File and Directory Permissions Explained

Looking back at the long listings provided by the `ls` command you see that the first bit of information displayed is the permissions for the given file or directory.

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
```

The first character in the permissions string reveals the type. For example, `-` is a regular file, `d` is a directory, and `l` is a symbolic link. Those are the most common types you will encounter. For a full listing read the `ls` man page.

Symbol	Type
`-`	Regular file
`d`	Directory
`l`	Symbolic link

You will also notice other characters in the permissions string. They represent the three main types of permissions which are read, write, and execute. Each one is represented by a single letter, also known as a symbol. Read is represented by `r`, write by `w`, and execute by `x`.

Symbol	Permission
`r`	Read
`w`	Write
`x`	Execute

Read, write, and execute are rather self explanatory. If you have read permissions you can see the contents of the file. If you have write permissions you can modify the file. If you have execute permissions you can run the file as a program. However, when these permissions are applied to directories they have a slightly different meaning than when they are applied to files.

Permission	File Meaning	Directory Meaning
Read	Allows a file to be read.	Allows file names in the directory to be read.
Write	Allows a file to be modified.	Allows entries to be modified within the directory.
Execute	Allows the execution of a file.	Allows access to contents and metadata for entries in the directory.

There are three categories of users that these permissions can be applied to. These categories or classes are user, group, and other. Like the permission types, each set is represented by a single letter. The user who owns the file is represented by `u`, the users that are in the file's group are represented by `g`, and the other users who do not own the file or are not in the file's group are represented by `o`. The character `a` represents all, meaning user, group, and other. Even though these characters do not show up in an `ls` listing, they can be used to change permissions.

Symbol	Category
`u`	User
`g`	Group

`o`	Other
`a`	All - user, group, and other.

Every user is a member of at least one group called their primary group. However, users can and often are members of many groups. Groups are used to organize users into logical sets. For example, if members of the sales team need access to some of the same files and directories they can be placed into the `sales` group.

Run the `groups` command to see what groups you are a member of. If you supply another users ID as an argument to the `groups` command you will see the list of groups to which that user belongs. You can also run `id -Gn [user]`to get the same result.

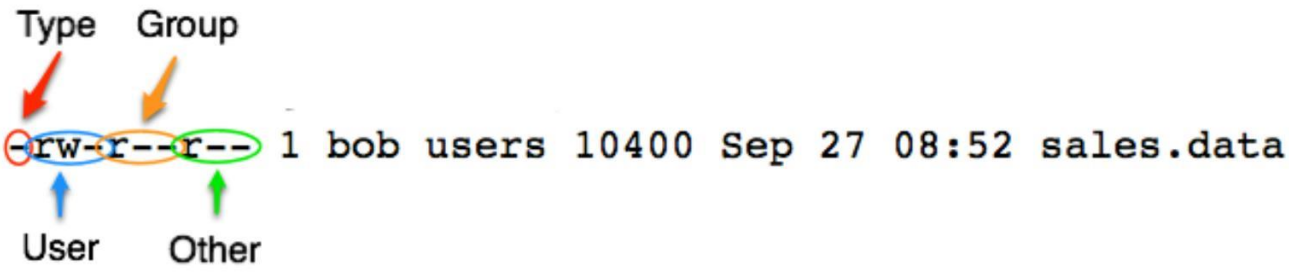
```
$ groups
users sales
$ id -Gn
users sales
$ groups pat
users projectx apache
$ groups jill
users sales manager
```

Secret Decoder Ring for Permissions

Now you have enough background information to start decoding the permissions string. The first character is the type. The next three characters represent the permissions available to the user, also known as the owner of the file. The next three characters represent the permissions available to the members of the file's group. The final three characters represent the permissions available to all others.

In this case order has meaning. Permission types will be displayed for user, followed by group, and finally for others. Also, the permission types of read, write, and execute are displayed in that order. If a particular permission is not granted a hyphen (`-`) will take its place.

Here is a graphical representation of the permission information displayed by `ls -l`.



If you happen to see an extra character at the end of the permissions string an alternative access control method has been applied. If you see a period (`.`), the file or directory has an SELinux (Security Enhanced Linux) security context applied to it. If you see a plus sign (`+`), ACLs (Access Control Lists) are in use. SELinux and ACLs are beyond the scope of this book. However, you will be pleased to know that the use of either of these is rare. If you are having troubles with permissions and notice an extra character in the permissions string, know that further investigation may be necessary.

```
$ ls -l sales.data.selinux
-rw-r--r--. 1 bob users 10400 Sep 27 08:52 sales.data.selinux
$ ls -l sales.data.acl
-rw-r--r--+ 1 bob users 10400 Sep 27 08:52 sales.data.acl
```

Changing Permissions

Permissions are also known as modes. That is why the command you use to change permissions is called `chmod`, short for "change mode." The format of the `chmod` command is `chmod mode file`. There are two ways to specify the mode. The first way is called symbolic mode. The symbolic mode format is `chmod user_category operator permission`. Here is a table view of the `chmod` command symbolic mode format.

Item	Meaning
`chmod`	The change mode command
`ugoa`	The user category. One or more of `u` for user, `g` for group, `o` for other, `a` for all.
`+-=`	One of `+`, `-`, or `=`. Use `+` to add permissions, `-` to subtract them, or `=` to explicitly set them.
`rwx`	The permissions. One or more of `r` for read, `w` write, and `x` execute.

You can add, subtract, or set permissions using user category and permission pairs. For example, if you want to add the write permission for the members of a file's group, you would specify `chmod g+w file`.

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod g+w sales.data
$ ls -l sales.data
-rw-rw-r-- 1 bob users 10400 Sep 27 08:52 sales.data
```

Notice that after running `chmod g+w sales.data`the permissions string changed from '-rw-r--r--' to '-rw-rw-r--'. Remember that the permissions are displayed in the order of user, group, and other. The group permission set now includes the `w` symbol indicating that the write permission has been granted. Now the owner of the file (`bob`) and members of the group (`users`) can read and write to the `sales.data` file. Here is the reverse. This is how you would subtract the write permission.

```
$ ls -l sales.data
-rw-rw-r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod g-w sales.data
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
```

You can change more than one permission at a time. This time write and execute permissions are added for the file's group.

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod g+wx sales.data
$ ls -l sales.data
-rw-rwxr-- 1 bob users 10400 Sep 27 08:52 sales.data
```

You can even set permissions on different user categories simultaneously. Here is how to change permissions for the user and group. Notice that before running this command that the user already has the write permissions. Using `+` to add permissions does not negate any existing permissions, it just adds to them.

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod ug+wx sales.data
```

```
$ ls -l sales.data
-rwxrwxr-- 1 bob users 10400 Sep 27 08:52 sales.data
```

If you want to set different permissions for different user categories, you can separate the specifications with a comma. You can mix and match to produce the outcome you desire. Here is how you can specify `rwx`for user while adding `x` for group.

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod u=rwx,g+x sales.data
$ ls -l sales.data
-rwxr-xr-- 1 bob users 10400 Sep 27 08:52 sales.data
```

If you want to set the file to be just readable by everyone, run `chmod a=r file`. When you use the equal sign (`=`) the permission are set to exactly what you specify. If you specify just read, then only read will be available regardless of any existing permissions.

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod a=r sales.data
$ ls -l sales.data
-r--r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
```

If you do not specify permissions following the equal sign, the permissions are removed. Here is an illustration of this behaviour

```
$ ls -l sales.data
-rw-r--r-- 1 bob users 10400 Sep 27 08:52 sales.data
$ chmod u=rwx,g=rx,o= sales.data
$ ls -l sales.data
-rwxr-x--- 1 bob users 10400 Sep 27 08:52 sales.data
```

Numeric Based Permissions

In addition to symbolic mode, octal mode can be used with `chmod` to set file and directory permissions. Understanding the concepts behind symbolic mode will help you learn octal mode. However, once you learn octal mode you may find that it is even quicker and easier to use than symbolic mode. Since there are only a few common and practical permission modes they can be readily memorized and recalled.

In octal mode permissions are based in binary. Each permission type is treated as a bit that is either set to off (0) or on (1). In permissions, order has meaning. Permissions are always in read, write, and execute order. If `r`, `w`, and `x` are all set to off, the binary representation is `000`. If they are all set to on, the binary representation is `111`. To represent read and execute permissions while omitting write permissions, the binary number is `101`.

r	w	x	
0	0	0	Binary value for off
1	1	1	Binary value for on

r	w	x	
0	0	0	Base 10 (decimal) value for off
4	2	1	Base 10 (decimal) value for on

To get a number that can be used with `chmod`, convert the binary representation into base 10 (decimal). The shortcut here is to remember that read equals 4, write equals 2, and execute

equals 1. The permissions number is determined by adding up the values for each permission type. There are eight possible values from zero to seven, hence the name octal mode. This table demonstrates all eight of the possible permutations.

Octal	Binary	String	Description
0	000	`---`	No permissions
1	001	`--x`	Execute only
2	010	`-w-`	Write only
3	011	`-wx`	Write and execute (2 + 1)
4	100	`r--`	Read only
5	101	`r-x`	Read and execute (4 + 1)
6	110	`rw-`	Read and write (4 + 2)
7	111	`rwx`	Read, write, and execute (4 + 2 + 1)

Again, in permissions order has meaning. The user categories are always in user, group, and other order. Once you determine the octal value for each category you specify them in that order. For example, to get `-rwxr-xr--`, run `chmod 754 file`. That means the user (owner) of the file has read, write, and execute permission; the members of the file's group have read and execute permission; and others have read permissions.

	U	G	O
Symbolic	rwx	r-x	r--
Binary	111	101	100
Decimal	7	5	4

Commonly Used Permissions

Here are the most commonly used permissions. These five permissions will let you do just about anything you need to permissions wise.

Symbolic	Octal	Use Case / Meaning
`-rwx------`	`700`	Ensures a file can only be read, edited, and executed by the owner. No others on the system have access.
`-rwxr-xr-x`	`755`	Allows everyone on the system to execute the file but only the owner can edit it.
`-rw-rw-r--`	`664`	Allows a group of people to modify the file and let others read it.
`-rw-rw----`	`660`	Allows a group of people to modify the file and not let others read it.
`-rw-r--r--`	`644`	Allows everyone on the system to read the file but only the owner can edit it.

When you encounter `777` or `666` permissions, ask yourself "Is there a better way to do this?" "Does everybody on the system need write access to this?" For example, if a script or program is set to `777`, then anyone on the system can make changes to that script or program. Since the execute bit is set for everyone, that program can then be executed by anyone on system. If malicious code was inserted either on purpose or on accident it could cause unnecessary trouble. If multiple people need write access to a file consider using groups and limiting the access of others. It is good practice to avoid using `777` and `666` permission modes.

Working with Groups

If you work on the sales team and each member needs to update the `sales.report` file, you would set the group to `sales` using the `chgrp` command and then set the permissions to `664` (`rw-rw-r--`). You could even use `660` (`rw-rw---`) permissions if you want to make sure only members of the sales team can read the report. Technically `774` (`rwxrwxr--`) or `770` (`rwxrwx---`) permissions work also, but since `sales.report` is not an executable program it makes more sense to use `664` (`rw-rw-r--`) or `660` (`rw-rw----`).

When you create a file its group is set to your primary group. This behaviour can be overridden by using the `newgrp` command, but just keep in mind when you create a file it typically inherits your default group. In the following example Bob's primary group is `users`. Note that the format of the `chgrp` command is `chgrp GROUP FILE`.

```
$ nano sales.report
$ ls -l sales.report
-rw-r--r-- 1 bob users 6 Dec  4 20:41 sales.report
$ chgrp sales sales.report
$ ls -l sales.report
-rw-r--r-- 1 bob sales 6 Dec  4 20:41 sales.report
$ chmod 664 sales.report
$ ls -l sales.report
-rw-rw-r-- 1 bob sales 6 Dec  4 20:41 sales.report
```

Instead of keeping files in the home directories of various team members, it is easier to keep them in a location dedicated to the team. For example, you could ask the system administrator of the server to create a /usr/local/sales directory. The group should be set to `sales` and the permissions should be set to `775` (`rwxrwxr-x`) or `770` (`rwxrwx---`). Use `770` (`rwxrwx---`) if no one outside the sales team needs access to any files, directories, or programs located in /usr/local/sales.

```
$ ls -ld /usr/local/sales
drwxrwxr-x 2 root sales 4096 Dec  4 20:53 /usr/local/sales
$ mv sales.report /usr/local/sales/
$ ls -l /usr/local/sales
total 4
-rw-rw-r-- 1 bob sales 6 Dec  4 20:41 sales.report
```

Directory Permissions Revisited

This example demonstrates how permissions effect directories and their contents. A common problem is having proper permissions set on a file within a directory only to have the incorrect permissions on the directory itself. Not having the correct permissions on a directory can prevent the execution of the file, for example. If you are sure a file's permissions have been set correctly, look at the parent directory. Work your way towards the root of the directory tree by running `ls -ld .` in the current directory, moving up to the parent directory with `cd ..`, and repeating those two steps until you find the problem.

```
$ ls -dl directory/
drwxr-xr-x 2 bob users 4096 Sep 29 22:02 directory/
$ ls -l directory/
total 0
-rwxr--r-- 1 bob users    0 Sep 29 22:02 testprog
$ chmod 400 directory
$ ls -dl directory/
dr-------- 2 bob users 4096 Sep 29 22:02 directory/
$ ls -l directory/
ls: cannot access directory/testprog: Permission denied
total 0
```

```
-????????? ? ? ? ?            ? testprog
$ directory/testprog
-su: directory/testprog: Permission denied
$ chmod 500 directory/
$ ls -dl directory/
dr-x------ 2 bob users 4096 Sep 29 22:02 directory/
$ ls -l directory/
total 0
-rwxr--r-- 1 bob users 0    Sep 29 22:02 testprog
$ directory/testprog
This program ran successfully.
$
```

Default Permissions and the File Creation Mask

The file creation mask is what determines the permissions a file will be assigned upon its creation. The mask restricts or masks permissions, thus determining the ultimate permission a file or directory will be given. If no mask were present directories would be created with `777` (`rwxrwxrwx`) permissions and files would be created with `666` (`rw-rw-rw-`) permissions. The mask can and is typically set by the system administrator, but it can be overridden on a per account basis by including a `umask` statement in your personal initialization files.

`umask [-S] [mode]` - Sets the file creation mask to mode if specified. If mode is omitted, the current mode will be displayed. Using the `-S` argument allows umask to display or set the mode with symbolic notation.

The mode supplied to `umask` works in the opposite way as the mode given to `chmod`. When you supply 7 to `chmod`, that is interpreted to mean all permissions on or `rwx`. When you supply 7 to `umask`, that is interpreted to mean all permissions off or `---`. Think of `chmod` as turning on, adding, or giving permissions. Think of `umask` as turning off, subtracting, or taking away permissions.

A quick way to estimate what a umask mode will do to the default permissions is to subtract the octal umask mode from `777` in the case of directories and `666` in the case of files. Here is an example of a `umask 022` which is typically the default umask used by Linux distributions or set by system administrators.

```
                     Dir    File
Base Permission       777     666
Minus Umask          -022    -022
                     ----    ----
Creation Permission   755     644
```

Using a umask of `002` is ideal for working with members of your group. You will see that when files or directories are created the permissions allow members of the group to manipulate those files and directories.

```
                     Dir    File
Base Permission       777     666
Minus Umask          -002    -002
                     ----    ----
Creation Permission   775     664
```

Here is another possible umask to use for working with members of your group. Use `007` so that no permissions are granted to users outside of the group.

```
                     Dir    File
Base Permission       777     666
```

```
Minus Umask          -007    -007
                     ----    ----
Creation Permission   770     660 *
```

Again, using this octal subtraction method is a good estimation. You can see that the method breaks down with the umask mode of `007`. In reality, to get an accurate result each time you need to convert the octal permissions into binary values. From there you use a bitwise NOT operation on the umask mode and then perform a bitwise AND operation against that and the base permissions.

It is fine to gloss over the subtleties here since there are only a few practical umask modes to use. They are `022`, `002`, `077`, and `007`. Save yourself the binary math homework and look at the following table containing all the resulting permissions created by each one of the eight mask permutations.

Octal	Binary	Dir Perms	File Perms
0	000	rwx	rw-
1	001	rw-	rw-
2	010	r-x	r--
3	011	r--	r--
4	100	-wx	-w-
5	101	-w-	-w-
6	110	--x	---
7	111	---	---

Special Modes

Look at this output of umask when the mask is set to `022`.

```
$ umask
0022
```

You will notice an extra leading `0`. So far you have only been dealing with three characters that represent permissions for user, group, and other. There is a class of special modes. These modes are setuid, setgid, and sticky. Know that these special modes are declared by prepending a character to the octal mode that you normally use with `umask` or `chmod`. The important point here is to know that `umask 0022` is the same as `umask 022`. Also, `chmod 644` is the same as `chmod 0644`.

Even though special modes will not be covered in this book, here they are for your reference. There are links at the end of this chapter so you can learn more about these modes if you are so inclined.

`setuid permission` - Allows a process to run as the owner of the file, not the user executing it.

`setgid permission` - Allows a process to run with the group of the file, not of the group of the user executing it.

`sticky bit` - Prevents a user from deleting another user's files even if they would normally have permission to do so.

`umask` Examples

Here are two examples of the effects umask modes have on file and directory creation.

```
$ umask
0022
$ umask -S
u=rwx,g=rx,o=rx
$ mkdir a-dir
$ touch a-file
$ ls -l
total 4
drwxr-xr-x 2 bob users 4096 Dec  5 00:03 a-dir
-rw-r--r-- 1 bob users    0 Dec  5 00:03 a-file
$ rmdir a-dir
$ rm a-file
$ umask 007
$ umask
0007
$ umask -S
u=rwx,g=rwx,o=
$ mkdir a-dir
$ touch a-file
$ ls -l
total 4
drwxrwx--- 2 bob users 4096 Dec  5 00:04 a-dir
-rw-rw---- 1 bob users    0 Dec  5 00:04 a-file
$
```