Unix File and Directory Permissions and Modes

Overview

Unix and Linux systems (including Mac OS X and other POSIX compliant systems) have a (relatively) simple system for controlling access to files and directories. This system is defined in POSIX.1:2008 standard, also known as the *Single Unix Specification (SUS version 4*). And since devices such as disks, ports, etc. have file names (under `/dev`) you control access to them the same way. (Note Windows systems up through Windows ME don't support permissions, just a “read only” attribute.)

This systems works by assigning a user and a group to every file. Then users of that file are put into one of three classes:

- the *owner* (the process' UID matches the *user* of the file),
- the *group* (not the owner, but the process' GID is a member of the file's group),
- and *other* (everyone else).

For each class of users there are three possible permissions that can be granted:

- read,
- write, and
- execute.

Thus there are nine permissions you can set, in any combination. (Not all combinations make sense however.)

Any attempt to access a file's data requires *read* permission. Any attempt to modify a file's data requires *write* permission. Any attempt to execute a file (a program or a script) requires *execute* permission.

In *nix systems directories are also files and thus use the same permission system as for regular files. Note permissions assigned to a directory **are not inherited** by the files within that directory.

Because directories are not used in the same way as regular files, the permissions work slightly (but only slightly) differently. An attempt to list the files in a directory requires *read* permission for the directory, but not on the files within. An attempt to add a file to a directory, delete a file from a directory, or to rename a file, all require *write* permission for the directory, but (perhaps surprisingly) not for the files within. *Execute* permission doesn't apply to directories (a directory can't also be a program). But that permission bit is reused for directories for other purposes.

Execute permission is needed on a directory to be able to `cd` into it (that is, to make some directory your current working directory).

Execute is needed on a directory to access the "inode" information of the files within. You need this to search a directory to read the inodes of the files within. For this reason the *execute* permission on a directory is often called *search* permission instead.

> *You can think of **read** and **execute** on directories this way: directories are data files that hold two pieces of information for each file within, the file's name and it's inode number. **Read** permission is needed to access the names of files in a directory. **Execute** (a.k.a. **search**) permission is needed to access the inodes of files in a directory, if you already know the file's name.*

Search permission is required in many common situations. Consider the command "`cat /home/user/foo`". This command clearly requires *read* permission for the file `foo`. But unless you have *search* permission on `/`, `/home`, and `/home/user` directories, `cat` can't locate the inode of `foo` and thus can't read it! You need *search* permission on every ancestor directory to access the inode of any file (or directory), and you can't read a file unless you can get to its inode.

Various other commands will need to access the inodes of files to work. Earlier I said you need *write* permission on a directory to add, rename, or delete files within. But all those actions also require changing or at least reading the inodes of the affected files, so *search* permission is also needed.

> *Permissions don't determine what commands can access files, they determine what "system calls" can access files. The required permissions for system calls are documented in their man pages (section 2). So to know what permissions are needed to run the command X on a file Y, you need to know (or guess) what system calls X will try to make.*

In addition to these standard permissions there are three standard "attributes" that can be set on any file (these are commonly also referred to as permissions):

- The set user ID (or *SUID*),
- the set group ID (or *SGID*), and
- the *text* (or *sticky*) attributes.

Finally, there are non-standard attributes and additional permissions (access control lists or ACLs) that may or may not be available on some systems.

The Details

Each file or directory contains 12 settable permission (or *mode*) bits, which means there are 2**12 = 4096 possible permission settings! The 12 bits are either *on* (set to 1) or *off* (set to zero). Each can be changed independently.

All permission and attribute information about a file is kept in the file's *inode*. Only the `owner` of a file (or the root user) can modify information in the *inode* such as the permission bits and group. Note the owner needs no permissions

set to change permissions; it is enough to be the owner. Also, only the super-user `root` can change the owner of a file on most Unix and Linux systems.

Unix doesn't support the idea of inherited permissions. So, unlike other systems, setting read permission on a directory for some user does not give that user read permission on the files within that directory.

Note that permissions do not grant users the right to run certain *programs*, rather they grant the right to use certain *system calls* (of the Unix API). A command such as `cat` or `more` is written using the `read()` system call, and only files and directories that have the "`r`" permission for a user permit the use of this system call. This is why a user can't use `more`, `vi`, etc., on any file on which they don't have "`r`" permission. Similarly, a user must have "`w`" permission to `write()` a file or directory, which is the system call used to modify files and directories (which are files too). The "`x`" permission permits a user to `exec()` a file, which means to execute it as a program. (In Unix, a program or application is just a file that has execute ("x") permission.)

In short any program makes one or more system calls to access files and directories. A user process must have been granted the appropriate permissions (one or more of "`r`" for read, "`w`" for write, or "`x`" for execute) or the access will fail. Note that other system calls (such as `stat()`) will also fail if the right permissions aren't granted.

To see all the system calls a given program uses, you can use the `strace` (or a similar) command. (This may produce a lot of output!) Once you know which system call is used, you can check the *man pages* for that system call to see what permissions are needed to use it. (See note.)

Classes of Users

The basic permissions of "`r`", "`w`", and "`x`", are applied to three different categories or *classes* of users. Note that every file and directory in Unix is identified with an *owner* and a *group*. The categories/classes are `owner` (occasionally referred to as the file's *user* or *user owner*), `group` (or *group owner*), and `others`. (See note.)

In addition to these nine mode bits ("`r`", "`w`", and "`x`", for each of three categories of `owner`, `group`, and `others`), there are three others: the set User ID (`SUID` or `setuid`), the set Group ID (`SGID` or `setgid`), and the `sticky` (or `text`) bit. The effect of these three bits depends on what other modes are set, and differs for files and directories.

If the person (*) attempting to read, write, or execute a file is the same as the owner, the first set of permissions is used and the remaining six bits (three for group and three for others) are ignored. But if the person is not the same as the owner, the system will check the group of the file against all the groups the person is a member of. If there is a match then the second set of permissions are used. If the person is not the owner and not a member of the group for the file, then the third set of permissions is used to determine what access the person is allowed.

To see the permissions for files and directories use the `ls -l` *`filename`*

command. (On a directory, use the `ls -ld` *`directoryname`* command since otherwise `ls` gives information on the files within that directory and not on the directory itself.) To illustrate, suppose the permissions for a file named `foo` are listed as follows:

```
-rw-r-----    1 Hymie   staff         78 Aug 14 13:08 foo
```

The first dash indicates an ordinary file. On a directory you would see a "`d`" instead. The next nine characters tell what permissions have been granted. The first three ("`rw-`") indicate what permissions have been granted to the owner ("`Hymie`") of the file. In this case the owner has been granted read and write permission, but not execute permission. The next three show what permissions have been granted to members of the file's group. Here ("`r--`") they show that group "`Staff`" members have read access only. The last three characters show that the others (i.e., not the owner "`Hymie`" and not members of group "`Staff`") have no permissions granted.

Note the `SUID`, `SGID`, and `sticky` bits have no columns of their own in an `ls` output, but if turned on they show up as special characters (that is, not "`x`" or "`-`") in the execute columns for the owner, group, and others. The `SUID` bit displays as an "`S`" in the owner's execute column of the output. If the execute bit is also set then an "`s`" is used. The `SGID` bit appears similarly in the group's execute column. The `sticky` bit appears in the others' execute column, as a "`T`" or as a "`t`" if the *other* execute bit is also set.

An example:

```
-rwsr-S--t    1 Hymie   staff         78 Aug 14 13:08 foo
```

Here, the owner (`Hymie`) is granted read, write, and execute permission, the group members (`staff`) are granted read permission, and others are granted execute permission. In addition, the `SUID`, `SGID`, and `sticky` bits are all set.

Files

Suppose some user attempts to read a file with some Unix command such as `cat`. The system call `read()` is used and the "`r`" permission is required. The system checks to see if the user is in fact the `owner` of the file. If so, the access is permitted if the owner has been granted "`r`" permission. If not, the system check to see if the user is a member of the `group` of the file. If so, access is permitted if the group has been granted "`r`" permission.

If the user is neither the `owner` nor a member of the file's `group` then the access is permitted if `others` has been granted "`r`" permission.

The same logic holds for attempts to modify the file (write) or to run it (execute).

Changing the name of a file or deleting it completely are **not** tasks that require the `write()` system call. So a user doesn't need read ("`r`") or write ("`w`") permission to rename or delete a file. You don't even have to be the owner of a file to delete it (*)! However, when using `mv` to move a file to another directory on another disk, (e.g., "`mv foo /floppy`") the system must copy the file to the other disk and therefore does need read permission.

Special Considerations on Files:

Execute permission and scripts

If a user has execute ("x") permission on some file but not read ("r") permission, he or she can execute the file. In other words, the file is an application program. However if users don't also have read permission they cannot copy the file, since the `cp` command requires read ("r") permission to work.

On the other hand a shell script file with execute permission only will not run! This is because any script file (including Perl scripts) cannot be executed directly by the system with an `exec()` system call. Instead the proper script interpreter (usually shell) is actually executed. This interpreter in turn attempts to *read* the script file. It is possible to *run* a script without execute permission by entering "`sh` *`script`*" (or "`perl` *`script`*".)

The proper permissions on a script are both read and execute. Setting the execute bit on causes the kernel to start up the shell (*) which reads the script. This is one reason why scripts are less secure than compiled programs; scripts must be readable and executable but compiled programs need only be executable.

The three attribute mode bits can also affect access to files. Their effects depend on the other permissions set.

SUID on a file

If any class of user is granted execute permission, then this bit causes the `owner` of the resulting process to be that of the file and not of the user running the program. So if the program attempts to `read()` something, the permissions that apply would be for the `owner` of the file and not the user of the program.

For example, suppose user `Jane` runs the command "`view memo.txt`", and the permissions on the `view` command and the file `memo.txt` are as follows:

```
-rwx--x--x    1 root    bin         4515 Aug 14 13:08 view
-rw-------    1 root    bin          218 Aug 14 13:08 memo.txt
```

Jane has permission to run `view`, but not permission to read `memo.txt`. So when this `view` program attempts to `read()` the file a "`permission denied`" error will occur.

Suppose we change the view program to have the `SUID` bit on:

```
-rws--x--x    1 root    bin         4515 Aug 14 13:08 view
```

Now, when Jane runs this `SUID` program, the access to `memo.txt` is permitted. When `view` attempts to `read()` the file, the system doesn't think `Jane` is attempting to read, it thinks "`root`" is the user. So the access is allowed.

A similar substitution occurs if the `SGID` bit is set and any execute bits are set. The `group` ID checked is not the current user, but the `group` of the program.

Technically, every process has a real user (RUID) and a real group (RGID). These are the user and group of the person who started the process by running some program. Every process also has an *effective* user id EUID and EGID. By default these are the same. But if you run a program that has the SUID or SGID bits on, the effective UID or effective GID become those of the file, not of the person.

> ***WARNING:*** *SUID and SGID programs can be dangerous. They are not usually needed. SUID and SGID scripts are incredibly dangerous and can easily allow evil-doers super-user access to your system!!* ***Never*** *allow a SUID or SGID writable program on your system for even a minute!*

SUID and Shell Scripts

The standard is clear that executing a shell script is treated as an execution of "sh *script*", which means the process started uses the permissions of sh (or other interpreter for other types of scripts such as Perl, Python, Ruby, etc.) This implies that scripts will run ignoring the SUID and SGID bits. That said, many systems in the past have honored these modes for scripts. The security of scripts is low compared to compiled programs and even if your system allows a script to run as SUID you shouldn't do so.

SUID and DLLs

Note that today, many executables use *dynamic link libraries*. (DLLs have the extension .so or .so.*number* on Unix and Linux, where the "so" stands for *shared object*.) Such a program controls which libraries to link to at runtime. This process uses configuration files in /etc but those system-wide defaults can be over-ridden by setting certain environment variables. This could be a very dangerous security hole for SUID or SGID programs (I write an evil library, then set the environment variables so that your SUID program runs using my evil code). So when the EUID or EGID differ from the RUID or RGID, a SUID program ignores the environment variables (e.g., "LD_LIBRARY_PATH") and only uses DLLs from the standard, preconfigured locations.

SUID and SGID on non-executable files

If the SUID bit is set on a file with no execute bits set (i.e. a data file), the SUID has no effect. However, if the SGID bit is set on a file without any execute bits set, then some sort of file and/or record *locking* may be enabled. This means that if one process has that file open, any other attempts to open it will block. In Linux and System V systems, when SGID is set on a file that does not have group execute privileges, this indicates a file that is subject to *mandatory locking* during access (if the filesystem is mounted to support mandatory locking with mount -o mand). This overload of meaning surprises many and is not universal across Unix-like systems. In fact, the Open Group's Single Unix Specification for chmod(3) permits systems to ignore requests to turn on SGID for files that aren't executable if such a setting has no meaning.

The Sticky (a.k.a. *Text*) Bit on Files

The `sticky` bit was used to keep programs (executable files) in memory, so that the next time any user runs that program it would start faster. This is obsolete on modern systems which use *virtual memory*, and no longer has any effect. On non-executable files, the bit never had any effect.

Some versions of Unix called this the *save program text* bit (or the *text* bit). Old systems that honored this bit on executable files ensured that only the root user could set this bit; otherwise users could have crashed systems by forcing *everything* into memory. Modern POSIX systems ignore this bit on regular files but allow any user (not just root) to set/clear this bit on the files they own.

Directories

Directories (and nearly everything else) in Unix are just files. They contain little information, just the name of a file and its `inode` number. System calls that read or modify directories work similarly as for ordinary files. However the permission bits on directories control access to additional system calls (such as `chdir`) then just the few used for regular files (such as `read`, `write`, and `exec`).

To read the names of files in a directory using `read()`, or using `opendir()` or `readdir()` system calls, requires read permission. Note the `ls` command needs this permission to access the names of files in a directory. Directories also contain *inode* numbers for each file, but read permission does not grant access to these.

To modify the contents of a directory requires write permission. If you have write permission on some directory, you can add files to it, rename and delete files from it.

> *Deleting, linking, and renaming files require execute permission too, as discussed below). This is because such operations also require access to a file's inode in addition to the file's name. While read permission will allow access to the name of a file in a directory, execute permission is needed to access the inodes of files in that directory.*

Note you don't have to be the `owner` of a file or have write permission on it to rename or delete it! You only need write permission on the directory that contains the file.

Execute Permission for Directories

The `chdir()` system call is one of many that requires execute permission on a directory. Of course a directory isn't really a program that you can run even if it has execute permission. The execute bit is reused rather than waste space with additional permission bits.

Besides controlling a user's ability to `cd` into some directory, the execute permission is required on a directory to use the `stat()` system call on files within that directory. The `stat()` system called is used to access the information in a file's `inode`, and must be done before you can open or delete (via the `unlink()` system call) that file. (See Note.)

Because of its role in file access the execute bit on a directory is sometimes

called *search* permission. For example, to read a file `foo/bar`, you must have read permission for the file itself, but before the file can be accessed you must first search the directory `foo` for the inode of file `bar`. This requires search ("x") permission on the directory `foo`. (Note you don't need read permission on the directory in this case! You only need read permission on a directory to list its contents.)

Special Considerations on Directories:

Execute Permission

The use of the execute permission on a directory has some non-obvious effects on file access. Note that if execute permission is required for a directory, it is usually required for each directory component on the full pathname of that directory.

Without execute permission on a directory, a user can't access files in a directory even if they own them and have all permissions on them.

With read but not execute, you can do `ls someDir` but not `ls -l someDir`. With execute but not read permission, you can `ls -l someDir/file` but not `ls someDir` or `ls -l someDir`. Thinking of the system calls involved (`read` and `stat`) may help clarify this. Also, make sure `ls` isn't aliased to something such as `ls --color` or `ls -F`, since these options change the listing to identify directories, links, and executables by using `stat`, which requires execute permission. (Try `/bin/ls` each time, or `unalias ls`.)

Remember that to use `ls -l file`, or on some systems `ls -i dir` (i.e., to use `stat()` system call), you must have execute on the directory, the directory's parent, and all ancestor directories up to and including "`/`" (the *root* directory).

With execute but not read permission on a directory, users cannot list the contents of the directory but can access files within it if they know about them.

A common situation illustrating all this is user web sites. If a user's web page is `/home/auser/public_html/index.htm`, then 'x' permission is needed for everyone on `/`, `/home`, `/home/auser`, and `/home/auser/public_html`, and the file `index.htm` needs 'r' permission for everyone ('x' is not needed for the file.)

To delete a file requires both write (to modify the directory itself) and execute (to `stat()` the file's `inode`) on a directory. Note a user needs no permissions on a file nor be the file's `owner` to delete it!

To put or create a file in a directory required both "w" and "x" permissions. Write permission is needed because you are modifying the directory with a new hard link, and execute permission is needed in order to use `stat`, `open`, and `creat` system calls. (Creating a file involves trying to `open` the file first to see if it already exists and `stat` if it does, and using either `ln` to create a new hard link or `creat` to create a new file.)

SUID and SGID for Directories

The SUID bit has no effect on directories.

In Linux and Solaris, when SGID is set on a directory files created in that directory will have their GID automatically reset to that of the directory's GID. This means that setting the SGID bit on a directory causes any new files or directories created within to *inherit* the group identity of that directory rather than that of the user. Also, new sub-directories will inherit the SGID bit as well.

The purpose of this approach is to support *project directories*: users can save files into such SGID directories and the group identity of the file automatically changes. This is useful for example on the *Document Root* of a website or other directories containing a set of files worked on by a specific group of users. (It works especially well if each user's primary group (*) is a private group for that user, and the umask (*) setting is 002). However, setting the setgid bit on directories is not specified by standards such as the *Single Unix Specification* [Open Group].

Sticky bit on Directories

The sticky bit is used on directories that are writable by group or others. As noted earlier, a user doesn't have to be the owner of a file to delete it, nor have been granted any permissions on that file. A user needs only write and execute permission on the directory to delete any file contained within it. However if the sticky bit is also set on the directory, only the owner of a file or the owner of the directory (and the super-user of course) will be able to delete that file. Public directories such as /tmp use this feature. (Not all versions of Unix support this use of the sticky bit, unfortunately.)

In Linux 3.6 and newer, the sticky bit has another effect on directories. If enabled via /proc/sys or sysctl, the kernel won't follow symlinks from a directory with the sticky bit set, to locations outside that directory. (That is, creating a link in /tmp to a file in, say, /etc won't be followed. But a link in /tmp to, say, /tmp/foo/bar would be allowed.) Additionally, hard links can only be created when the user is already the existing file's owner, or if they already have read/write access to the existing file. These changes should prevent a common trick used by attackers to escalate their privileges, a problem noted in the mid-1990s but never addressed. (Reference: git.kernel.org commit log.)

Other Permission and Attribute Information:

ACLs

Some Unixes (notably HP-UX and Solaris) support the idea of file and directory ACLs (Access control lists), which are a means of granting sets of individual users permissions. You can think of it this way: Normally a file is associated with a single user (the user owner) and a single group (the group owner). With ACLs a file can be associated with multiple users and groups, not just the owner user and group. Each of these groups and users can be granted any of the normal permissions (read, write, or execute). Note that only the real file owner can change permissions, just as before.

Some ACL implementations do support inheritance of permissions. However

these ACLs are independent of the standard methods described here, which use the 12 permission bits. (There is a POSIX standard for ACLs, but it was withdrawn and is not widely used except on Linux.)

If you set the "default" ACL on a directory, then any subsequently created files/directories will also have their ACL set to a copy of this default ACL. (This is spoken of as new files "inheriting" the default ACL of their parent directory, but this term can be confusing; subsequently changing the default ACL on the parent directory will not change the ACLs of any existing files within that directory.)

Try this (using POSIX ACLs):

```
cd
mkdir test
touch test/foo
setfacl -d -m user:nobody:r-- test
touch test/bar
getfacl -R test
```

(`-d` means to change the default ACL, `-m` means to modify)

You should see that `bar` is readable by "`nobody`", but that `foo` is not. When changing the *default* ACL of a directory, there is a recursive option for `setfacl` you can use to change the ACLs of existing files as well. (In the example above, use:

```
setfacl -R -m user:nobody:r-- test
```

ACLs can be used to solve the *per-directory `umask`* problem. In a highly secure system, `umask` is set to `077`. But creating a new file in a project workgroup directory, that is a directory holding a group's project's files, this is the wrong value since you would want new files to be accessible by group members. For directories the value is also wrong since you normally want those to have group read, write, and executable permissions. On a web site, new files need to be group accessible and also read by others; new subdirectories need execute by others too.

Keeping `umask` set to a highly secure value and setting a default ACL on a directory to add the desired extra group and other permissions per directory works well, especially if the SetGID is also set on that directory.

> *The POSIX permission model is showing its age. Even with the addition of ACLs and extended attributes, it can be difficult to assign exactly the permissions desired. the lack of permission inheritance and more finely-grained permissions has prompted newer models to be explored. These are often based on Microsoft's NTFS permission model. Solaris 10 uses this (and will approximate POSIX permissions for older utilities), as does NFSv4.*

Additional and Extended Attributes

Many types of filesystems support additional attributes on files. Some examples include the NTFS and the ext family of filesystems. Usually special utilities are

provided to view and change these, such as the ext2 filesystem's `lsattr` and `chattr`. Modern systems also support *extended attributes*. These are not pre-defined by the system but consist of *name-value* pairs. These can usually be accessed with the utilities `getfattr` and `setfattr`.

> *The output of `ls -l` indicates when a file has ACLs, additional attributes, extended attributes, or any combination of these. The eleventh character (the first character following the ten permission/mode characters) will be a space or period unless the file has ACLs or attributes set. In that case the eleventh character is a "+" (plus sign).*

Rootly Powers

Some operations don't use the permission (mode) bits to allow or deny access. In some cases permission solely depends on who is making the request. For example, only the user owner (or root) can change the permissions. Other operations require the user to be root. Examples include halting the system and starting daemons (servers) that listen on "privileged ports" (i.e., TCP and UDP port numbers below 1024). The kernel simply checks the UID of the process to see if it is "0" (root), and grants or denies access accordingly.

While common in Unix and Linux systems this scheme is flawed, in that many programs must be run as root. Thus, if an attacker finds some exploit in such a program then that user has gained complete control! This meant that a web server, print server, DNS server, etc., would all run as root. Many times in the past this has indeed led to security problems.

In some modern systems (notably Solaris and Linux), internally the rootly powers have been split up into about a dozen separate *privileges* (the term used on Solaris) or *capabilities* (the term used on Linux). This internal change is invisible to most users—root gets all these rights and regular users get none, so the system works exactly as before.

Where it gets interesting is that a program that was started by root (and thus has all rootly power) can selectively give up the rights it doesn't need. All modern server programs thus start as root, give up the rights they don't need, uses the remaining rights, and finally sets the UID to a completely non-privileged user (and dropping all rootly powers they still hold). In this way, even if an attacker finds an exploit in some server daemon there is very little privilege they can exploit.

You can view a process's privileges on Solaris using the `ppriv` command, and on Linux using the `getpcaps` command. On either system one can also use the `/proc` system to see this information. On Linux for example:

```
cat /proc/pid/status | grep Cap
```

MAC and DAC

The twelve permission bits (or mode bits) discussed above, the three special bits (SUID, SGID, text) and the three groups of user, group, and other permissions, can be changed on any file or directory at the discretion of the owner (or by root). For this reason such permissions are called *discretionary access controls*

(or “DACs”). DACs can be considered weak because if an attacker gains access to your system they can change these permissions and do whatever they want.

In modern versions of Unix and Linux an alternative can be used. A separate permission system can be enabled that loads a policy at boot time that determines who can do what. This policy cannot be modified without a reboot. (See Note.) Because the system will require a process to have these permissions to proceed with some operation, this system is called *mandatory access controls* (or “MACs”).

If MAC is enabled, both MAC and DAC systems must allow some operation. For example, if the DAC permissions allow some user to read a file but the MAC policy doesn't, or if the MAC policy does allow a user to read some file but the DAC doesn't, then access is denied.

Several MAC systems are available. For Linux, consider using SELinux or AppArmor.

Send comments and questions to
pollock@acm.org
Last updated by Wayne Pollock on 04/25/2018 00:20:22.