Using Power Wisely

By Jerry Peek

One Linux user is special: the superuser or *root*. If you manage any number of Linux boxes — even just your own desktop workstation — you'll inevitably need root access to configure or tune the system or manage system resources. But with great wizardly powers also come risks. This month, let's see how to manage root access and learn some powerful tricks to juggle Linux processes and shells.

What is Root? Who is Root?

As last month's column "Sharing Files (Carefully)" (available online at http://www.linux-mag.com/2002-11/power_01.html) explained, each account on a Linux system has a name and a corresponding user ID (UID). The root user has UID 0.

The Linux kernel gives special privileges to any process running with UID 0. These processes can read or write *any* file, no matter what its access permissions are. The root user — actually, any process with UID 0 — can also become any other user without entering a password. Root processes can also access any network port. The UID 0 is the key to superuser power.

Becoming Root

So how do processes — shells, text editors, and so on — get root privileges? One way is to log in with the username *root*. When any user logs in successfully, the system starts a process (a shell or a window system) running with that user's UID. That first process' UID propagates to all child processes it spawns. So, if your window system starts as root, all windows and programs also run as the superuser.

However, this method can be dangerous: as root, you can accidentally remove all devices or mistakenly start trojan horse programs. It's best to invoke superuser privileges only when absolutely necessary.

You can also use the *su* command to become root. *su* allows you to assume the UID of another user. For example, if your system has a user *joe*, typing `su joe` and entering *joe*'s password would start *joe's* shell and give you *joe*'s privileges. Indeed, you *are joe*.

If you run *su* (as shown below) with no user name, and provide root's password, you become root.

```
zsh% su
Password:
bash#
```

Note that after you enter root's password, you're running root's shell, in this example, *bash*. The # in the shell prompt reflects your newfound powers as the superuser.

Leaving Root (Temporarily?)

When you're done working as root, it's a good idea to leave that shell. Again, superuser privileges should be used only when needed. If you type `exit` (or, in many cases, just press CTRL-D), root's shell terminates, *su* also terminates, and

LOGIN VS. NON-LOGIN SHELLS

Years ago, before window systems were common, many users connected to their Unix systems through a *tty* — a physical port connected to a dialup modem or directly to a character terminal on their desk. In fact, some still do.

When first logging in, these users may need to do first-time setup, such as initializing the terminal. The Unix *login* command facilitates this by telling the user's shell to act as a *login shell*. *login* does this by calling the user's shell with a dash prepended to argument zero, like `-sh`.

A login shell reads a special shell setup file in the user's home directory, like *.profile* for Bourne shells, and *.login* for *csh*. Terminal initialization and other "first-time" commands should be stored in these special "dot" files. Any subsequent shells (child processes of the parent login shell) aren't started in this special way, so they don't read the *.profile* or *.login* setup file.

You can emulate the behavior of *login* with *su*. You can launch a login shell with the `su -` command. Plain `su` launches a non-login shell.

Because login shells are at the top of the user's process tree, there's no other shell to take control of the terminal session! This is why you can't suspend a login shell — and why, with many shells, you also can't suspend an `su -` session.

The lines between login and non-login shells have gotten fuzzy in recent years. Window managers may start login or non-login shells, for instance. This "fuzzy-ness" can cause *su* problems! For example, if a user's *.profile* runs commands that emit a terminal initialization string, and the superuser uses `su - `*`username`* to become that user, that initialization could corrupt the root user's terminal window. On the other hand, if a non-root user becomes root by typing plain *su*, it's possible that the user's PATH (command search path) could be used by root — which might cause a serious security breach!

Look for more information about this puzzle in a future column — or see the book *Unix Power Tools* from O'Reilly & Associates.

your original shell prints another prompt.

If you use root access often though, it's nice to be able to keep the root shell handy — with its command history, current directory (and directory stack), and everything else just as you left them. How can you stop a shell in its tracks, leaving it just as it was? With job control.

Modern shells ignore the CTRL-Z "stop" signal. If they didn't, they couldn't provide job control commands like `fg`, `bg`, and `%1`. To stop a shell, use the command **`suspend`**. For example, to suspend the *bash* shell and the *su* process that started it, type **`suspend`** at the prompt:

```
...
bash# suspend
[1]+ Stopped su
zsh%
```

After **`suspend`**, you're back to your original shell, running with *your* privileges. Like other jobs you might suspend, the *su* process (and the shell it spawned) is waiting — you can't run commands as root until you type **`fg`** to resume the job.

Not all shells have a **`suspend`** command. If yours doesn't, try the command **`kill -STOP $$`** instead. The *kill* command sends a signal (here, `SIGSTOP`) to the process with the given process ID (or PID) number. In general, the shell's special operator `$$` expands into the current shell's PID number, so this command sends a STOP signal to the current shell.

If you need to access several different accounts, it's handy to have several *su* commands stopped, ready to resume where you left off when you bring one into the foreground. For example, if you're *zoe*, and you first *su* to *root* as above, and then need to work on your FTP server, you could start a second *su* job:

```
...
bash# suspend
zsh: suspended su
zsh% su ftp
Password:
$ cd ~ftp/pub
...
$ suspend
zsh: suspended su ftp
zsh% jobs
[1] - suspended (signal) su
[2] + suspended (signal) su ftp
zsh%
```

The shell's `jobs` command lists stopped jobs. Stopped jobs are identified by their *job number:* `%1` for the first job, `%2` for the second, and so on. So, to resume work as root, *zoe* can type `fg %1`. Or, to work as *ftp*, she can type `fg %2`. When she restarts her *ftp* job, for instance, she'll be in the same current directory (*~ftp/pub*) where she left off.

TOOL TIP

If you stop shells often, store the *suspend* or *kill* command in a shell alias named *z*.

You may have been told to use `su -` instead of plain `su`. `su -` starts a *login shell*, which has some special features. One of those features is that most login shells can't be suspended. Also, starting a login shell in a terminal window can cause problems! The sidebar "Login vs. Non-login Shells" explains.

su Isn't Just for Shells

Yes, *su* starts shells. But it also can start any arbitrary program. If you want to start another commnd, type the command as a single argument (surrounded by single quotes) to the `-c` option. For instance, if you're logged on as a non-root user and you want to run the command **`chattr +a log`** as root, simply type:

```
$ su -c 'chattr +a log'
Password:
```

That's a quick and easy way to become root for the shortest time possible — just long enough to run one command — without even getting a single root shell prompt. Here's another example. If you want to start a terminal window running as root, try this command:

```
$ su -c 'xterm -title "root shell" &'
Password:
```

This runs the command `xterm -title "root shell" &` as root, which starts an *xterm* process running in the background, thanks to the `&`. (Notice the nested quotes: in this particular command, the double quotes must appear inside single quotes.) *xterm* starts a shell, and because *xterm* is running as root, the shell runs as root, too.

Let's see another way to do the same sort of thing, but with a root shell (from *su*) running inside a non-root *xterm* window. Because *xterm* isn't running as root, this may be a bit safer:

```
$ xterm -title "root shell" -e su &
```

su Versus *sudo*

A popular substitute for *su* is *sudo*. *sudo* allows the system administrator to delegate the use of some programs that would

TOOL TIP

For security, some accounts (like *ftp*) may have an "impossible" password (encrypted as `*` or `x`). If an account has an impossible password, you can't access those accounts using *su*. But, as we saw earlier, *root* can become any user without typing the user's password. So, if you need to work as one of these special users, first *su* to root, then *su* again to the special account.

There is one exception: this tip may not work if the special account also has a non-existent shell, such as */bin/false*, defined in */etc/passwd*. You can work around that problem, too, if your version of *su* has the `-s` option. Then the command `su -s /bin/bash ftp` will start a *bash* shell running on the *ftp* account.

otherwise only be executable by root. In short, *sudo* associates users with privileges, where a privilege describes what you can execute, as which users, and on what hosts. (For more information and instructions on how to configure *sudo*, see the November 2001 "Tech Support" column, available online at http://www.linuxmagazine.com/2001-11/tech_support_01.html). Here's a *sudo* command:

```
zsh% sudo grep allen /etc/shadow > apwd
Password:
zsh%
```

sudo prompts for *your* (not the superuser's) password, then runs the command `grep allen /etc/shadow` as *root*. You never get a root prompt.

Even better, *sudo* provides an audit trail (for you or your system administrator to monitor). *sudo* logs the commands you run, and only allows the command to run for a "short" time before killing its subprocess.

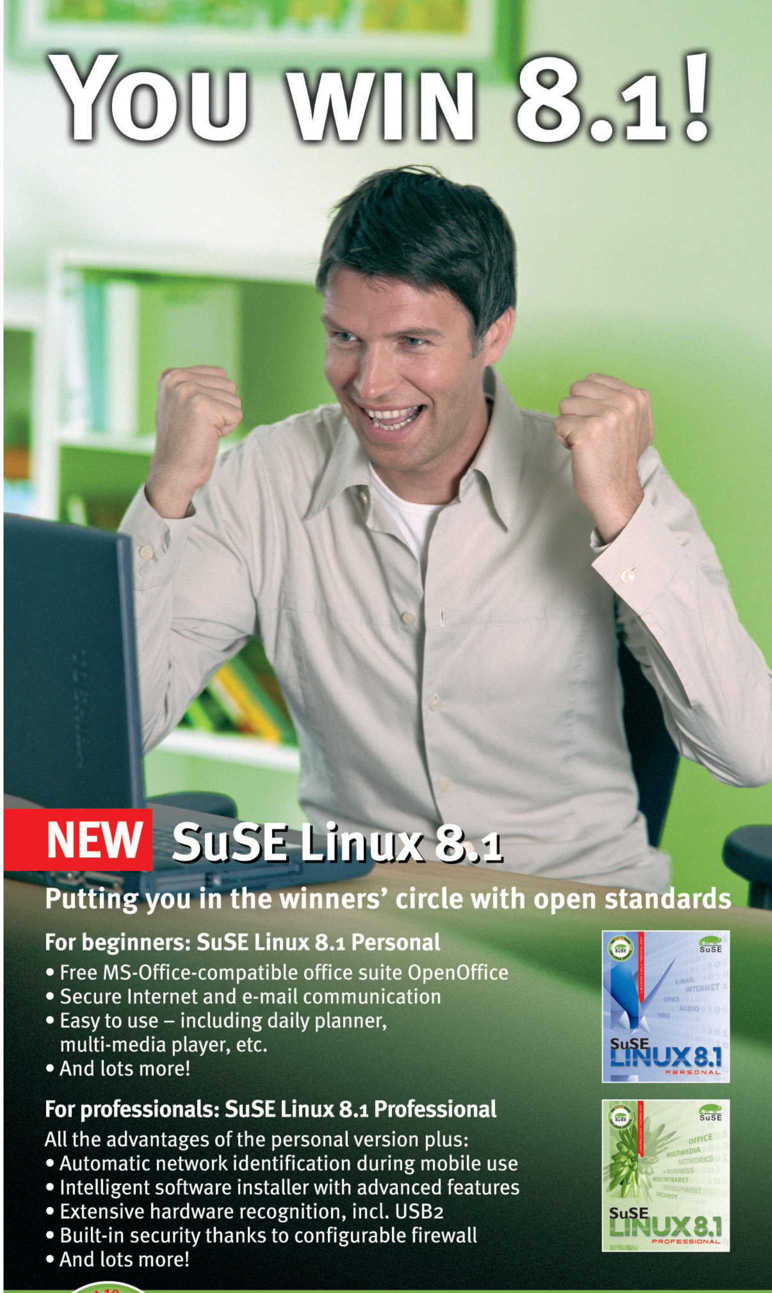
There's another difference between *su* and *sudo*. Which user owns the *apwd* file? It's not root. Why? Because the shell creates *apwd* and redirects the output of the *sudo* process to that file. So text from the "protected" shadow password file is now readable by a normal user. If you had run *grep* from a root prompt, the file would have been created as root.

A Bird, A Plane, A Super User

We've seen a hodge-podge of related concepts this month.

While you may never want to stop multiple *su* sessions or to open a new terminal window with an *su* job inside it, understanding the concepts in this column should make you a better user — "super" or not. Next month: Mozilla!

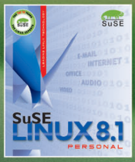
Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.



NEW SuSE Linux 8.1

Putting you in the winners' circle with open standards

For beginners: SuSE Linux 8.1 Personal
- Free MS-Office-compatible office suite OpenOffice
- Secure Internet and e-mail communication
- Easy to use – including daily planner, multi-media player, etc.
- And lots more!



For professionals: SuSE Linux 8.1 Professional

All the advantages of the personal version plus:
- Automatic network identification during mobile use
- Intelligent software installer with advanced features
- Extensive hardware recognition, incl. USB2
- Built-in security thanks to configurable firewall
- And lots more!

THE 1st 10 YEARS SuSE THE LINUX EXPERTS

SuSE Linux is celebrating its 10 year anniversary! We owe our success to you, thus we would like to thank you for your loyalty.

Please bill the following address:

Company		☐ Personal 8.1 $ 39.95
Name		
Street		☐ Professional 8.1 $ 79.95
Postal-Code	Zip-Code	
E-mail	Phone	

Order online today! http://shop.suse.com/lm

info@suse.com (888) 875-4689 (510) 628-3381

SuSE Inc. — The Linux Experts
318 Harrison Street, Suite 301
Oakland · CA 94607

SuSE