1 Introduction

Since most computers running UNIX have many users accessing the same file systems, it is imperative that there are safeguards to prevent users from accidentally or maliciously overwriting or removing important system files or the files of other users. The first line of defense against such unauthorized access is a system of file permissions. All of the routines in the UNIX kernel which access files (which include devices like tape drives, CDs and diskettes) are governed by these permissions. On a well-administered UNIX system, it's possible that you could never run into any problems related to file permissions, but if you want to provide a little more security for some of your files, or understanding the workings of the UNIX system more thoroughly, it's a good idea to know something about file permissions.

In this article, I'll cover the basics of what file permissions are and how to manipulate them and touch on a few more advanced permission issues which you might encounter.

2 The Basics

Every file or directory on a UNIX system is associated with an owner(usually the creator of the file), as well as a group. The group associated with a file is the default group of it's owner. You can see what groups you are in by typing the command `groups`; the first group listed is your default group.

Thus, permissions can be controlled separately for three classes of users: the owner (usually referred to as "user" in UNIX jargon) members of the same group as is associated with the file, and everyone else ("other" in UNIX jargon). For each of these classes there are three categories of permissions, namely read, write and execute. So, for example, you could have a file which was only readable by yourself and members of the group you are in, but not others, or which was writable only by you, executable by others in your group, and restricted completely for other users.

One point about file permissions which should always be kept in mind is that the superuser (root) or anyone who has access to the root password, or to your password, can do whatever they want to any of your files. So if you have files which absolutely should not be seen or manipulated by anyone else, you should either encrypt them in some fashion, or store them somewhere other than a computer system which is accessible by others.

To display information about permissions, the `-l` and `-g` options of the unix command `ls` can be used. The `-l` option specifies the long form of a file or directory listing; it includes all the information about the three forms of permission for the three classes of users, as well as the name of the owner of the file, and the size of the file in bytes. The `-g` option shows the group association of the file. Here's an example, listing a few files from the `/usr/bin` directory, using the `-lg` option:

```
-rwxr-xr-x  1 root     staff        5504 Jan 20  1994 cmp
-rwxr-xr-x  1 root     staff        8136 Jan 20  1994 col
-rwsr-xr-x  1 root     staff       24576 Jan 20  1994 crontab
```

The first column contains an indication of the type of the file. For a regular file, this column will contain a hyphen (`-`), as in the example above. For a directory, the letter `d` will appear, while symbolic links are designated with the letter `l`. (A symbolic link is a filename which really represents a file at a different location in the file system. When trying to access these files, the permissions which are used are those of the actual file, not of the link. See the manual page for the `ln` command for more information.). There are one or two other letters which may appear in the first column of the `ls -l` listing to indicate special files such as storage devices and terminal sessions.

The next nine columns display the permissions of the file. These nine columns can be thought of as three sets, each of three columns. The first set of columns applies to the owner (user) of the file, followed by columns for the group and finally those for everyone else (other). For each of the three sets of columns, the individual columns refer to read, write and execute permissions in that order. So in the example above, for the file `cmp`, the user (`root`) has read, write and execute access, users in the group `staff` have read and execute permissions, as do all other users. Usually, these permissions are represented by a `r`, `w`, or `x`, but as can be seen for the `crontab` entry, an `s` may appear. This is explained in the section on Special Permissions.

The next column following the block of permission columns shows the number of links to the file, followed by the owner, the group, and the size of the file in bytes. The next few columns display the date on which the file was last modified, finally followed by the name of the file.

3 What the Permissions Mean

The actual meaning of read, write and execute are different depending on whether they are referring to a normal file or to a directory. When you use the `ls` command on a directory, it will not normally show you information about the directory itself, but on the files found in the directory. To get specific information about the directory, you should use the `-ld` or `-ldg` flag of the `ls` command.

For a normal file, read permission allows opening the file so, for example, it can be viewed with an editor or pager, or read by an executing program. It also allows copying of the file to a different location (provided, of course, that you have permission to write files to that location). For a directory, read permission allows listing of the files in the directory, usually using the `ls` command. However, since individual file permissions override the permission of the directory, there is no guarantee that you can read all the files in a directory for which you have read access.

For a normal file, write access allows changing or deleting the file. To create a file, however, you must have write access to the directory in which you want the file to reside. Write access in a directory also allows removal of files in that directory. If a file is located in a directory for which you have write permission, but you don't have write permission for the individual file, you will be given the option of overriding the file's write protection, so it's a good idea to remove write permission in a directory if you don't want the files in that directory deleted.

Finally, execute permission for a file allows you to have the commands in the file executed by typing the name of the file. When you use a compiler to create an executable binary file, it automatically gives the file execute permission. However, if you create a file of executable commands like a shell script or a `perl` program, you'll have to set the execute permission for the file as described in the next section, or the system will respond with "`Permission denied`" when you try to execute the file. For a directory, execute permission allows you to use that directory as your current directory, as well as allowing you to open and execute files in the directory. If another user reports seeing lots of error messages when trying to access files in one of your directories, even though they can eventually access the files individually, check the execute permission of the directory in question.

4 Changing Permissions

The command which is used to control permissions is `chmod`, short for changing the "mode" of the file. First, keep in mind that you can only modify permissions on the files which you own, that is, those where the `ls -l` command shows you as the owner of the file. If you accidentally remove your own permissions to a file, don't despair; as the owner, you can always reinstate them. There are two forms of syntax for using the `chmod` command: symbolic and absolute. With symbolic syntax, you specify requests as a combination of a class of users, an action and the permissions involved; multiple requests are separated by a comma. The characters used with symbolic syntax and their meanings are displayed in the table below:

Class		Action		Permission	
value	meaning	value	meaning	value	meaning
`u`	owner	`+`	add	`r`	read
`g`	group	`-`	remove	`w`	write
`o`	other	`=`	set	`x`	execute
`a`	all				

A common example mentioned in the previous section is allowing a file containing shell commands to be made executable. The command `chmod +x myscript` would allow you to type `myscript` and have the commands in the file `myscript` executed by the shell. If you wanted to remove read and write access to a file called `secret` for other users (not in the file's group), you could use the command `chmod o-rw secret`. To give yourself read, write and execute access to a file called `myfile`, while giving everyone else only execute access, you could use the command `chmod a=x,u=rwx myfile` since the comma-separated requests are processed in they order they appear. As an alternative to symbolic syntax, you can use absolute syntax, where the permissions of a file are represented as three octal digits, one each for user, group and other. Each of the digits is composed of the sum of 4 (for read access), 2 (for write access) and 1 (for execute access). So in the above example for `myfile`, you could use the command `chmod 711 myfile`; `7=4+2+1`, representing full access for user, and `1` representing only execute access for group and other. The choice between symbolic and absolute syntax is strictly a personal one, although symbolic syntax is usually easier if all you want to do is add or delete one class of permissions, as in the `myscript` example above.

5 Default Permissions

When you create a file or directory through a UNIX command (like an editor, or the `cat` or `mkdir` commands, the operating system provides default permissions of read and write permission for everyone (666 in absolute syntax) for files, or read, write and execute permission for everyone (777) for directories or executable files. These system defaults, however, are always modified by the shell using a value known as a umask. The umask value, which is generally set in the `.profile` or `.cshrc` file, is an octal number which is used to mask the system default value as described above, effectively removing those permissions which are represented by the value of umask. A commonly used value for umask is 022; this removes write permission for group and other, resulting in absolute permission 644 (`rw-r--r--`) for files and 755 (`rwxr-xr-x`) for directories and executables. (You can check your value of umask by typing `umask` with no arguments.) If you wish to provide more security for your files, you could set your umask to a more restrictive value, for example 077, which would provide no permissions for anyone but you. This would be done with the command `umask 077`, which you could either enter in your `.cshrc` or `.profile` file to have it in effect every time you log in, or type as a normal UNIX command to have it in effect for a single shell session. (Keep in mind, however, the point raised in the Introduction, namely that anyone with the root password or your password will be able to override the permissions you set.)

6 Special Permissions

In the example of the output of the `ls` command, the entry for `crontab` had an `s` in place of the `x` which would otherwise by expected for execute permission. This indicates that a the file has a special type of execute permission known as setuid permission. This means that when this file is executed, it is executed utilizing the file permissions of the owner of the file instead of the usual file permissions of the user executing the file. By using this technique, system files can be modified by ordinary users, without giving those users direct access to the files. A closely related form of permission, known as setgid, executes the file with the permissions of the group which owns the file, instead of the user who is executing the file. Setgid permissions are indicated by an `s` in place of the expected `x` for the group permission in an `ls -l` display. This description of setuid and setgid assumes that the usual execute permissions for owner or group are in effect; if not, or if the file in question is a directory, the meaning of these permissions varies from system to system; consult your system's manual page for the `chmod command` for more information.

Since the execute bit can be modified for user and group, it is not surprising that a similar modification exists for other. This permission is known as the "sticky bit", and its meaning varies from system to system. Once again the manual page for the `chmod` command should explain the particulars on your system.

In absolute mode, these permissions are controlled using a leading octal digit formed in a similar way to the three digits following it, with setuid = 4, setgid = 2 and the sticky bit = 1.