

Eliminating Root with Sudo

by Michael Lucas

While proper implementation of groups can help reduce the need for the root password, at times, users must absolutely run commands as another user (usually root). As the system administrator, you're stuck between deciding to hand out the root password or doing everything for your users. sudo provides a third way, one that can help solve this dilemma. It's a tricky program, however, and needs some care in implementation. sudo is integrated into OpenBSD, and is an add-on package for just about every other Unix-like operating system out there.

Sudo is a setuid root wrapper that implements fine-grained access control for commands that need to be run as root. It takes the command you want to run and compares it to its internal list of permissions. If sudo's permissions allow that particular user to run that command, sudo runs that command for you, with its privileges. As root can run commands as any user, sudo can execute commands as any arbitrary system user.

With proper setup, the system administrator can allow any user to run any command as any other user. Sudo is a very powerful tool, and can be configured to allow or deny almost any set of commands. As a result of this flexibility, the documentation tends to scare off new users. We're going to do a basic sudo setup that will cover almost all uses, but you should be aware that many more combinations are possible, and are documented in sudo (8) and sudoers (5).

Other than the obvious fine-grained access control sudo provides, there are a few other benefits to using sudo. One of the biggest advantages is the command logging. Every sudo command is logged, making it very easy to track who made what changes. And once you have sudo configured correctly, you can change the root password and not give it to anyone. Nobody should need the root password if they have the correct sudo permissions, after all! Reducing the number of people who have the root password can help improve security. Finally, a single sudo configuration file can be used on all of these systems, vastly easing administrator overhead.

The most overwhelmingly common disadvantage to sudo is that users and junior administrators don't like it. If people have traditionally had root access on a system, they will perceive that they're losing something when you implement sudo. The key to overcoming this is to make sure that people have the ability to do their jobs. If users think that they need the root password to perform other tasks, then your need to settle just who is responsible for what. These users may have been taking extra duties upon themselves, rather than troubling you with jobs that you should do.

A faulty sudo setup can create security holes. A thoughtless Systems configuration will create holes in the system that a clever user can use to actually become root. This problem is best dealt with by a combination of careful configuration and administrative policy.

sudo has three pieces. The first is the actual sudo (8) command, the setuid root wrapper that users will actually use. There's also sudo's configuration file, /etc/sudoers. This file is sudo's permissions table, saying who may run what commands as which user, and is fully documented in sudoers (5). Finally, the visudo (8) command allows administrators to edit the sudoers file without risking locking themselves out of the system. We'll consider each component in turn: visudo, the sudoers file, and sudo itself.

If the syntax in your sudoers file is incorrect, sudo will not run. If you're relying on sudo to provide access to the sudoers file, and you corrupt the sudoers file, you can lock yourself out of root-level activities on the system and be unable to correct your error. This is bad. visudo (8) provides some protection against this sort of error.

Much like vipw (8), visudo (8) locks the file so only one person can edit the configuration file at a time. It then opens the sudo configuration file in an editor vi (1) by default, but it respects the EDITOR environment variable. When you exit the editor, visudo parses the file and confirms that there

Eliminating Root with Sudo

by Michael Lucas

are no sudo syntax errors. This is not a guarantee that the configuration will do what you want, merely a confirmation that the file is actually a valid. visudo (8) will accept a configuration file that says "nobody may do anything via sudo" if the rules are properly formatted.

If visudo finds an error when you exit the editor, it will print out the line number and ask you what you want to do.

```
# visudo >>> sudoers file: syntax error, line 44 <<<What now?
```

Here, we've made an error on line 44. You have three choices: edit the file again, quit without saving any of the changes you made, or forcing visudo to write the sudoers file you created.

If you press e, visudo will send you back to the editor. You can go to the line it complained about, and try to find your error.

If you enter x, visudo will quit and revert the configuration file to what it was before you started editing. Your changes will be lost, but that may be all right. It's better to have the old, working configuration, than have a new, non-functional configuration.

Entering Q forces visudo to accept the file, syntax error and all. If your configuration file has an error, sudo will not run. Essentially, you're telling visudo to break sudo until such time as you log in as root to fix the problem. This is almost certainly not what you want to do!

The sudoers file tells sudo who may run which commands as which users. OpenBSD stores the sudoers file as /etc/sudoers, FreeBSD stores it as /usr/local/etc/sudoers. Never edit this file directly, even if you think you know exactly what change you want to make; always use visudo (8).

The sudo permissions syntax can be confusing until you understand it. Getting everything correct can be difficult the first time. Once you understand how sudo sets things up, however, it's very quick and easy.

The various sample sudoers files you'll find on the Internet frequently look quite complicated and difficult to understand, as they demonstrate all the nifty things you can do with sudo. The basic syntax is very simple. Each rule entry in sudoers has the following format.

```
username host = command
```

The username is the username of the user who may execute the command. host is the hostname of the system where this rule applies. sudo is designed so you can use one sudoers file on all of your systems. This space allows you to set per-host rules.

The commandfield lists the commands this rule applies to. You must have a full path to each command name, or sudo will not recognize it! (You wouldn't want people to be able to adjust their \$PATH variable to access renamed versions of commands, now would you?)

sudo defaults to not allowing anything to happen. To let a user run a command, you must create a rule that gives that user permission on that host to run that command. If any of the three fields don't match, the user cannot run the command.

You can use the ALL keyword in any of these fields to match all possible options. For example, suppose I trust user "chris" to run absolutely any command as root, on any system.

Eliminating Root with Sudo

by Michael Lucas

```
chris ALL = ALL
```

Giving a junior system administrator total control of one of my systems isn't very likely. As senior system administrator, I should know what commands Chris needs to run to do his job. Suppose Chris is in charge of the nameserver portion of this system. We control actual editing of the zone files with group permissions, but that won't help when the nameserver must be started, reloaded, or stopped. Here, I'll give him permission to run just the name daemon controller program, `ndc` (8).

```
chris ALL = /usr/sbin/ndc
```

If I'm sharing this file across several machines, it's quite probable that many of those machines are not even running a nameserver program. Here, I'll restrict which machine Chris may run this program on to the server called "dns1."

```
chris dns1 = /usr/sbin/ndc
```

On the other hand, Chris is the administrator of the email server "mail". This server is his responsibility, and he can run any commands on it whatsoever. I can set entirely different permissions for him on the mail server, and yet use the same sudoers file on both the systems.

```
chris dns1 = /usr/sbin/ndcchris mail = ALL
```

You can specify multiple entries in a single field by separating them with commas. Here, I'd like Chris to be able to mount floppy disks with `mount` (8), as well as control the nameserver.

```
chris dns1 = /usr/sbin/ndc, /bin/mount
```

You can tell sudoers that a user can run commands as a particular user, instead of root, by putting the username in parenthesis before a command. For example, suppose we have our nameserver set to run as the user "named" and all commands to control the server must be run as that user.

```
chris dns1 = (named) /usr/sbin/ndc
```

Every entry in `/etc/sudoers` must be on a single line. This can make the lines very long. If you have a long list of alias members or rules, you can skip to another line by using the `¥` character at the end of each incomplete line.

```
chris server1 =  
/sbin/fdisk,/sbin/fsck,/sbin/kldload,¥/sbin/newfs,/sbin/newfs_msdos,/sbin/mount
```

Now that you understand how sudo permissions are set, let's look at how to actually use sudo. Use `visudo`, and give your account privileges to run any command. (If you've installed sudo correctly you already have root, so this won't be a security issue.)

The first time you run sudo, sudo will prompt you for a password. Enter the password for your own account, not the root password. If you give an incorrect password, sudo will insult your typing abilities, mental facilities, or ancestry, and let you try again. After three incorrect passwords, sudo gives up on you. You'll have to re-enter the command you want to run.

Once you enter a correct password, sudo records the time. If you run sudo again within five minutes, it won't ask you for a password. After you don't use sudo for five minutes, however, you must re-authenticate. This makes work easier when you're issuing a series of commands under sudo, but times things out quickly in case you walk away from the computer.

Eliminating Root with Sudo

by Michael Lucas

When you're a regular user on a system with sudo, one thing you'll probably want to know is what commands the system administrator has permitted you to run. sudo's -l flag will tell you this.

```
# sudo -l Password: User mwlucas may run the following commands on this host:
(root) ALL#
```

If you had tighter restrictions, they would be displayed.

To run commands via sudo, just put the word "sudo" before the command you actually want to run.

For example, here's how we would become root using su via sudo.

```
# sudo su Password: #
```

Using sudo to become root simply allows the senior system administrator keep the root password a closely-held secret. This isn't entirely useful, as with unrestricted sudo access junior administrators can change the root password. Still, it's a start towards keeping the system more secure, and towards implementing sudo for all commands.

You can run more complicated commands under sudo, with all of their regular arguments. For example, tail -fis excellent to view the end of a log file, and to have new log entries appear on the end of the screen. Some log files are only visible to root, or should be -- for example, the log that contains sudo use information. You might want to view these logs without bothering to become root.

```
# sudotail -f /var/log/authlogopenbsd/usr/src/usr.bin/sudo;sudotail -f /var/log/secure
Jul 29 13:24:19 openbsd sudo: mwlucas : TTY=ttyp0 ; PWD=/home/mwlucas ;
USER=root ; COMMAND=list
Jul 29 13:30:03 openbsd sudo: mwlucas : TTY=ttyp0 ; PWD=/home/mwlucas ;
USER=root ; COMMAND=/usr/bin/tail -f /var/log/authlog ...
```

You can choose to run commands as a user other than root, if you have the appropriate permissions. For example, suppose we have our database application where commands must be run as the database user. You tell sudo to run as a particular user by using the -u flag and a username. For example, the operator user has the privileges necessary to run dump and back up the system.

```
# sudo -u operator dump /dev/sd0s1
```

All this tracking and accountability is nice, but where does it account to? sudo messages are logged to LOCAL2. Each log message contains a timestamp, the name of the user, the directory where sudo was run, and the command that was run.

```
Jul 29 11:21:02 openbsd sudo: chris : TTY=ttyp0 ; PWD=/home/chris ;
USER=root ; COMMAND=/sbin/mount /dev/fd0 /mnt
```

In the worst case, you can backtrack exactly what happened when something breaks. For example, if one of my systems doesn't reboot correctly because /etc/rc.confis missing or corrupt, I can check the sudo logs to who touched it.

```
Jul 29 11:34:56 openbsd sudo: chris : TTY=ttyp0 ; PWD=/home/chris ;
USER=root ; COMMAND=/bin/rm /etc/rc.conf
```

If everyone had been using su, or even using "sudo su" instead of sudo to run each individual command, I would have had no clue as to why the system broke. With sudo logs, once I get this computer up and running again I know who to blame. This alone makes sudo worth implementing.