A veteran Unix and Linux enthusiast, Chris Brown has written and delivered open source training from New Delhi to San Francisco, though not on the same day.

CORE TECHNOLOGY

Prise the back off Linux and find out what really makes it tick.

Access control

Get to grips with file access controls in Linux as we take "rwx" to the max.

Check into a hotel, and the chances are you'll get a little plastic card. Take the lift to your room, slot the card in the door, and either it will let you in or you will have to retrace your steps to the check-in desk to get a new card. That's access control. Within an operating system, it can be applied at various points. Consider the common scenario of browsing to a website that's delivered by a LAMP stack. Access controls are applied at several places in the data path:

1. Packet filtering rules in the Linux kernel can block incoming traffic except to a few favoured ports, or from a favoured IP address range perhaps.
2. The *Apache* web server can apply access controls based on directives in its config file. For example, "allow from" and "deny from" rules control access based on the client machine identity, and it also supports user logins to restrict access to parts of the site to specific authenticated users. Many servers such as Samba and Postfix can impose similar rules.
3. The PHP code on a web page queries a back-end database, probably *MySQL*. Here again, there's a need to authenticate, and there's a fine-grained set of privileges defined in *MySQL*'s grant tables.
4. Underneath it all, we have the Linux filesystem. Again, access control decisions are made based on the file's ownership and permissions, and on the identity of the process attempting the access.

Indeed, there are so many points along the way at which something might say "no" that it's a miracle it ever works at all. And don't even get me started on the mandatory access control mechanisms like SELinux and App Armor.

In my list, items 2 and 3 are implemented by rules in userspace (ie within the code of the server itself), whereas items 1 and 4 rely on decision-making code in the kernel.

It's the bottom layer (item 4 – file access control) that I want to focus on here. The detail of how this works is a well-travelled road and probably familiar to you. But I'll try to turn over a few stones along the way that you haven't looked underneath before.

Burn after reading

A good place to start is by examining a couple of lines from **ls -l**:

```
# ls -ld /home/chris /etc/shadow
-rw-r-----  1 root  shadow 1284 Sep 29 13:51 /etc/shadow
drwxr-xr-x 93 chris chris  4096 Sep 29 12:08 /home/chris
```

The first field of the line includes the classic nine permission bits, the third and fourth fields show the owner and group of the file. The table 'rwx explained' shows the meaning of these three permissions as applied to files and directories. The meaning of the execute bit, in particular, is totally different for files and directories. You should also notice that you need to have both read and execute permission to have any sort of useful access to a directory, and they invariably go together; you are very unlikely to come across a directory that has one but not the other.

This is pretty basic stuff. "Sure", I hear you say. "I knew all that". But do you really understand how it works? Consider this:

```
$ ls -l test
-r--rw-rw- 1 chris chris 0 Sep 19 16:10 test
```

My question is, can I (chris) as the owner of the file, write to it? Before you read on, have a guess!

Then try this:

```
$ echo hello > test
bash: test: Permission denied
```

...which will convince you that you can't. It is tempting to think that because members

Create a shared directory

Our mission here is to create a directory that can be used as a shared read/write resource by two users: andrew and graham. We will accomplish this by making them both members of the group "editors", and the directory will be **/home/editors**.

First, create the group:

```
# groupadd editors
```

Now create the directory and set its group. Then set its **setgid** bit so that files created here will have the group "editors":

```
# mkdir /home/editors
# chgrp editors /home/editors
# chmod g+ws,o-rwx /home/editors
```

It should now look like this:

```
# ls -ld /home/editors
drwxrws--- 2 root editors 4096 Sep 29 20:38 /home/editors
```

Now make sure that andrew and graham are members of the group (we assume they already have accounts):

```
# usermod -a -G editors andrew
# usermod -a -G editors graham
```

Finally, establish a **umask** of 007 for andrew and graham by adding a line:

```
umask 007
```

in the **~/.profile** file of the two users. This will ensure that files they create are accessible by the group but not by others. Allowing the files they create to be group writable is OK because outside of **/home/editors** any files they will create will have their regular primary group (andrew or graham) and only the accounts andrew and graham will have membership of those groups. For more on **umask**, see Command Of The Month.

of the file's group (and for that matter everyone else) have permission to write to the file, then you should too. But it doesn't work that way. If I am the owner of the file, I see the first set of permissions. End of story. The tests are not applied in a hierarchical way. In practice this situation almost never arises. It is very rare (and not particularly useful) for the permissions to become more liberal as you move from left to right (owner/group/other). It's usually the other way round.

The other interesting question is "Can the owner (chris) delete the file?" Have another guess, then try this:

```
$ rm temp1
rm: remove write-protected regular empty file
'temp1'? yes
$
```

You will get a warning from **rm** about deleting a file on which you don't have write permission, but if you answer 'yes' it will do it. The point is this: whether you can delete a file does not depend on the permissions on the file itself; it depends on the permissions on the directory you're trying to delete it from (you need write permission). This is logical if you think about it – we're not actually trying to write to the file, we're just trying to write to the directory (to remove the link). But in the training classes I run I usually see a few raised eyebrows and wrinkled noses when I explain this. The fact is, there is no "you can delete me" file permission in Linux.

The warning you receive from **rm** in this case is not characteristic Linux behaviour. The usual philosophy of these commands is "if you asked to do it, and if you have permission to do it, I'll do it". For example, if you have files **foo** and **bar**, neither of these commands will raise a query:

```
$ cp foo bar
$ mv foo bar
```

even though both of them will result in the loss of the original file **bar**.

The nine "rwx" mode bits on a file are well known, but there are actually three more,

Fool your boss!

Do you have to contend with an insufferable Linux know-it-all at work? Would you like to outsmart them? Then try this:

```
$ touch temp
$ chmod 7000 temp
$ ls -l temp
---S--S--T 1 chris chris 491 Jun 30 15:49 temp
```

Ask them to do an **ls -l** on the file then explain the permissions. Of course, there's a small risk that they'll actually know, in which case they'll become even more insufferable!

The rwx permissions: files vs directories

Permission	On a file	On a directory
r (read)	Read the file. (Make a copy of it, display it, compile it.)	List the names of the files in a directory.
w (write)	Write to the file. (Edit or append to it or copy over it.)	Create file (links) in the directory, or remove them.
x (execute)	Run the file as a command. (Applicable to compiled programs or to scripts.)	Access the files in the directory, use the directory in a pathname.

known as the "set user ID bit", the "set group ID bit", and the "sticky bit" which are less well understood. The "set user ID" bit (**setuid** to its friends) is probably the most interesting, and by way of introducing it, consider these three facts:

1. Passwords in Linux are stored in a file called **/etc/shadow**. (Actually, it's the password hashes that get stored.)
2. The shadow file is heavily protected – ordinary users can neither read or write to it.
3. Users can change their passwords without administrative help.

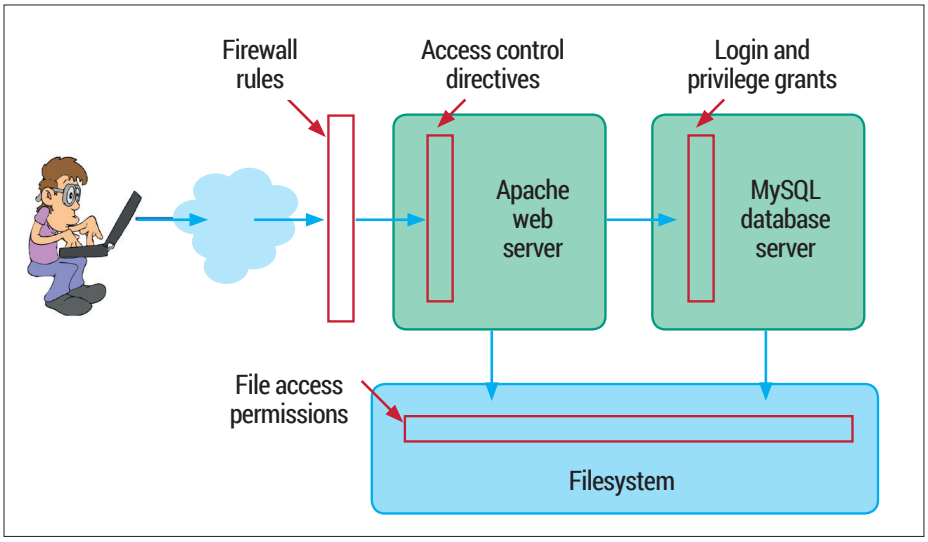
So how does this work, exactly? The following commands give us a clue:

```
$ which passwd
/usr/bin/passwd
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 47032 Feb 17  2014 /usr/bin/
passwd
```

The thing to notice is the **s** in the fourth character position, where the **x** would normally be. This is the famous **setuid** bit. Let me explain.

If a file is executable, and it has the **setuid** bit set, then while it is running it runs with the effective privileges of its owner – in this case root. The program is said to run "setuid to root". As root, of course, it can do anything, including writing to the shadow file. Once the program terminates, you're back to your normal identity at the shell prompt. This mechanism lies at the heart of all "privilege escalation" events in Linux. It's how commands like **su** and **sudo** work, for example. It's also used by programs like **ping**, not because they need to elevate their access into the file system, but because they need to reach right down to the IP layer and construct unusual network packets, which only root processes can do.

The **setuid** bit was invented by the late, great Dennis Ritchie while he was working for Bell Telephone Laboratories. They obtained a patent (US patent 4135240 – look it up of you're interested) though they never charged licensing fees and later placed the invention into the public domain. Interestingly, the operation of **setuid** was explained on the patent by a diagram of logic gates and signals, it being unclear at the time whether a software algorithm could actually be patented.



Linux applies access control at several points within a LAMP (Linux, *Apache*, *MariaDB*/*MySQL*, PHP/Perl/Python) stack, some in the kernel and some in userspace.

Finding out who you are

We can demonstrate the effect of the **setuid** bit with a trivial C program using the **getuid()** and **geteuid()** system calls, which report on the real and effective UID of the process respectively. Here's the code:

```
#include <stdio.h>
main()
{
 printf("Real ID = %d\n", getuid());
 printf("Effective ID = %d\n", geteuid());
}
```

I won't insult your intelligence by dissecting this! Now compile and run it:

```
gcc uiddemo.c -o uiddemo
$ ./uiddemo
Real ID = 1000
Effective ID = 1000
```

No surprises here. 1000 is the UID of my regular account. Now try this:

```
$ sudo chown root uiddemo
$ sudo chmod u+s uiddemo
$ ls -l uiddemo
-rwsrwxr-x 1 root chris 8620 Sep 28 18:52 uiddemo
$ ./uiddemo
Real ID = 1000
Effective ID = 0
```

Now that the program is running **setuid** to root, it reports an effective UID of zero, and at this point, it is all-powerful.

The **setuid** bit is not honoured on shell scripts. To demonstrate this, begin by resetting the ownership and mode of **uiddemo** back to what it was:

```
$ sudo chown chris uiddemo
$ sudo u-s uiddemo
```

Now wrap the program in a tiny shell script called **uuidemo-bash** like this:

```
#!/bin/bash
./uiddemo
```

and make the shell script itself run **setuid** to root:

```
$ sudo chown root uiddemo-bash
$ sudo chmod u+s uiddemo-bash
$ ls -l uiddemo-bash
-rwsrwxr-x 1 root chris 24 Sep 28 18:57 uiddemo-bash
$ ./uiddemo-bash
Real ID = 1000
Effective ID = 1000
```

The output shows clearly that the script is owned by root and has the **setuid** bit on; nonetheless, no privilege change has occurred.

Programs that run **setuid** to root need to be coded with great care to avoid design flaws, buffer overflows, or other vulnerabilities which would put them at risk of being subverted.

You can find all the **setuid** programs on your system with the command:

```
$ sudo find / -user root -perm +4000 -ls
```

Hopefully, you shouldn't find too many. And if you claim to be a security-conscious system adminstrator, you really ought to be able to say what all of them are for.

The "set group id" (**setgid**) bit is similar. Whilst the program is running, it runs with the effective group of its owner. Again, **find** will locate these for you:

```
$ sudo find / -perm +2000 -ls
```

You'll probably see quite a few administrative group ownerships showing up in this list. Running **setuid** is potentially more dangerous than running **setgid**, and running **setuid** to root is most dangerous of all. But using **setgid** effectively takes more thought about setting file groups and permissions. And sometimes **setuid**-to-root is necessary to get the job done.

The **setgid** bit has a totally different meaning when applied to a directory. Normally, when a file is created, its group is taken from the primary group of its creator. However, if the file is being created in a directory that has the **setgid** bit set, the group will be inherited from that of the directory. (This obscure feature makes a wonderful Linux interview question, by the way!) See the box Create A Shared Directory for an example of how to exploit this feature.

The sticky bit

In the very early days of Unix, the "sticky bit" would be set on a frequently used executable to encourage the operating system to keep it loaded in memory after use, thus speeding things up the next time it was run. These days, a program's code is paged in as part of the virtual memory management, and the sticky bit's original role has long been redundant. It has, however, found an important retirement job when applied to directories.

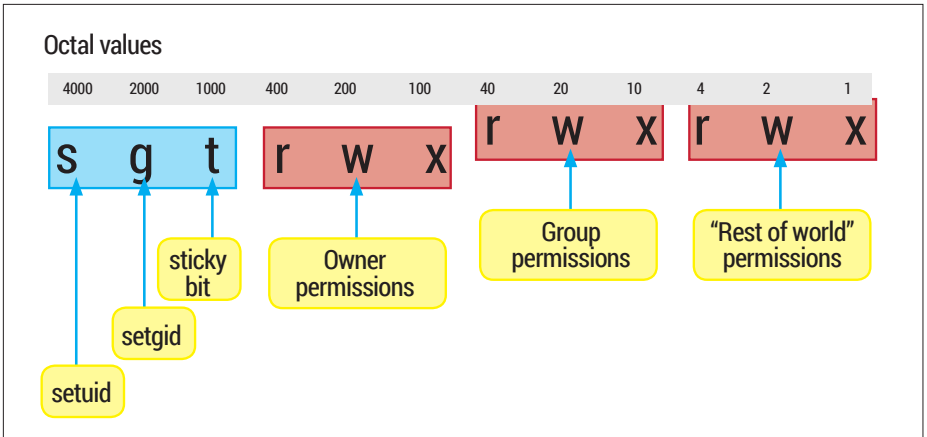
As we have seen, you can delete any file if you have write permission on the directory you're deleting it from. This doesn't work too well for world-writable directories (**/tmp** is the obvious example; you may find a few others) because you would be able to delete other peoples' files. The sticky bit changes the rules so that you can only delete files that you own.

The way that **ls -l** shows you these three extra bits (putting an **s** or **t** over the top of the 'x' bits) is confusing to say the least. It would have been better, perhaps, for **ls** to squander three more character positions on the line to represent them explicitly, but I guess there are too many scripts out there that make assumptions about the output format from **ls -l** to consider changing it now. The assumption is that the 'x' bit that's underneath is set (for example it doesn't make sense to set the **setuid** bit on a file if it's not executable). If the underlying 'x' isn't set, you'll see an upper-case S or T instead (see the box "Fool your boss!") but you need to be pretty eagle-eyed to spot this.

The scoop on groups

Every user has a primary group. The primary group assigned to them when they log in is taken from the fourth field of **/etc/passwd**. On many systems, adding a user account will, by default, add a new group with the same name, so that each user has a personal primary group that only they belong to.

However, it's easy to create new groups and assign a user any primary group of your choice when you create their account. A user can also have a number of secondary group memberships. These are defined in **/etc/group**. For example, if you see this line in **/etc/group**:



Every file and directory carries these 12 access control bits; each can be identified by its octal value.

```
editors:x:1002:andrew,graham
```

then andrew and graham have "editors" as a secondary group.

You can create a new group like this:

```
$ sudo addgroup editors
Adding group `editors' (GID 1002) ...
Done.
```

And you can give a user secondary group membership like this:

```
$ sudo usermod -a -G editors andrew
```

When a user creates a file, the file's group is normally set to the user's primary group (for an exception to this see the box Create A Shared Directory). When a user accesses a file that he's not the owner of, if the file's group matches any of the user's groups, (primary or secondary) then he'll see the group permissions on the file. In effect, you are "in" all your groups simultaneously.

Move users around

So for example, user andrew with primary group "andrew" and secondary group "editors" will be able to read the file **/tmp/target** with ownership and permissions like this:

```
$ ls -l /tmp/target
-rw-r----- 1 root editors 24 Sep 29 13:47 /tmp/target
```

It's also possible for a user to switch his primary group to be any of his secondary groups, though this feature is rarely used in my experience. So, for example, andrew can switch his primary group to be "editors", like this:

```
$ newgrp editors
```

This command starts a new shell with the new primary group identity. (And by the way, **newgrp** runs **setuid**-to-root to allow it to do this.) The following sequence of commands shows **newgrp** in action. Assume andrew has just logged in, and pay attention to the group of the files **f1** and **f2**:

```
$ touch /tmp/f1; ls -l /tmp/f1
-rw-rw-r-- 1 andrew andrew 0 Sep 29 14:33 /tmp/f1
$ newgrp editors
$ touch /tmp/f2; ls -l /tmp/f2
-rw-rw-r-- 1 andrew editors 0 Sep 29 14:33 /tmp/f2
```

You can set a password on a group, though hardly anyone uses this feature. If you do, a user can make any group his primary group as long as he knows the password. Let's try. First, we'll add a new group called "hackers":

```
# groupadd hackers
```

Now we'll set a password on it:

```
# gpasswd hackers
Changing the password for group hackers
New Password:
Re-enter new password:
```

The password hashes are stored in **/etc/gshadow**, analogous to **/etc/shadow** for user passwords. Now, if andrew tries to make hackers his primary group, although he's not a member, he'll succeed if he knows the password:

```
$ newgrp hackers
Password:
$ id
uid=1001(andrew) gid=1004(hackers) groups=1001(andrew),1002(editors),1004(hackers)
```

Be aware, though, that telling someone the password for a group is not the same as making them a member of the group.

Although file permissions are not the only place that Linux applies access controls, they are absolutely essential to system security. In my experience a high proportion of system problems or security holes come down to misunderstanding or misapplying file permissions. Knowing how they really work is important.

"It's possible for a user to switch his primary group to be any of his secondary groups."

The three "extra" mode bits -- less well known but very important

Permission	On a file	On a directory
s (setuid)	Run an executable file with the effective user privilege of its owner	Unused
g (getuid)	Run an executable file with the effective user privilege of its group	Files created in this directory will inherit their group from the group of the directory
t (sticky bit)	Unused	Only allow users to delete files from this directory if they own the file

Command of the month: umask

Strictly speaking, **umask** isn't a command, it's a shell built-in, but I don't mind thinking of it as a command if you don't. The **umask** setting (usually established in a shell startup file such as **~/.profile**) is used to limit the permissions that will be assigned to any file you create. It's a bit confusing because it sort of works 'upside down' – the permission bits that are set in **umask** will be withheld from any files you create.

Maybe a couple of examples will help. First we'll set our **umask** to zero:

```
$ umask 000
```

Now if we create a directory and check its permissions we see something like this:

```
$ mkdir test1
$ ls -ld test1
drwxrwxrwx 2 andrew andrew 4096 Sep 29 21:20 test1
```

Notice the very liberal permissions on the directory. Now let's tighten things up by setting a **umask** and repeating the experiment:

```
$ umask 027
$ mkdir test2
$ ls -ld test2
drwxr-x--- 2 andrew andrew 4096 Sep 29 21:20 test2
```

We see that the bits that are set in the umask (we might represent it as ----w-rwx) are NOT set in the file's permissions. To be exact, to compute the permissions that will be assigned to a file when it is first created, take the one's complement of **umask** and perform a bit-wise logical **and** operation with the permissions requested by the program that created the file. Also, you should understand that **umask** only takes effect at the instant a file or directory is created. It doesn't apply retrospectively to existing files.