

# File Protection in UNIX

*Matt Bishop*

Research Institute for Advanced Computer Science  
NASA Ames Research Center  
Moffett Field, CA 94035

## Introduction

UNIX file protection is very simple and straightforward. However, many people are not aware of the richness of this powerful mechanism. In this article we shall discuss file protection in detail, and describe some simple tricks to help users protect the information in their files.

## Background

The UNIX file system is organized as a hierarchy of files. Think of a tree turned upside down, and replace branches with lines. Wherever lines join or end, put a node. If a node is *internal* to the tree, it corresponds to a UNIX *directory* file, which contains a list of files in that directory. Otherwise, it corresponds to some other type of file. When the operating system needs to locate a file, it starts at the beginning of the given path and walks down the tree searching each successive directory until the file is found. For example, in Figure 1, to locate the file `/usr/mab/mbox` UNIX begins at the root directory `/` and looks for the file or directory `usr`. It then searches `usr` for the file `mab`. Finally, it searches that directory for the file `mbox`. Note that the root directory `/`, `usr`, and `mab` must be directories and must be searchable; otherwise the system will fail to locate the file.

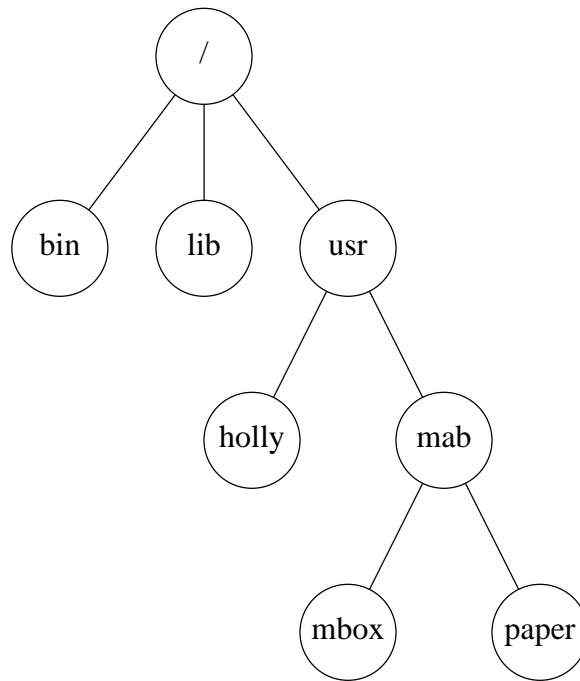


Figure 1. UNIX File System Structure.

UNIX files are *owned* in the sense that a particular user creates them, and they are counted towards that user's disk space. The owner of a file is usually the person who creates it. Every user has a unique identification number, called the *UID*, and when a file is created, the UID of the creator is associated with the file. For example, if my UID is 213, and I create a file named *xxx*, the UID of *xxx* is 213. So, the person with UID 213 (in this case, me) owns the file. Notice the way that last sentence was written. If my UID were changed one day, say to 625, the UID of file *xxx* must also be changed to 625. This can cause problems, because if the administrator changes someone's UID and forgets to change the UID of all his files, the wrong person will have access to them!

The *group* of a file is analogous to the owner of a file. UNIX users are divided into groups, and each group has an identification number called a *GID*. How the GID is assigned to the file varies from system to system—for example, 4.2 BSD UNIX assigns the file the GID of its parent directory, System V UNIX assigns the GID of its owner—but the important fact is that a file can have exactly one GID.

With these preliminaries fresh in our minds, we can now examine the way files are protected.

### Permission Bits for Non-Directory Files

There are twelve permission bits in the protection word of a file; they are customarily divided into groups of three. Let us look at the lower three sets of three bits.

The first set of three bits encodes permissions for the file's owner, the second set, permissions for the members of the file's group, and the third set, permissions for everybody else. (See Figure 2.) When someone tries to access the file, UNIX first checks to see

if that person is the  $\hat{file}$  owner and, if not, if that person is a member of the  $\hat{file}$  group. For example, suppose a  $\hat{file}$  has permissions set so the owner can only read it, and anyone in the  $\hat{file}$  group can only write it. Even if the owner is a member of the  $\hat{file}$  group, he will be unable to write on the  $\hat{file}$ . Figure 3 illustrates this procedure.

permission bits								
miscellaneous			owner bits			group bits		
su	sg	t	r	w	x	r	w	x

*Legend*

su	setuid bit	t	sticky bit	w	write bit
sg	setgid bit	r	read bit	x	execute bit

Figure 2. Permission Bits.

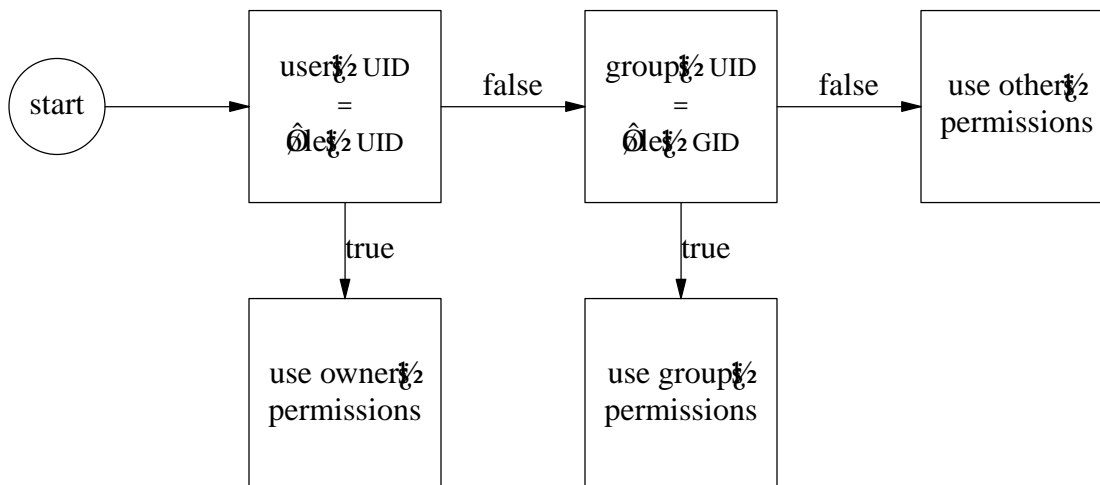


Figure 3. How User Permissions Are Determined for a File.

UNIX associates three rights with each  $\hat{file}$ : *read* (the ability to display the contents of the  $\hat{file}$ ), *write* (the ability to change the contents of the  $\hat{file}$ ), and *execute* (the ability to run the program contained in the  $\hat{file}$ .) The first bit in each triplet of permission bits is set if read permission is granted, the second bit if write permission is granted, and the third bit if execute permission is granted.

Here are some examples. Suppose the lower nine bits of the permission word are 110110100. Split this into sets of three: 110 110 100. So, the owner of the  $\hat{file}$  can read or write the  $\hat{file}$ , the members of the group of the  $\hat{file}$  can read or write the  $\hat{file}$ , and anyone else can only read the  $\hat{file}$ . As another example, suppose the  $\hat{file}$ 's permission bits were 111101101; following the above procedure, the owner can read, write, and execute

the file, and anyone else can read and execute the file (but not write it.)

As you can gather from the above paragraph, writing permission bits in binary is painful and hard to read. Since permission bits are grouped into sets of three, and octal digits correspond to three binary digits each, permission bits are written as a string of octal digits. So, 110110100 would normally be written as 640 and 111101101 as 755.

We have not discussed the highest three bits of the permission bits. The first two are the *setuid* and *setgid* bits. Normally, when a program is executed, the UID and GID of that process are those of the person executing the program. However, if the setuid bit is set, the UID of the process will be that of the owner of the file. Similarly, if the setgid bit is set, the GID of the process will be that of the owner of the file.

For example, suppose a file owned by *mab* has the setuid bit set. User *holly* executes that file. The resulting process will have *mab*'s UID rather than that of *holly*. Had the setuid bit not been set, the resulting process would have had *holly*'s UID.

The third bit in this group is called the *sticky bit*, and on many systems can only be set by the superuser. It simply prevents the program text from leaving main memory once it has been placed there, and is used to keep heavily-used programs in main memory. This speeds the program's startup time noticeably on many systems.

## Permissions for a Directory File

Although the permission bits are called the same for both directory and non-directory files, the interpretation of the three rights is somewhat different for directory files. *Reading* a directory means being able to find out what files are in that directory. *Writing* a directory means being able to create new files, or delete existing ones, in that directory. *Executing* a directory means being able to search that directory to see if a named file is contained in that directory.

Let's look at an example. Suppose the directory */usr* contains three subdirectories called *bin*, *mab* and *src*. (See Figure 4.) If you can read */usr* saying

```
ls usr
```

will print

```
bin
mab
src
```

If you do not have execute permission on */usr* and you say

```
ls usr/bin
```

you will get the error message */usr/bin not found*. (In fact, if you don't have search permission on */usr* and you type

```
ls usr
```

you will get

```
usr/bin not found
usr/mab not found
usr/src not found
```

because `ls` cannot obtain any information about the three subdirectories other than their names!)

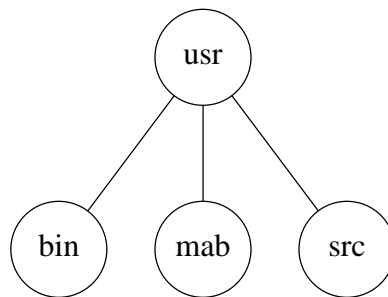


Figure 4. Sample File System.

### The `chmod(1)` Command

How do you change the permission mode of a file? The command `chmod(1)` exists for this purpose. Its general format is

```
chmod permission_mode filenames ...
```

The *permission\_mode* can be in one of two forms. The first one is the permission mode you want to assign; it *must* be in octal. For example,

```
chmod 4755 rogue
```

makes the file `rogue` readable and executable by everyone on the system, writable only by the owner, and setuid to the owner of that file. As another example,

```
chmod 644 rogue.help
```

makes `rogue.help` readable by everyone on the system and writable only by the owner.

The second form exists only on some systems; look in the manual to see if it will work on yours. The form is

```
chmod who op permission filename ...
```

*Who* is one of `u` (for `user` permission), `g` (for `group` permission), `o` (for `others` permission), or `a` (for `all users` permission). *Op* indicates how rights are to be assigned; `+` means the permissions are to be added to those already there, `-` means the permissions are to be taken away from those already there, and `=` means the permissions are to replace those already there. *Permission* is the code letter for the desired permission; they are `r` (for `read`), `w` (for `write`), `x` (for `execute`), `s` (for `setuid` if used with `g` it means `setgid`, and if used with `u` it means `setuid`), and `t` (for `sticky`).

Let's suppose, in the example with `rogue`, the file originally was mode 664 (that is, readable by all users but writable only by the owner and members of the group.) To

change it to mode 4755, you could use any of the previous commands,

```
chmod a+x,u+s,g-w rogue.help
```

(which changes the permission bits as indicated; it leaves all other bits alone), or

```
chmod u=srwx,g=rx,o=rx rogue.hints
```

(which sets all permission bits to the indicated state.) If the mode of *rogue.help* were 700, to make it mode 644 you can use the command

```
chmod u-x,g+r,o+r rogue.help
```

## The *umask*

When you create a file from a C program, you have to specify the permission mode of the file. For example, you might say

```
creat("xxx", 0666)
```

to make the file readable and writable by all users. However, before the file protection mode is set, the given permission is changed, by bitwise *and'ing* it with the bitwise negation of some variable called the *umask*. If, for example, your *umask* were 022, the bitwise negation of that is 0755, so the file *xxx* would be created with permission mode 644 (that is, you can read or write the file, but everyone else can only read it.) Some convenient values of the *umask* are:

077	only the owner has any permissions
022	only the owner can write the file
002	only the owner and group members can write the file

The *umask* can be set by using the command

```
umask nnn
```

(which sets it to *nnn*) from the shell level, or by using the system call *umask*.

Be aware that on many systems the default value of the *umask* is 0, which means that the mode you specify when creating a file is the mode of a file. Unfortunately, some programs create files in modes 666 or 777, assuming the user's *umask* will cancel any unwanted permissions. So be sure you set your *umask* to a sane value!

## Conclusion

In this article we have covered the most important aspects of the built-in controls UNIX provides for file protection. These mechanisms are simple yet powerful. They provide a consistent framework for controlling who can access a file, as well as the ability to prevent dangerous errors in protection when creating files. With these mechanisms, you can improve the protection of files containing data that you wish to keep secret.