Dr Brown's Administeria

Dr Chris Brown
The Doctor provides Linux training, authoring and consultancy. He finds his PhD in particle physics to be of no help in this work at all.

Esoteric system administration goodness from the impenetrable bowels of the server room.

Sorting the men from the boys

Last week I taught a Linux security class. The group was a typical mixture of folks who had an adequate background in driving Linux and folks who, well… didn't. A snatch of conversation during a lab exercise might go something like this:
Me: "Could you check the permissions on the log file?"
Student (after a long pause): "How do I do that?"
Me: "Well, you could try **ls -l**."
Student: "OK, but how was I supposed to know?"

Then there are the guys who've only allocated six minutes of their entire lives to learn *Vi*. They don't think that editors should need to be learned and don't see why *Vi* should be any different. So they never get the thing out of first gear. Or they'll copy the file on to a memory stick, edit it on Windows with *Notepad* and copy it back. (Sadly, I'm not making this up!) And some of them have been wasting time like this for years. They can type *grep* commands under supervision, but would never think of using it as a problem-solving tool, and couldn't design a regular expression if their life depended on it.

Class act

And then there's the 'let's not rush into this' brigade, who'll type in a command then sit and admire it awhile, waiting for it to mature, perhaps, before entrusting it to the tender mercies of the shell by actually pressing Enter.

Am I being a little cruel? Maybe. But if I applied for a job building circuit boards and picked up the soldering iron at the wrong end, or turned up at your quilting class and couldn't thread a needle, you'd probably send me away. If you come to my security class and can't tell a SIGHUP from an inode, you'll probably get to stay. But expect a hard week!

Toys for the boys

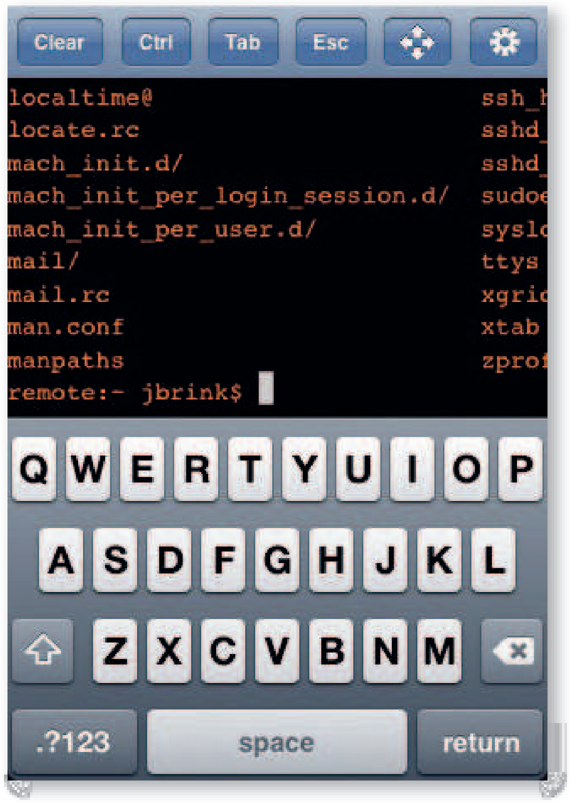
Want to reformat your server's hard drive from your iPhone? Here's how it's done…

Anyone who'd like to be able to securely administer their Linux servers from, say, Margate beach (and who wouldn't?) might be interested in a piece of software called *TouchTerm*, which is basically an SSH client and terminal emulator for the Apple iPhone.

TouchTerm includes a direct port of the OpenSSL and OpenSSH software to the iPhone and offers RSA/DSA key-based authentication and public key distribution via email. It also includes an emulator for a VT100 terminal, giving you a standard command line interface and even enabling *curses*-based programs such as *Vi* and *Top* to be used. (The VT100, for those of you too young to remember, was a character-based terminal made by Digital Equipment Corporation, which was popular around 1980. The control sequences of ASCII characters that the VT100 used for things such as cursor positioning became something of a *de facto* standard and is often supported by terminal emulators.)

I've included a screenshot that gives just a glimpse of the user interface; there are additional screens for SSH key management and for editing server connections. (Once you've defined a connection you can connect with a single tap.)

Find out more at **www.jbrink.net**. You can download *TouchTerm* from the *iTunes* App Store for the princely sum of $2.99.



```
localtime@                          ssh_h
locate.rc                           sshd_
mach_init.d/                        sshd_
mach_init_per_login_session.d/      sudoe
mach_init_per_user.d/               syslo
mail/                               ttys
mail.rc                             xgrid
man.conf                            xtab
manpaths                            zprof
remote:~ jbrink$ 
```

› *TouchTerm*, shown in immediate mode. Characters are sent to the server as you type them. Buffered entry modes, supporting local editing of command lines, are also available.

The joy of RWX

The Doctor reviews file permissions and answers questions no one thinks to ask.

You know about basic file permissions, right? The good ol' 'read, write, execute' for 'owner, group and other'. Of course you do! So, if we have a file like this:

```
$ ls -l foo
-rw-r----- 1 chris student 1550 2008-08-21 15:05 foo
```

We can see that chris can read and write the file, members of the student group can read it and others have no access. But what about this one?

```
$ ls -l foo
-r--rw---- 1 chris student 1550 2008-08-21 15:05 foo
```

Assuming chris is a member of the student group, can he write to the file? If I ask this in class, my students are typically split three ways, between yes, no and 'I'm not going to risk embarrassment by expressing a possibly incorrect opinion'. The correct answer is no. Since chris owns the file, he sees the first three permission bits. End of story. Linux doesn't go on to say: "Aha, but chris is a member of the student group, so he can write the file." In practice, the question rarely arises – you hardly ever see permissions that become less restrictive as you move from left to right.

Based on the same file permissions, the next question is: Can chris delete the file? In class, after I've convinced everyone that chris can't write to the file, most students decide he can't delete it either. In fact, chris probably can delete the file; it depends entirely on whether he has write permission on the directory he's trying to delete it from. That's right – you don't need write permission on a file to delete it, you only need write permission on the directory.

What about execute permission? In the case of a fully-compiled binary file, such as those you'll find in **/usr/bin**, execute permission means you're allowed to run the command. Execute and read permission are entirely separate, and in this case you don't need read permission on the file to execute it. For a script, however, the waters are slightly murkier. Suppose I have a shell script called *demo*. If I have read permission, there are a couple of ways I can run it. First, I can explicitly invoke a new shell and tell it to take the file demo as input, like this:

```
$ sh demo
```

Second, I can tell the current shell to execute the commands in the script using the shell's built-in *source* command, like this:

```
$ source demo
$ . demo      # . is a shorthand for source
```

For both of these, I only need read (not execute) permission on the script. It's also possible to arrange to run the script as if it were a command, by just typing its name like this:

```
$ ./demo
```

Linux in print

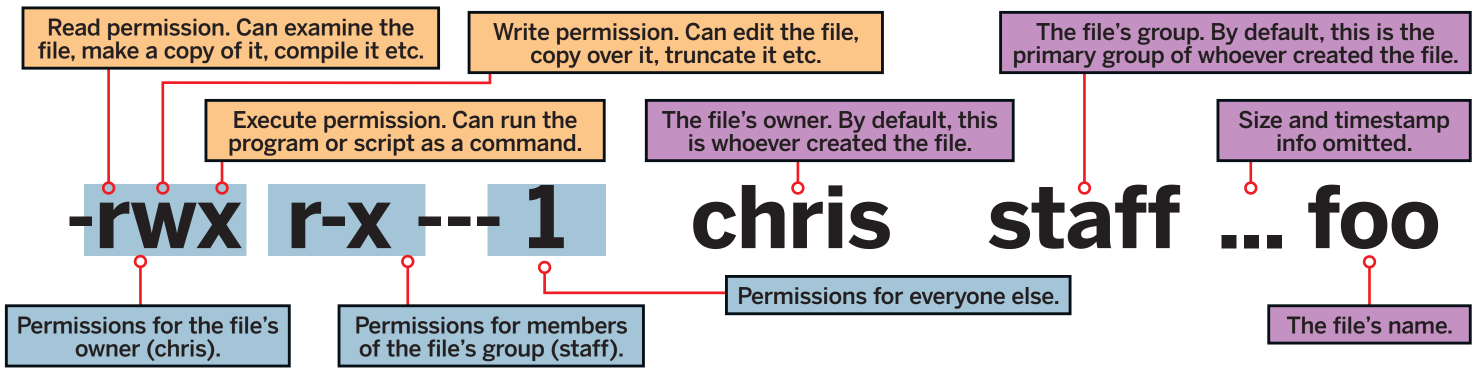
I'd love to be able to offer you some book reviews, but the market for Linux system administration books is very quiet at the moment. Before the technical book industry fell into decline (starting around 2001), people were buying a lot more books and you could publish on anything. These days, publishers have to go for topics with a bigger audience in order to avoid losing money on the title. Almost by definition, system administrators are significantly less numerous than end users, and publishers are unwilling to take the risk. There are exceptions to this – niche audiences that are more enthusiastic about buying books (known in the trade as their 'attach rate') – and occasionally a publisher will take a chance on something unusual and do well, but most of the more specialist topics simply won't sustain a book in the present economic climate. I guess that the 800-page tome I was planning about porting Linux to your microwave will just have to wait.

In this case, a new shell is automatically invoked to run the script. For this to work I need read and execute permission; execute permission alone isn't sufficient.

Another area that causes confusion is what the permissions mean when applied to a directory. The difference between read and execute permissions on a directory are well defined but subtle. If I have read permission on a directory but not execute permission, I can list the contents of a directory but not **cd** into it. If I have execute permission but not read permission, I can **cd** into the directory but not list its contents, although surprisingly I can access a file by name. The message is: to have sane access to a directory, you need both read and execute permission. It doesn't make sense to have one without the other and, as far as I can tell, no Linux directories have these combinations of permissions.

The sticky bit

Finally, a mention of the 'sticky bit'. This flag originated in Unix in the mid 70s. When set, it told the kernel to keep the code segment of the program in swap space after the process ended. This speeded subsequent executions by enabling the kernel to make a single operation of moving the program from swap space into memory. Thus, frequently used programs such as editors would load noticeably faster. Although it was a good idea at the time, no current versions of Unix honour the sticky bit on regular files and Linux never has. But it does have an important meaning for directories: it changes the rules about who can delete files. If the sticky bit is set on a directory, files within it can be deleted only by their owner, the directory's owner or the superuser. »


Read permission. Can examine the file, make a copy of it, compile it etc.
Write permission. Can edit the file, copy over it, truncate it etc.
The file's group. By default, this is the primary group of whoever created the file.
Execute permission. Can run the program or script as a command.
The file's owner. By default, this is whoever created the file.
Size and timestamp info omitted.
-rwx r-x --- 1 chris staff ... foo
Permissions for the file's owner (chris).
Permissions for members of the file's group (staff).
Permissions for everyone else.
The file's name.

› The output from ls -l foo (well, most of it) and what it all means.

Leaving an audit trail

Sudo and auditctl Armed with Fedora 9 and a large bag of breadcrumbs, we examine a couple of ways to provide a record of system admin activity.

Providing a permanent record of a system administrator's actions – a list of what was done (and when, and who by) – can be invaluable when your system has got into a muddle and you'd like to know who to blame, or if you're garnering evidence of a break-in. In some companies, audit trails are a mandatory part of the security policy. Here, I'll suggest a couple of ways of ensuring that any significant system admin activity leaves behind some sort of permanent record in the log files.

The first approach is to prevent direct root logins and force all rootly activity to be performed using the **sudo** command. If *sudo* is appropriately configured, this will result in every command that root executes being logged – typically to **/var/log/secure**.

To prevent a complete free-for-all, we'll restrict this use of *sudo* to members of one specific group called 'wheel'. So, to begin, we need to configure *sudo* to say that members of the wheel group can run anything as root. The required line in the **sudoers** file is:

```
%wheel    ALL=(ALL)  ALL
```

You may find that the line is already there and simply needs 'commenting in'. Next, make at least one user a member of the wheel group. I added the user 'chris' to the group, like this:

```
# usermod -G wheel chris
```

Now I can log in as chris and use *sudo* to run single commands with root privilege, like this:

```
$ sudo /usr/sbin/useradd ellie
[sudo] password for chris:
```

In this example, I'm adding a user account. The password I'm prompted for is my own password, not the root password.

When you're sure this is working (and not before!) you can disable direct root login by locking the root password:

```
$ sudo passwd -l root
Locking password for user root.
passwd: Success
```

Now, root can't log in. You can't even use **su** to become root. You must do everything through *sudo*. (Ubuntu users will be aware that Ubuntu is set up this way by default.) While this arrangement can be irritating at times, there are some significant benefits. First, it forces you to say, very explicitly: "I need to do this as root." Second, it minimises the amount of time you spend running commands with root privilege. Third, *sudo* will log all the commands it executes. As an example, suppose I need to edit my *Apache* configuration. I would have to do it like this:

```
$ sudo vi /etc/httpd/conf/httpd.conf
```

resulting in the following line being written to **/var/log/secure**:

```
Aug 18 11:51:54 fedora9 sudo:  chris : TTY=pts/2 ; PWD=/etc/
httpd/conf ; USER=root ; COMMAND=/bin/vi /etc/httpd/conf/
httpd.conf
```

So, now we know who edited **httpd.conf** and when, and which terminal they were logged in on.

Don't try this at home!

Don't lock the root password until you're absolutely certain that there's at least one user account that can use *sudo* to run commands as root, or you're certified to perform mouth-to-mouth resuscitation. It would also help to have a fire extinguisher handy and to know how to perform (say) a rescue boot from a Live CD.

Belt and braces

Another way in which you can restrict the use of *sudo* to the wheel group is by careful setting of the *sudo* executable's group ownership and execute permissions, like this:

```
# chown root:wheel /usr/bin/sudo
# chmod 4110 /usr/bin/sudo
# ls -l /usr/bin/sudo
---s--x--- 2 root wheel 148836 2008-03-31
15:13 /usr/bin/sudo
```

Note the unusual 4110 mode. The program runs **setuid to root**, and is executable by root and members of the wheel group. This is much more restrictive than simply putting the line

```
%wheel    ALL=(ALL)  ALL
```

into the **sudoers** file, because it entirely prevents you from using *sudo* in order to provide privilege escalation to non-wheel-group members.

If you'd like much finer control over auditing, you might want to experiment with the auditing system built into the kernel itself. Using this system, you can set a 'watch' on any file in the filesystem and log any operation that reads, writes, executes or changes the permissions of the file. You can audit every system call made by a specified process or user, or (for example) record every **open()** operation on a file that fails. You can use this system to detect misconduct by unauthorised users or to gain evidence of violations of your security policy.

If you liked that, you'll love this

Suppose we've noticed that someone is performing port scans of the machines on our local network. We have the port-scanning tool *Nmap* installed; the question is, is anyone using it? Using **auditctl** we ask the kernel to audit any attempt to execute *Nmap*, like this:

```
# auditctl -w /usr/bin/nmap -p x -k port-scan
```

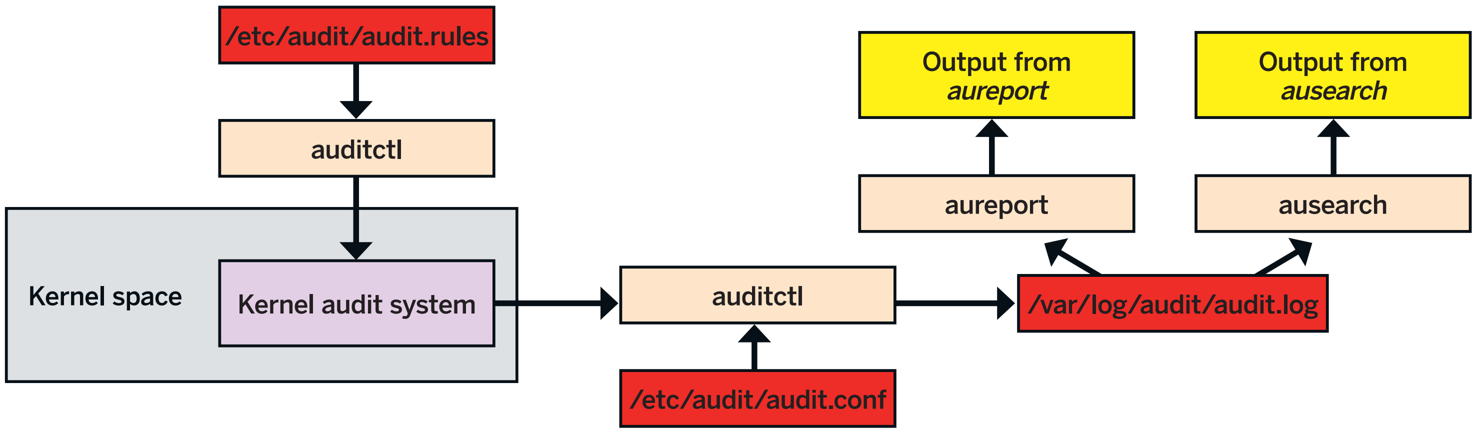
Let's dissect this command. The arguments **-w /usr/bin/nmap** specify the file on which we want to set a watch. We can't use wildcards here, just simple filenames. The arguments **-p x** specifies the kind of activity we want to log – some combination of **r** (read), **w** (write), **x** (execute) or **a** (attribute change). Finally, the arguments **-k port-scan** specify a filter key (an arbitrary text string) that will be included in the event log and can be searched on by *ausearch* or, of course, good ol' *grep*.

Later, we check the audit log by asking *ausearch* to show us those logged entries that contain our keyword **port-scan**:

```
# ausearch -k port-scan
----
```

› The main components of the auditing system are shown in the table below.

Component	Description
The kernel	The kernel generates audit events according to a specified set of event rules.
auditctl	This user-space program loads event-matching rules into the kernel. In a sense, it's a bit like *iptables*, which loads packet-filtering rules into the kernel. At boot time, the startup script for the *auditd* daemon runs **auditctl** to load an initial rule set from the file **/etc/audit/audit.rules**.
auditd	This daemon captures the event audit output from the kernel and writes it to a log file. The daemon can also manage rotation of the log files. The config file for the daemon is **/etc/audit/auditd.conf**.
aureport	A utility that's used for producing human-readable summary reports of the audit logs. It has lots of flags that control the type of audited event and the timeframe of events of interest.
ausearch	A utility that displays detailed audit records. This tool also has many options for selecting the events of interest.



```
time->Mon Aug 18 21:15:42 2008
type=PATH msg=audit(1219072542.201:117): item=1 name=(null)
inode=354635 dev=fd:00 mode=0100755 ouid=0 ogid=0
rdev=00:00 obj=system_u:object_r:ld_so_t:s0
type=PATH msg=audit(1219072542.201:117): item=0 name="/usr/
bin/nmap" inode=33539 dev=fd:00 mode=0100755 ouid=0 ogid=0
rdev=00:00 obj=system_u:object_r:traceroute_exec_t:s0
type=CWD msg=audit(1219072542.201:117):  cwd="/home/ellie"
type=EXECVE msg=audit(1219072542.201:117): argc=2
a0="nmap" a1="192.168.0.1-20"
type=SYSCALL msg=audit(1219072542.201:117): arch=40000003
syscall=11 success=yes exit=0 a0=83096e0 a1=83079d8
a2=830fd48 a3=0 items=2 ppid=2766 pid=2790 auid=0 uid=501
gid=501 euid=501 suid=501 fsuid=501 egid=501 sgid=501
fsgid=501 tty=pts2 ses=10 comm="nmap" exe="/usr/bin/nmap"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
key="port-scan"
```

All of this output resulted from a single invocation of *Nmap*. If you look carefully you'll see that the user ID 501 ran the command **nmap 192.168.0.1-20** at 9.15 pm on Monday August 18. A check for UID 501 in the password file reveals that the culprit is ellie, the scallywag we created an account for earlier in the tutorial.

The *auditctl* command does for auditing what the *iptables* command does for packet filtering – it loads rules into the kernel. Its command syntax defines a sort of language for auditing rules, just as the command syntax of *iptables* defines a language for packet-filtering rules. To present a slightly more complex example, let's create an audit rule that will log all of user ellie's unsuccessful attempts to open a file. The rule might look like this:

```
#  auditctl -a exit,always -S open -F uid=501 -F success=0
```

Let's dissect the rule again. We're appending (**-a**) a rule to the **exit** system call list. This list is used upon exit from a system call to determine if an audit event should be created. We're selecting the **open** system call for auditing. (This is the system call a program must use in order to gain access to the data in a file.) And we're auditing only those events for which the user ID is 501 (ellie's account) and the system call wasn't successful. There are many more conditions we could filter on in this way; the real or effective user and group IDs, the exit code of the system call, the inode number of the file being accessed, the process ID and so on.

Some time later, we can examine the audit log. Here, we ask *aureport* to show us failed file-related events:

```
#aureport -f
...
14. 19/08/08 18:47:58 /etc/passwd 5 no /bin/cp 0 310
...
```

I've removed many lines from *aureport*'s output to focus on one of interest. At 6.47 pm on August 19, someone failed to open the file **/etc/passwd** using the program **/bin/cp**. At the end of the line we see the event ID (310), which is effectively an index into the audit log. Use it as an argument to *ausearch* to drill deeper:

```
# ausearch -a 310
time->Tue Aug 19 18:47:58 2008
type=PATH msg=audit(1219168078.990:310): item=0 name="/etc/
passwd" inode=256011 dev=fd:00 mode=0100644 ouid=0 ogid=0
rdev=00:00 obj=system_u:object_r:etc_t:s0
type=CWD msg=audit(1219168078.990:310):  cwd="/home/ellie"
type=SYSCALL msg=audit(1219168078.990:310): arch=40000003
syscall=5 success=no exit=-13 a0=bfd559d8 a1=8201 a2=0
a3=8201 items=1 ppid=4954 pid=4978 auid=0 uid=501 gid=501
euid=501 suid=501 fsuid=501 egid=501 sgid=501 fsgid=501
tty=pts2 ses=33 comm="cp" exe="/bin/cp" subj=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023 key=(null)
```

I apologise for the inscrutability of the output, but look carefully and you can see the timestamp and user ID of the event. Bad ellie.

The audit rules established by *auditctl* are just for the 'here and now' – they won't survive a reboot. To make the rules permanent, place them into the file **/etc/audit/audit.rules**, which is read at boot time. The rules in this file are simply the parameters that would be passed to *auditctl*. So, for example, the line in **audit.rules** corresponding to the rule we played with earlier looks like this:

```
-w /usr/bin/nmap -p x -k port-scan
```

To learn more about the Linux audit system, look at the man pages for *auditd*, *auditd.conf*, *auditctl*, *aureport* and *ausearch*. Try visiting **www.intersectalliance.com**; its tool *Snare* provides GUIs for building audit rulesets and viewing the results. LXF

› The kernel audit system records key events according to rules specified by *auditctl*.

What if?

What if lawyers applied their accumulated wisdom to the C language? We'd have ended up with LEGOL, of course, but it would've been a very different language. For example, the C statement:

```
int i = 1;
```

translates to LEGOL as:

"Be it understood and acknowledged by those present that the newly created object is herewith to be named and referred to as 'i' within the scope determined by the preferred embodiment of the previously submitted namespace patent hereby incorporated by reference and further that 'i' being of the type declared known and widely recognised as integer it shall straightway without let or hindrance be assigned and alloted the value 1 (ONE) and shall retain that value until such time or times if any that some other value within the jurisdiction of the Type Safety (Promotions) Act may be assigned and allotted thereto."

I leave it as an exercise for the reader to translate

```
x += *p++;
```

into LEGOL.

Here's another worrying thought. What if Bjarne Stroustrup had decided to start with COBOL, not C? We'd now all be writing in POSTFIX INCREMENT COBOL BY ONE. And if you subscribe to the theory that C# is really C++++, the .NET folks would presumably be coding in POSTFIX INCREMENT POSTFIX INCREMENT COBOL BY ONE BY ONE.

See Stan Kelly-Bootle's book *The Computer Contradictionary* for more.