CHAPTER 6

# Security

SECURITY IS DEFINITELY A HOT TOPIC both inside and outside the computer world. It can be difficult to distinguish legitimate threats from basic paranoia, but as anyone who has connected to a high-speed connection and monitored the logs knows, these days there are armies of servers out there trying to attack you.

Even though other operating systems and products seem to get the majority of the press for their security breaches, Ubuntu users aren't completely in the clear. Even though Ubuntu has good security out of the box, the moment you set up new services you risk opening holes to attack. This chapter discusses some common security practices and simple steps you can take to keep your Ubuntu server secure. Just in case you still get attacked, I also include a section at the end on how to respond to a security breach.

## General Security Principles

There is a saying in security circles: "Security is a process, not a product." What that means is that despite what your vendor might tell you, you can't solve all your security problems with some appliance or software. Instead, you find real security when you follow sound security principles and develop sound security procedures. While I cover some specific tools and options you can use to increase your system's security later in the chapter, there's no way I can discuss how to lock down every major service under Ubuntu. These principles, though, are something that you can apply no matter what software you might run:

- **Keep it simple.**
  Another saying you will hear in security circles is "Complexity is the enemy of security." The more complex a system, the more difficult it is to understand every part of it and the greater the likelihood that the security of some aspect of the system was overlooked. Whenever you design a system, try to keep the number of interoperating pieces as small as you can. Not only will it help with security, it will help with troubleshooting and overall administration as well.

- **Follow the principle of least privilege.**
  The principle of least privilege is the idea that programs and people should operate with the lowest possible level of power. This is the

concept behind the separation of root and the regular users. Since most daily tasks don't require full system privileges, give users fewer privileges. Programs like Apache and Postfix follow this principle; they use the root privileges only when they are absolutely necessary, and then the bulk of the work is done via child processes owned by a different user. When these practices are in place, and you do get attacked, the amount of damage an attacker can do is limited.

- **Provide layers of protection.**
  Some refer to this as "defense in depth." The best security occurs in layers. Slapping a firewall in front of your servers won't automatically make them secure, but it will help increase their security. Instead, you want multiple layers of defense, such as a firewall between you and the outside world, a local software firewall, strong passwords, and sudo roles.

- **Avoid security by obscurity.**
  On the surface it might seem as if moving the SSH server from port 22 to port 257 would add extra security. After all, no one will think to look for it there. Unfortunately, steps like this slow down, but don't stop, an attacker. The real danger of these sorts of security methods is that they create a false sense of security. This isn't to say that moving ports around and using other means of obscurity are completely bad, just that they should be recognized for what they are—things that only slow down attackers and that must be combined with other security procedures.

- **Keep on top of security patches.**
  You can have all sorts of security procedures in place and still be attacked if a vulnerability is found in a service and you fail to patch it. It's important to monitor security updates and use Ubuntu's package management to keep your systems up to date.

## AppArmor

The UNIX permissions model has long been used to lock down access to users and programs. Even though it works well, there are still areas where extra access control can come in handy. For instance, many services still run as the root user, and therefore if they are exploited, the attacker potentially can run commands throughout the rest of the system as the root user. There are a number of ways to combat this problem, including sandboxes, `chroot` jails, and so on, but Ubuntu has included a system called AppArmor, installed by default, that adds access control to specific system services.

AppArmor is based on the security principle of least privilege; that is, it attempts to restrict programs to the minimal set of permissions they need to function. It works through a series of rules assigned to particular programs. These rules define, for instance, which files or directories a program is allowed to read and write to or only read from. When an application that is being managed by AppArmor violates these access controls, AppArmor steps in and prevents it and logs the event. A number of services include AppArmor profiles that are enforced by default, and more are being added in each Ubuntu release. In addition to the default profiles, the universe repository has an `apparmor-profiles` package you can install to add more profiles for other services. Once you learn the syntax for AppArmor rules, you can even add your own profiles.

Probably the simplest way to see how AppArmor works is to use an example program. The BIND DNS server is one program that is automatically managed by AppArmor under Ubuntu, so first I install the BIND package with `sudo apt-get install bind9`. Once the package is installed, I can use the `aa-status` program to see that AppArmor is already managing it:

```
$ sudo aa-status
apparmor module is loaded.
5 profiles are loaded.
5 profiles are in enforce mode.
   /sbin/dhclient3
   /usr/lib/NetworkManager/nm-dhcp-client.action
   /usr/lib/connman/scripts/dhclient-script
   /usr/sbin/named
   /usr/sbin/tcpdump
0 profiles are in complain mode.
2 processes have profiles defined.
1 processes are in enforce mode :
   /usr/sbin/named (5020)
0 processes are in complain mode.
1 processes are unconfined but have a profile defined.
   /sbin/dhclient3 (607)
```

Here you can see that the /usr/sbin/named profile is loaded and in enforce mode, and that my currently running /usr/sbin/named process (PID 5020) is being managed by AppArmor.

## AppArmor Profiles

The AppArmor profiles are stored within /etc/apparmor.d/ and are named after the binary they manage. For instance, the profile for /usr/sbin/named is located at /etc/apparmor.d/usr.sbin.named. If you look at the contents of the file, you can get an idea of how AppArmor profiles work and what sort of protection they provide:

```
# vim:syntax=apparmor
# Last Modified: Fri Jun  1 16:43:22 2007
#include <tunables/global>
```

```
/usr/sbin/named {
  #include <abstractions/base>
  #include <abstractions/nameservice>

  capability net_bind_service,
  capability setgid,
  capability setuid,
  capability sys_chroot,

  # /etc/bind should be read-only for bind
  # /var/lib/bind is for dynamically updated zone (and journal) files.
  # /var/cache/bind is for slave/stub data, since we're not the origin
  #of it.
  # See /usr/share/doc/bind9/README.Debian.gz
  /etc/bind/** r,
  /var/lib/bind/** rw,
  /var/lib/bind/ rw,
  /var/cache/bind/** rw,
  /var/cache/bind/ rw,

  # some people like to put logs in /var/log/named/
  /var/log/named/** rw,

  # dnscvsutil package
  /var/lib/dnscvsutil/compiled/** rw,

  /proc/net/if_inet6 r,
  /usr/sbin/named mr,
  /var/run/bind/run/named.pid w,
  # support for resolvconf
  /var/run/bind/named.options r,
}
```

For instance, take a look at the following excerpt from that file:

```
  /etc/bind/** r,
  /var/lib/bind/** rw,
  /var/lib/bind/ rw,
  /var/cache/bind/** rw,
  /var/cache/bind/ rw,
```

The syntax is pretty straightforward for these files. First there is a file or directory path, followed by the permissions that are allowed. Globs are also allowed, so, for instance, `/etc/bind/**` applies to all of the files below the /etc/bind directory recursively. A single `*` would apply only to files within the current directory. In the case of that rule you can see that /usr/sbin/

named is allowed only to read files in that directory and not write there. This makes sense, since that directory contains only BIND configuration files—the `named` program should never need to write there. The second line in the excerpt allows `named` to read and write to files or directories under /var/lib/bind/. This also makes sense because BIND might (among other things) store slave zone files here, and since those files are written to every time the zone changes, `named` needs permission to write there.

## Enforce and Complain Modes

You might have noticed that the `aa-status` output mentions two modes: enforce and complain modes. In enforce mode, AppArmor actively blocks any attempts by a program to violate its profile. In complain mode, AppArmor simply logs the attempt but allows it to happen. The `aa-enforce` and `aa-complain` programs allow you to change a profile to be in enforce or complain mode, respectively. So if my /usr/sbin/named program did need to write to a file in /etc/bind or some other directory that wasn't allowed, I could either modify the AppArmor profile to allow it or I could set it to complain mode:

```
$ sudo aa-complain /usr/sbin/named
Setting /usr/sbin/named to complain mode
```

If later on I decided that I wanted the rule to be enforced again, I would use the `aa-enforce` command in the same way:

```
$ sudo aa-enforce /usr/sbin/named
Setting /usr/sbin/named to enforce mode
```

If I had decided to modify the default rule set at /etc/apparmor.d/usr.sbin.named, I would need to be sure to reload AppArmor so it would see the changes. You can run AppArmor's init script and pass it the `reload` option to accomplish this:

```
$ sudo /etc/init.d/apparmor reload
```

Be careful when you modify AppArmor rules. When you first start to modify rules, you might want to set that particular rule into complain mode and then monitor /var/log/syslog for any violations. For instance, if

/usr/sbin/named were in enforce mode and I had commented out the line in the /usr/sbin/named profile that granted read access to /etc/bind/**, then reloaded AppArmor and restarted BIND, not only would BIND not start (since it couldn't read its config files), I would get a nice log entry in /var/log/syslog from the kernel to report the denied attempt:

```
Jan  7 19:03:02 kickseed kernel: [ 2311.120236]
  audit(1231383782.081:3): type=1503 operation="inode_permission"
  requested_mask="::r" denied_mask="::r" name="/etc/bind/named.conf"
  pid=5225 profile="/usr/sbin/named" namespace="default"
```

### Ubuntu AppArmor Conventions

The following list details the common directories and files AppArmor uses, including where it stores configuration files and where it logs:

- **/etc/apparmor/**
  This directory contains the main configuration files for the AppArmor program, but note that it does *not* contain AppArmor rules.
- **/etc/apparmor.d/**
  You will find all of the AppArmor rules under this directory along with subdirectories that contain different sets of include files to which certain rule sets refer.
- **/etc/init.d/apparmor**
  This is the AppArmor init script. By default AppArmor is enabled.
- **/var/log/apparmor/**
  AppArmor stores its logs under this directory.
- **/var/log/syslog**
  When an AppArmor rule is violated in either enforce or complain mode, the kernel generates a log entry under the standard system log.

## SSH Security

If you are going to run services on your servers, these days it's a safe bet that one of them will be SSH. SSH provides a secure, encrypted channel between your desktop and a server so that you can run commands and manage the machine without having to physically be there with a keyboard and mouse. Even though SSH was designed with security at the forefront, poor management of the service can open you up to attack. In fact, one of the most common ways that Linux servers are attacked at the moment is via SSH brute-force attacks. I cover how to manage those attacks, but first I discuss a few other methods to enhance the security of SSH.

## sshd_config

The /etc/ssh/sshd_config file is where you will find all of the settings for the SSH server. The default Ubuntu sshd_config file is pretty secure out of the box, as it allows only SSH protocol 2, uses privilege separation, and allows authentication keys to be used. The only questionable setting is `PermitRootLogin yes`. This option allows the root user to log in via SSH. In a way this setting is useless on a default Ubuntu install, since the root account is disabled, but if you decide to enable the root account, you might want to set this option to `no` and run `sudo service ssh reload` to save the settings. This way you force users to log in with their regular accounts and `sudo` up to root, and you also prevent a user from being able to guess the root password and gain access.

## Key-Based Authentication

If there is a weak link in SSH security, password authentication would probably be it. I know plenty of people who have been hacked simply because of a weak user password. There are many brute-force SSH scripts active in the wild that constantly scan for new machines and run through a dictionary full of passwords until one works. I know of a honeypot server intentionally set with weak passwords that was hacked and used as part of a botnet within hours of showing up online.

The good news is that you don't need password authentication to log in to an SSH server. SSH supports key-based authentication. In this approach the user generates a public and private key. The public key is then placed in a special file on the remote server. When the user logs in, these keys are used to authenticate the user instead of a password. It's certainly more convenient to be able to log in to a machine without typing a password every time, although if you want an extra layer of security, you can set a passphrase on your keys as well.

It is relatively simple to set up key-based authentication. In the following example we have a user named ubuntu on desktop1 who wants to set up key authentication on server1. The first step is to use the `ssh-keygen` program to create an RSA public and private key on desktop1. At each prompt you can press Enter to accept the defaults.

```
ubuntu@desktop1:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Created directory '/home/ubuntu/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.
The key fingerprint is:
91:ae:0c:ff:16:a2:67:98:19:34:71:5b:71:e3:d2:2c ubuntu@ubuntu
```

The script creates the keys in the .ssh directory under your home directory, in this case /home/ubuntu/.ssh. The private key and public key are named id_rsa and id_rsa.pub respectively. It's very important (especially if you chose an empty passphrase) to keep the private key (id_rsa) safe! If anyone else gets access to this file, he or she can copy it and will be able to log in to any machines you have set up with this key.

Once you have created the keys, the next step is to copy the id_rsa.pub key to the server and then append it to the ~/.ssh/authorized_keys file. There are a number of ways you can do this. You could SSH into the remote machine, open ~/.ssh/authorized_keys with a text editor, and paste in the contents of id_rsa.pub, for instance. Here are two other ways to do this. The first way is simple to understand but takes multiple steps. The second method does the entire operation in one command. In method one I use the `scp` command to copy the id_rsa.pub file to the home directory on the remote server, then I append it to the ~/.ssh/authorized_keys file:

```
ubuntu@desktop1:~$ scp ~/.ssh/id_rsa.pub
ubuntu@server1:/home/ubuntu/id_rsa-desktop1.pub
ubuntu@desktop1:~$ ssh ubuntu@server1
ubuntu@server1:~$ mkdir ~/.ssh
ubuntu@server1:~$ chmod 700 ~/.ssh
ubuntu@server1:~$ cat ~/id_rsa-desktop1.pub >>
  ~/.ssh/authorized_keys
```

It turns out that you can skip all of the preceding steps with an included script named ssh-copy-id, which safely copies your local public key to the remote host:

```
ubuntu@desktop1:~$ ssh-copy-id ubuntu@server1
```

Once you have keys set up on a machine, you should be able to log in without a password prompt, unless you set a passphrase for your key, in which case you will need to type it. After your keys work, you might want to disable SSH password authentication altogether. Just make sure that your SSH keys work first or you could lock yourself out! To disable password authentication, edit /etc/ssh/sshd_config and locate the line that says

```
#PasswordAuthentication yes
```

Uncomment that line and set it to `no`:

```
PasswordAuthentication no
```

Finally, run `sudo service sshd reload` to load the new change.

## SSH Brute-Force Attacks

As mentioned earlier in this chapter, SSH brute-force attacks have become a very common threat to Linux servers. Even if your password is hard to guess, unless you impose strong password restrictions on the server, there's no way of knowing that every other user has a strong password. The best way to combat SSH brute-force attacks is to simply disable password authentication and use SSH keys. Unfortunately, that isn't an option for every administrator. If you must use password authentication, there is another way to protect against these attacks: a package named `denyhosts`.

The way that `denyhosts` works is to monitor for failed SSH logins. When a host attempts to log in either as a user that doesn't exist or too many times, that host is added to /etc/hosts.deny and blocked from future SSH access. Also, if a host tries to log in as a valid user but fails too many times, the host is blocked.

A number of administrators use this tool or tools like it to protect against brute-force attacks and like the results, but I find it hard to recommend. I mention it so that you know it is available, since you might disagree with my opinion. In case you do decide to deploy it, here are some things to watch out for:

- Any program that automatically modifies firewall (or TCP wrappers) rules is dangerous. If an attacker can detect that such a tool exists, he or she can remotely modify your firewall rules. What happens if the attacker can appear to come from a different host, such as your desktop, and lock you out?
- Set your thresholds carefully. Even with reasonably large thresholds, such as ten failed attempts for a valid user, you might still lock out valid users who forgot their password. I've even seen this happen with a user who set up keys and had a cron script log in and perform various tasks. When the server got overloaded and the SSH connections timed out, the failed SSH connections crossed the threshold and locked out the script.
- Set whitelists for trusted hosts. Be sure to add any hosts or networks that you can't risk being blocked into /etc/hosts.allow. Be sure to keep your whitelists up-to-date with new hosts or networks. Just keep in mind that if attackers do manage to hack into another machine on any of these networks, they will be able to attack these machines.
- Botnets know about `denyhosts` and can work around it. It's true that `denyhosts` makes a brute-force attack more difficult, but a large-enough botnet can work around this problem by having a particular host attack only a few times, or shift to a different host once the first is locked out.