIX systems designed for multiple users run on computers that once were, or sometimes still are, used as personal computers. When a filesystem not made for multiple users is made accessible from UNIX, system administrators don't always stop to check whether or not any user can poke around there.

temporary files Many programs need temporary files for various purposes. Most of these are stored the */tmp* directory, but temporary files can be anywhere the program has write access. Temporary files include not only short-term data storage, but also interprocess communication. In many ways, a sockets and lock files are like temporary files. Sockets are used to send messages between processes and/or devices. Lock files are indicators to other programs that files or devices are in use and shouldn't be disturbed. All of these files on a UNIX filesystem have permissions, and even though the files don't stay long, they need correct permissions to prevent unauthorized access or tampering. Sometimes, a file may only exist long enough for a process to stop and start another, but this could still leave a "race condition" where a malicious program races to get at the file first. Sometimes programs create files without regard to their permissions, in which case the *umask* is used. The umask is a setting which determines the default permissions for new files. umask is in octal. *umask 077* allows only the user who created the file to access it, and is generally a good choice.

Other times, however, the programs themselves have problems built into them. Section 5 discusses appropriate actions to protect against these problems. With temporary files, a program should always set proper permissions and check filenames. The latter is important because programs may overwrite other programs' files, especially when symbolic links are maliciously created. A symbolic link is a filesystem entry that indicates that a particular filename really points to another filename. If the superuser runs a program that writes temporary data to, for example, */tmp/data*, any user could type *ln -s /etc/passwd /tmp/data* and when the superuser runs the program the passwd file would end up getting the data, thus corrupting both the temporary data and the passwd file, even though the user didn't have permission to write to /etc/passwd himself.

To summarize, every file on your UNIX system serves a purpose and that purpose can be defeated or abused if its permissions are wrong.

Section 3 - Login daemons

Login daemons are the primary ways to connect to your UNIX system. There are other ways, which we will examine later, but first we need to understand typical connections and how they can be abused. Login daemons typically wait for a connection, authenticate the connection, then provide some type of service.

getty One of the simplest login daemons is *getty*. Getty does not use virtual connections like TCP/IP. Instead, getty handles text console and serial port connections, including modem connections. Getty is being used less and less on UNIX systems as digital network protocols replace older, slower modem technology. Even dialup internet service providers typically use a specialized terminal server to handle modem connections because of better performance and less cost.

Attaching a modem to a UNIX system adds a great deal of concern for security. Someone with root access (or whatever access permissions are set for the modem device) could use the modem to dial anywhere and connect to other computers, sticking you with the bill. Also, many modems support an "escape sequence" (normally '+++') which puts the modem into command mode without always disconnecting. The result can be that if an intruder logs in over the modem, establishes a connection with another computer, and sends the escape sequence from the other computer, the other computer can gain control of the modem, even without root access. Control over a dial-in modem also means that the troublemaker can answer incoming calls, issue fake prompts, and obtain other users' passwords.

If you must attach a modem directly to a UNIX system, take these precautions: Realize that root access means free calls on you. If the modem is used only for incoming calls, see if your telephone company or PBX can restrict outgoing calls. Unfortunately, many times emergency (911) calls can't be restricted, and can result in hefty abuse fees. Be sure to set the modem to ignore escape sequences, but disconnect when DTR (data terminal ready) drops (usually AT&D2). This provides a safe way for getty to control the modem. Set the modem to use CD (carrier detect) (usually AT&C1), as opposed to leaving CD always on. This way, if the user hangs up unexpectedly, the next caller won't be able to get the previous user's login session. (Otherwise, even someone without an account could get into your system on someone else's. It has happened.) Finally, have users report problems logging in on the modem, and have them change their passwords when any problem occurs.

inetd A wide variety of connection protocols have been developed for UNIX, but due to popularity and flexibility, TCP/IP is rapidly pushing out the rest. There are two TCP/IP protocols suitable for transferring significant amounts of data, TCP (transfer control protocol), and UDP (universal datagram protocol). These two protocols are supported by *inetd*.

Inetd is a "super-server" for TCP and UDP data. It detects incoming data and sends it to the appropriate daemon. Inetd is **not** a login daemon itself, but an understanding of it is important because it runs login daemons.

The main advantage of inetd is that one relatively small daemon can be used to wait for many different connections. Daemons for the individual protocols are only started as needed, keeping more RAM available. Not all TCP and UDP services need to use inetd, however, and there are instances where it might not be beneficial. For example, a busy web server has to support many connections each second, and if inetd needs to run a new program for each connection, the service may be slower than a "stand-alone" server.

TCP wrappers The biggest security advantage to an inetd approach is that it can support TCP wrappers. TCP wrappers are programs that act like traffic cops for incoming connections. They can log each connection and decide whether or not to allow it. One little trick that TCP wrappers can protect against is false reverse DNS records. Many protocols use the name of the connecting computer to determine what access to give it. In addition, logs frequently store names of computers that connect. When a computer connects, however, the server only has its IP address, not its name. Reverse DNS lookup obtains a name from an IP address. This name is supposed to translate back into the IP address when forward looked-up, but if it doesn't, a computer is claiming to be someone that it's not.

Inetd can also protect against "brute-force" attacks, where connections are repeatedly make to a server, resulting in an overloaded server or guessed password.

telnetd The most common way for logging in to a UNIX system is using telnet. Telnet simply establishes a TCP connection on port 23 and exchanges some data about terminal type. The login process is the same as a modem or text console.

One risk that telnet shares with most other protocols is its ability to be "sniffed". A telnet connection sends all data, including the password, in unencrypted form. This means that any computer along the connection path is able to read and scan the data. Your network may be secure, but your router connects your network to another, to another, and so on, up to 30 networks. Anyone with superuser access on any system of any one of those networks can eavesdrop on the telnet connection.

rlogind Rlogin functions a lot like telnet, but uses UDP and has a dangerous authentication process. When rlogin connects, it sends two login names, the user on the client machine and the server account to log into. Rlogind first examines */etc/hosts.equiv*. If it finds the connecting computer's name, and both login names are the same, the user does not need to enter a password. If it doesn't find the connecting computer's name, it checks the user's home directory for *.rhosts*. If .rhosts contains the connecting computer's name and originating login name, again, no password is needed. What's worse, rlogin allows the character '+' in a file to represent any user or any host. Simply putting '+ +' in a user's .rhosts file lets anyone access the account.

It's not hard to see how such an authentication method might be convenient and help prevent password-entry carpal tunnel syndrome. It's also not hard to see how dangerous it can be. It's an easy way for someone to allow himself or herself to get back into an account that was compromised. It's also an easy way for trojan programs to make an account accessible.

True, a password can't be sniffed if it's never entered, but a computer along the network path can "spoof". Spoofing is like sniffing, but it involves sending fake packets. When along the same network path, a computer can claim to have an IP address belonging to someone else. The result is like fake reverse DNS, only harder to detect. Rlogind thinks it's dealing with a trusted computer.

ftpd The File Transfer Protocol (FTP) is TCP/IP's main method for transferring files between accounts. Unlike telnet and rlogin, *ftp* is considered a "non-interactive" login because programs are not executed. Restricting a user to only FTP logins, however, does not prevent a user from executing programs. (See *.forward* under *sendmail* in the next Section.)

Typically, ftpd will only allow access if the user's default shell is listed in */etc/shells*. This is supposed to prevent restricted-use accounts from gaining additional access. Other login daemons and shells, however, are likely to see an account and matching file permissions and allow access without necessarily checking */etc/shells* like they should. When you make an account, don't depend on things other than file permissions to restrict access, because adding a login daemon in the future could otherwise be risky.

FTP also supports "anonymous" login, where no account or password is needed to access a limited group of files. Don't allow anonymous uploads to an anonymously accessible directory (use *-wx*, not *rwx*). Doing so allows anyone to store and share anything through your computer, and you could be held responsible if copyrighted material is exchanged without permission.

Anonymous FTP uses an *ls -l* type file listing, so the anonymous directory tree may need an */etc/passwd* file to associate file owner numbers with names. This passwd file should only contain accounts that need to be named, because an account name is half (though admittedly, the easier to find half) of the account/password combination. As for the other half, always use a shadowed /etc/passwd file in the anonymous directory tree.

sshd The Secure Shell (ssh) is an easy and effective answer to the problem of sniffing and spoofing. If you don't have *ssh* and *sshd* on your UNIX system, get them. Basically, ssh encrypts transferred data so it cannot be intercepted or faked.

SSH works by using public key encryption. The concept of public key encryption is simple, though a full understanding of it requires incredibly complex mathematics. The client and the user each sends a code to each other. This code, called the public key, tells how to encrypt information. The public key cannot be easily reversed to find a way to decrypt the

information. For that, the private key is needed, which matches the public key and is never sent over the connection. So only the intended recipient can decipher what is sent.

SSH implements a slightly safer alternative to rlogin's .rhosts authentication. To establish a trusted relationship between accounts, an identity file is used. This file contains a cryptographic signature that identifies you. Of course, an intruder can replace your identity file with his own if he gets into your account, so beware.

When implemented correctly, SSH can exponentially increase your system's security. SSH, unlike most UNIX programs, was designed with security in mind, so the common problems of UNIX security aren't present. If security is bypassed, SSH will usually warn you. My opinion is that all systems should make sshd available for use, if not required, as a replacement for unencrypted login daemons.

Section 4 - Non-login daemons

Security should never be about generalizations, but generally, login daemons are like the doors on a house. Exceptions will occur, but you'd expect that the door have a lock and not be too easy to kick in. Non-login daemons are like the windows. They're not made for people to pass through, but if you're not careful, it still could happen. The biggest concern is that non-login daemons require no account or password. Thus, any vulnerability in them may make your system open to attackers that would otherwise not be able to get a foot in the door. And people who don't normally use your system aren't likely to have as much respect for its successful operation. The solution is to make sure your non-login daemons do exactly what they're supposed to do and nothing more.

inetd Inetd is also used for some non-login connections. The file */etc/inetd.conf* specifies options for individual services. First, you may want to disable any service you don't use by commenting them out with a '#'. Every service you run has a chance of having a security flaw, so why have a tiny risk if there's absolutely no benefit?

Non-login daemons can also be flooded with an unreasonable number of requests. Inetd versions or wrappers which prevent this are a good idea. Also, any UDP entry in */etc/inetd.conf* should have the 'wait' parameter. Otherwise, an attacker can send a flood of UDP packets and inetd has to run the server for each one.

inetd.conf entries also let you specify what user the servers should be run as. Be sure to also check suid settings on the servers themselves and any programs they use in turn. Don't run a server as root unless absolutely necessary. Usually the user **nobody** or **daemon** will provide a more restricted level of access.

fingerd The finger daemon is a very useful tool for finding users, but I must admit that it presents many security risks. The first common risk is the *finger* @**host** form. Unless configured otherwise, this returns to anyone a list of users logged on, login/idle times, etc. For one thing, this gives away half of the account/password pair. If a perpetrator wants into your system, and knows how to intimidate or spy on one of your users, there's trouble. Also, a perpetrator can find users that have been idle for a long time, which may indicate that they forgot to log out. *fingerd* also returns the origin of users' connections, so the perpetrator can quickly find the logged in workstation.

The *finger* **user@host** format returns more detailed information. 'user' need not always be an account name, either. On most systems, fingerd searches users' real names as well, so 'smith' will find 'Mary Smith' and, more dangerously, 'account' will find 'Test Account' or 'Temporary Account'. This format also returns each user's last login time (if implemented on your UNIX system), so criminal hackers can schedule their attacks at times when they're less likely to be seen, or aim their attacks at accounts that are rarely used.

finger **user@host** also returns the entire *.plan* file in the user's home directory and the first line of the *.project* file. If fingerd runs as nobody or daemon this usually isn't a problem, but if it runs as root, beware of symbolic links. Unless fingerd is made to detect symbolic links (I've found a few versions that aren't), a user can issue a command like *ln -s /etc/top.secret.file ~/.plan* and then finger his or her own account to read any file on the system (in this case, */etc/top.secret.file*). You're best off not running fingerd as root, even though you'll have to teach your users to set proper permissions on their .plan and .project files.

httpd Sometimes I wonder if http will be the only service used on the internet in the future. It's sad too, because it's such an inefficient protocol. My main concern, however, is the countless system administrators who put together a fancy web server to show off and end up compromising their systems. There are a lot of ways to use and abuse web pages, but here I'm only addressing ways in which a web server can be a risk for a UNIX system's security.

If you don't use inetd for your web server, and you want to use TCP port 80 (the standard http port), you'll need to run the server as root. (Ports 1 through 1023 are only usable by root.) Fortunately, httpd provides a configuration file which lets you tell it what user to do its work as.

Even nobody, however, is still somebody. One of the most powerful features of a UNIX webserver, as opposed to most other operating systems, is its flexibility of CGI (Common

Gateway Interface) programming. CGI allows you to write a program that your webserver runs to determine what to send to the client. This program can take a picture of your office, count web page accesses, save an HTML form, search a database... almost anything. But how safe is your program? A faulty CGI script opens a door into your system. Just because your CGI works, doesn't mean that it can't be abused. See Section 6, "Safe Scripting".

sendmail Equallly as popular as httpd is *sendmail*. sendmail is the program that handles internet e-mail. There have been over a dozen major revisions in the sendmail program. Basically, when sendmail runs as a daemon, it listens on port 25 for incoming mail. When it gets mail, it routes the mail to the appropriate mailbox or remote system (via port 25 on the remote host specified by the MX field of the domain name service). There are two files (at least) that sendmail checks before delivering mail to a user. The first is the /etc/aliases file which lists addresses and how to handle messages for them. The second is ~/.forward, where ~ is the destination user's home directory. There are delivery formats for forwarding to another user (remote or local), writing to a specific file, or as the standard input to a program. The latter is the tricky part. An entry in the format of *|/directory/program* feeds the message to *program* every time the user gets a message. Accounts which are only supposed to have ftp access can gain shell access by using this technique. Fortunately, the program runs with no more authority than the user himself has. Many versions of sendmail allow you to disable the pipe format, limit it to specific programs, or specific users. If you disable it, keep in mind that useful implementations, like extended abscence autoresponders, will not work. Also, make sure that .forward will only be recognized when owned by the user. .forward is not the only vulnerability found in sendmail, but it is the most common and unique, which warrants its mention here.

identd UNIX security pros would probably think me less thorough if I didn't mention *identd* in this Section. Identd is a non-login daemon that identifies users making outgoing connections to other computers. The other computer connects, issues the originating and destination port numbers, and identd names the user. The idea here is to distinguish activities on multi-user systems and aid in logging. Logging, however, is not a preventive measure, so it's not a major priority and I'll discuss it later. Personally, I'd rather spend $5 on knee pads than $500 on an x-ray and fracture diagnosis, wouldn't you?

low-level flooding I've already mentioned flooding with TCP and UDP, but there are a couple of flooding concerns inherent in TCP/IP that you should know about. The first, and simplest, is ping flooding. IP has a low-level packet type called ICMP (Internet Control Message Protocol). One ICMP sub-type is the ping. Named (I assume) after the game ping-pong (table tennis), ping involves throwing out a packet for some destination to return. The idea is great; it can be used for testing network connections and availability. Too much pinging, however, is a flood. A UNIX kernel can be made to protect against this, but even unreturned ping packets can overload a network. A firewall can be configured to prevent this before it enters your network.

Another type of flood vulnerability inherent in TCP/IP is called SYN flooding. To understand SYN flooding, you have to examine how a TCP connection is made. The client first sends a SYN, the server acknowledges it (ACK1), and the client acknowledges the acknowledgement (ACK2). To SYN flood, a client sends multiple SYN packets without acknowledging the server's acknowledgement. Because each SYN packet is considered by your server to be a connection in progress, relatively few are needed to overload the server. The result is that legitimate connections can't get through. Kernels can be written to only allow a limited number of connections-in-progress from each host or subnet.

Flooding is called a "denial-of-service" attack, because it denies your server its ability to use and offer services. Technically, it's more of a nuisance than a risk. So unless you require maximum system availability at all times, you may be able to postpone major kernel or network changes until a problem occurs. See, my emphasis on prevention is within reason!

Section 5 - Stack Smashing

You'd think that when someone writes a program, it will work, and any flaws would be limited in scope. "Stack smashing" is a technique which exploits a frighteningly common programming error and has virtually no limit to what it can do.

The error is called "buffer overflow" and is possible in most traditional programming languages including UNIX's favorite, C. There are some languages where buffer overflow is impossible, but only because they incorporate a way to detect and prevent it, which usually hinders performance.

To understand buffer overflow you must first examine how a program is stored in memory. Whenever a program is executed, it is assigned a chunk of memory. Every function in the program gets a smaller piece within the chunk. These smaller pieces are divided into two uses, data and code. Basically, the code tells the computer what to do, and the data is where the computer does it. But what if the program finds that it needs more data space than it asked for at the start? Additional memory can be allocated as needed, so it shouldn't be a problem. The problem occurs when programs go ahead and use more memory without proper allocation. This excessive memory use, overflow, intrudes upon the memory which is for the code. The code gets messed up, and when the computer tries to follow it, anything can happen.

An historical example should help clarify the problem. The first versions of fingerd (I doubt any such versions still exist) read a user's entire query into a buffer (data memory assigned at the start). The programmer assumed that the request (user@host) would never exceed a certain length (say, 136 characters). A reasonable assumption, you would think. Then one day, someone entered more characters. The additional characters went beyond the assigned memory, into the code space. The computer then executed the additional characters as machine code. All an attacker had to do was have 136 characters of anything, followed by his or her own evil program, and the program was executed.

Not all buffer overflows occur in service daemons, but when they do they are especially dangerous because people probably don't even need an account on your system to exploit them. Even in user programs though, stack smashing can result in granting superuser access if the flawed program is suid root.

Stack smashing requires a thorough knowledge of the buffer overflow flaw as well as your operating system and processor's machine language. Nonetheless, stack smashing programs are common, and if a suid root program on your system has a buffer overflow, preventing a user from getting root access is impossible.

Solution The only solution is to get programs without flaws. Impossible, but you can stay up-to-date with bug fixes. Usually when a buffer overflow problem is found, the program author fixes it immediately. Update your software before someone attacks. Most authors and vendors provide notices or bug fixes. Don't ignore them, they usually take only a minute to install.

Writing your own programs You probably don't have the time to write every program you'll use on your UNIX system, but once in a while you may need a custom application. Keep buffer overflow protection in mind.

A buffer is an array. In C, you can't exactly have an array grow in size when you need it. Whether you declare your arrays in the source code or use malloc() to allocate them while the program is running, you have to specify a finite size. When writing to an array, don't exceed the array size. It's simple. Unfortunately, some standard C functions don't follow this rule. fscanf() and scanf() when used to read strings have no bounds checking. Use fgets() and gets() instead. strcpy(), strcat(), and others may also overflow buffers. strncpy() and strncat() let you specify a maximum length to avoid this. Generally, if you don't tell a function how big your buffer is, it won't know.

Section 6 - Safe Scripts

UNIX's wide variety of scripting languages make it extremely flexible. But before you sculpt your own functional masterpiece, keep security in mind. Even the most skilled and experienced programmers forget to stop and check for unexpected vulnerabilities.

Passing parameters One of the biggest scripting mistakes I've seen is restricting a user to a script-based menu that doesn't properly handle input. This is increasingly problematic with CGI scripts on web pages. Let's say that for whatever reason you're making a script that lets people send e-mail. You prompt the user for a destination address and then execute the command *mail* followed by the destination address. For a destination, the user enters |rm -rf /. Your program executes 'mail |rm -rf /', and your entire system (if running as root) is erased.

'|' is just one of the characters that shells interpret in a special manner. Called 'pipe', it takes the output of the first command and feeds it into the second command. ';' is sometimes used to separate command lines. '<' feeds the following file into the program (i.e., mail user < formletter) and '>' feeds program output into the file (ls -l > filelist). '$' is used to refer to an environment variable (echo $TERM). '\' is used to escape the following character so it isn't interpreted specially (echo \$TERM), and might be useful in securing your scripts. Backwards quotes containing a command are replaced by that command's output (echo `ls`). Basically, don't allow any characters unless you're sure they're safe.

Permissions Never forget permissions! If you write a script to print a page and record a billing amount, be sure the user can't print without the script or worse, edit the billing records. The way to do this would be to use suid for a special printer account and make the billing record and print device accessible only by that account.

Passwords Scripts which have to keep things secret (like passwords) aren't as simple as they may initially seem. Passwords should be encyrpted whenever possible. A password can be picked up from a file, a connection, or from memory. Passwords should never be passed as parameters to processes, because the *ps* command can show command lines to any user. Some systems use a restricted form of *ps* that only allows users to see their own processes. Also, keep in mind the common sense rules of password safety and remind the users.

Section 7 - 100% minus (never 100%)

I didn't want to include this Section at first. Security should be about prevention, not revenge or recovery. It seems that our society puts too much effort into trying to correct past problems and not enough effort into trying to prevent them. One man I know in here is here because he wrote bad checks. If people would have had a check validation system in place, there would have been no need for detectives, lawyers, district attorneys, and the cost of incarcerating someone. But I've already admitted that no computer is 100% secure, so it would be unfair for me to neglect the topic of "unavoidable" problems that arise.

Backups I think that just about everyone, especially everyone here in jail, has wished that they could go back to a previous stage in their lives. Well, you can't, but your computer can. Backups have a lot of uses for a lot of problems. Human error, drive failures, and malicious attacks can all destroy a lot of hard work. Backups can be an efficient way to replace it.

Tape drives are the most popular mass-storage backup devices. They have many favorable features including removable media, unattended operation, and relatively low cost. Removable media is important because anything that stays physically accessible by the computer is virtually accessible by the programs it runs. Many tape drives available today can eject a tape when a backup finishes, but not reload it unless someone puts it back in. Backups aren't nearly as effective if they're just as vulnerable as the originals. Backups aren't effective at all, however, if you don't use them. Unattended operation means less hassle for you, which means a higher likelihood that you'll backup on a regular basis. Remember that old exercise machine you have packed away? When you bought it, 20 minutes a day didn't seem like much for a healthy body. But routines get tiresome and the value of your time changes. Finally, you have to consider the cost of backups. You can't foresee how many times you'll need to use your backups, but consider the cost of replacing your data. Estimate how long it would take you and multiply that by what you think your time is worth. If the resulting amount is more than the cost of a backup device, just one emergency would make the purchase worthwhile. And if you're anything like me, there are some things that just can't be replaced.

How often you should backup, and how many backups you should have at a given time, depend on your use. If a thousand researchers use your system to store their findings, you may want to backup several times a day. Other systems, however, may find it reasonable to backup only monthly. The number of backups to have on hand need not be high if you can detect problems right away. But sometimes you don't notice a problem until it has already been duplicated to at least one backup, in which case multiple backups would be important.

You don't need a backup device large enough to store your whole system. If you have a CD-ROM with your UNIX operating system, you only need to backup user files, configuration files, and updated programs. If, however, you don't have your original install CD-ROM, be sure you have a recovery disk that can get data off of your backup in case everything gets wiped out. I once accidentally wiped out a computer's Flash ROM BIOS, leaving it with no way of even booting a disk. All my information was still in the computer, I just had no way of getting at it short of buying a new computer to put the hard drive in. Fortunately the computer had a built-in non-erasable routine to load a new BIOS off of a disk.

Paper Paper has been around for thousands of years, so people feel a sense of security in dealing with it. Sure, no computer error is going to cause your printouts to go up in smoke (unless computer manufacturers get really sloppy), but remember why you got a computer in the first place-- paper is hard to transport, expensive, difficult to search and manage, and non-reusable. As much as I'd like to save the trees, I won't say that paper should never be used (after all, what am I writing this on?), but don't let the threats of the new technology throw you into the past.

Logging When something goes wrong, there is a sequence of questions that anyone asks: "What went wrong?", "Why did it happen?", "How can I fix it?" and, "How do I prevent it from happening again?" Logs help you answer the first two, so you can then concentrate on the last two.

I think the story goes something like this: Long ago, foresters would cut down trees and

send them down the river so they could be used to build homes. Of course, these foresters wanted to keep track of how many tree logs they sent each day. They tried counting in their heads, but after a long day of hard work they began to lose count. So instead, each day they would take a branch and make a mark in it for each tree log they sent. These branches were called the logs, and kept an accurate report of the work they did. (Can you tell I'm not an epistemologist?)

On your UNIX system, logs record any notable events or errors with the time they occurred and other useful information. Many types of programs, webservers, for one, have their own logs. Others use a standard logging interface called *syslog.* Syslog runs a special daemon, syslogd, which takes messages of interest from programs, timestamps them, and stores them in a common file. Syslog can also exchange log entries with remote systems, providing a constantly updated backup. Remember, though, that security compromised on one system means that syslogd can be disabled and other systems on the network become more vulnerable.

Another log worth mentioning is the last logins log. When someone logs into a UNIX system by rlogin, ftp, ssh, telnet, or any other protocol that uses /bin/login, the system records the user name, tty (or 'ftp'), connecting location (IP address/hostname), and time of connect. Duration of the connection is added when the user logs out. This information is usually stored in a file called *wtmp* (the directory of the file depends on the type of UNIX) and is readable with the *last* program (see *man last*).

Any logs stored on the system can be modified or destroyed with superuser access. For this reason, security experts recommend "append-only" media for storing logs. Append-only media, such as a printer, can record additional log lines but cannot modify or destroy ones that have already been recorded. (It would be amusing to see someone try to get a printer to reverse-feed the paper and black out previous entries.)

When system administrator access is obtained by an intruder, nothing is safe. The best thing to do is inform all users and have them get new passwords. Restore from a backup that you're sure was taken before security was compromised, fix whatever let the intruder in, and then implement the new passwords. All this must be done at one time, or the intruder could get his foot in the door again. Don't try to save face by not telling the users immediately. I've seen system administrators try; it doesn't work. Also, check systems your system frequently connects to, to ensure that the intruder didn't use your system to get into others.

I hope you never have to rely on anything in this Section, but, like an ambulance, it's good to know it's there.

Conclusion

I've now provided you with the ability to prevent every broad type of UNIX security attack I know. I've even told you what to do if you mess up. Most importantly, though, I hope I've prepared you to be a security-minded UNIX system administrator. No two UNIX systems are exactly alike (why would you want them to be?), and I can't list for you all the problems that will come up in all possible installations. But if you keep in the mind the information in this book, reviewing it if necessary. I'm confident that your UNIX system can be one of the most secure of its type. Mainstream advertising will always try to scare the consumers into buying their security products, but you have the power to control your own security more thoroughly and at less cost... if you just use common sense. I know.