11

System Administration: Core Concepts

In This Chapter

The job of a system administrator is to keep one or more systems in a useful and convenient state for users. On a Linux system, the administrator and user may both be you, with you and the computer being separated by only a few feet. Alternatively, the system administrator may be halfway around the world, supporting a network of systems, with you being one of thousands of users. On one hand, a system administrator can be one person who works part-time taking care of a system and perhaps is also a user of the system. On the other hand, several administrators can work together full-time to keep many systems running.

RUNNING COMMANDS WITH root PRIVILEGES

Some commands can damage the filesystem or crash the operating system. Other commands can invade users' privacy or make the system less secure. To keep a Linux system up and running as well as secure, Ubuntu is configured not to permit ordinary users to execute some commands and access certain files. Linux provides several ways for a trusted user to execute these commands and access these files. The default username of the trusted user with these systemwide powers is **root**; a user with these privileges is also sometimes referred to as *Superuser*. As this section explains, Ubuntu enables specified ordinary users to run commands with **root** privileges while logged in as themselves.

A user running with **root** privileges has the following powers—and more:

- Some commands, such as those that add new users, partition hard drives, and change system configuration, can be executed only by a user with **root** privileges. Such a user can configure tools, such as sudo, to give specific users permission to perform tasks that are normally reserved for a user running with **root** privileges.
- Read, write, and execute file access and directory access permissions do not affect a user with **root** privileges. A user with **root** privileges can read from, write to, and execute all files, as well as examine and work in all directories.

- Some restrictions and safeguards that are built in to some commands do not apply to a user with **root** privileges. For example, a user with **root** privileges can change any user's password without knowing the old password.

Console security

security Ubuntu Linux is not secure from a user at the console. Additional security measures, such as setting bootloader and BIOS passwords, can help secure the console. However, when a user has physical access to the hardware, as console users typically do, it is very difficult to secure a system from that user.

Least privilege

caution When you are working on any computer system, but especially when you are working as the system administrator (with **root** privileges), perform any task using the least privilege possible. When you can perform a task logged in as an ordinary user, do so. When you must run a command with **root** privileges, do as much as you can as an ordinary user, use sudo so that you have **root** privileges, complete the part of the task that has to be done with **root** privileges, and revert to being an ordinary user as soon as you can. Because you are more likely to make a mistake when you are rushing, this concept becomes more important when you have less time to apply it.

When you are running with **root** privileges in a command-line environment, by convention the shell displays a special prompt to remind you of your status. By default, this prompt is (or ends with) a pound sign (#). You can gain or grant **root** privileges in a number of ways:

- When you bring the system up in recovery mode (page 428), you are logged in as the user named **root**.
- The sudo utility allows specified users to run selected commands with **root** privileges while they are logged in as themselves. You can set up sudo to allow certain users to perform specific tasks that require **root** privileges without granting them systemwide **root** privileges. See page 406 for more information on sudo.
- Some programs ask for *your* password when they start. If sudo is set up to give you **root** privileges, when you provide your password, the program runs with **root** privileges. When a program requests a password when it starts, you stop running as a privileged user when you quit using the program. This setup keeps you from remaining logged in with **root** privileges when you do not need or intend to be.
- Any user can create a *setuid* (set user ID) file. Setuid programs run on behalf of the owner of the file and have all the access privileges that the owner has. While you are running as a user with **root** privileges, you can change the permissions of a file owned by **root** to setuid. When an ordinary user executes a file that is owned by **root** and has setuid permissions,

the program has *effective* ***root*** *privileges*. In other words, the program can do anything a user with **root** privileges can do that the program normally does. The user's privileges do not change. Thus, when the program finishes running, all user privileges are as they were before the program started. Setuid programs owned by **root** are both extremely powerful and extremely dangerous to system security, which is why a system contains very few of them. Examples of setuid programs that are owned by **root** include passwd, at, and crontab. For more information refer to "Setuid and Setgid Permissions" on page 204.

root-owned setuid programs are extremely dangerous

security Because a **root**-owned setuid program allows someone who does not know the **root** password and cannot use sudo to gain **root** privileges, it is a tempting target for a malicious user. Also, programming errors that make normal programs crash can become **root** exploits in setuid programs. A system should have as few of these programs as necessary. You can disable setuid programs at the filesystem level by mounting a filesystem with the **nosuid** option (page 490). See page 437 for a command that lists all setuid files on the local system.

optional The following techniques for gaining **root** privileges depend on unlocking the **root** account (setting up a **root** password) as explained on page 415.

- You can give an su (substitute user) command while you are logged in as yourself. When you then provide the **root** password, you will have **root** privileges. For more information refer to "su: Gives You Another User's Privileges" on page 415.
- Once the system is up and running in multiuser mode (page 431), you can log in as **root**. When you then supply the **root** password, you will be running with **root** privileges.

Some techniques limit how someone can log in as **root**. For example, PAM (page 461) controls the who, when, and how of logging in. The **/etc/securetty** file controls which terminals (ttys) a user can log in on as **root**. The **/etc/security/access.conf** file adds another dimension to login control (see the comments in the file for details).

Do not allow root access over the Internet

security Prohibiting **root** logins using login over a network is the default policy of Ubuntu and is implemented by the PAM **securetty** module. The **/etc/security/access.conf** file must contain the names of all users and terminals/workstations that you want a user to be able to log in as **root**. Initially every line in **access.conf** is commented out.

You can, however, log in as **root** over a network using ssh (page 627). As shipped by Ubuntu, ssh does not follow the instructions in **securetty** or **access.conf**. In addition, in **/etc/ssh/sshd_config**, Ubuntu sets **PermitRootLogin** to **yes** to permit **root** to log in using ssh (page 644).

sudo: Running a Command with root Privileges

Classically a user gained **root** privileges by logging in as **root** or by giving an su (substitute user) command and providing the **root** password. When an ordinary user executed a privileged command in a graphical environment, the system would prompt for the **root** password. More recently the use of sudo (www.sudo.ws) has taken over these classic techniques of gaining **root** privileges.

There is a root account, but no root password

tip As installed, Ubuntu locks the **root** account by not providing a **root** password. This setup prevents anyone from logging in to the **root** account (except when you bring the system up in recovery mode [page 428]). There is, however, a **root** account (a user with the username **root**—look at the first line in **/etc/passwd**). This account/user owns files (give the command **ls –l /bin**) and runs processes (give the command **ps –ef** and look at the left column of the output). The **root** account is critical to the functioning of an Ubuntu system.

The sudo utility enables you to run a command as though it had been run by a user logged in as **root**. This book uses the phrase "working with **root** privileges" to emphasize that, although you are not logged in as **root**, when you use sudo you have the powers of the **root** user.

Ubuntu strongly encourages the use of sudo. In fact, as shipped, Ubuntu locks the **root** account (there is no password) so you cannot use the classic techniques. There are many advantages of using sudo over using the **root** account for system administration:

- When you run sudo, it requests *your* password—not the **root** password—so you have to remember only one password.
- The sudo utility logs all commands it executes. This log can be useful for retracing your steps if you make a mistake and for system auditing.
- The sudo utility allows implementation of a finer-grained security policy than does the use of su and the **root** account. Using sudo, you can enable specific users to execute specific commands—something you cannot do with the classic **root** account setup.
- Using sudo makes it harder for a malicious user to gain access to a system. When there is an unlocked **root** account, a malicious user knows the username of the account she wants to crack before she starts. When the **root** account is locked, the user has to determine the username *and* the password to break into a system.

Some users question whether sudo is less secure than su. Because both rely on passwords, they share the same strengths and weaknesses. If the password is compromised, the system is compromised. However, if the password of a user who is allowed by sudo to do one task is compromised, the entire system may not be at risk. Thus, *if used properly,* the finer granularity of sudo's permissions structure *can* make it a more secure tool than su. Also, when sudo is used to invoke a single command, it is less likely that a user will be tempted to keep working with **root** privileges than if the user opens a **root** shell with su.

Run graphical programs using gksudo not sudo

caution Use gksudo (or kdesu from KDE) instead of sudo when you run a graphical program that requires **root** privileges. Although both utilities run a program with **root** privileges, sudo uses your configuration files, whereas gksudo uses **root**'s configuration files. Most of the time this difference is not important, but sometimes it is critical. Some programs will not run when you call them with sudo. Using gksudo can prevent incorrect permissions from being applied to files related to the X Window System in your home directory. In a few cases, misapplying these permissions can prevent you from logging back in. In addition, you can use gksudo in a launcher (page 107) on the desktop or on a panel.

Using sudo may not always be the best, most secure way to set up a system. On a system used by a single user, there is not much difference between using sudo and carefully using su and a **root** password. In contrast, on a system with several users, and especially on a network of systems with central administration, sudo can be set up to be more secure than su. If you are a dyed-in-the-wool UNIX/Linux user who cannot get comfortable with sudo, it is easy enough to give the **root** account a password and use su. See page 415.

When you install Ubuntu, the first user you set up is included in the **admin** group. As installed, sudo is configured to allow members of the **admin** group to run with **root** privileges. Because there is no **root** password, initially the only way to perform privileged administrative tasks from the command line is for the first user to run them using sudo. Graphical programs call other programs, such as gksudo (see the adjacent tip), which in turn call sudo for authentication.

Timestamp By default, sudo asks for *your* password (not the **root** password) the first time you run it. At that time, sudo sets your *timestamp*. After you supply a password, sudo will not prompt you again for a password for 15 minutes, based on your timestamp.

In the following example, Sam tries to set the system clock working as the user **sam**, a nonprivileged user. The date utility displays an error message followed by the expanded version of the date he entered. When he uses sudo to run date to set the system clock, sudo prompts him for his password, and the command succeeds.

```
$ date 10151620
date: cannot set date: Operation not permitted
Wed Oct 15 16:20:00 PDT 2008

$ sudo date 10151620
[sudo] password for sam:
Wed Oct 15 16:20:00 PDT 2008
```

Next Sam uses sudo to unmount a filesystem. Because he gives this command within 15 minutes of the previous sudo command, he does not need to supply a password:

```
$ sudo umount /music
$
```

Now Sam uses the **–l** option to check which commands sudo will allow him to run. Because he was the first user registered on the system (and is therefore a member of the **admin** group), he is allowed to run any command as any user.

```
$ sudo -l
User sam may run the following commands on this host:
    (ALL) ALL
```

Spawning a **root** shell When you have several commands you need to run with **root** privileges, it may be easier to spawn a **root** shell, give the commands without having to type **sudo** in front of each one, and exit from the shell. This technique defeats some of the safeguards built in to sudo, so use it carefully and remember to return to a non**root** shell as soon as possible. (See the tip on least privilege on page 404.) Use the sudo **–i** option to spawn a **root** shell:

```
$ pwd
/home/sam
$ sudo -i
# id
uid=0(root) gid=0(root) groups=0(root)
# pwd
/root
# exit
$
```

In this example, sudo spawns a **root** shell, which displays a **#** prompt to remind you that you are running with **root** privileges. The id utility displays the identity of the user running the shell. The exit command (you can also use CONTROL-D) terminates the **root** shell, returning the user to his normal status and his former shell and prompt.

sudo's environment The pwd builtin in the preceding example shows one aspect of the modified environment the **–i** option (page 409) creates. This option spawns a **root** login shell (a shell with the same environment as a user logging in as **root** would have) and executes **root's** startup files (page 277). Before issuing the **sudo –i** command, the pwd builtin shows **/home/sam** as Sam's working directory; after the command it shows **/root**, **root**'s home directory, as the working directory. Use the **–s** option (page 409) to spawn a **root** shell without modifying the environment. When you call sudo without an option, it runs the command you specify in an unmodified environment. To demonstrate, the following example has sudo run pwd without an option. The working directory of a command run in this manner does not change.

```
$ pwd
/home/sam
$ sudo pwd
/home/sam
```

Redirecting output The following command fails because, although the shell that sudo spawns executes ls with **root** privileges, the nonprivileged shell that the user is running redirects the output. The user's shell does not have permission to write to **/root**.

```
$ sudo ls > /root/ls.sam
-bash: /root/ls.sam: Permission denied
```

There are several ways around this problem. The easiest is to pass the whole command line to a shell running under sudo:

```
$ sudo bash -c 'ls > /root/ls.sam'
```

The bash –c option spawns a shell that executes the string following the option and then terminates. The sudo utility runs the spawned shell with **root** privileges. You can quote the string to prevent the nonprivileged shell from interpreting special characters. You can also spawn a **root** shell with **sudo –i**, execute the command, and exit from the privileged shell. (See the preceding section.)

optional Another way to deal with the problem of redirecting output of a command run by sudo is to use tee (page 240):

```
$ ls | sudo tee /root/ls.sam
...
```

This command writes the output of ls to the file but also displays it. If you do not want to display the output, you can have the nonprivileged shell redirect the output to **/dev/null** (page 471). The next example uses this technique to do away with the screen output and uses the **–a** option to tee to append to the file instead of overwriting it:

```
$ ls | sudo tee -a /root/ls.sam > /dev/null
```

OPTIONS

You can use command-line options to control how sudo runs a command. Following is the syntax of an sudo command line:

*sudo [**options**] [**command**]*

where ***options*** is one or more options and ***command*** is the command you want to execute. Without the **–u** option, sudo runs ***command*** with **root** privileges. Some of the more common ***options*** follow; see the sudo man page for a complete list.

–b (**background**) Runs ***command*** in the background.

–i (**initial login environment**) Spawns the shell that is specified for **root** (or another user specified by **–u**) in **/etc/passwd**, running **root**'s (or the other user's) startup files, with some exceptions (e.g., **TERM** is not changed). Does not take a ***command***.

–k (**kill**) Resets the timestamp (page 407) of the user running the command, which means the user must enter a password the next time she runs sudo.

–L (**list defaults**) Lists the parameters that you can set on a Defaults line (page 413) in the **sudoers** file. Does not take a ***command***.

–l (**list commands**) Lists the commands the user who is running sudo is allowed to run on the local system. Does not take a ***command***.

–s (**shell**) Spawns a new **root** (or another user specified by **–u**) shell as specified in the **/etc/passwd** file. Similar to **–i** but does not change the environment. Does not take a ***command***.

–u *user* Runs ***command*** with the privileges of ***user***. Without this option sudo runs ***command*** with **root** privileges.

sudoers: Configuring sudo

As installed, sudo is not as secure and robust as it can be if you configure it carefully. The sudo configuration file is **/etc/sudoers**. The best way to edit **sudoers** is to use visudo by giving this command: **sudo visudo**. The visudo utility locks, edits, and checks the grammar of the **sudoers** file. By default, visudo calls the nano editor. You can set the **VISUAL** environment variable to cause visudo to call vi with the following command:

```
$ export VISUAL=vi
```

Replace **vi** with the textual editor of your choice. Put this command in a startup file (page 277) to set this variable each time you log in.

Always use visudo to edit the sudoers file

caution A syntax error in the **sudoers** file can prevent you from using sudo to gain **root** privileges. If you edit this file directly (without using visudo), you will not know that you introduced a syntax error until you find you cannot use sudo. The visudo utility checks the syntax of **sudoers** before it allows you to exit. If it finds an error, it gives you the choice of fixing the error, exiting without saving the changes to the file, or saving the changes and exiting. The last is usually a poor choice, so visudo marks the last choice with **(DANGER!)**.

In the **sudoers** file, comments, which start with a pound sign (**#**), can appear anywhere on a line. In addition to comments, this file holds two types of entries: aliases and user privilege specifications. Each of these entries occupies a line, which can be continued by terminating it with a backslash (****).

User Privilege Specifications

The format of a line that specifies user privileges is as follows (the whitespace around the equal sign is optional):

user_list host_list = *[(**runas_list**)]* ***command_list***

- The ***user_list*** specifies the user(s) this specification line applies to. This list can contain usernames, groups (prefixed with **%**), and user aliases (next section).
- The ***host_list*** specifies the host(s) this specification line applies to. This list can contain one or more hostnames, IP addresses, or host aliases (discussed in the next section). You can use the builtin alias ALL to cause the line to apply to all systems that refer to this **sudoers** file.
- The ***runas_list*** specifies the user(s) the commands in the ***command_list*** can be run as when sudo is called with the **–u** option (page 409). This list can contain usernames, groups (prefixed with **%**), and runas aliases (discussed in the next section). Must be enclosed within parentheses. Without ***runas_list,*** sudo assumes **root**.

- The ***command_list*** specifies the utilities this specification line applies to. This list can contain names of utilities, names of directories holding utilities, and command aliases (discussed in the next section). All names must be absolute pathnames; directory names must end with a slash (**/**).

If you follow a name with two adjacent double quotation marks (" "), the user will not be able to specify any command-line arguments, including options. Alternatively, you can specify arguments, including wildcards, to limit the arguments a user is allowed to use.

Examples The following user privilege specification allows Sam to use sudo to mount and unmount filesystems (run mount and umount with **root** privileges) on all systems (as specified by **ALL**) that refer to the **sudoers** file containing this specification:

```
sam     ALL=(root) /bin/mount, /bin/umount
```

The (**root**) ***runas_list*** is optional. If you omit it, sudo allows the user to run the commands in the ***command_list*** with **root** privileges. In the following example, Sam takes advantage of these permissions. He cannot run umount directly; instead, he must call sudo to run it.

```
$ whoami
sam
$ umount /music
umount: only root can unmount /dev/sdb7 from /music
$ sudo umount /music
[sudo] password for sam:
$
```

If you replace the line in **sudoers** described above with the following line, Sam is not allowed to unmount **/p03**, although he can still unmount any other filesystem and can mount any filesystem:

```
sam     ALL=(root) /bin/mount, /bin/umount, !/bin/umount /p03
```

The result of the preceding line in **sudoers** is shown below. The sudo utility does not prompt for a password because Sam has entered his password within the last 15 minutes.

```
$ sudo umount /p03
Sorry, user sam is not allowed to execute '/bin/umount /p03' as root on localhost.
```

The following line limits Sam to mounting and unmounting filesystems mounted on **/p01**, **/p02**, **/p03**, and **/p04**:

```
sam     ALL= /bin/mount /p0[1-4], /bin/umount /p0[1-4]
```

The following commands show the result:

```
$ sudo umount /music
Sorry, user sam is not allowed to execute '/bin/umount /music' as root on localhost.
$ sudo umount /p03
$
```

Default privileges for **admin** group

As shipped, the **sudoers** file contains the following lines:

```
# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL
```

This user privilege specification applies to all systems (as indicated by the **ALL** to the left of the equal sign). As the comment says, this line allows members of the **admin** group (specified by preceding the name of the group with a percent sign: **%admin**) to run any command (the rightmost **ALL**) as any user (the **ALL** within parentheses). When you call it without the **–u** option, the sudo utility runs the command you specify with **root** privileges, which is what sudo is used for most of the time.

If the following line were in **sudoers**, it would allow members of the **wheel** group to run any command as any user with one exception: They would not be allowed to run passwd to change the **root** password.

```
%wheel ALL=(ALL) ALL, !/usr/bin/passwd root
```

optional

In the **%admin ALL=(ALL) ALL** line, if you replaced (**ALL**) with (**root**), or if you omitted (**ALL**), you would still be able to run any command with **root** privileges. You would not, however, be able to use the **–u** option to run a command as another user. Typically, when you can have **root** privileges, this limitation is not an issue. Working as a user other than yourself or **root** allows you to use the least privilege possible to accomplish a task, which is a good idea.

For example, if you are in the **admin** group, the default entry in the **sudoers** file allows you to give the following command to create and edit a file in Sam's home directory. Because you are working as Sam, he will own the file and be able to read from and write to it.

```
$ sudo -u sam vi ~sam/reminder
$ ls -l ~sam/reminder
-rw-r--r-- 1 sam sam 15 Mar  9 15:29 /home/sam/reminder
```

Aliases

An alias enables you to rename and/or group users, hosts, or commands. Following is the format of an alias definition:

alias_type alias_name = *alias_list*

where ***alias_type*** is the type of alias (**User_Alias**, **Runas_Alias**, **Host_Alias**, **Cmnd_Alias**), ***alias_name*** is the name of the alias (by convention in all uppercase letters), and ***alias_list*** is a comma-separated list of one or more elements that make up the alias. Preceding an element of an alias with an exclamation point (**!**) negates it.

User_Alias

The ***alias_list*** for a user alias is the same as the ***user_list*** for a user privilege specification (discussed in the previous section). The following lines from a **sudoers** file define three user aliases: OFFICE, ADMIN, and ADMIN2. The ***alias_list*** that defines the first alias includes the usernames **mark**, **sam**, and **sls**; the second includes two usernames and members of the **admin** group; and the third includes all members of the **admin** group except Max.

```
User_Alias      OFFICE = mark, sam, sls
User_Alias      ADMIN = max, zach, %admin
User_Alias      ADMIN2 = %admin, !max
```

Runas_Alias The ***alias_list*** for a runas alias is the same as the ***runas_list*** for a user privilege specification (discussed in the previous section). The following SM runas alias includes the usernames **sam** and **sls**:

```
Runas_Alias     SM = sam, sls
```

Host_Alias Host aliases are meaningful only when the **sudoers** file is referenced by sudo running on more than one system. The ***alias_list*** for a host alias is the same as the ***host_list*** for a user privilege specification (discussed in the previous section). The following line defines the LCL alias to include the systems named **dog** and **plum**:

```
Host_Alias      LCL = dog, plum
```

If you want to use fully qualified hostnames (**hosta.example.com** instead of just **hosta**) in this list, you must set the **fqdn** flag (discussed in the next section), which can slow the performance of sudo.

Cmnd_Alias The ***alias_list*** for a command alias is the same as the ***command_list*** for a user privilege specification (discussed in the previous section). The following command alias includes three files and, by including a directory (denoted by its trailing /), incorporates all the files in that directory:

```
Cmnd_Alias     BASIC = /bin/cat, /usr/bin/vi, /bin/df, /usr/local/safe/
```

Defaults (Options)

You can change configuration options from their default values by using the **Defaults** keyword. Most values in this list are flags that are implicitly Boolean (can either be on or off) or strings. You turn on a flag by naming it on a Defaults line, and you turn it off by preceding it with a **!**. The following line in the **sudoers** file would turn off the **lecture** and **fqdn** flags and turn on **tty_tickets**:

```
Defaults        !lecture,tty_tickets,!fqdn
```

This section lists some common flags; see the **sudoers** man page for a complete list.

env_reset Causes sudo to reset the environment variables to contain the **LOGNAME**, **SHELL**, **USER**, **USERNAME**, and **SUDO_*** variables only. The default is on. See the **sudoers** man page for more information.

fqdn (**fully qualified domain name**) Performs DNS lookups on *FQDNs* (page 1109) in the **sudoers** file. When this flag is set, you can use FQDNs in the **sudoers** file, but doing so may negatively affect sudo's performance, especially if DNS is not working. When this flag is set, you must use the local host's official DNS name, not an alias. If hostname returns an FQDN, you do not need to set this flag. The default is on.

insults Displays mild, humorous insults when a user enters a wrong password. The default is off. See also **passwd_tries**.

lecture=*freq* Controls when sudo displays a reminder message before the password prompt. Possible values of ***freq*** are **never** (default), **once**, and **always**. Specifying **!lecture** is the same as specifying a ***freq*** of **never**.

mailsub=*subj* (**mail subject**) Changes the default email subject for warning and error messages from the default ***** SECURITY information for %h ***** to ***subj***. The sudo utility expands **%h** within ***subj*** to the local system's hostname. Place ***subj*** between quotation marks if it contains shell special characters (page 146).

mailto=*eadd* Sends sudo warning and error messages to ***eadd*** (an email address; the default is **root**). Place ***eadd*** between quotation marks if it contains shell special characters (page 146).

mail_always Sends email to the **mailto** user each time a user runs sudo. The default is off.

mail_badpass Sends email to the **mailto** user when a user enters an incorrect password while running sudo. The default is off.

mail_no_host Sends email to the **mailto** user when a user whose username is in the **sudoers** file but who does not have permission to run commands on the local host runs sudo. The default is off.

mail_no_perms Sends email to the **mailto** user when a user whose username is in the **sudoers** file but who does not have permission to run the requested command runs sudo. The default is off.

mail_no_user Sends email to the **mailto** user when a user whose username is not in the **sudoers** file runs sudo. The default is on.

passwd_tries=*num*
The ***num*** is the number of times the user can enter an incorrect password in response to the sudo password prompt before sudo quits. The default is 3. See also **insults** and **lecture**.

rootpw Causes sudo to accept only the **root** password in response to its prompt. Because sudo issues the same prompt whether it is asking for your password or the **root** password, turning this flag on may confuse users. ***Do not turn on this flag if you have not unlocked the root account*** (page 415) as you will not be able to use sudo. To fix this problem, bring the system up in recovery mode (page 428) and turn off (remove) this flag. The default is off, causing sudo to prompt for the password of the user running sudo. See the adjacent tip.

Using the root password in place of your password

tip If you have set up a **root** password (page 415), you can cause graphical programs that require a password to require the **root** password in place of the password of the user who is running the program by turning on **rootpw**. The programs will continue to ask for *your* password, but will accept only the **root** password. Making this change causes an Ubuntu system to use the **root** password in a manner similar to the way some other distributions use this password.

shell_noargs Causes sudo, when called without any arguments, to spawn a **root** shell without changing the environment. The default is off. This option is the same as the sudo **–s** option.

timestamp_timeout=*mins*
The ***mins*** is the number of minutes that the sudo timestamp (page 407) is valid. The default is 15; set ***mins*** to **–1** to cause the timestamp to be valid forever.

umask=*val* The ***val*** is the umask (page 442) that sudo uses to run the command that the user specifies. Set ***val*** to **0777** to preserve the user's umask value. The default is 0022.

Unlocking the root Account (Assigning a Password to root)

Except for a few instances, there is no need to unlock the **root** account on an Ubuntu system and Ubuntu suggests that you do not do so. The following command unlocks the **root** account by assigning a password to it:

```
$ sudo passwd root
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Relocking the **root** account If you decide you want to lock the **root** account after unlocking it, give the command **sudo passwd –l root**. You can unlock it again with the preceding command.

Securing a Server

Two ways you can secure a server are by using TCP wrappers and by setting up a chroot jail. This section describes both techniques.

TCP Wrappers: Secure a Server (hosts.allow and hosts.deny)

Follow these guidelines when you open a local system to access from remote systems:

- Open the local system only to systems you want to allow to access it.
- Allow each remote system to access only the data you want it to access.
- Allow each remote system to access data only in the appropriate manner (readonly, read/write, write only).

libwrap As part of the client/server model, TCP wrappers, which can be used for any daemon that is linked against **libwrap**, rely on the **/etc/hosts.allow** and **/etc/hosts.deny** files as the basis of a simple access control language (ACL). This access control language defines rules that selectively allow clients to access server daemons on a local system based on the client's address and the daemon the client tries to access. The output of ldd shows that one of the shared library dependencies of **sshd** is **libwrap**:

```
$ ldd /usr/sbin/sshd | grep libwrap
        libwrap.so.0 => /lib/libwrap.so.0 (0xb7ec7000)
```

hosts.allow and **hosts.deny**

Each line in the **hosts.allow** and **hosts.deny** files has the following format:

daemon_list : client_list [: command]

where ***daemon_list*** is a comma-separated list of one or more server daemons (such as **portmap**, **vsftpd**, or **sshd**), ***client_list*** is a comma-separated list of one or more clients (see Table 11-2, "Specifying a client," on page 444), and the optional ***command*** is the command that is executed when a client from ***client_list*** tries to access a server daemon from ***daemon_list***.

When a client requests a connection to a server, the **hosts.allow** and **hosts.deny** files on the server system are consulted as follows until a match is found:

1. If the daemon/client pair matches a line in **hosts.allow**, access is granted.
2. If the daemon/client pair matches a line in **hosts.deny**, access is denied.
3. If there is no match in the **hosts.allow** or **hosts.deny** file, access is granted.

The first match determines whether the client is allowed to access the server. When either **hosts.allow** or **hosts.deny** does not exist, it is as though that file was empty. Although it is not recommended, you can allow access to all daemons for all clients by removing both files.

Examples

For a more secure system, put the following line in **hosts.deny** to block all access:

```
$ cat /etc/hosts.deny
...
ALL : ALL : echo '%c tried to connect to %d and was blocked' >> /var/log/tcpwrappers.log
```

This line prevents any client from connecting to any service, unless specifically permitted to do so in **hosts.allow**. When this rule is matched, it adds a line to the file named **/var/log/tcpwrappers.log**. The **%c** expands to client information and the **%d** expands to the name of the daemon the client attempted to connect to.

With the preceding **hosts.deny** file in place, you can include lines in **hosts.allow** that explicitly allow access to certain services and systems. For example, the following **hosts.allow** file allows anyone to connect to the OpenSSH daemon (ssh, scp, sftp) but allows telnet connections only from the same network as the local system and users on the 192.168. subnet:

```
$ cat /etc/hosts.allow
sshd: ALL
in.telnet: LOCAL
in.telnet: 192.168.* 127.0.0.1
...
```

The first line allows connection from any system (ALL) to **sshd**. The second line allows connection from any system in the same domain as the server (LOCAL). The third line matches any system whose IP address starts **192.168.** and the local system.

Setting Up a chroot Jail

On early UNIX systems, the root directory was a fixed point in the filesystem. On modern UNIX variants, including Linux, you can define the root directory on a per-process basis. The chroot utility allows you to run a process with a root directory other than /.

The root directory appears at the top of the directory hierarchy and has no parent. Thus a process cannot access files above the root directory because none exists. If, for example, you run a program (process) and specify its root directory as **/tmp/jail**, the program would have no concept of any files in **/tmp** or above: **jail** is the program's root directory and is labeled **/** (not **jail**).

By creating an artificial root directory, frequently called a (chroot) jail, you prevent a program from accessing, executing, or modifying—possibly maliciously—files outside the directory hierarchy starting at its root. You must set up a chroot jail properly to increase security: If you do not set up a chroot jail correctly, you can make it easier for a malicious user to gain access to a system than if there were no chroot jail.

Using chroot

Creating a chroot jail is simple: Working with **root** privileges, give the command **/usr/sbin/chroot *directory***. The ***directory*** becomes the root directory and the process attempts to run the default shell. The following command sets up a chroot jail in the (existing) **/tmp/jail** directory:

```
$ sudo /usr/sbin/chroot /tmp/jail
/usr/sbin/chroot: cannot run command '/bin/bash': No such file or directory
```

This example sets up a chroot jail, but when it attempts to run the bash shell, it fails. Once the jail is set up, the directory that was named **jail** takes on the name of the root directory, **/**. As a consequence, chroot cannot find the file identified by the pathname **/bin/bash**. In this situation the chroot jail works correctly but is not useful.

Getting a chroot jail to work the way you want is more complicated. To have the preceding example run bash in a chroot jail, create a **bin** directory in **jail** (**/tmp/jail/bin**) and copy **/bin/bash** to this directory. Because the bash binary is dynamically linked to shared libraries, you need to copy these libraries into **jail** as well. The libraries go in **lib**.

The next example creates the necessary directories, copies **bash**, uses ldd to display the shared library dependencies of bash, and copies the necessary libraries to **lib**. The **linux-gate.so.1** file is a dynamically shared object (DSO) provided by the kernel to speed system calls; you do not need to copy it.

```
$ pwd
/tmp/jail
$ mkdir bin lib
$ cp /bin/bash bin
$ ldd bin/bash
        linux-gate.so.1 =>  (0xffffe000)
        libncurses.so.5 => /lib/libncurses.so.5 (0xb7f44000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7f40000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7dff000)
        /lib/ld-linux.so.2 (0xb7f96000)
```

```
$ cp /lib/{libncurses.so.5,ld-linux.so.2} lib
$ cp /lib/tls/i686/cmov/{libdl.so.2,libc.so.6} lib
```

Now start the chroot jail again. Although all the setup can be done by an ordinary user, you must be working with **root** privileges to run chroot:

```
$ sudo /usr/sbin/chroot /tmp/jail
bash-3.2# pwd
/
bash-3.2# ls
bash: ls: command not found
bash-3.2# exit
exit
$
```

This time chroot finds and starts **bash**, which displays its default prompt (**bash-3.2#**). The pwd command works because it is a shell builtin (page 247). However, bash cannot find the ls utility because it is not in the chroot jail. You can copy **/bin/ls** and its libraries into the jail if you want users in the jail to be able to use ls. An **exit** command allows you to escape from the jail.

If you provide chroot with a second argument, it takes that argument as the name of the program to run inside the jail. The following command is equivalent to the preceding one:

```
$ sudo /usr/sbin/chroot /home/sam/jail /bin/bash
```

To set up a useful chroot jail, first determine which utilities the users of the chroot jail need. Then copy the appropriate binaries and their libraries into the jail. Alternatively, you can build static copies of the binaries and put them in the jail without installing separate libraries. (The statically linked binaries are considerably larger than their dynamic counterparts. The base system with bash and the core utilities exceeds 50 megabytes.) You can find the source code for most common utilities in the **bash** and **coreutils** source packages.

The chroot utility fails unless you run it with **root** privileges. The preceding examples used sudo to gain these privileges. The result of running chroot with **root** privileges is a **root** shell (a shell with **root** privileges) running inside a chroot jail. Because a user with **root** privileges can break out of a chroot jail, it is imperative that you run a program in the chroot jail with reduced privileges (i.e., privileges other than those of **root**).

There are several ways to reduce the privileges of a user. For example, you can put su or sudo in the jail and then start a shell or a daemon inside the jail, using one of these programs to reduce the privileges of the user working in the jail. A command such as the following starts a shell with reduced privileges inside the jail:

```
$ sudo /usr/sbin/chroot jailpath /usr/bin/sudo -u user /bin/bash &
```

where *jailpath* is the pathname of the jail directory, and *user* is the username under whose privileges the shell runs. The problem with this scenario is that sudo and su as compiled for Ubuntu, call PAM. To run one of these utilities you need to put all of PAM, including its libraries and configuration files, in the jail, along with sudo (or su) and the **/etc/passwd** file. Alternatively, you can recompile su or sudo.

The source code calls PAM, however, so you would need to modify the source so it does not call PAM. Either one of these techniques is time-consuming and introduces complexities that can lead to an insecure jail.

The following C program[1] runs a program with reduced privileges in a chroot jail. Because this program obtains the UID and GID of the user you specify on the command line before calling **chroot**(), you do not need to put **/etc/passwd** in the jail. The program reduces the privileges of the specified program to those of the specified user. This program is presented as a simple solution to the preceding issues so you can experiment with a chroot jail and better understand how it works.

```
$ cat uchroot.c

/* See svn.gna.org/viewcvs/etoile/trunk/Etoile/LiveCD/uchroot.c for terms of use. */

#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>

int main(int argc, char * argv[])
{
    if(argc < 4)
    {
        printf("Usage: %s {username} {directory} {program} [arguments]\n", argv[0]);
        return 1;
    }
    /* Parse arguments */
    struct passwd * pass = getpwnam(argv[1]);
    if(pass == NULL)
    {
        printf("Unknown user %s\n", argv[1]);
        return 2;
    }
    /* Set the required UID */
    chdir(argv[2]);
    if(chroot(argv[2])
        ||
        setgid(pass->pw_gid)
        ||
        setuid(pass->pw_uid))
    {
        printf("%s must be run as root.  Current uid=%d, euid=%d\n",
                argv[0],
                (int)getuid(),
                (int)geteuid()
                );
        return 3;
    }
    char buf[100];
    return execv(argv[3], argv + 3);
}
```

1. Thanks to David Chisnall and the Étoilé Project (etoileos.com) for the **uchroot.c** program.

The first of the following commands compiles **uchroot.c**, creating an executable file named uchroot. Subsequent commands move uchroot to **/usr/local/bin** and give it appropriate ownership.

```
$ cc -o uchroot uchroot.c
$ sudo mv uchroot /usr/local/bin
$ sudo chown root:root /usr/local/bin/uchroot
$ ls -l /usr/local/bin/uchroot
-rwxr-xr-x 1 root root 7922 Jul 17 08:26 /usr/local/bin/uchroot
```

Using the setup from earlier in this section, give the following command to run a shell with the privileges of the user **sam** inside a chroot jail:

```
$ sudo /usr/local/bin/uchroot sam /tmp/jail /bin/bash
```

Keeping multiple chroot jails

tip If you plan to deploy multiple chroot jails, it is a good idea to keep a clean copy of the **bin** and **lib** directories somewhere other than one of the active jails.

RUNNING A SERVICE IN A chroot JAIL

Running a shell inside a jail has limited usefulness. Instead, you are more likely to want to run a specific service inside the jail. To run a service inside a jail, make sure all files needed by that service are inside the jail. Using uchroot, the format of a command to start a service in a chroot jail is

```
$ sudo /usr/local/bin/uchroot user jailpath daemonname
```

where *jailpath* is the pathname of the jail directory, *user* is the username that runs the daemon, and *daemonname* is the pathname (inside the jail) of the daemon that provides the service.

Some servers are already set up to take advantage of chroot jails. You can set up DNS so that **named** runs in a jail (page 808), for example, and the **vsftpd** FTP server can automatically start chroot jails for clients (page 667).

SECURITY CONSIDERATIONS

Some services need to be run by a user/process with **root** privileges but release their **root** privileges once started (Apache, Procmail, and **vsftpd** are examples). If you are running such a service, you do not need to use uchroot or put su or sudo inside the jail.

A process run with **root** privileges can potentially escape from a chroot jail. For this reason, always reduce privileges before starting a program running inside the jail. Also, be careful about which setuid (page 204) binaries you allow inside a jail—a security hole in one of them could compromise the security of the jail. In addition, make sure the user cannot access executable files that he uploads to the jail.

DHCP: Configures Network Interfaces

Instead of storing network configuration information in local files on each system, DHCP (Dynamic Host Configuration Protocol) enables client systems to retrieve network configuration information from a DHCP server each time they connect to the network. A DHCP server assigns IP addresses from a pool of addresses to clients as needed. Assigned addresses are typically temporary but need not be.

This technique has several advantages over storing network configuration information in local files:

- A new user can set up an Internet connection without having to deal with IP addresses, netmasks, DNS addresses, and other technical details. An experienced user can set up a connection more quickly.
- DHCP facilitates assignment and management of IP addresses and related network information by centralizing the process on a server. A system administrator can configure new systems, including laptops that connect to the network from different locations, to use DHCP; DHCP then assigns IP addresses only when each system connects to the network. The pool of IP addresses is managed as a group on the DHCP server.
- IP addresses can be used by more than one system, reducing the total number of IP addresses needed. This conservation of addresses is important because the Internet is quickly running out of IPv4 addresses. Although a particular IP address can be used by only one system at a time, many end-user systems require addresses only occasionally, when they connect to the Internet. By reusing IP addresses, DHCP has lengthened the life of the IPv4 protocol. DHCP applies to IPv4 only, as IPv6 (page 371) forces systems to configure their IP addresses automatically (called autoconfiguration) when they connect to a network.

DHCP is particularly useful for an administrator who is responsible for maintaining a large number of systems because individual systems no longer need to store unique configuration information. With DHCP, the administrator can set up a master system and deploy new systems with a copy of the master's hard disk. In educational establishments and other open-access facilities, the hard disk image may be stored on a shared drive, with each workstation automatically restoring itself to pristine condition at the end of each day.

More Information

Web www.dhcp.org
DHCP FAQ: www.dhcp-handbook.com/dhcp_faq.html

HOWTO *DHCP Mini HOWTO*

How DHCP Works

Using dhclient, the client contacts the server daemon, **dhcpd**, to obtain the IP address, netmask, broadcast address, nameserver address, and other networking parameters. In turn, the server provides a *lease* on the IP address to the client. The client can request the specific terms of the lease, including its duration; the server can limit these terms. While connected to the network, a client typically requests extensions of its lease as necessary so its IP address remains the same. This lease may expire once the client is disconnected from the network, with the server giving the client a new IP address when it requests a new lease. You can also set up a DHCP server to provide static IP addresses for specific clients (refer to "Static Versus Dynamic IP Addresses" on page 366). DHCP is broadcast based, so both client and server must be on the same subnet (page 369).

When you install Ubuntu, the system runs a DHCP client, connects to a DHCP server if it can find one, and configures its network interface. You can use firestarter (page 824) to configure and run a DHCP server.

DHCP CLIENT

A DHCP client requests network configuration parameters from the DHCP server and uses those parameters to configure its network interface.

PREREQUISITES

Make sure the following package is installed:

- **dhcp3-client**

dhclient: THE DHCP CLIENT

When a DHCP client system connects to the network, dhclient requests a lease from the DHCP server and configures the client's network interface(s). Once a DHCP client has requested and established a lease, it stores the lease information in a file named **dhclient.*interface*.leases**, which is stored in **/var/lib/dhcp3**. The ***interface*** is the name of the interface that the client uses, such as **eth0**. The system uses this information to reestablish a lease when either the server or the client needs to reboot. You need to change the default DHCP client configuration file, **/etc/dhcp3/dhclient.conf**, only for custom configurations.

The following **/etc/dhcp3/dhclient.conf** file specifies a single interface, **eth0**:

```
$ cat /etc/dhcp3/dhclient.conf
interface "eth0"
{
send dhcp-client-identifier 1:xx:xx:xx:xx:xx:xx;
send dhcp-lease-time 86400;
}
```

In the preceding file, the 1 in the **dhcp-client-identifier** specifies an Ethernet network and **xx:xx:xx:xx:xx:xx** is the *MAC address* (page 1118) of the device controlling that interface. See page 457 for instructions on how to determine the MAC address of a device. The **dhcp-lease-time** is the duration, in seconds, of the lease on the IP address. While the client is connected to the network, dhclient automatically renews the lease each time half of the lease time is up. The lease time of 86,400 seconds (or one day) is a reasonable choice for a workstation.

DHCP Server

A DHCP server maintains a list of IP addresses and other configuration parameters. Clients request network configuration parameters from the server.

Prerequisites

Install the following package:

- **dhcp3-server**

dhcp3-server init script When you install the **dhcpd3-server** package, the **dpkg postinst** script attempts to start the **dhcpd3** daemon and fails because **dhcpd3** is not configured—see **/var/log/syslog** for details. After you configure **dhcpd3**, call the **dhcp3-server** init script to restart the **dhcpd3** daemon:

```
$ sudo /etc/init.d/dhcp3-server restart
```

dhcpd: The DHCP Daemon

A simple DCHP server (**dhcpd**) allows you to add clients to a network without maintaining a list of assigned IP addresses. A simple network, such as a home LAN sharing an Internet connection, can use DHCP to assign a dynamic IP address to almost all nodes. The exceptions are servers and routers, which must be at known network locations to be able to receive connections. If servers and routers are configured without DHCP, you can specify a simple DHCP server configuration in **/etc/dhcp3/dhcpd.conf**:

```
$ cat /etc/dhcp3/dhcpd.conf
default-lease-time 600;
max-lease-time 86400;

option subnet-mask 255.255.255.0;
option broadcast-address 192.168.1.255;
option routers 192.168.1.1;
option domain-name-servers 192.168.1.1;
option domain-name "example.com";

subnet 192.168.1.0 netmask 255.255.255.0 {
   range 192.168.1.2 192.168.1.200;
}
```

The **/etc/default/dhcp3-server** file specifies the interfaces that **dhcpd** serves requests on. By default, **dhcpd** uses **eth0**. To use another interface or to use more than one interface, set the **INTERFACES** variable in this file to a SPACE-separated list of the interfaces you want to use; enclose the list within quotation marks.

The preceding configuration file specifies a LAN where both the router and DNS server are located on **192.168.1.1**. The **default-lease-time** specifies the number of seconds the dynamic IP lease will remain valid if the client does not specify a duration. The **max-lease-time** is the maximum time allowed for a lease.

The information in the **option** lines is sent to each client when it connects. The names following the word **option** specify what the following argument represents. For example, the **option broadcast-address** line specifies the broadcast address of the network. The **routers** and **domain-name-servers** options allow multiple values separated by commas.

The **subnet** section includes a **range** line that specifies the range of IP addresses the DHCP server can assign. If case of multiple subnets, you can define options, such as **subnet-mask**, inside the **subnet** section. Options defined outside all **subnet** sections are global and apply to all subnets.

The preceding configuration file assigns addresses in the range from 192.168.1.2 to 192.168.1.200. The DHCP server starts at the bottom of this range and attempts to assign a new IP address to each new client. Once the DHCP server reaches the top of the range, it starts reassigning IP addresses that have been used in the past but are not currently in use. If you have fewer systems than IP addresses, the IP address of each system should remain fairly constant. Two systems cannot use the same IP address at the same time.

Once you have configured a DHCP server, restart it using the **dhcpd** init script (page 456). When the server is running, clients configured to obtain an IP address from the server using DHCP should be able to do so.

Static IP Addresses

As mentioned earlier, routers and servers typically require static IP addresses. Although you can manually configure IP addresses for these systems, it may be more convenient to have the DHCP server provide them with static IP addresses.

When a system that requires a specific static IP address connects to the network and contacts the DHCP server, the server needs a way to identify the system so it can assign the proper IP address to that system. The DHCP server uses the *MAC address* (page 1118) of the system's Ethernet card (NIC) as an identifier. When you set up the server, you must know the MAC address of each system that requires a static IP address.

Determining a MAC address

The ifconfig utility displays the MAC addresses of the Ethernet cards in a system. In the following example, the MAC addresses are the colon-separated series of hexadecimal number pairs following **HWaddr**:

```
$ ifconfig | grep -i hwaddr
eth0      Link encap:Ethernet  HWaddr BA:DF:00:DF:C0:FF
eth1      Link encap:Ethernet  HWaddr 00:02:B3:41:35:98
```

Run ifconfig on each system that requires a static IP address. Once you have determined the MAC addresses of these systems, you can add a **host** section to the **/etc/dhcp3/dhcpd.conf** file for each one, instructing the DHCP server to assign a specific address to that system. The following **host** section assigns the address **192.168.1.1** to the system with the MAC address of **BA:DF:00:DF:C0:FF**:

```
$ cat /etc/dhcp3/dhcpd.conf
...
host router {
   hardware ethernet BA:DF:00:DF:C0:FF;
   fixed-address 192.168.1.1;
   option host-name router;
}
```

The name following **host** is used internally by **dhcpd**. The name specified after **option host-name** is passed to the client and can be a hostname or an FQDN. After making changes to **dhcpd.conf**, restart **dhcpd** using the **dhcpd** init script (page 456).

nsswitch.conf: Which Service to Look at First

With the advent of NIS and DNS, finding user and system information was no longer a simple matter of searching a local file. Where once you looked in **/etc/passwd** to get user information and in **/etc/hosts** to find system address information, you can now use several methods to obtain this type of information. The **/etc/nsswitch.conf** (name service switch configuration) file specifies which methods to use and the order in which to use them when looking for a certain type of information. You can also specify which action the system should take based on whether a method succeeds or fails.

Format Each line in **nsswitch.conf** specifies how to search for a piece of information, such as a user's password. A line in **nsswitch.conf** has the following format:

```
info:          method [[action]] [method [[action]]...]
```

where ***info*** specifies the type of information the line describes, ***method*** is the method used to find the information, and ***action*** is the response to the return status of the preceding ***method***. The action is enclosed within square brackets.

How nsswitch.conf Works

When called upon to supply information that **nsswitch.conf** describes, the system examines the line with the appropriate ***info*** field. It uses the methods specified on the line, starting with the method on the left. By default, when it finds the desired information, the system stops searching. Without an ***action*** specification, when a method fails to return a result, the system tries the next action. It is possible for the search to end without finding the requested information.

Information

The **nsswitch.conf** file commonly controls searches for usernames, passwords, host IP addresses, and group information. The following list describes most of the types of information (***info*** in the syntax given earlier) that **nsswitch.conf** controls searches for.

automount	Automount (**/etc/auto.master** and **/etc/auto.misc**, page 756)
bootparam	Diskless and other booting options (See the **bootparam** man page.)
ethers	MAC address (page 1118)
group	Groups of users (**/etc/group**, page 474)
hosts	System information (**/etc/hosts**, page 475)
networks	Network information (**/etc/networks**)
passwd	User information (**/etc/passwd**, page 476)
protocols	Protocol information (**/etc/protocols**, page 477)
publickey	Used for NFS running in secure mode
rpc	RPC names and numbers (**/etc/rpc**, page 478)
services	Services information (**/etc/services**, page 479)
shadow	Shadow password information (**/etc/shadow**, page 479)

Methods

Following is a list of the types of information that **nsswitch.conf** controls searches for (***method*** in the format above). For each type of information, you can specify one or more of the following methods:[2]

files	Searches local files such as **/etc/passwd** and **/etc/hosts**
nis	Searches the NIS database; **yp** is an alias for **nis**
dns	Queries the DNS (**hosts** queries only)
compat	**±** syntax in **passwd**, **group**, and **shadow** files (page 460)

Search Order

The information provided by two or more methods may overlap: For example, both **files** and **nis** may provide password information for the same user. With overlapping information, you need to consider which method you want to be authoritative (take precedence); place that method at the left of the list of methods.

The default **nsswitch.conf** file lists methods without actions, assuming no overlap (which is normal). In this case, the order is not critical: When one method fails, the system goes to the next one and all that is lost is a little time. Order becomes critical when you use actions between methods or when overlapping entries differ.

The first of the following lines from **nsswitch.conf** causes the system to search for password information in **/etc/passwd** and, if that fails, to use NIS to find the information. If the user you are looking for is listed in both places, the information in the local file is used and is considered authoritative. The second line uses NIS to find an IP address given a hostname; if that fails, it searches **/etc/hosts;** if that fails, it checks with DNS to find the information.

```
passwd        files nis
hosts         nis files dns
```

Action Items

Each method can optionally be followed by an action item that specifies what to do if the method succeeds or fails. An action item has the following format:

*[[!]STATUS=**action**]*

where the opening and closing square brackets are part of the format and do not indicate that the contents are optional; ***STATUS*** (uppercase by convention) is the status being tested for; and ***action*** is the action to be taken if ***STATUS*** matches the status returned by the preceding method. The leading exclamation point (**!**) is optional and negates the status.

2. There are other, less commonly used methods. See the default **/etc/nsswitch.conf** file and the **nsswitch.conf** man page for more information. Although NIS+ belongs in this list, it is not implemented as a Linux server and is not discussed in this book.

STATUS Values for ***STATUS*** are

NOTFOUND The method worked but the value being searched for was not found. The default action is **continue**.

SUCCESS The method worked and the value being searched for was found; no error was returned. The default action is **return**.

UNAVAIL The method failed because it is permanently unavailable. For example, the required file may not be accessible or the required server may be down. The default action is **continue**.

TRYAGAIN The method failed because it was temporarily unavailable. For example, a file may be locked or a server overloaded. The default action is **continue**.

action Values for ***action*** are

return Returns to the calling routine with or without a value.

continue Continues with the next method. Any returned value is overwritten by a value found by a subsequent method.

Example The following line from **nsswitch.conf** causes the system first to use DNS to search for the IP address of a given host. The action item following the DNS method tests whether the status returned by the method is not (**!**) UNAVAIL.

```
hosts          dns [!UNAVAIL=return] files
```

The system takes the action associated with the ***STATUS*** (**return**) if the DNS method does not return UNAVAIL (**!UNAVAIL**)—that is, if DNS returns SUCCESS, NOTFOUND, or TRYAGAIN. The result is that the following method (**files**) is used only when the DNS server is unavailable. If the DNS server is *not un*available (read the two negatives as "is available"), the search returns the domain name or reports that the domain name was not found. The search uses the **files** method (checks the local **/etc/hosts** file) only if the server is not available.

compat Method: ± in passwd, group, and shadow Files

You can put special codes in the **/etc/passwd**, **/etc/group**, and **/etc/shadow** files that cause the system, when you specify the **compat** method in **nsswitch.conf**, to combine and modify entries in the local files and the NIS maps.

A plus sign (**+**) at the beginning of a line in one of these files adds NIS information; a minus sign (**–**) removes information. For example, to use these codes in the **passwd** file, specify **passwd: compat** in **nsswitch.conf**. The system then goes through the **passwd** file in order, adding or removing the appropriate NIS entries when it reaches each line that starts with a **+** or **–**.

Although you can put a plus sign at the end of the **passwd** file, specify **passwd: compat** in **nsswitch.conf** to search the local **passwd** file, and then go through the NIS map, it is more efficient to put **passwd: file nis** in **nsswitch.conf** and not modify the **passwd** file.