Linux Filesystem Security

Part I and Part II

By Mick Bauer

(Reprinted from The Linux Journal)

Linux Filesystem Security Part II

October 1, 2004

For most of the Paranoid Penguin's illustrious four years with *Linux Journal*, I've tended to write tools-focused columns. I've described how to secure Sendmail, how to add SSL encryption to things by using Stunnel and how to get any number of other powerful security software tools configured and running.

Over the next couple of columns, however, I am going to address one of the most basic and important, yet often-overlooked aspects of Linux security; filesystem permissions. If used wisely, it will be harder for users and intruders to abuse their system privileges. If you set them carelessly, however, minor vulnerabilities can lead to major system compromises.

These articles should be especially useful to Linux newcomers who wonder what all the drwxr-xr-x gobbledygook in file listings means. But, even if you're an intermediate user—perhaps the kind who doesn't yet understand the precise ramifications of setuid and setgid—these articles, especially Part II, may have something for you too.

Prelude: Everything Is a File

Did you know that in UNIX and UNIX-like systems, basically everything is represented by files? Documents, pictures and even executable programs are easy to conceptualize as files on your hard disk. Although we think of a directory as a container of files, a directory actually is a file containing, you guessed it, a list of other files.

Similarly, the CD-ROM drive attached to your system seems tangible enough, but to your Linux kernel, it too is a file—the special device file /dev/cdrom. To send data from it or to write it to the CD-ROM drive, the kernel actually reads to and writes from this special file. Actually, on most systems, /dev/cdrom is a symbolic link to /dev/hdb or some other special file. And wouldn't you know it, a symbolic link is in turn nothing more than a file containing the location of another file.

Other special files, such as named pipes, act as input/output (I/O) conduits, allowing one process or program to pass data to another.

My point here is not to describe each and every type of file that exists in Linux or UNIX. It's to illustrate how nearly everything is represented by a file. Once you understand this, it's much easier to understand why filesystem security is such a big deal and how it works.

Commands and Man Pages

In this article, I focus on filesystem concepts rather than the precise syntax and usage of actual commands. But if you're a beginner, you may be wondering how to execute commands at all and where can you find syntax/usage help.

First, in all of my examples and example scenarios, I'm working in a terminal window. Microsoft Windows users can think of a terminal as like a DOS prompt or command window. A terminal window

provides the most direct means of interacting with Linux, that is, by letting you enter all your commands manually rather than by triggering them with mouse clicks.

To start your own shell session from GNOME, click the Main Menu button and select System Tools→Terminal. In KDE, the terminal command is called konsole, and it has its own icon on the taskbar, a clamshell in front of a computer screen. Alternatively, you can start the Run Program dialog and type konsole at the prompt.

For fast help with practically any Linux command from within a terminal/shell session, you can type that command followed by the **--help** option. For example, if I can't remember all the command-line options for the ls command, which lists files and directories, I enter the command ls --help.

The **--help** option is quick, but it doesn't work for all commands. Even when it does work, its output can be quite terse. The best way to get command help is by using the man command. Man pages provide complete instructions on how to use most Linux commands and are present on practically all UNIX-like systems. To see the man page for the ls command, for example, type the command man ls. Within the man page listing, press the spacebar to advance forward one page, the B key to go back one page and type */somestring* to search the man page for *somestring*.

But, what if you don't know the name of the command you need? That's what apropos is for. For example, type apropos list to see a variety of commands that list things, and then pull up a man page for whichever of those commands seem to be what you need.

Users, Groups and Permissions

Actually, two things on a Linux system aren't represented by files, user accounts and group accounts, which we call users and groups for short. Various files contain information about a system's users and groups, but none of those files actually represents them. A user account represents someone or something capable of using files. This is to say, both human beings and system processes can use user accounts. For example, a user account named webmaster typically represents a human being who maintains Web sites. But the standard Linux user account lp is used by the line printer dæmon (lpd); the lpd program runs as the user lp. I explain later what it means for a program to run as one user vs. another.

A group account simply is a list of user accounts. Each user account is defined with a main group membership but may in fact belong to as many groups as needed. For example, the user maestro may have a main group membership in conductors and also belong to pianists.

A user's main group membership is specified in the user account's entry in /etc/password. You can add that user to additional groups by editing /etc/group and adding the user name to the end of the entry for each group to which the user should belong. Alternatively, you could use the usermod command; see the usermod(8) man page for more information.

Listing 1 shows maestro's entry in the file /etc/password, and Listing 2 shows part of the corresponding /etc/group file.

Listing 1. An /etc/password Entry for the User maestro

```
maestro:x:200:100:Maestro Edward Hizzersands:/home/maestro:/bin/bash
```

Listing 2. Two /etc/group Entries

```
conductors:x:100:
pianists:x:102:maestro,volodyia
```

In Listing 1, we see that the first field contains the name of the user account, maestro. The second field (x) is a placeholder for maestro's password, which actually is stored in /etc/shadow. The third field shows maestro's numeric user ID, or uid; in this case it's 200. The fourth field shows the numeric group ID, or gid—in this case it's 100—of maestro's main group membership. The remaining fields specify a comment, maestro's home directory and maestro's default login shell.

In Listing 2, from /etc/group, each line simply contains a group name, a group password (usually unused—x is a placeholder), numeric group ID (gid) and a comma-delimited list of users with secondary memberships in the group. Thus, we see that the group conductors has a gid of 100, which corresponds to the gid specified as maestro's main group in Listing 1. We also see that the group pianists includes the user maestro, plus another named volodyia, as a secondary member.

The simplest way to modify /etc/password and /etc/group in order to create, modify and delete user accounts is by using the commands useradd, usermod and userdel, respectively. I'd rather concentrate here on concepts than command syntax, so suffice it to say that all three of these commands can be used to set and modify group memberships and all three commands are well documented in their respective man pages. To see a quick usage summary, you also can type the command followed by **--help**, for example, useradd --help.

Simple File Permissions

Each file has two owners, a user and a group, each with its own set of permissions that specify what the user or group may do with the file—read it, write to it and execute it. A third set of permissions pertains to what others, user accounts that don't own the file or belong to the group that owns it, can do with the file. Listing 3 shows a long file listing for the file /home/maestro/baton_dealers.txt.

Listing 3: File Listing Showing Permissions

```
-rw-rw-r--  1  maestro  conductors  35414  Mar 25 01:38 baton_dealers.txt
```

Permissions are listed in the order of user permissions, group permissions and other permissions. For the file shown in Listing 3, its user owner (maestro) can read and write the file (rw-); its group owner (conductors) also can read and write the file (rw-), but other users can only read the file. Permissions are a little more complicated, however. Users classified as other, in terms of permissions on a particular file, can delete any file in a directory to which they have write permissions. In other words, users with read-only permission on a file cannot edit the file but can delete it if they have write permission on the file's directory.

There's a third permission besides read and write: execute, which is denoted by x when set. If maestro writes a shell script named punish_bassoonists.sh and if he sets its permissions to **-rwxrw-r--**, he then can execute this script by entering the name of the script at the command line. If, however, he forgets to set the execute permission, he is not able to run the script, even though he owns it.

Permissions and root

In practical terms, file permissions simply do not apply to the root user; root can do anything to any file, at any time. This is why it's so important never to log on as root or use the su command to become root, except when absolutely necessary. When you're root, file permissions do not protect you from your own mistakes.

This is not to say that all programs entirely disregard file permissions when you're root. If, for example, root tries to alter a read-only file using the vim editor, root must use the :w! command (force write). The normal ZZ or :w commands return an error in this case. However, many other commands have no such sanity-check feature.

Permissions usually are set with the chmod command, short for change mode. Continuing with our example, suppose maestro has second thoughts about allowing other members of the conductors group to read his list of baton dealers. He could remove the group read/write permissions using the commands shown in Listing 4.

Listing 4. Changing a File's Permissions with chmod

```
bash-$ ls -l baton_dealers.txt

-rw-rw-r--  1  maestro  conductors  35414  Mar 25 01:38 baton_dealers.txt

bash-$ chmod go-rw baton_dealers.txt
bash-$ ls -l baton_dealers.txt

-rw-------  1  maestro  conductors  35414  Mar 25 01:38 baton_dealers.txt
```

In Listing 4's sample chmod command (chmod go-rw), go tells chmod to change the group permissions and other permissions; -rw says to remove read and write permissions for those two categories of permissions, group and other. Thus, a chmod command has three parts: a permission category, some combination of u, g and o or a for all; an operator, - to remove, + to add; and a list of permissions to add or remove, usually r, w or x.

Directory Permissions

We now know how to change basic permissions on regular files, but what about directories? Directory permissions work slightly differently from permissions on regular files. Read and write are similar; for directories these permissions translate to list the directory's contents and create or delete files within the directory, respectively.

Execute is a little less intuitive on directories, however. Here, execute translates to use anything within or change working directory to this directory. That is, if a user or group has execute permissions on a

given directory, the user or group can list that directory's contents, read that directory's files (assuming those individual files' own permissions include this) and change the working directory to that directory with the command cd. If a user or group does not have execute permission on a given directory, the user or group is unable to list or read anything in it, regardless of the permissions set on the things inside. If you lack execute permission on a directory but do have read permission and you try to list its contents with ls, you receive an error message that, in fact, lists the directory's contents. But this doesn't work if you have neither read nor execute permissions on the directory.

Suppose our example system has a user named biff who belongs to the group drummers. Also suppose that his home directory contains a directory called extreme_casseroles that he wishes to share with his fellow percussionists. Listing 5 shows how biff might set that directory's permissions.

Listing 5. A Group-Readable Directory

```
bash-$ chmod g+rx extreme_casseroles
bash-$ ls -l extreme_casseroles

drwxr-x---   8   biff   drummers   288   Mar 25 01:38 extreme_casseroles
```

Per Listing 5, only biff has the ability to create, change or delete files inside extreme_casseroles. Other members of the group drummers can list its contents and cd to it. Everyone else on the system, however, is blocked from listing, reading, cd-ing or doing anything else with the directory.
Conclusion (for Now)

Those are the most basic concepts and practical uses of Linux filesystem security. In Part II, we'll go further in depth and discuss (among other things) setuid, setgid and numeric permission modes. Until then, be safe!

Linux Filesystem Security Part II
November 1, 2004

Last time, we looked at file and directory permissions from the ground up—what users and groups are and how to set and remove read, write and execute permissions on files and directories. In this column, we look at some more advanced types of permissions, explore permission numeric modes and the command umask and see how to delegate root's authority with su and sudo. This article contains more intermediate-level information than last month's, but hopefully it should make sense, even if all you know about permissions is what you read here last time.

The Sticky Bit

```
Recall last month's long listing of the extreme_casseroles/ directory:
drwxr-x--- 8  biff drummers 288 Mar 25 01:38 extreme_casseroles
```

Recall also that we set the group permissions on this directory to r-x, that is, group-readable and group-executable, so that our fellow members of the drummers group could enter this directory and enjoy the recipes stored therein.

Suppose that our drummer friend Biff wants to allow his fellow drummers not only to read his recipes but to add their own as well. As we saw last time, all he needs to do is set the group-write bit for this directory, like this:

```
chmod g+w ./extreme_casseroles
```

There's only one problem with doing that, however. Write permissions include both the ability to create new files in this directory and also to delete them. What's to stop one of his drummer pals from deleting other people's recipes? The sticky bit, that's what.

In olden times, the sticky bit was used to write a file (program) to memory so it would load more quickly when invoked. On Linux, however, it serves a different function. When you set the sticky bit on a directory, it limits people's ability to delete things in that directory. That is, to delete a given file in the directory you either must own that file or own the directory, even if you belong to the group that owns the directory and group-write permissions are set on it.

To set the sticky bit, issue the command:

```
chmod +t directory_name
```

In our example, this would be chmod +t extreme_casseroles. If we now do a long listing of the directory itself, by using ls with the -d option to list the directory's permissions rather than its contents, that is, ls -ld extreme_casseroles, we see:

```
drwxrwx--T 8  biff drummers  288  Mar 25 01:38 extreme_casseroles
```

Notice the T at the end of the permissions. We'd normally expect to see either x or - there, depending on whether the directory is other-writable. The T denotes that the directory is not other-executable and has the sticky bit set. A lowercase t would denote that the directory is other-executable and has the sticky bit set.

To illustrate what effect this restriction has, suppose a listing of the contents of extreme_casseroles/ looks like Listing 1.

Listing 1. Contents of extreme_casseroles/

```
drwxrwxr-T  3 biff drummers  192 2004-08-10 23:39 .
drwxr-xr-x  3 biff drummers 4008 2004-08-10 23:39 ..
-rw-rw-r--  1 biff drummers   18 2004-07-08 07:40 chocolate_turkey_casserole.txt
-rw-rw-r--  1 biff drummers   12 2004-08-08 15:10 pineapple_mushroom_surprise.txt
drwxr-xr-x  2 biff drummers   80 2004-08-10 23:28 src
```

Suppose further that the user crash tries to delete the file pineapple_mushroom_surprise.txt, which crash finds offensive. crash expects this to work, because he belongs to the group drummers and the group-write bit is set on this file. Remember, though, that biff set the parent directory's sticky bit. Therefore, crash's attempted deletion fails, as we see in Listing 2.

Listing 2. Attempting Deletion with Sticky Bit Set

```
crash> rm pineapple_mushroom_surprise.txt
rm: cannot remove `pineapple_mushroom_surprise.txt':
```

Operation not permitted

One more note on the sticky bit: it only applies to the directory's first level downward. In Listing 1, you may have noticed that besides the two nasty recipes, extreme_casseroles/ also contains another directory, src. The contents of src will not be affected by extreme_casseroles' sticky bit, although the directory src itself is. If biff wants to protect src's contents from group deletion, he needs to set src's own sticky bit.

setuid and setgid

Now we come to two of the most dangerous permissions bits in the world of UNIX and Linux, setuid and setgid. If set on an executable binary file, the setuid bit causes that program to run as its owner, no matter who executes it. Similarly, when set on an executable, the setgid bit causes that program to run as a member of the group that owns it, again regardless of who executes it.

When I say run as, I mean the program runs with the same privileges as. For example, suppose biff writes and compiles a C program, killpms, that behaves the same as the command rm /extreme_casseroles/pineapple_mushroom_surprise.txt. Suppose further that biff sets the setuid bit on killpms, with the command chmod +s ./killpms and also makes it group-executable. A long listing of killpms might look like this:

```
-rwsr-xr--   1 biff drummers   22 2004-08-11 23:01 killpms
```

If crash runs this program, he finally can succeed in his quest to delete the Pineapple-Mushroom Surprise recipe: killpms runs as though biff had executed it. When killpms attempts to delete pineapple_mushroom_surprise.txt, it succeeds because the file has user-write permissions and killpms is acting as its user/owner, biff.

IMPORTANT WARNING

setuid and setgid are very dangerous if set on any file owned by root or any other privileged account or group. I'm illustrating setuid and setgid so you understand what they do, not because I think you actually should use them for anything important. The command sudo, described later in this article, is a much better tool for delegating root's authority.

If you want a program to run setuid, that program must be group-executable or other-executable for what I hope are obvious reasons. In addition, the Linux kernel ignores the setuid and setgid bits on shell scripts. These bits work only on binary (compiled) executables.

setgid works the same way but with group permissions. If you set the setgid bit on an executable file with the command chmod g+s *filename*, and if the file also is other-executable (-r-xr-sr-x), when that program is executed it runs with the group ID of the file rather than of the user who executed it.

In the above example, if we change killpms' other permissions to r-x (chmod o+x killpms) and make it setgid (chmod g+s killpms), no matter who executes killpms, killpms exercises the permissions of the drummers group, because drummers is the group owner of killpms.

setgid and Directories

What about directories? Well, setuid has no effect on directories, but setgid does, and it's a little non-intuitive. Normally, when you create a file, it's automatically owned by your user ID and your (primary) group ID. For example, if biff creates a file, the file has a user owner of biff and a group owner of drummers, assuming that drummers is biff's primary group, as listed in /etc/passwd.

Setting a directory's setgid bit, however, causes any file created in that directory to inherit the directory's group owner. This is useful if users on your system tend to belong to secondary groups and routinely create files that need to be shared with other members of those groups. For example, if the user animal is listed in /etc/group as being a secondary member of drummers but is listed in /etc/passwd has having a primary group of muppets, then animal has no trouble creating files in the extreme_casseroles/ directory, whose permissions are set to drwxrwx--T. However, by default, animal's files belong to the group muppets, not to drummers, so unless animal manually reassigns his files' group ownership (chgrp drummers *newfile*) or resets their other permissions (chmod o+rw *newfile*), other members of drummers cannot read or write animal's recipes.

If, on the other hand, biff or root sets the setgid bit on extreme_casseroles/ (chmod g+s extreme_casseroles), when animal creates a new file therein, the file has a group owner of drummers, exactly like extreme_casseroles/ itself. All other permissions still apply; if the directory in question isn't group-writable to begin with, the setgid bit has no effect, because group members are not able to create files inside it.

Now we've covered all possible permissions: read, write, execute, sticky bit, setuid and setgid. If you understand all six of these, you're probably in the minority of Linux users. But wait, there's more!

Numeric Modes

So far we've been using mnemonics to represent permissions—r for read, w for write and so on. Needless to say, as with everything else, your system actually uses numbers to represent permissions. The chmod command recognizes both mnemonic permission modifiers (u+rwx,go-w) and numeric modes.

A numeric mode consists of four digits: as you read left to right, these represent special permissions, user permissions, group permissions and other permissions. Recall that other is short for other users not covered by user permissions or group permissions. For example, 0700 translates to no special permissions set, all user permissions set, no group permissions set and no other permissions set.

Each permission has a numeric value, and the permissions in each digit place are additive: the digit represents the sum of all permission bits you want to set. If, for example, user permissions are set to 7, this represents 4 (the value for read) plus 2 (the value for write) plus 1 (the value for execute).

As I just mentioned, the basic numeric values are 4 for read, 2 for write and 1 for execute. (I remember these by mentally repeating the phrase, read-write-execute, 4-2-1.) Why no 3, you might wonder? Because this way, no two combination of permissions have the same sum.

Special permissions are as follows: 4 stands for setuid, 2 stands for setgid and 1 stands for sticky bit. For example, the numeric mode 3000 translates to setgid set, sticky bit set and no other permissions set, which is, actually, a useless set of permissions.

Here's one more example of a numeric mode. If I issue the command chmod 0644 mycoolfile, I am setting the permissions of mycoolfile, as shown in Figure 1.

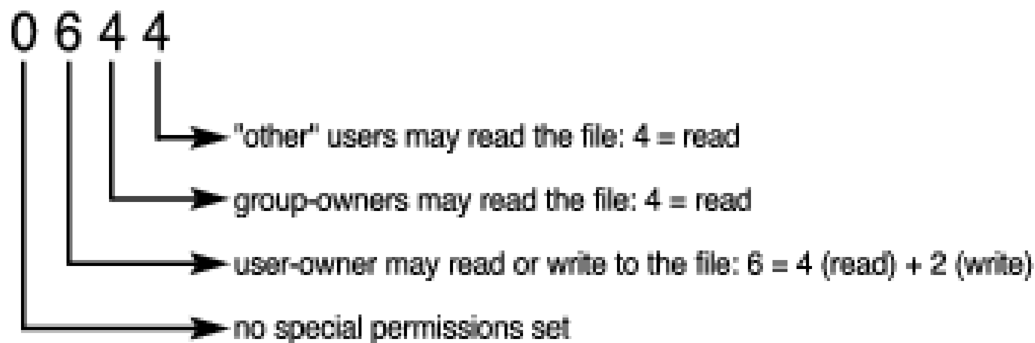

Figure 1. Permissions for mycoolfile

For a more complete discussion of numeric modes, see the info page for coreutils, node Numeric Modes. That is, enter the command info coreutils numeric.

umask

I want to cover one last command specific to permissions before closing with a couple of other topics. umask is a command built into the bash shell that prints or sets your default permissions mask. To see yours, simply enter the umask command without any arguments; it returns a four-digit number. On my system, it looks like Listing 3.

Listing 3. Checking My Default Permissions Mask

```
mick@localhost:/home/mick> umask
0022
```

Mode 0022 means no special permissions, no user-owner permissions, group and other permissions set to write, right? How can that be?

Actually, umask deals in masks, not in modes per se. 0022 is what is subtracted from the number 0777 to determine the numeric mode of files you create: 0777 – 0022 = 0755.

Aha! So, files I create have user-owner permissions set to read-write-execute (7 = 4 + 2 + 1) and group and other permissions set to read-execute (5 = 4 + 1)? Right? Almost. It also happens that umask sets the execute bit automatically only on directories. Even if your permissions mask includes execute permissions, the execute bit does not set automatically on regular files you create. So, if my permissions mask is 0022, resulting in default permissions of 0755, and I create a file named default_file and a directory named default_dir, long listing output for those two items look like Listing 4.

Listing 4. File and Directory with Mask of 0022

```
-rwxr-xr-x   2 mick users      48 2004-08-13 08:31 default_dir
-rw-r--r--   1 mick users       4 2004-08-13 08:31 default_file
```

To change your default permissions mask, simply issue the command umask with the new mask as its argument. For example, if I want all my files to have group-read permissions but no other permissions, this translates to a numeric mode of 0740. If I subtract that from 0777 I get a mask of 0037. Therefore, the umask command I enter is umask 0037. This new mask, however, applies only to my current session and any new shells I start from it. To make it persistent, I can add the line umask 0037 to my .bashrc file.

su and sudo

I should say a few words about the reality of users, groups and permissions. The whole problem with UNIX security is that far too often, permissions and authority on a given system boil down to root can do anything, although users can't do much of anything.

Sadly, it's much easier to do a quick su - to become root for a while than it is to create a granular system of group memberships and permissions that allows administrators and sub-administrators to have exactly the permissions they need. Sure, you can use the su command with the -c option, which allows you to specify a single command to run as root rather than an entire shell session (for example, su -c rm *somefile.txt*), but this requires you to enter the root password. It's never good for more than a small number of people to know root's password.

Another approach to solving the root-takes-all problem is to use role-based access control (RBAC) systems, such as SELinux, which enforce access controls that reduce root's effective authority. However, this makes things even more complicated than setting up effective groups and group permissions. This is not to say that SELinux and the rest aren't good things—I love RBAC.

A better middle ground is to use the sudo command. sudo is short for superuser do, and it allows users to execute single commands as root, without actually needing to know the root password. sudo is now a standard package on most Linux distributions.

sudo is configured with the file /etc/sudoers, but you shouldn't edit this file directly. Rather, use the visudo command, which opens a editor on the file; vi is the editor by default. You can use a different editor by setting the EDITOR environment variable. For example, to use /usr/bin/gedit, do this:
export EDITOR=/usr/bin/gedit

Space doesn't permit me to explain sudoers' syntax in detail; see the sudoers(5), sudo(8) and visudo(8) man pages for complete information. In the space available here, let's run through a quick example.

Remember the user crash's quest to rid the world of Pineapple-Mushroom Surprise? Although in this case it would be overkill—the permissions techniques I've already illustrated are sufficient—you could use sudo to allow crash to realize his goal, assuming you (biff) have root privileges. First, become root (su -). Next, execute the command visudo. You're now in a vi session, editing the file /etc/sudoers; see the vi(1) man page if you're new to vi. Go down to the bottom of the file and add this line:

```
crash localhost=/bin/rm
/home/biff/extreme_casseroles/pineapple_mushroom_surprise.txt
```

Save and exit the file.

Now, to do his thing, crash enters the command:

```
sudo rm /home/biff/extreme_casseroles/pineapple_mushroom_surprise.txt
```

whereupon he is prompted to enter his password. After he enters this correctly, the command:

```
/bin/rm /home/biff/extreme_casseroles/pineapple_mushroom_surprise.txt
```

is executed as root, and the offending file is gone. Alternately, the line in /etc/sudoers could look like this:

```
crash localhost=/bin/rm /home/biff/extreme_casseroles/*
```

That way, crash can delete anything in extreme_casseroles/, regardless of the sticky bit setting.
As handy as it is, sudo is a powerful tool, so use it wisely; root privileges never should be trifled with. It really is better to use user and group permissions judiciously than to hand out root privileges, even with sudo. Better still, use an RBAC-based system such as SELinux if the stakes are high enough.
That's it for now. I hope you've found this tutorial useful. Until next time, be safe!