Grymoire Navigatic

Unix/Linux
Quotes
Bourne Shell
C Shell
File Permissions
Regular Expressions
grep
awk UPDATED
sed UPDATED
find
tar
inodes
Security
IPv6
Wireless
Hardware
spam
Deception
PostScript
Halftones
Privacy
Bill of Rights
References
Top 10 reasons to

Also see my **Unix Page** for more tutorials on Unix

Updated: Wed Feb 1 23:18:23 EST 2012

# Unix/Linux Permissions - a tutorial

G+

## Table of Contents





Original version written in 1995 and published in the Sun Observer

avoid CSH
sed Chart
PDF
awk
Reference
HTML
Magic
Search
About
Donate NEW

How to limit access to your files. How to make files read-only, executable, and so on. How to prevent others from deleting files in your directories. How to prevent others from even looking at your directories. "Special-purpose" access modes for executable files.

In this tutorial, I'll go into detail over file permissions, and discuss some of the more subtle aspects. Unfortunately I have go cover the basics, so I'll try to be brief. I use aliases below. These are tcsh aliases, not bash aliases.

## Basic File Attributes - Read, Write and Execute

There are three basic attributes for plain file permissions: read, write, and execute.

## Read Permission of a file

If you have read permission of a file, you can see the contents. That means you can use more(1), cat(1), etc.

## Write Permission of a file

If you have write permission of a file, you can change the file. This means you can add to a file, or overwrite a file. You can empty a file called "yourfile" by copying the empty (/dev/null) file on top of it

```
cat /dev/null yourfile
```

## Execute Permission of a file

If the file has execute permission, then you can ask the operating system to run the file as if it were a program. If it's a binary file/program, you can execute it like any other program. In other words, of there is a file called "xyz", and it is in your searchpath, and the file is executable, all you need to do to run the program is type

xyz

If the file is a shell script, then the execute attribute says you can treat it as if it were a program. To put it another way, you can create a file using your favorite editor, add the execute attribute to it, and it "becomes" a program. However, since a shell has to read the file, a shell script has to be readable and executable. A compiled program does not need to be readable.

## The basic permission characters, "r", "w", and "x"

**r** means read **w** means write, and **x** means e**X**ecute.

Simple, eh?

## Using chmod to change permissions

The chmod(1) command is used to change permission. The simplist way to use the chmod command is to add or subtract the permission to a file. A simple plus or minus is used to add or sumtract the permission.

You may want to prevent yourself from changing an important file. Remove the write permission of the file "myfile" with the command

```
chmod -w myfile
```

If you want to make file "myscript" executable, type

```
chmod +x myscript
```

You can add or remove more than one of these attributes at a time

```
chmod -rwx file
chmod +wx file
```

You can also use the "=" to set the permission to an exact combination This command removes the write

and execute permisison, while adding the read permission:

```
chmod =r myfile
```

Note that you can change permissons of files you own. That is, you can remove all permissions of a file, and then add them back again. You can make a file "read only" to protect it. However, making a file read only does not prevent you from deleting the file. That's because the file is in a directory, and directories also have read, write and execute permission. And the rules are different. Read on.

## Basic Directory Attributes - Read, Write and Search

Directories use these same permissions, but they have a different meaning. Yes, very different meanings. This is classic Unix: terse, flexible and very subtle. Okay - let's cover the basic directory permissions.

## Read permission on a directory

If a directory has read permission, you can see what files are in the directory. That is, you can do an "ls" command and see the files inside the directory. However, read permission of a directory does **not** mean you can read the **contents** of files in the directory.

## Write permission on a directory

Write permission means you can add a new file to the directory. It also means you can **rename** or **move** files in the directory.

## Execute permission on a directory

Execute allows you to **use** the directory name when accessing files inside that directory. The "x"

permission means the directory is "searchable" when searching for executables. If it's a program, you can execute the program.

## Let's examine all this a bit closer.

This makes sense when you realise that directories are files as well. The file is not stored "inside" a directory. Files are stored in data blocks scattered alll over the disk partition. The directory is a special file that contains access information about all of the files references "inside" the directory.

Suppose you have read access to a directory, but you do not have read access to the files in the directory. You can still read the directory, or more corectly, the status information for that file, as returned by the *stat*() system call. That is, you can see the file's name, permissions, size, access times, owner and group, and number of links because you have access to the directory. The file itself is stored elsewhere on the disk (this spot is identified by the i-node number.) The "ls -i" command shows the i-node number for a file.

Write permission in a directory allows you to change the contents of a directory. Because the name of the file is stored in the directory, and not in the file, **write permission in a directory allows renaming or deletion of files** and does not require write permission of the file. To be specific, if someone has write permission to your home directory, they can **rename or delete** your **.login** file and put a new file in its place. The permissions of your **.login** file do not matter. Someone can rename a file even if they can't read the contents of a file.

Execute permission on a directory is sometimes called search permission. If you found a directory that gave you execute permission, but not read permission, you could use any file in that directory. However, you **must** know the name. You cannot look inside the directory to find out the names of the files. Think of this type of directory as a black box. You can throw filenames at this directory, and sometimes you find a file, sometimes you don't.

# User, Group and World

so far I have treated permissions as either your permission or not your permission. The read, write and execute permissions are stored in three different places, called user (u), group (g) or world or other (o).

When you execute

```
chmod =r myfile
```

it changes the permissions in three places. When you list this file with "ls -l" you will see

```
-r--r--r--    1 grymoire  admin         0 Feb  1 19:30 myfile
```

Note that there are three "r"'s for the three different types of permissions.

All files have an owner and group associated with them. There are three sets of read/write/execute permissions: one set for the user of the file, one set for the group of the file, and one set for everyone else (other). These permissions are determined by 9 bits in the *i-node* information, and are represented by the characters "rwxrwxrwx." The first three characters specify the user, the middle three the group, and the last three the world. If the permission is not true, a dash is used to indicate lack of privilege. If you wanted to have a data file that you could read or write, but don't want any one else to see, the permission would be "rw-------."

Everyone belongs to at least one group in a Unix system. Some people belong to more than one group. If the computer is only used by one person, then groups aren't that useful except for set group-id programs, but that comes later.

Let's assume you have several people using a computer, and you want to allow people in a group to have access to a directory. Let's also say they belong to the same group as you. Assume the file is in a group directory, with the group "admin", and you wanted to allow them to read and You can create a directory that has read, write and execute permission for the group. But you want to prevent people

outside of the group from reading or changing the file. You want the file to have the permission "rw-rw----" for user and group=read and write, and others have none. The chmod command can do this. You should remember that the command

```
chmod =rw myfile
```

will create the permission "rw-rw-rw-" which means user, group and other have read and write. So how can you change it to "rw-rw----"?

The chmod command has options, of course. using "=", "-" or "=" changes user (u), group (g)and other (o) permissons. You can explicitly specify u, g or o in the chmod command:

```
chmod u=rw myfile
chmod g=rw myfile
chmod ug=rw myfile
```

This is handy, but the three commands above do not change the "other" permission. They only change what is specified. To remove read and write permission for other, you can instead type

```
chmod o= myfile
chmod o-rw myfile
```

The first sets the permission to nothing, and the second removes the read and write permission.

If you want to change the group permission, use "g" instead of "o":

```
chmod g+r myfile
chmod g-w myfile
```

These comands will add read and remove write permission. You can combine these two commands

```
chmod g+r-w myfile
```

if you want to combine an operaiton on group, and other, you can put a comma between the permissions:

```
chmod g+r-w,o=rwx myfile
```

Besides "u", "g" or "o", you can use "a" to mean all three. The following commands do the same thing

```
chmod a=rw myfile
chmod =rw myfile
```

An easier way to specify these 9 bits is with 3 octal digits instead of 9 characters. The octal representative of the read, write and execute bits, "rwx" are

```
Read 4
Write 2
Execute 1
```

Octal representation is pure geek talk, and was the only form that worked in the early versions of Unix. The order is the same as the "rwx", so read/write permission, or "rw-" can be described the the octal number **6**. However, we have to express the permission of all three parts, so the permission "rw-------" (read/write for the user, and group and world get nothing) is **b00**. The first number specifies the file owner's permission. The second number specifies the group permissions. The last number specifies permissions to everyone who is not the owner or not in the group of the file.

Let's review the different combinations. I will show the letter representaiton, the octal representaiton, and the meaning

```
+-----+---+-------------------------+
| rwx | 7 | Read, write and execute  |
| rw- | 6 | Read, write              |
| r-x | 5 | Read, and execute        |
| r-- | 4 | Read,                    |
| -wx | 3 | Write and execute        |
| -w- | 2 | Write                    |
| --x | 1 | Execute                  |
| --- | 0 | no permissions           |
+-----------------------------------+
```

You can use the octal notation, where the three digits correspond to the user, then group, then other.

Perhaps this might help

```
+------------+------+-------+
| Permission | Octal| Field |
+------------+------+-------+
| rwx------  | 700  | User  |
| ---rwx---  | 070  | Group |
| ------rwx  | 007  | Other |
+------------+------+-------+
```

let's put this all together. I will list some chmod commands in both character and octal representaion.

```
+------------------------+-----------+
| chmod u=rwx,g=rwx,o=rx | chmod 775 | For world executables files
| chmod u=rwx,g=rx,o=    | chmod 750 | For executables by group only
| chmod u=rw,g=r,o=r     | chmod 644 | For world readable files
| chmod u=rw,g=r,o=      | chmod 640 | For group readable files
| chmod u=rw,go=         | chmod 600 | For private readable files
| chmod u=rwx,go=        | chmod 700 | For private executables
+------------------------+-----------+
```

Let's also review the same permissions for directories

```
+------------------------+-----------+
| chmod u=rwx,g=rwx,o=rx | chmod 775 | For world readable directories
|                        |           |   Members of group can change fil
| chmod u=rwx,g=rx,o=    | chmod 750 | For group readable directories
|                        |           |   Members of group can change fil
| chmod u=rwx,go=        | chmod 700 | For private direcories
+------------------------+-----------+
```

## The importance of order in Unix Permissions

This last point is subtle. When testing for permissions, the system looks at the groups in order. When Unix checks permissions, the order is this

> If the file is owned by the user, the user permissions determine the acces
> If the group of the file is the same as the user's group, the group permisson determien the access.
> If the user is not the file owner, and is not in the group, then the other permission is used.

If you are denied permission, Unix does not examine the next group. Consider the case of a file that is owned by user **jo**, is in the group **guests**, and has the permissions **-----xrwx** or **017** in octal. It would be listed as

```
------xrwx    1 jo  guests         0 Feb  1 20:47 myfile
```

Let's assume the directory has the permission 775 (world readable and searchable). When considering this file, the exact permissions above mean:

> **jo** cannot use the file.
> Anyone in group **guests** can execute the program.

Everyone else besides **jo** and the members of the **guests** group can read write, and execute the program.

This is not a very common set of permissions. But there are ways it can be used. However, to really understand it, we have to consider the permission of the directory. Remember, as I said, if jo of any one in the guest group has write permission of the directory, then they can rename or delete the file.

Let's say the directory /testme is owned by the superuser, and has the permission 711. What does this mean? First of all, notice that the directory does not have group and world read or write - just search.

This means that users cannot see what files are in this directory. It also means they have to know the name of the file to execute it.

Any user can type

```
/testme/myfile
```

and because the directory is world searchable, the program can be executed.

Now let's return to Jo and the Guests (Hmm. sounds like a 60's pop rock group). If the file "/testme/myfile" has the permission 017 then Jo cannot execute the program. Anyone in group Guests can, but only if the file is a compiled program (and not a shell script). ANd the rest of the world can execute the program. However, they have to know the name of the file. They cannot list the contents of the directory.

people use a similar mechanism to deny one group of users from accessing or using a file. In the above case, **jo** cannot read or write the file she owns. She could use the **chmod** command to grant herself permission to read the file. However, if the file was in a directory owned by someone else (root), and the directory did not give Jo read or search permission, she would not be able to find the file to change its permission.

## Another example - using chmod 510 on a directory to provide group access

Let's change the situation around a bit. Let's make the directory mode 510. Let's also make the file "myfile" and the directory "/testme" owned by Jo. Let's also assume the program "myfile" is a compiled program, and has the permission 711.

Anyone in group "guests" can execute the program. However, if the administrator removed someone from the group, they can no longer execute the program. They do need to log off and log on again, and group permission is granted at logon time.

## Typical Permissions

Most of the time permissions fall into three cases:

The information is personal. Many people have a directory or two they store information they do not wish to be public. Mail should probably be confidential, and all of your mailbox files should be in a directory with permission of 700, denying everyone but yourself and the system administrator read access to your letters.
The information is not personal, yet no one should be able to modify the information. Most of my directories are set up this way, with the permission of 755.
The files are managed by a team of people. This means group write permission, or directories with the mode 775.

You could just create a directory with the proper permission, and put the files inside the directory, hoping the permissions of the directory will "protect" the files in the directory.

## Using Permissions in Work Groups

This is not adequate. Suppose you had a directory with permission 755 and a file with permission 666 inside the directory. Anyone could change the

contents of this file because the world has search access on the directory and write access to the file.

## The umask command

What is needed is a mechanism to prevent any new file from having world write access. This mechanism exists with the **umask** command. If you consider a new directory would get permissions of 777, and new files get permissions of 666, the **umask** command specifies permissions to **take away** from all new files. To "subtract" world write permission from a file, 666 must have 002 "subtracted" from the default value to get 664. To subtract group and world write, 666 must have 022 removed to leave 644 as the permission of the file. These two values of **umask** as so common it is useful to has some tcsh aliases defined:

```
alias open umask 002
alias shut umask 022
```

With these two values of **umask**, new directories will have permissions of 775 or 755. Most people have a **umask** value of one of these two values.

In a friendly work group, people tend to use the **umask** of 002, which allows others in your group make changes to your files. Someone who uses the mask of 022 will cause grief to others working on a project. Trying to compile a program is frustrating when someone else owns files that you must delete but can't. You can rename files if this is the case, or ask the system administrator for help.

Members of a team who normally use a default umask of 022 should find a means to change the mask value when working on the project. (Or else risk flames from your fellow workers!) Besides the **open** alias above, some people have an alias that changes directories and sets the mask to group write permission:

```
alias proj "cd /usr/projects/proj;umask 002"
```

This isn't perfect, because people forget to use aliases. You could have a private shell file in each project directory called **.dir** that contains the line

```
umask 002
```

If you had the following alias

```
alias cd 'chdir !*; if ( -f .dir && -o .dir ) source .dir '
```

You would automatically set your mask value when to change to the project directory. Other people could have similar files in the project directory with a different name. Still another method is to run **find** three times a day and search for files owned by you in the project directory that have the wrong permission:

```
find /usr/projects -user $USER ! -perm -020 -print | \
xargs chmod g+w
```

You can use the command **crontab -e** to define when to run this command.

## Which group is which?

Since group write permission is so important in a team project, you might be wondering how the group of a new file determined? The answer depends on several factors. Before I cover these, you should note that Berkeley and AT&T based systems would use different mechanisms to determine the default group. These two variations were merged by Sun, and Linux has inherited the Sun approach.

Originally Unix required you to specify a new group with the **newgrp** command. If there was a password for this group in the **/etc/group** file, and you were not listed as one of the members of the group, you had to type the password to change your group.

Berkeley based versions of Unix would use the current directory to determine the group of the new file. That is, if the current directory has **cad** as the group of the directory, any file created in that directory would be in the same group. To change the default group, just change to a different directory.

Both mechanisms had their good points and bad points. The Berkeley based mechanism made it convenient to change groups automatically. However, there is a fixed limit of groups one could belong to,

which was 8 groups. SunOS 4 changed this to a limit of 16 groups.

Sun supports both mechanisms for backwards compability. The entire disk can be mounted with either the AT&T or Berkeley mechanism. If it is necessary to control this on a directory by directory basis, a special bit in the file permissions are used. If a disk partition is mounted without the Berkeley group mechanism, then a directory with this special bit will make new files have the same group as the directory, Without the special bit, the group of all new files depends on the current group of the user.

## The other three bits

Besides the nine bits that specify read, write, and execute (or search) permissions for the owner, group and world, there are three other bits that have special characteristics. The most famous bit is the **set uid** or **set user identification** bit. Any person who executes a program with this bit set has their user identification changed to be the same as the owner of the file. With this simple ability, Unix allows users to gain special priviledges in a controlled, temporary fashion.

Another similar bit is the **set gid** or **set group identification** bit, which changes the group instead of the user. This is the permission bit that can be applied to a directory to force it to follow the Berkeley group semantics.

The last bit is called the **sticky** bit. It is used to reduce the start up time when executing a program. Earlier versions of Unix would keep a "sticky" program in the swapping area of the disk. The second time a sticky program is executed, the system doesn't have to search for the file in the file system. However, with diskless workstations, and kernels that cache recent files in memory, this isn't as much a benefit as before. In fact, SunOS used it to indicate special files used for diskless clients.

When a directory is made sticky, it adds a special security feature. It prevents someone from deleting or renaming files in a directory unless they own the

file. This is called the "append-only" mode for a directory. The **/tmp** directory is a good example where this is needed. The directory must be world writable to be useful to others. By adding the sticky bit to this directory, you prevent someone from replacing a file owned my another user.

## Changing special permissions

The octal value of the **set uid** bit is **4000**. the **set gid** bit is **2000**, and the sticky bit is **1000**. These strange octal values aren't shown when you list them. Instead, the character representation is used. When using the chmod command

```
chmod u+s myfile
```

adds the setuid bit. The set group iD bit can be set using

```
chmod g+s myfile
```

To make a program **set uid** using the octal representaiton,the command would be

```
chmod 4755 program
```

Alternately, you could use the symbolic form:

```
chmod u+s,g=rx,o=rx program
```

To make a **set gid** program, use one of the following:

```
chmod 2755 program
chmod g+s program
```

If the file is a directory, you **must** use the symbolic form. Sticky files and directories can be created using one of these two forms:

```
chmod 1755 file
chmod u+t file
```

## Examining the permissions

Besides using **find** to search for these permissions bits, **ls** displays the permissions when the **-l** flag is

used. If a program is **set uid**, the "x" in the user area is displayed as a "s" A sticky file or directory is indicated with the last "x" displayed as a "t." If the corresponding execute bit is not set, the letter is capitalized. The capitalization of the letter is a flag that an unusal combination was chosen.

## Conditional Changing

Linux supports the +X option.

For example, this mode:

```
chmod      a+X *
```

gives all users permission to execute files (or search directories) if anyone could before.

## Conclusion

Here is a chart of the permissions displayed by **ls** and the corresponding octal values:

```
+------------------+
|rwxrwxrwx     777 | all permissions granted
|rwxr-xr-x     755 | Group and world readbale/eecutable
|rwx------     700 | Private
|rwsr-xr-x    4755 | set UID
|rwxr-sr-x    2755 | set GID
|rwxr-xr-t    1755 | Sticky bit
|rwSw-xr-x    4655 | setUID but not executable by user
|rwxr-Sr-x    2745 | getGID, but not executable by group members
|rwxr-xr-T    1754 | Sticky bit, but not world executable
+------------------+
```

*This document was translated by troff2html v0.21 on September 22, 2001.*