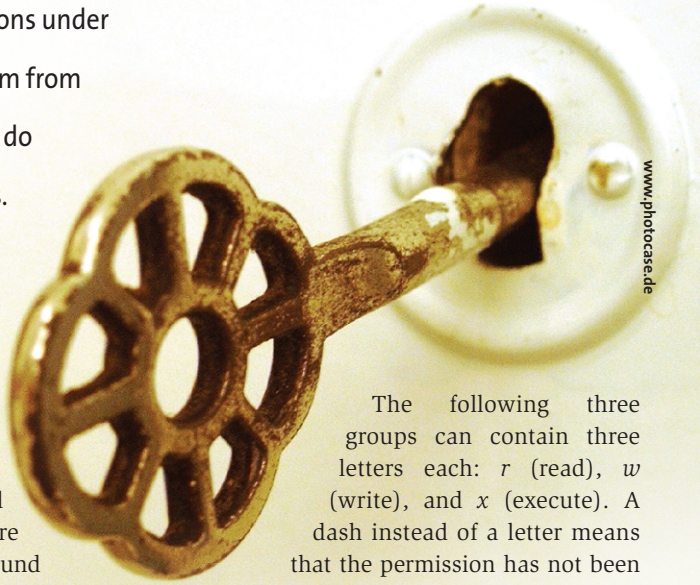
chmod, chown, chgrp and umask

Secure access rights?

Hands off other people's files. An intelligent system of permissions under Linux means that you can allow others access to, or prevent them from accessing, your files. Commands like chmod and chgrp help you do this. Additionally, you can use umask to define user permissions.

BY HEIKE JURZIK



Linux has a sophisticated system of permissions. Each file has an owner, with detailed access rights to define who is allowed to read, write, or execute files. Linux distinguishes between access privileges for the file owner, the members of a group, and all other users on the system. As usual, the administrative user, *root*, is allowed to do everything, so you should be careful with the commands we will be looking at in this month's column.

The *chmod* allows you to change file access permissions, assuming that you are the file owner or system administrator, that is. As the *root* user, you can also assign a new owner to the file using the *chown* command. If you want to assign default access permissions, then the *umask* command is the place to do this. So let's start off on our path through the "permissions jungle".

Users and Groups

There are three ways of controlling user permissions. First, there are permissions for the file owner, then for a whole group of users (you can assign each file to exactly one group), and finally for all other users on the system that do not fit into one of the other categories.

The principle of organizing multiple users into groups has a lot of advantages. Many distributions have a group called *audio*, where users are allowed to access the sound device (*/dev/dsp*) among other things. Most distros organize access to CD and DVD-ROM drives in a similar way. Thus, the members of a group have the same permissions. Users that do not belong to this group are known collectively as "others".

The *ls* command with the *-l* flag not only lists the filename, but also its type, permissions, owner, group and size (see Box 1).

Note the letters and dashes in the first column of this output representing the access privileges for the file. The third column shows the owner (user *huhn*), followed by the group (*video* or *users*). Permissions are easy to decipher: the first character denotes the file type. A dash indicates a "normal" file, *d* stands for "directory": the other possibilities being *b* (block device), *c* (character device), *l* (symbolic link), *p* (FIFO file, named pipe), or *s* (Unix domain socket).

The following three groups can contain three letters each: *r* (read), *w* (write), and *x* (execute). A dash instead of a letter means that the permission has not been assigned. The first group of three is for the file owner, the second for the group, and the third for other users. Let's take another look at the files in the first example: the file *foo.mpg* is readable and writable for the owner (*huhn*), and the members of the *video* group. Other users on the system are not permitted to read or write to the file. Nobody can execute this file, not even its owner.

The nine permissions flags have a slightly different meaning for directories. *r* still means "readable" – that is, the owner (a user, a group, or others) can enter *ls* to list the directory contents. *x* means you can change to the directory using the *cd* command, but you need both *x* and *w* to store files in the directory. If you do not have these permissions, an error message to that effect is displayed.

Box 1: "ls" output

```
01 huhn@open:~$ ls -l
02 [...]
03 -rw-rw----    1 huhn     video     3002156 2004-05-24 20:29 foo.mpg
04 -rw-------    1 huhn     users        3526 2004-05-24 20:26
   permissions.html
05 drwx------    3 huhn     users        4096 2004-05-24 19:16 titlepix/
06 [...]
```

Command Line

Although GUIs such as KDE or GNOME are useful for various tasks, if you intend to get the most out of your Linux machine, you will need to revert to the good old command line from time to time. Besides that, you will probably be confronted with various scenarios where some working knowledge will be extremely useful in finding your way through the command line jungle.

Box 2: Assigning permissions

```
01 huhn@open:~$ ls -l permissions.html
02 -rw-------  1 huhn  users  3526 2004-05-24 20:26 permissions.html
03 huhn@open:~$ chmod g+rw permissions.html
04 huhn@open:~$ ls -l permissions.html
05 -rw-rw----  1 huhn  users  3526 2004-05-24 20:26 permissions.html
```

More Permissions

However, there are more permissions than just read, write, and execute – for example, the *s* and *t* bits. The *s* (setuid) bit is a special case for executable files. If you set this bit for the owner or group, it will appear in place of the *x* in *ls* output. When you execute a file that has the *s* bit set, Linux will run the file with the permissions assigned to the file owner and/or group.

Setting the *s* bit for programs that belong to *root* can be a considerable security risk, as it allows normal users to run the file as if they were the *root* user.

The *s* bit has a different meaning for directories. Let's assume that you have a group of users who all work in the same department of an enterprise. The administrator can create a group for these users and assign them to the group. After doing so, the admin could create a directory, assign it to the new group, and set the *s* bit for the group. Any new files created in this directory would then automatically belong to this particular group.

The *t* bit is also known as the sticky bit. It also replaces the *x* flag in *ls* output, just like the *s* bit. The sticky bit is not used often for programs, where it tells the system to keep a program in memory after execution, rather than releasing the memory (this is one approach to launching programs more quickly). The *t* bit has a completely different effect on directories. You can normally delete any file in a directory where you have write privileges, even if this file belongs to another user. In directories with shared access, such as */tmp*, that store files for multiple users, it makes sense to set the *t* bit. Although users have write permissions for the directory, they can only delete their own files (and of course any files for which they have explicit write permissions). The *t* bit does affect the owner of the directory – who is allowed to delete any file in the directory.

GLOSSARY

Octal numbers: *Octal numbers, just like normal decimals, run from numbers 0 to 9, are one of three common bases used in computing. Octals use only eight numbers (0 through 7), so that 10 follows 7 in this base. Octals are particularly useful for access permission management, as permissions can use exactly the same values, 0 through 7, that is all octal numbers.*

All Change!

The *chmod* command expects the new access permissions, and one or multiple files to be changed, as parameters. *chmod* recognizes two approaches to describing permissions: the symbolic and the octal approach.

The symbolic approach to writing parameters uses the letters *u* (for "user"), *g* (for "group"), and *o* (for "others") to define whose permissions it should change. You can use combinations of these letters, simply by entering them without a space. This is followed (again without a space) either by a + (if you are adding permissions), a - (if you are removing permissions), or a = (to assign exactly the permissions defined in the command). The last parameter specifies the permissions: *r*, *w*, *x*, *s*, and *t*.

The syntax in Box 2 assigns additional read and write permissions (*rw*) to a group (*g*): *chmod g+rw ...*.

You can remove the same permissions from the group with a similar command:

```
chmod g-rw permissions.html
```

You can combine multiple steps – *chmod ug=rx,u+s file* first assigns read and execute to the owner and group, and then sets the *s* bit for the owner:

```
-r-sr-x--- 1 huhn users 7641 20➥
04-05-25 12:14 permissions.html*
```

If you want to change the permissions for the owner, the group, and all others, there is no need to type *ugo*. Instead, use *a* (for "all").

Mathematical Approach

Besides the symbolic, letter-based approach, *chmod* also has a variant that uses three or four-digit **octal numbers**. This allows you to reassign all permissions with a single command. Instead of the mnemonics *u*, *g* and *o*, the *chmod* expects numbers such as those in Box 3.

The numbers express the sum of *4* (read permission), *2* (write permission), and *1* (execute permission). In other words, you need to add the permissions that you want to assign to the owner, the group, and all others. Read and write are collectively expressed as 4 + 2 = 6. The first number applies to the owner, the second to the group, and the third to all others. Thus, *644* translates to *-rw-r-r--*, and *777* to *rwxrwxrwx*.

You can also express special flags, such as the *s* and *t* bits, as octals. To do so, simply place another number in front of the group of three: *4* represents the *s* bit for the owner, *2* the *s* bit for the group, and *1* the *t* bit. The following:

```
chmod 4755 file
```

sets the permissions to *rwsr-xr-x* – all users are allowed to read and execute the file, and the *s* bit is set for the owner.

Recursion

You can tell *chmod* to act recursively by adding the *-R* parameter. If you need to remove a group's permissions for a directory, its subdirectories and any files in them, you can simply type:

```
chmod -R g-w directory
```

The *x* flag is more tricky. The need *x* flag has to be set, for you to change to a directory. This means that a command such as *chmod -R 600 directory* will soon produce a *Permission denied* message. This is what happens under the hood:

Box 3: Chmod with numbers

```
01 huhn@open:~$ chmod 644 permissions.html
02 huhn@open:~$ ls -l permissions.html
03 -rw-r--r--  1 huhn  users  7641 2004-05-25 12:14 permissions.html
```

before the command reaches the first file in the directory, it will already have set the permissions for the directory itself to *600*. After doing so, the command is no longer able to change to the directory.

You can use a trick to remove the permissions from files only. This involves using the *find* command to create a chain of commands that firsts searches a subdirectory for files, before running the *chmod 600 …* on the files. Subdirectories are not affected.

```
find directory/ -type f -exec ➥
chmod 600 \{\} \;
```

Changing Group and Owner Assignments

The *chgrp* allows you to change the group assignment for files, the only restriction being that you need to be a member of the new group yourself. To discover your group assignments, enter:

```
huhn@open:~$ groups
users uucp dialout audio video
```

In this scenario, the user wants to assign a file to the *video* group:

```
huhn@open:~$ chgrp video foo.mpg
huhn@open:~$ ls -l foo.mpg
-rw-r--r-- 1 huhn video 3002156➥
 2004-05-25 14:03 foo.mpg
```

Of course, you are out of luck if you attempt to assign one of your own files to the *root* group:

```
huhn@open:~$ chgrp root foo.mpg
chgrp: changing group of `foo➥
.mpg': Operation not permitted
```

This command also supports recursion when the *-R* is set.

The *chown* assigns files to new users, however, use of the command is restricted to the administrative user, *root*. Enter *su* to become *root*, and then enter *chown newOwner file*.

Masked

umask allows you to assign specific permissions directly to newly created files. Typing *umask* without any parameters displays the default setting:

```
huhn@open:~$ umask
0022
```

umask outputs a four-digit octal number at this point, however, this does not specify the permissions that are assigned, but the permissions that are removed. Thus, *0022* means that *rw-r--r--* (644) permissions will be assigned to new files, and *rwxr-xr-x* (755) permissions to new directories (in other words *2*, "write" will be disabled for the group and other users). Note that new files are never executable by default; in contrast to this, the *x* bit is set for directories, unless you define a umask to change this.

Table 1: Bit magic

Bit	Permissions
0	—
1	--x
2	-w-
3 (= 2+1)	-wx
4	r--
5 (= 4+1)	r-x
6 (= 4+2)	rw-
7 (= 4+2+1)	rwx

To change the mask, enter *umask* followed by a value. The following command makes new files writable for group members, for example:

```
umask 0002
```

Note that any changes will be temporary, and restricted to the current bash session. You need to enter the *umask* command in *.profile* or *.bashrc* to make the change permanent. The entry will then be *umask 0002* – without quotes. The change becomes effective next time you launch a bash session, or parse the modified configuration file by entering *. .bashrc*.

If you prefer to avoid octal numbers, you can use a syntax similar to that of *chmod* with *umask*.

```
umask u=rwx,g=rx,o=
```

It makes all new files readable for the owner and group, and writable for the owner. The same values apply to directories, but the owner and group members are also permitted to change to the directory.