by Peter Collinson, **Hillside Systems**

BEK SHAKIROV

Files: Permissions & Ownership

One of the key aspects of a multiuser operating system such as UNIX is the notion of file ownership. Files belong to users and the system operates a protection scheme that enforces a set of access rules on each file. Users log into the system to identify themselves and prove their identity using a password. Each of the processes started by a user inherits the knowledge of who owns the process, and the file protection system matches the ownership information in the process with that stored for the files in the file system. The system uses a set of rules to determine whether the particular user running the process has appropriate rights to access the file.

File ownership is used to prevent someone else inspecting or using your files without your consent. It's also used to stop you from doing bad things accidentally, perhaps deleting or overwriting files in system areas on the file system. Since nearly everything on a UNIX system is represented by files, the key to system security on UNIX is the file protection mechanism.

I've talked hazily about ownership. How is it represented? When you log into the system, you present it with a login name and a password. The `login` program looks up your name in the system's password file, and assuming that the passwords match, accesses a pair of numbers also stored in the password file. The numbers are the user id (UID) and the group id (GID). As we've seen, the UID and GID pair are also stored in the file system as part of the file's information. The file system has a small block of information that's used to tell it where the file resides on the disk. This information block is called the *inode*. The inode is used to store other information about the file, including its ownership. The file protection mechanism matches the UID and GID pair you obtain by logging in against the pair stored with each file to determine your access rights to the file.

Incidentally, the password that you supply when you log in isn't decrypted and matched against a string stored in plain text in the password file. The password file is readable by everyone to enable programs to access it and find out the names of users on the system. For security, the password is stored in encrypted form in the password file. When you log in, your newly entered password is encrypted and a string comparison is done using the two encrypted strings. This is why a UNIX system administrator can never tell you your password. They can only reset it.

As CPUs have become much faster, having a visible encrypted password has proved to be a weak link, and many people have written exhaustive password cracking programs, matching passwords in the file against lists of possible words. Morris' Internet Worm caused the first massive Internet security incident by employing password cracking techniques using a small dictionary to determine passwords. It met with some success. For a short time, the password file on your

system became the target for would-be system hackers. Some of the early Web attacks were aimed at getting password files from remote systems. In response to the problem that encrypted passwords were visible to users, many systems have moved them from the password file into a special file (called `/etc/shadow` on Solaris) which normal users cannot read.

Permissions

The UID must be unique in the password file because it identifies a specific person, and the file protection mechanism can be used to permit or deny access to a file based on the UID alone. The GID is intended to allow users to be placed into groups, and different permissions can be supplied to the file for the group. This permits me to make my files readable but not writeable by you because you are in my group, while I can continue to access them for reading and writing.

Of course, I need some way of indicating my preferences for how the file should be protected. UNIX has a simple system–it retains the settings in a mode word in the inode for the file. A single bit is stored allowing a process with a matching UID to open the file for reading, and another bit to allow a process with a matching UID to open the file for writing. A further pair of bits is used to control GID access. A third set is used to control access by "others," (i.e., if the process accessing the file doesn't have a matching UID or GID). So we have three pairs of bits to control read/write access.

UNIX needs to have some way of determining that a file is executable.

We don't just read and write files on UNIX, though, we can also use them as commands. UNIX needs to have some way of determining that a file is executable. When we type a command into the system, like

```
$ ls
```

the system looks for a file called `ls` in all directories given by your search path shell variable. It will find the file `/bin/ls` and attempt to execute it. A further bit for each of the three groups in the mode word is used to indicate whether the file is executable. The idea is that the user can control who can execute their commands. The bit says "you can treat this file as a command, load it into memory and run it." We now have three bits (read, write and execute) for each type of access (owner, group and other).

You can see these bits in the output from the `ls` command with the `-l` (the letter *ell* not the digit one) option:

```
$ ls -l file
-rw-r--r-- 1 pc staff .....
```

I've omitted the remainder of the `ls` listing for printing reasons. The `rw-` here shows that the file is readable by me, writeable by me, but not executable by me. In addition, members of my group (*staff*) can read the file (`r--`), and all others can too. Incidentally, the tests are exclusive, so some file permission settings can be a little counter intuitive and less than useful:

```
$ ls -l file
-r--r--rw- 1 pc staff .....
```

allows everyone to read the file, but I cannot write it, nor can members of my group. Only other users can write to the file. This can seem strange, because shouldn't I be included as a member of my group, or in "others?" The answer is no. It is simply a consequence of the order in which the tests are done.

Here's a typical setting for a public command:

```
$ ls -l /bin/ls
-r-xr-xr-x 1 root bin .....
```

Everyone is allowed to read or execute it, but no one is allowed to write to it. Well, that's not quite true. If a process comes along with a UID of zero, then the permission tests are skipped and it's allowed to do anything, and this is the basis of the notion of the *superuser* on UNIX.

Command Files

To convert a file containing a script into a command that can be run like any other command, all you need to do is turn on its execute bit. In the early systems, the shell would look for a magic number at the start of the file to determine whether it could load the file as a command. If one was not found, it would open the file and pass it into a new invocation of `/bin/sh` for interpreting. When `csh` was invented, the designers at Berkeley needed a way of saying "this script is written in `csh` syntax" and hit on the method of making the first character of the script a "#"–which shows that the line is a comment. When any shell opened the file to see what type of file it was, it would look for the # and use `/bin/csh` as the interpreter, and otherwise pass the script to `/bin/sh`.

A little later on, it became apparent that it was restrictive that shells had to work out whether they were dealing with a script and take appropriate action. For example, you may want to call a command from a program directly without using a shell to decode a command line. It was a good idea to take the shell out of the loop and make the kernel understand that it was dealing with a script. At the same time, it was possible to supply the kernel with the name of the interpreter needed to execute the script, rather than creating adhoc rules. Modern kernels understand that if the first two bytes of a script are `#!` then the name of an interpreter follows, and the kernel will arrange to call the interpreter automatically. So for example, you'll now find that Perl commands start with:

```
#!/usr/bin/perl
```

and the files in which the commands live will have their execute bits turned on. All this magic allows us to create executable files from scripts, which behave identically to a file containing a binary image that's created by compiling a program written in C or C++.

Incidentally, scripts are read by an interpreter and need to have their "read" bits set on to permit the program to open the file. If you look on your system, nearly all executable files have their read bits turned on. Occasionally we all have legitimate reasons to open an executable file for reading. For example, the `file` command that guesses at file contents opens the file for reading:

```
$ file /bin/ls
/bin/ls: ELF 32-bit MSB executable SPARC
       Version 1, dynamically linked,
       stripped
```

I've wrapped the line for printing.

Directories

Directories are files, but we shouldn't execute them as commands. However, we do want to supply them with an additional capability. We want to be able to control access to the files that they contain, so for a directory the `x` bit becomes an *access* bit (or *aXess*). Some people call this bit in directories the *search access* bit to make its use clearer. The read bit on a directory allows a process to open the directory and read its contents. The write bit on a directory allows a process to write to the directory when creating a new file. The access bit allows the directory to be searched looking for file names.

Consider a directory that I own with these permissions:

```
drw-rw-rw-
```

I've put the "d" there because `ls` will do that for directories. What can I do? Well I can use `ls` on the directory because the `read` bit is on and `ls` will read the directory to list its contents. However, I cannot move into the directory using the `cd` command, and I can't access any files in the directory, or use the directory to move down the file system tree. Although the write bit is on, I can't create any new files in the directory because I am unable to search the directory to find the filenames that it contains, which is needed to ensure that I can create a unique filename in the directory.

The bottom line is that for a directory to function as a directory, the `x` bits need to be on. The most common settings for directories are:

```
drwxr-xr-x
```

which allows everyone to list its contents and search it for files. But only the owner can create new files there. Home directories are often created:

```
drwxr-x---
```

allowing the owner full access rights. The directory is usually created in the group of the owner, so members of the owner's group can search and read the owner's files. Everyone else is denied access. Some sites will make home directories really private by setting:

```
drwx------
```

It may seem that you never want to turn the read bits off in a directory, but it can be useful in certain circumstances. If you have an anonymous FTP set-up, and want to have one directory into which you place files for people to pick up, then you can set the mode:

```
d--x--x--x
```

When you (as superuser) place a file there (e.g., `getme`), you can tell the recipient to come into the FTP server, and obtain the file `hidden/getme`. When they arrive at the server, they cannot list the directory because they cannot read it, but they can pull the file by supplying an explicit name. There's a caveat to this example. Many current GUI-based FTP clients need to be able to list the directory in order to operate, so your recipient may need to be able to see the file in a listing before being able to obtain it. Another problem is that they can pull any other file from the directory if they are able to guess its name. So the mechanism works, but does have problems.

The interaction between the permissions on a file and those on the directory in which the file lives can have subtle effects that are not often well understood. To create or delete a file, I need to have search access and write rights to the directory where the file lives. Removing write permission from the file itself doesn't stop its deletion. If I have search and write access to one of your directories, then I can delete your files.

I can also create new files in the directory. So one way for me to hack you is to delete one of your files and replace it with one of mine. The new file will be owned by me and not by you, so you can determine what has happened and who was responsible. However, if I can install a file in your private `bin` directory that you expect to execute as a command, I may be able to get away with it. You will unwittingly execute the nasty thing I have planned and it can delete the evidence as it runs.

Changing File Permissions

The `chmod` command is used to alter or establish appropriate permissions on a file or directory. It has two basic forms. First, you can specify the necessary permissions using octal numbers. Second, you can use a symbolic form. Your mileage will probably vary with how friendly this appears to be to you. The manual page for `chmod` goes into considerable detail on how to drive the command, so I will not bother too much about it here. If you are frightened by the command line `chmod` command, you can also change file permissions using the CDE File Manager GUI. Select a file in the CDE File Manager

and choose *Properties* from the *Selected* menu.

I personally tend to use the octal form when I am setting permissions, saying something like:

```
$ chmod 664 file
```

to set

```
-rw-rw-r--
```

The symbolic form is useful when augmenting or removing certain bits from the permissions, so

```
$ chmod +x file
```

will set all the execute bits on the file and

```
$ chmod -x file
```

will remove them. Be careful when removing permissions from several files, saying

```
$ chmod -x *
```

where the current directory contains other directories which will remove the search access permissions from the directories, which may not be what was desired.

Setuid and *setgid*

Occasionally we want to write a program that delivers privileged access to a system resource normally barred from use by mortals. Take for example the `lp` command that queues print jobs. We want this to be a privileged operation because we don't want people to circumvent the queuing policies of the print spooler and we want to protect the spool directory against unauthorized access. We do this by letting the spool directory be owned by a special user and group, and prohibiting normal mortals from writing into it. We now need to allow any user to come along and run a program that places files into that spooler directory.

Two special bits in the mode word, the *setuid* and *setgid* bits, are used to provide the functionality. When a command is run with these bits set, it is not run with the UID and GID pair of the user. If the setuid bit is on, the process is run with the UID set to that of the owner of the command file. If the setgid bit is set, then the GID is set to the group of the command file. The mechanism allows a user to temporarily change their access rights for the duration of the command.

It's typical to boost the access rights for the command up to superuser status, and this is what happens for the `lp` command. I'll guess there's a moment where it needs to access your files and also create files as the `lp` user, so it needs a foot in both camps. In the past, many commands were elevated to superuser status in order to give them special access rights. It's been realized that more care needs to be taken, because

each program that's setuid to root represents a potential security hole. Actually, Sun seems to have done a good job on removing these setuid/setgid programs on Solaris. Many of the programs traditionally boosted to superuser status no longer have the bits set on.

The `ls` command tells you that the setuid or setgid bits are turned on for a file by changing the `x` in the permissions listing to an `s`, so here's the lp command on my machine:

```
-r-s--x--x 1 root lp .....
```

Note that this is also set to be unreadable by mortals, for further security.

More than One Group

UNIX skipped merrily along with the simple UID and GID file permission mechanism for some time. However, groups never really worked too well. They were fine for creating sets of users that cooperated amongst themselves, but things became very complicated when you wanted a user to have access rights in more than one group. This would happen if you started to use groups to manage shared project files. The user needs to have normal group access to their home directory and also to some shared project directory that is owned by another group.

The problem, as I observed above with the `lp` command, is there are always occasions where you want to have access to two groups, usually when moving files in the file system. On the whole, we don't permit normal users to change the UID and GID pair stored with a file, because this would circumvent the normal security system. So mortals cannot change the group ownership of a file to make it accessible to a specific group.

The BSD systems changed the semantics of groups to make them more useful. First, each user can now be "in" more than one group. You log in with a primary group as before, but the system scans the `/etc/group` file that's used for mapping group names to GID values, and adds each group to which you have rights into a list that's passed from process to process. Now you can be in several groups at once, and can access files based on your set of groups without having to take any special action.

There is now a problem. If you create a file, it will normally be owned by you and placed in your primary login group. If this file is in a project directory, then it's reasonable to assume that the file should be able to be read by members of the group. To achieve that, it needs to have its GID set to the group's GID and not be set to your primary login GID. The BSD team decided that when a file was created, it should be owned by the group that owned the directory. Now when you create a file, its GID is set depending on its context.

There is apparently a strange side effect to the new semantics. If you have a file on your own home directory, and copy it into the shared project directory using the `cp` command, then it will appear in the shared directory with the GID set to that of the directory. This is what we would expect. However, if you use `mv` to place the file into the target directory, then the GID will be unchanged. This is because `mv` doesn't actually move the file contents and open a new file, which is the action of the `cp` command. It just changes the directory contents in the source and target directories.

Groups now begin to be useful, with no specific user action needed to select which group you are in. When you create a shared file, you need to ensure that the group permissions are set, so the actual group of the file is taken from the directory. Anyone can now be placed into that group, just by editing the `/etc/group` file.

When you create a shared file, you need to ensure that the group permissions are set, so the actual group of the file is taken from the directory.

Sun decided this new functionality was useful, but didn't want to apply the change in directory semantics everywhere. They decided to use the unused *setgid* bit on a directory to imply that when a file was created in that directory, then it should be given the GID of the directory and not that of the process creating the file. This allows the system administrator to control where the new semantics should apply.

Access Control Lists

In many ways, the UID and GID mechanisms of UNIX are too restrictive. They work, but elaborate dances with groups are sometimes needed to achieve particular access policies. They have the benefit of simplicity and speed, both of which were considerations in the original UNIX systems. However, it's better to be able to control access on a per-user basis, and many systems provide more specific controls on files called Access Control Lists (ACLs). Sun has glued this ability into Solaris since 2.5.

ACLs allow fine-grained control over access to files or directories, so you can allow a single user appropriate rights to your files while not enabling access to a wider population. In addition, as long as you own the file, then you can change its access control lists. There is a pair of commands to get and set ACLs: `getfacl` and `setfacl`. You can also access them from the CDE file property mechanism. There's some help for all this in the CDE User's Guide Answerbook.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* pc@cpg.com.