Hardening Linux

JAMES TURNBULL

Apress®

Hardening Linux

Copyright © 2005 by James Turnbull

ISBN (pbk): 1-59059-444-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jim Sumser

Technical Reviewer: Judith Myerson

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Kylie Johnston

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks-Copony

Production Editor: Kelly Winquist

Compositor: Linda Weidemann

Proofreader: Lori Bring

Indexer: Kevin Broccoli

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 5

Understanding Logging and Log Monitoring

One of the key facets of maintaining a secure and hardened environment is knowing what is going on in that environment. You can achieve this through your careful and systematic use of logs. Most systems and most applications, such as Apache or Postfix, come with default logging options. This is usually enough for you to diagnose problems or determine the ongoing operational status of your system and applications. When it comes to security, you need to delve a bit deeper into the logging world to gain a fuller and clearer understanding of what is going on with your systems and applications and thus identify potential threats and attacks.

Logs are also key targets for someone who wants to penetrate your system—for two reasons. The first reason is that your logs often contain vital clues about your systems and their security. Attackers often target your logs in an attempt to discover more about your systems. As a result, you need to ensure your log files and `/var/log` directory are secure from intruders and that log files are available only to authorized users. Additionally, if you transmit your logs over your network to a centralized log server, you need to ensure no one can intercept or divert your logs.

The second reason is that if attackers do penetrate your systems, the last thing they want to happen is that you detect them and shut them out of your system. One of the easiest ways to prevent you from seeing their activities is to whitewash your logs so that you see only what you expect to see. Early detection of intrusion using log monitoring and analysis allows you to spot them before they blind you.

I will cover a few topics in this chapter, including the basic `syslog` daemon and one of its successors, the considerably more powerful and more secure syslog-NG. I will also cover the Simple Event Correlation (SEC) tool, which can assist you in highlighting events in your logs. I will also discuss logging to databases and secure ways to deliver your logs to a centralized location for review and analysis.

Syslog

Syslog is the ubiquitous Unix tool for logging. It is present on all flavors of Linux and indeed on almost all flavors of Unix. You can add it using third-party tools to Windows systems, and most network devices such as firewalls, routers, and switches are capable of generating Syslog messages. This results in the Syslog format being the closest thing to a universal logging standard that exists.

■**Tip** RFC 3164 documents the core Syslog functionality.[1]

I will cover the Syslog tool because not only is it present on all distributions of Linux, but it also lays down the groundwork for understanding how logging works on Linux systems. The `syslog` utility is designed to generate, process, and store meaningful event notification messages that provide the information required for administrators to manage their systems. Syslog is both a series of programs and libraries, including `syslogd`, the `syslog` daemon, and a communications protocol.

The most frequently used component of `syslog` is the `syslogd` daemon. This daemon runs on your system from startup and listens for messages from your operating system and applications. It is important to note that the `syslogd` daemon is a passive tool. It merely waits for input from devices or programs. It does not go out and actively gather messages.

■**Note** Syslog also uses another daemon, `klogd`. The Kernel Log Daemon specifically collects messages from the kernel. This daemon is present on all Linux systems and starts by default when your system starts. I will talk about that in some more detail in the "syslog-NG" section.

The next major portion of the `syslog` tools is the `syslog` communications protocol. With this protocol it is possible to send your log data across a network to a remote system where another `syslog` daemon can collect and centralize your logs. As presented in Figure 5-1, you can see how this is done.

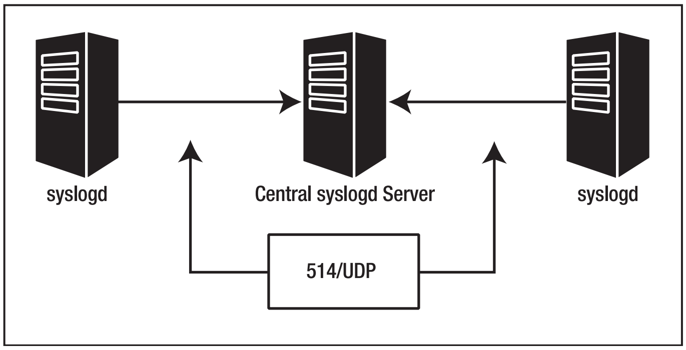

Figure 5-1. *Remote* `syslogd` *logging*

But my recommendation, though, is that if you have more than one system and either have or want to introduce a centralized logging regime, then do not use `syslog`. I make this

1. See `http://www.faqs.org/rfcs/rfc3164.html`. Also, some interesting work is happening on a new RFC for Syslog; you can find it at `http://www.syslog.cc/ietf/protocol.html`.

recommendation as a result of `syslog`'s reliance on the User Datagram Protocol (UDP) to transmit information. UDP has three major limitations.

- On a congested network, packets are frequently lost.
- The protocol is not fully secure.
- You are open to replay and Denial of Service (DoS) attacks.

If you are serious about secure logging, I recommend the syslog-NG package, which I will discuss later in the "syslog-NG" section.

The `syslog` communications protocol allows you to send `syslog` messages across your network via UDP to a centralized log server running `syslogd`. The `syslogd` daemon usually starts by default when your system boots. It is configured to collect a great deal of information about the ongoing activities of your system "out of the box."

■ **Tip** Syslog traffic is usually transmitted via UDP on port 514.

Configuring Syslog

The `syslog` daemon is controlled by a configuration file located in `/etc` called `syslog.conf`. This file contains the information about what devices and programs `syslogd` is listening for (filtered by facility and priority), where that information is to be stored, or what actions are to be taken when that information is received. You can see in Listing 5-1 that each line is structured into two fields, a selector field and an action field, which are separated by spaces or a tab.

Listing 5-1. `syslog.conf` *Syntax*

```
mail.info     /var/log/maillog
```

This example shows a facility and priority selector, `mail.info`, together with the action `/var/log/maillog`. The facility represented here is `mail`, and the priority is `info`. Overall the line in Listing 5-1 indicates that all messages generated by the `mail` facility with a priority of `info` or higher will be logged to the file `/var/log/maillog`. Let's examine now what facilities, priorities, and actions are available to you on a Linux system.

Facilities

The facility identifies the source of the `syslog` message. Some operating-system functions and daemons and other common application daemons have standard facilities attached to them. The `mail` and `kern` facilities are two good examples. The first example is the facility for all mail-related event notification messages. The second example is the facility for all kernel-related messages. Other processes and daemons that do not have a prespecified facility are able to log to the `local` facilities, ranging from `local0` to `local7`. For example, I use `local4` as the facility for all messages on my Cisco devices. Table 5-1 lists all Syslog facilities.

Table 5-1. *Syslog Facilities on Linux*

Facility	Purpose
`auth`	Security-related messages
`auth-priv`	Access control messages
`cron`	`cron`-related messages
`daemon`	System daemons and process messages
`kern`	Kernel messages
`local0–local7`	Reserved for locally defined messages
`lpr`	Spooling subsystem messages
`mail`	Mail-related messages
`mark`	Time-stamped messages generated by `syslogd`
`news`	Network News–related messages (for example, Usenet)
`syslog`	Syslog-related messages
`user`	The default facility when no facility is specified
`uucp`	UUCP-related messages

■ **Tip** On Mandrake and Red Hat systems `local7` points at `/var/log/boot.log`, which contains all the messages generated during the boot of your system.

The `mark` facility is a special case. It is used by the time-stamped messages that `syslogd` generates when you use the `-m` (minutes) flag. You can find more on this in the "Starting syslogd and Its Options" section.

You have two special facilities: `*`, which indicates all facilities, and `none`, which negates a facility selection. As shown in the following example, you can use these two facilities as wildcard selectors. See Listing 5-2.

Listing 5-2. `syslog.conf` * *Wildcard Selector*

```
*.emerg     /dev/console
```

This will send all messages of the `emerg` priority, regardless of facility, to the console. You can also use the `none` wildcard selector to not select messages from a particular facility.

```
kern.none     /var/log/messages
```

This will tell `syslog` to not log any kernel messages to the file `/var/log/messages`.

Priorities

Priorities are organized in an escalating scale of importance. They are `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert`, and `emerg`. Each priority selector applies to the priority stated and all higher priorities, so `uucp.err` indicates all `uucp` facility messages of `err`, `crit`, `alert`, and `emerg` priorities.

As with facilities, you can use the wildcard selectors `*` and `none`. Additionally, you can use two other modifiers: `=` and `!`. The `=` modifier indicates that only one priority is selected; for example, `cron.=crit` indicates that only `cron` facility messages of `crit` priority are to be selected. The `!` modifier has a negative effect; for example, `cron.!crit` selects all `cron` facility messages except those of `crit` *or higher priority*. You can also combine the two modifiers to create the opposite effect of the `=` modifier so that `cron.!=crit` selects all `cron` facility messages except those of `crit` priority. Only one priority and one priority wildcard can be listed per selector.

Actions

Actions tell the `syslogd` what to do with the event notification messages it receives. Listing 5-3 lists the four actions `syslogd` can take, including logging to a file, device file, named pipes (fifos) and the console or a user's screen. In Listing 5-2 you saw device logging at work with all the `emerg` messages on the system being sent to the console.

Listing 5-3. *File, Device, and Named Pipe Actions*

```
cron.err          /var/log/cron
auth.!=emerg      /dev/lpr3
auth-priv         root,bob
news.=notice      |/var/log/newspipe
```

In the first line all `cron` messages of `err` priority and higher are logged to the file `/var/log/cron`. The second line has all `auth` messages except those of `emerg` priority being sent to a local printer `lpr3`. The third line sends all `auth-priv` messages to the users `root` and `bob` if they are logged in. The fourth sends all `news` messages of `notice` or greater priority to a named pipe called `/var/log/newspipe` (you would need to create this pipe yourself with the `mkfifo` command).

■Caution When logging to files, `syslogd` allows you to add a hyphen (`-`) to the front of the filename like this: `-/var/log/auth`. This tells `syslog` to not sync the file after writing to it. This is designed to speed up the process of writing to the log. But it can also mean that if your system crashes between write attempts, you will lose data. Unless your logging system is suffering from performance issues, I recommend you do not use this option.

You can also log to a remote system (see Listing 5-4).

Listing 5-4. *Logging to a Remote System*

```
mail     @puppy.yourdomain.com
```

In this example all `mail` messages are sent to the host `puppy.yourdomain.com` on UDP port 514. This requires that the `syslogd` daemon on `puppy` is started with the `-r` option; otherwise, the `syslogd` port will not be open.

Caution Opening syslogd to your network is a dangerous thing. The syslogd daemon is not selective about where it receives messages from. There are no access controls, and any system on your network can log to the syslogd port. This opens your machine to the risk of a DoS attack or of a rogue program flooding your system with messages and using all the space in your log partition. I will briefly discuss some methods by which you can reduce the risk to your system, but if you are serious about remote logging I recommend you look at the "syslog-NG" section. I will also discuss secure logging using the syslog-NG tool in conjunction with Stunnel in the "Secure Logging with syslog-NG" section.

Combining Multiple Selectors

You can also combine multiple selectors in your `syslog.conf` file, allowing for more sophisticated selections and filtering. For example, you can list multiple facilities separated by commas in a selector. See Listing 5-5.

Listing 5-5. *Multiple Facilities*

```
auth,auth-priv.crit     /var/log/auth
```

This sends all `auth` messages and all `auth-priv` messages with a priority of `crit` or higher to the file `/var/log/auth`.

You cannot do this with priorities, though. If want to list multiple priorities, you need to list multiple selectors separated by semicolons, as shown in Listing 5-6.

Listing 5-6. *Multiple Priorities*

```
auth;auth-priv.debug;auth-priv.!=emerg     /var/log/auth
```

This example shows you how to send all `auth` messages and all `auth-priv` messages with a priority of `debug` or higher, excluding `auth-priv` messages of `emerg` priority to the file `/var/log/auth`.

Tip Just remember with multiple selectors that filtering works from left to right; `syslogd` will process the line starting from the selectors on the left and moving to the right of each succeeding selector. With this in mind, place the broader filters at the left, and narrow the filtering criteria as you move to the right.

You can also use multiple lines to send messages to more than one location, as shown in Listing 5-7.

Listing 5-7. *Logging to Multiple Places*

```
auth           /var/log/auth
auth.crit      bob
auth.emerg     /dev/console
```

Here all `auth` messages are logged to `/var/log/auth` as previously, but `auth` messages of `crit` or higher priority are also sent to user `bob`, if he is logged in. Those of `emerg` priority are also sent to the console.

Starting syslogd and Its Options

The `syslogd` daemon and its sister process, the `klogd` daemon, are both started when your system boots up. This is usually in the form of an `init` script; for example, on Red Hat the `syslog` script in `/etc/rc.d/init.d/` starts `syslogd` and `klogd`. You can pass a number of options to the `syslogd` program when it starts.

■ **Tip** On most Red Hat and Mandrake systems the `syslog` file in `/etc/sysconfig/` is referenced by the `syslog init` script and contains the options to be passed to `syslogd` and `klogd` when it starts.

The first option you will look at is the debug option (see Listing 5-8).

Listing 5-8. *Running* `syslogd` *with Debug*

```
puppy# syslogd -d
```

This will start `syslogd` and prevent it from forking to the background. It will display a large amount of debugging information to the current screen (you will probably want to pipe it into `more` to make it easier to read). A lot of the information the debug option displays is not useful to the everyday user, but it will tell you if your `syslog.conf` file has any syntax errors, which is something that becomes useful if your file grows considerably.

The next option you will look at tells `syslogd` where to find the `syslog.conf` file. By default `syslogd` will look for `/etc/syslog.conf`, but you can override this (see Listing 5-9).

Listing 5-9. *Starting* `syslogd` *with a Different Config File*

```
puppy# syslogd -f /etc/puppylog.conf
```

In this example `syslogd` would look for `/etc/puppylog.conf`. If this file does not exist, then `syslogd` will terminate. This is useful for testing a new `syslog.conf` file without overwriting the old one.

I discussed earlier `mark` facility messages. These are time stamps that are generated at specified intervals in your logs that look something like this:

```
Feb 24 21:46:05 puppy -- MARK -
```

They are useful, amongst other reasons, for acting as markers for programs parsing your log files. These time stamps are generated using the `-m` *`mins`* option when you start `syslogd`. To generate a `mark` message every ten minutes, you would start `syslogd` as shown in Listing 5-10.

Listing 5-10. *Generating* `mark` *Messages*

```
puppy# syslogd -m 10
```

Remember that `mark` is a facility in its own right, and you can direct its output to a particular file or destination (see Listing 5-11).

Listing 5-11. *Using the* `mark` *Facility*

```
mark     /var/log/messages
```

In Listing 5-11 all `mark` facility messages will be directed to `/var/log/messages`. By default most `syslogd` daemons start with the `-m` option set to `0`.

Often when you set up a `chroot` environment, the application in the jail is unable to log to `syslog` because of the restrictive nature of the `chroot` jail. In this instance, you can create an additional log socket inside the `chroot` jail and use the `-a` option when you start `syslogd` to allow `syslog` to listen to it. You will see how this works in more detail in Chapter 11 when I show how to set up a BIND daemon in a `chroot` jail. See Listing 5-12.

Listing 5-12. *Listening to Additional Sockets*

```
puppy# syslogd -a /chroot/named/dev/log -a /chroot/apache/dev/log
```

Here the `syslogd` daemon is listening to two additional sockets: one in `/chroot/named/dev/log` and the other in `/chroot/apache/dev/log`.

Lastly you will look at the `-r` option, which allows `syslogd` to receive messages from external sources on UDP port 514. See Listing 5-13.

Listing 5-13. *Enabling Remote Logging*

```
puppy# syslogd -r
```

By default most `syslogd` daemons start without `-r` enabled, and you will have to specifically enable this option to get `syslogd` to listen.

■ **Tip** If you enable the `-r` option, you will need to punch a hole in your firewall to allow remote `syslogd` daemons to connect to your system.

If you are going to use `syslogd` for remote logging, then you have a couple of ways to make your installation more secure. The most obvious threat to `syslogd` daemons are DoS attacks in which your system is flooded with messages that could completely fill your disks. If your logs are located in the root partition, your system can potentially crash. To reduce the risk of this potential crash, I recommend you store your logs on a nonroot partition. This means that even if all the space on your disk is consumed, the system will not crash. The second way to secure your `syslogd` for remote logging is to ensure your firewall rules allow connections only from those systems that will be sending their logging data to you. Do not open your `syslog` daemon to all incoming traffic!

syslog-NG

Syslog and `syslogd` are useful tools; however, not only are they dated, but they also have limitations in the areas of reliability and security that do not make them the ideal tools to use in a hardened environment. A worthy successor to `syslog` is syslog-NG. Developed to overcome the limitations of `syslog`, it represents a "new-generation" look at logging with an emphasis on availability and flexibility and considerably more regard for security.

Additionally, syslog-NG allows for more sophisticated message filtering, manipulation, and interaction. syslog-NG is freeware developed by Balazs Scheidler and is available from `http://www.balabit.com/products/syslog_ng/`.

■ **Note** syslog-NG goes through a lot of active development, and new features are added all the time. With the active development cycle of the product, sometimes the documentation becomes out-of-date. If you want to keep up with all the activity and need help for something that is not explained in the documentation, then I recommend you subscribe to the syslog-NG mailing list at `https://lists.balabit.hu/mailman/listinfo/syslog-ng`. syslog-NG's author, Balazs Scheidler, is a regular and helpful participant on the list. As a result of this busy development cycle, I also recommend you use the most recent stable release of `libol` and syslog-NG to get the most out of the package.

The following sections cover installing and compiling syslog-NG and then configuring it as a replacement for syslog. I will also cover configuring syslog-NG to allow you to store and query log messages in a database. Finally, I will cover secure syslog-NG logging in a distributed environment.

Installing and Configuring syslog-NG

Download syslog-NG and `libol` (an additional library required for installing syslog-NG) from `http://www.balabit.com/products/syslog_ng/upgrades.bbq`. You will need to build `libol` first. So unpack the tar file, and compile the `libol` package.

```
puppy# ./configure && make && make install
```

■ **Tip** If you do not want to install `libol`, you can omit the `make install` command, and when you configure syslog-NG, you need to tell it where to find `libol` using `./configure --with-libol=/path/to/libol`.

Now unpack syslog-NG, enter the syslog-NG directory, and configure the package.

```
puppy# ./configure
```

By default syslog-NG is installed to `/usr/local/sbin`, but you can override this by entering the following:

```
puppy# ./configure --prefix=/new/directory/here
```

Also by default syslog-NG looks for its `conf` file in `/usr/local/etc/syslog-ng.conf`. You can override this also. I recommend using `/etc/syslog-ng`.

```
puppy# ./configure --sysconfdir=/etc/syslog-ng
```

Then make and install syslog-NG.

```
puppy# make && make install
```

This will create a binary called `syslog-ng` and install it either to the `/usr/local/sbin/` directory or to whatever directory you have specified if you have overridden it with the prefix option.

The contrib Directory

Within the syslog-NG package comes a few other useful items. In the `contrib` directory is a collection of `init` scripts for a variety of systems including Red Hat and SuSE. These can be easily adapted to suit your particular distribution. Also in the `contrib` directory is an `awk` script called `syslog2ng`, which converts `syslog.conf` files to `syslog-ng.conf` files. See Listing 5-14.

Listing 5-14. *Using the* `syslog2ng` *Script*

```
puppy# ./syslog2ng < /etc/syslog.conf > syslog-ng.conf
```

This will convert the contents of your `syslog.conf` file into the file called `syslog-ng.conf`. This is especially useful if you have done a lot of work customizing your `syslog.conf` file.

Lastly, in the `contrib` directory are several sample `syslog-ng.conf` files, including `syslog-ng.conf.RedHat`, which provides a syslog-NG configuration that replicates the default `syslog.conf` file on a Red Hat system. (Note that it assumes you have disabled the `klogd` daemon and are using `syslog-ng` for kernel logging as well.) This file should also work on most Linux distributions.[2] Also, among the sample `syslog-ng.conf` files is `syslog-ng.conf.doc`, which is an annotated configuration file with the manual entries for each option and function embedded next to that option or function.

Running and Configuring syslog-NG

As mentioned previously, syslog-NG comes with a number of sample `init` scripts that you should be able to adapt for your system. Use one of these scripts, and set syslog-NG to start when you boot up. As shown in Table 5-2. the `syslog-ng` daemon has some command-line options.

2. I tested it on Mandrake 9.2, SuSE 9, and Debian 3, in addition to Red Hat Enterprise 3, Red Hat 8.0, Red Hat 9.0, and Fedora Core 1, and it logged without issues.

Table 5-2. `syslog-ng` *Command-Line Options*

Flag	Purpose
`-d`	Enables debug.
`-v`	Verbose mode (`syslog-ng` will not daemonize).
`-s`	Do not start; just parse the `conf` file for incorrect syntax.
`-f /path/to/conf/file`	Tells `syslog-ng` where the configuration file is located.

The first two flags, `-d` and `-v`, are useful to debug the `syslog-ng` daemon. In the case of the `-v` flag, `syslog-ng` will start and output its logging messages to the screen and will not fork into the background. The `-d` flag adds some debugging messages. The next flag, `-s`, does not start syslog-NG but merely parses through the `syslog-ng.conf` file and checks for errors. If it finds any errors, it will dump those to the screen and exit. If it exits without an error, then your `syslog-ng.conf` has perfect syntax!

But do not start up syslog-NG yet. You need to create or modify a configuration file first. The `syslog-ng.conf` contains considerably more options than the `syslog.conf` file, which is representative of the increased functionality and flexibility characteristic of the syslog-NG product. As such, setting up the configuration file can be a little bit daunting initially. I recommend you use the `syslog-ng.conf` sample file. When it starts, syslog-NG looks for `/usr/local/etc/syslog-ng.conf` as the default `conf` file unless you overrode that as part of the `./configure` process. I recommend you create your configuration file in `/etc/syslog-ng`.

Every time you change your `syslog-ng.conf` file, you need to restart the `syslog-ng` daemon. Use the provided `init` script to do this, and use the `reload` option. For example, on a Red Hat system, enter the following:

```
puppy# /etc/rc.d/init.d/syslog-ng reload
```

Let's start configuring syslog-NG by looking at a simple configuration file. Listing 5-15 shows a sample `syslog-ng.conf` file that collects messages from the device `/dev/log`, selects all the messages from the `mail` facility, and writes them to the console device.

Listing 5-15. *A Sample* `syslog-ng.conf` *File*

```
options { sync (0); };

source s_sys { unix-dgram ("/dev/log"); };
destination d_console { file("/dev/console"); };
filter f_mail { facility(mail); };

log { source(s_sys); filter(f_mail); destination(d_console); };
```

Listing 5-15 is a functioning (if limited) syslog-NG configuration. It may look intimidating at first, but it is actually a simple configuration model when you break it down. The key line in this example is the last one, the `log{}` line. The `log{}` line combines three other types of statements: a source statement to tell syslog-NG where to get the messages from; a filter statement to allow you to select messages from within that source according to criteria, such as their facility or priority; and finally a destination statement to tell syslog-NG where to write the messages to, such as a file or a device. The `options{}` statement allows you to configure some global options for syslog-NG.

Let's take you through the basics of configuring syslog-NG by running through each of the statement blocks available to you. The `syslog-ng.conf` file uses five key statement blocks (see Table 5-3).

Table 5-3. `syslog-ng.conf` *Statement Blocks*

Directive	Purpose
`options{}`	Global options to be set
`source{}`	Statements defining where messages are coming from
`destination{}`	Statements defining where messages are sent or stored
`filter{}`	Filtering statements
`log{}`	Statements combining source, destination, and filter statements that do the actual logging

Each statement block contains additional settings separated by semicolons. You can see that I have used all these statements in Listing 5-15.

options{}

These are global options that tell syslog-NG what to do on an overall basis. The options themselves consist of their name and then their value enclosed in parentheses and terminated with a semicolon. As shown in Listing 5-16, these options control functions such as the creation of directories and the use of DNS to resolve hostnames, and they provide control over the process of writing data to the disk.

Listing 5-16. *A Sample* `syslog-ng options{}` *Statement*

```
options {
sync(0);
time_reopen(10);
use_dns(yes);
use_fqdn(no);
create_dirs(no);
keep_hostname(yes);
chain_hostnames(no);
};
```

Quite a number of options are available to you. In this section I will cover the key options. Probably the most confusing options to new users of syslog-NG are those associated with hostnames. I recommend two key options in this area that every user should put in the `syslog-ng.conf` file. They are `keep_hostname(yes | no)` and `chain_hostnames(yes | no)`.

■ **Tip** The syslog-NG documentation also refers to `long_hostnames()`. This is an alias for `chain_hostnames()` and is identical in function.

When syslog-NG receives messages, it does not automatically trust that the hostname provided to it by a message is actually the hostname of the system on which the message originated. As a result, syslog-NG tries to resolve the hostname of the system that generated the messages. If the resolved hostname is different, it attempts to rewrite the hostname in the message to the hostname it has resolved. This behavior occurs because by default the `keep_hostname()` option is set to `no`. If `keep_hostname(yes)` is set (as it is in Listing 5-16), then this prevents syslog-NG from rewriting the hostname in the message.

So where does the `chain_hostnames()` option come into all this? Well, it works in conjunction with `keep_hostname()`. If `keep_hostname()` is set to `no`, then it checks whether `chain_hostnames()` is set to `yes`. If `chain_hostnames()` is set to `yes`, then syslog-NG appends the name of the host that syslog-NG received the message from to the resolved hostname. So, for example, if the hostname in the message is `puppy` but the hostname that syslog-NG has resolved the IP address to is `puppy2`, then the message will change from this:

```
Jul 14 16:29:36 puppy su(pam_unix)[2979]: session closed for user bob
```

to the following:

```
Jul 14 16:29:36 puppy/pupp2 su(pam_unix)[2979]: session closed for user bob
```

If `chain_hostnames()` is set to `no`, then syslog-NG simply replaces the hostname with a resolved hostname.

This can be a little confusing, so I will now illustrate it with another example. In Table 5-4 you have a message that has a hostname of `server`. When syslog-NG resolves this hostname, DNS tells it that the real hostname of the system is `server2`. The table shows the resulting hostname that will be displayed in the message with all possible combinations of the options.

Table 5-4. `chain_hostnames()` *and* `keep_hostname()` *Interaction*

Option Setting	keep_hostname(yes)	keep_hostname(no)
chain_hostnames(yes)	server	server/server2
chain_hostnames(no)	server	server2

■ **Tip** By default `chain_hostnames()` is set to `yes`, and `keep_hostname()` is set to `no`.

Also related to hostnames are `use_dns()` and `use_fqdn()`. The `use_dns()` option allows you to turn off DNS resolution for syslog-NG. By default it is set to `yes`. The `use_fqdn()` option specifies whether syslog-NG will use fully qualified domain names. If `use_fqdn()` is set to `yes`, then all hosts will be displayed with their fully qualified domain names; for example, `puppy` would be `puppy.yourdomain.com`. By default `use_fqdn()` is set to `no`.

You have a whole set of options available that deal with the creation of directories and files (see Table 5-5). They control ownership, permissions, and whether `syslog-ng` will create new directories.

Table 5-5. *File and Directory Options*

Option	Purpose
`owner(`*`userid`*`)`	The owner of any file `syslog-ng` creates
`group(`*`groupid`*`)`	The group of any file `syslog-ng` creates
`perm(`*`permissions`*`)`	The permission of any file `syslog-ng` creates
`create_dirs(yes | no)`	Whether `syslog-ng` is allowed to create directories to store log files
`dir_owner(`*`userid`*`)`	The owner of any directory `syslog-ng` creates
`dir_group(`*`groupid`*`)`	The group of any directory `syslog-ng` creates
`dir_perm(`*`permissions`*`)`	The permission of any directory `syslog-ng` creates

A few additional options could be useful for you. They are `sync(`*`seconds`*`)`, `stats(`*`seconds`*`)`, `time_reopen(`*`seconds`*`)`, and `use_time_recvd()`. The `sync()` option tells syslog-NG how many messages to buffer before it writes to disk. It defaults to `0`. The `stats(`*`seconds`*`)` option provides you with regular statistics detailing the number of messages dropped.

■ **Note** Messages are dropped, for example, if syslog-NG reaches the maximum available number of connections on a network source (as defined with the `maxconnections()` option). The `stats` option will record how many messages were dropped.

The *`seconds`* variable in the option indicates the number of seconds between each stats message being generated. In the `time_reopen(`*`seconds`*`)` option, *`seconds`* is the amount of time that syslog-NG waits before retrying a dead connection. This currently defaults to 60 seconds, but you may want to reduce this. I have found around ten seconds is a sufficient pause for syslog-NG. The last option you will look at is `use_time_recvd()`. When this option is set to `yes`, then the time on the message sent is overridden with the time the message is received by syslog-NG on the system. The default for this setting is `no`. The `use_time_recvd()` option is important to consider when you use the `destination{}` file-expansion macros that I will discuss in the "destination{}" section.

source{}

Your source statements are the key to telling syslog-NG where its message inputs are coming from. You can see an example of a `source{}` statement in Listing 5-17.

Listing 5-17. *A syslog-NG* `source{}` *Statement*

```
source s_sys { unix-stream("/dev/log" max-connections(20)); internal(); };
```

The `source{}` statement block is much like the `options{}` statement block in that it contains different possible input sources and is terminated with a semicolon. The major difference is that the first part of each `source{}` statement block is its name you need to define. You can see that in Listing 5-17 I gave `s_sys` as the name of the `source{}` statement. For these purposes, you use a naming convention that allows you to easily identify each source: `s_sys` for Linux system logs and syslog-NG internal logging, `s_tcp` for logs that come in over TCP, `s_udp` for logs that come in over UDP, and `s_file` for file-based input sources.

Inside the `source{}` statement you have a number of possible input sources so that one source statement can combine multiple messages sources; for example, the `source{}` statement in Listing 5-17 receives both internal syslog-NG messages and standard Linux system messages. Table 5-6 describes the sources you are most likely to use.

Table 5-6. *syslog-NG Sources*

Source	Description
`unix-stream()`	Opens an `AF_UNIX` socket using SOCK_`STEAM` semantics (for example, `/dev/log`) to receive messages
`unix-dgram()`	Opens an `AF_UNIX` socket using `SOCK_DGAM` semantics
`tcp()`	Opens TCP port 514 to receive messages
`udp()`	Opens UDP port 514 to receive messages
`file()`	Opens a specified file and processes it for messages
`pipe()`	Opens a named pipe

You can use both `unix-stream()` and `unix-dgram()` to connect to an `AF_UNIX` socket, such as `/dev/log` (which is the source of most Linux system messages). You can also use it to specify a socket file in a `chroot` jail, as shown in Listing 5-18.

Listing 5-18. *Opening a Socket in a* `chroot` *Jail*

```
source s_named { unix-stream("/chroot/named/dev/log"); };
```

Listing 5-18 shows syslog-NG opening a log socket for a named daemon inside a `chroot` jail.

The `unix-stream()` and `unix-dgram()` sources are similar but have some important differences. The first source, `unix-steam()`, opens an `AF_UNIX` socket using `SOCK_STREAM` semantics, which are connection orientated and therefore prevent message loss. The second source, `unix-dgram()`, opens an `AF_UNIX` socket using `SOCK_DGRAM` semantics, which are not connection orientated and can result in messages being lost. The `unix-dgram()` source is also open to DoS attacks because you are unable to restrict the number of connections made to it. With `unix-stream()` you can use the `max-connections()` option to limit the maximum number of possible connections to the source.

■ **Tip** You can see the `max-connections()` setting in the first line of Listing 5-17; it is set to `10` by default, but on a busy system you may need to increase that maximum. If you run out of connections, then messages from external systems will be dropped. You can use the `stats` option, as described in the "options{}" section, to tell you if messages are being dropped.

As such, I recommend you use the unix-stream() source, not the unix-dgram() source.

The next types are tcp() and udp() sources.

```
source s_tcp { tcp(ip(192.168.0.1) port(514) max-connections(15)); };
source s_udp { udp(); };
```

These sources both allow syslog-NG to collect messages from external systems. As discussed during the "Syslog" section of this chapter, I do not recommend you use udp() for this purpose. Unlike syslog, however, syslog-NG also supports message send via Transmission Control Protocol (TCP) using the tcp() source. This delivers the same functionality as UDP connections but with the benefit of TCP acknowledgments, which greatly raise the level of reliability. The tcp() connections are also able to be secured by introducing a tool such as Stunnel. Stunnel encapsulates TCP traffic inside a Secure Sockets Layer (SSL) wrapper and secures the connection with public-key encryption. This means attackers are not able to read your log traffic and that you are considerably more protected from any potential DoS attacks because syslog-NG is configured to receive only from those hosts you specify. I will discuss this capability in the "Secure Logging with syslog-NG" section later in the chapter.

The previous tcp() source statements specify 192.168.0.1 as the IP address to which syslog-NG should bind. This IP address is the address of a local interface, not that of the sending system. It also specifies 514 as the port number to run on and 15 as the maximum number of simultaneous connections. If the max-connections() option is not set, then it defaults to 10. This is a safeguard against DoS attacks by preventing an unlimited number of systems from simultaneously connecting to your syslog-NG server and overloading it. I will show how to further secure your TCP connections in the "Secure Logging with syslog-NG" section.

The next type of source is a file() source statement. The file() is used to process special files such as those in /proc.

```
source s_file { file("/proc/kmsg" log_prefix("kernel: ")); };
```

It is also commonly used to collect kernel messages on systems where you have replaced klogd as well as syslogd.

■Tip This will not follow a file like the tail -f command. In the "Testing Logging with logger" section I will explain how you can use logger to feed a growing file to the syslog-NG daemon.

It is easy to replace klogd with syslog-NG. To add kernel logging to syslog-NG, adjust your source{} statement to include file("/proc/kmsg"). A source{} statement used to log most of the default system messages would now look something like this:

```
source s_sys { file("/proc/kmsg" log_prefix("kernel: ")); ➥
unix-stream("/dev/log"); internal(); };
```

The log prefix option ensures all kernel messages are prefixed with "kernel: ". You need to ensure you have stopped the klogd daemon before you enable and start syslog-NG with kernel logging. Otherwise syslog-NG may stop, and all local logging will be disabled.

The last source is the `pipe()` source. This is used to open a named pipe as an input source.

```
source s_pipe { pipe("/var/programa"); };
```

This allows programs that use named pipes for their logging to be read by syslog-NG. This source can be also used to collect system messages from `/proc/kmsg`.

```
source s_kern { pipe("/proc/kmsg"); };
```

destination{}

The `destination{}` statement block contains all the statements to tell syslog-NG where to put its output. This output could be written to a log file, output to a program, or output to a database. Listing 5-19 contains an example of a `destination{}` statement.

Listing 5-19. *A syslog-NG* `destination{}` *Statement*

```
destination d_mult { file("/var/log/messages"); usertty("bob"); };
```

The `destination{}` statement block is constructed like that of the `source{}` statement block. I have continued in the vein of the naming convention I started in the "source{}" section and prefixed the name of the destination blocks with `d_` (for example, `d_console`).

As with `source{}` statements, you can combine more than one destination in a single source statement. As you can see in Listing 5-19, the destination `d_mult` logs both to a file and to a session signed on as the user `bob`. Various possible destinations for your messages are available.

Probably the most commonly used destination is `file()`, which logs message data to a file on the system. The next line shows a file destination of `/var/log/messages`, which will be owned by the `root` user and group and have its file permissions set to `0644`.

```
destination d_mesg { file("/var/log/messages" owner(root) group(root) perm(0644)); };
```

So as you can see, the `file()` destination statement consists of the name of the file you are logging to and a variety of options that control the ownership and permission of the file itself. These are identical to the file-related permissions you can set at the global `options{}` level, and the options for each individual destination override any global options specified.

In the next line, you can also see the use of the file-expansion macros:

```
destination d_host { file("/var/log/hosts/$HOST/$FACILITY$YEAR$MONTH$DAY"); };
```

File-expansion macros are useful for including data such as the hostname, facility, and date and time in the filenames of your log files to make them easier to identify and manipulate. Each is placed exactly like a shell script parameter, prefixed with `$`. Using this example, a `cron` message on the `puppy` system on March 1, 2005, would result in the following directory and file structure:

```
/var/log/puppy/cron20050301
```

You can see a list of all possible file-expansion macros in Table 5-7.

Table 5-7. *syslog-NG File-Expansion Macros*

Macro	Description
`FACILITY`	Name of the facility from which the message is tagged as coming
`PRIORITY`	Priority of the message
`TAG`	Priority and facility encoded as a two-digit hexadecimal number
`DATE`	The date of the message in the form of MMM DD HH:MM:SS
`FULLDATE`	The date in the form of YYYY MMM DD HH:MM:SS
`ISODATE`	The date in the form of YYYY-MM-DD HH:MM:SS TZ
`YEAR`	Year the message was sent in the form YYYY
`MONTH`	Month the message was sent in the form of MM
`DAY`	Day of month the message was sent in the form of DD
`WEEKDAY`	Three-letter name of the day of week the message was sent (for example, Mon)
`HOUR`	Hour of day the message was sent
`MIN`	Minute the message was sent
`SEC`	Second the message was sent
`TZOFFSET`	Time zone as hour offset from Greenwich mean time (for example, +1200)
`TZ`	Time zone or name or abbreviation (for example, AEST)
`FULLHOST`	Name of the source host from where the message originated
`HOST`	Name of the source host from where the message originated
`PROGRAM`	Name of the program the message was sent by
`MESSAGE`	Message contents

The time-expansion macros, such as `DATE`, can either use the time that the log message was sent or use the time the message was received by syslog-NG. This is controlled by the `use_time_recvd()` option discussed in the "options{}" section.

Also using the same file-expansion macros you can invoke the `template()` option, which allows you to write out data in the exact form you want to the destination. The `template()` option also works with all the possible destination statements, not just `file()`. Listing 5-20 shows a destination statement modified to include a template.

Listing 5-20. *The* `template()` *Option*

```
destination d_host { file("/var/log/hosts/$HOST/$FACILITY$YEAR$MONTH$DAY" ➥
template("$HOUR:$MIN:$SEC $TZ $HOST [$LEVEL] $MSG $MSG\n") ➥
template_escape(no) ); };
```

The `template_escape(yes | no)` option turns on or off the use of quote marks to escape data in your messages. This is useful if the destination of the message is a SQL database, as the escaping prevents the log data being treated by the SQL server as commands.

The `pipe()` destination allows the use of named pipes as a destination for message data. This is often used to send messages to `/dev/console`.

```
destination d_cons { pipe("/dev/console"); };
```

Here the destination `d_cons` sends all messages to the `/dev/console` device. You will also use the `pipe()` destination to send messages to the SEC log correlation tool and to a database.

Importantly for a distributed monitoring environment, it is also possible to forward messages to another system using either TCP or UDP with the `tcp()` and `udp()` `destination{}` statements.

```
destination d_monitor { tcp("192.168.1.10" port(514)); };
```

You can see for the third statement of Listing 5-18 where the destination `d_monitor` is a `syslog` server located at IP address `192.168.1.10` that listens to TCP traffic on the standard `syslog` port of 514. As stated elsewhere, I do not recommend using UDP connections for your logging traffic. You will see more of how this is used for secure logging in the "Secure Logging with syslog-NG" section.

The `usertty()` destination allows you to send messages to the terminal of a specific logged user or all logged users. In the next line, the destination `d_root` sends messages to terminals logged in as the `root` user:

```
destination d_root { usertty("root"); };
```

You can also use the wildcard option (*) to send messages to all users.

Finally, the `program()` destination invokes a program as the destination. You have quite a variety of possible uses for this, including mailing out certain messages or as an alternative method of integrating syslog-NG with log analysis tools such as SEC or Swatch. See Listing 5-21.

Listing 5-21. *Sample* `program()` *Destination*

```
destination d_mailout { program("/root/scripts/mailout" ➥
template("$HOUR:$MIN:$SEC $HOST $FACILITY $LEVEL $PROGRAM $MSG\n")); };
```

In Listing 5-21 the `d_mailout` destination sends a message to the script `/root/scripts/mailout` using a template. Note the `\n` at the end of the `template()` line. Because of the use of a template, you have to tell the script that it has reached the end of the line and that the loop must end. With a normal message, syslog-NG includes its own line break.

The `mailout` script itself is simple (see Listing 5-22).

Listing 5-22. `mailout` *Script*

```
#!/bin/bash
#  Script to mail out logs

while read line; do
     echo $line | /bin/mail -s "log entry from mail out" pager@yourdomain.com
done
```

■ **Caution** Always remember that any time you designate a message to be mailed out, you should ask yourself if you are making yourself vulnerable to a self-inflicted DoS attack if thousands of messages were generated and your mail server was flooded. Either choose only those most critical messages to be mailed out or look at modifying the script you are using to mail out messages to throttle the number of messages being sent out in a particular time period.

Lastly, you also have the `destination{}` statements `unix-stream()` and `user-dgram()` that you can use to send message data to Unix sockets. Neither `unix-stream()` nor `user-dgram()` has any additional options.

```
destination d_socket { unix-stream("/tmp/socket"); };
```

filter{}

The `filter{}` statement blocks contain statements to tell syslog-NG which messages to select. This is much like the facility and priority selector used by the `syslog` daemon. For example, the following line is a `syslog` facility and priority selector that would select all messages of the `mail` facility with a priority of `info` or higher:

```
mail.info
```

This can be represented by a syslog-NG filter statement block that looks like this:

```
filter f_mail     { facility(mail); priority(info .. emerg) };
```

In the previous line I have selected all messages from the facility mail using the `facility()` option and with a range of priorities from `info` to `emerg` using the `priority()` option.

But the syslog-NG equivalent is far more powerful than the selectors available to you in the `syslog` daemon. Selection criteria can range from selecting all those messages from a particular facility, host, or program to regular expressions performed on the message data itself.

The `filter{}` statement blocks are constructed like the `source{}` statement block, and each filter must be named. Again, continuing with the naming convention, I generally prefix all my filter statements with `f_` (for example, `f_kern`). In Table 5-8 you can see a complete list of items on which you can filter.

Table 5-8. *Items You Can Filter on in syslog-NG*

Filter	Description
`facility()`	Matches messages having one of the listed facility code(s)
`priority()`	Matches messages by priority (or the `level()` statement)
`program()`	Matches messages by using a regular expression against the program name field of log messages
`host()`	Matches messages by using a regular expression against the hostname field of log messages
`match()`	Tries to match a regular expression to the message itself
`filter()`	Calls another filter rule and evaluate its value
`netmask()`	Matches message IP address against an IP subnet mask

The simplest filters are facility and priority filters. With facility and priority filters you can list multiple facilities or priorities in each option separated by commas. For example, in the `f_mail` filter on the next line, you can see the filter is selecting all `mail` and `daemon` facility messages.

```
filter f_mail     { priority(mail,daemon) };
```

You can also list multiple priories in a range separated by `..`, as you can see in the `f_infotoemerg` filter on the next line:

```
filter f_infotoemerg { priority(info .. error); };
```

The `filter{}` statements can also contain multiple types of options combined using Boolean AND/OR/NOT logic. You can see these capabilities in the `f_boolean` filter statement on the next line where I am selecting all messages of priority `info`, `notice`, and `error` but not those from the `mail`, `authpriv`, or `cron` facilities.

```
filter f_boolean { priority(info .. error) and not (facility(mail) ➥
or facility(authpriv) or facility(cron)); };
```

Another type of filtering is to select messages from a specific host. You can see this in the `f_hostpuppy` filter on the next line:

```
filter f_hostpuppy { host(puppy); };
```

You can also select messages based on the netmask of the system that generated them. The filter on the next line selects all messages from the network `10.1.20.0/24`:

```
filter f_netmask { netmask("10.1.20.0/24") };
```

Another form of filtering you will find useful is to select only those messages from a particular program. The filter on the next line will select only those messages generated by the `sshd` daemon:

```
filter f_sshd { program("sshd.*") };
```

Finally, you can match messages based on the content of the messages using the `match()` option. You can use regular expressions to match on the content of a message.

```
filter f_regexp { match("deny"); };
```

Additionally, you can add a `not` to the front of your match statement to negate a match and not select those messages with a particular content.

```
filter f_regexp2 { not match("STATS: dropped 0")};
```

log{}

As described earlier, after you have defined `source{}`, `destination{}`, and `filter{}` statements, you need to combine these components in the form of `log{}` statement blocks that actually do the logging. Unlike the other types of statement blocks, they do not need to be named.

For a valid `log{}` statement, you need to include only a source and a destination statement. The following line logs all the messages from the source `s_sys` to the destination `d_mesg`:

```
log { source(s_sys); destination(d_mesg); };
```

But generally your `log{}` statement blocks will contain a combination of source, destination, and filter statements. You can also combine multiple sources and filters into a `log{}` statement. As you can see in Listing 5-23, I am selecting messages from two sources, `s_sys` and `s_tcp`, and then filtering them with `f_mail` and sending them to the `d_mesg` destination.

Listing 5-23. *A syslog-NG* `log{}` *Statement with Multiple Sources*

```
log { source(s_sys); source(s_tcp); filter(f_mail); destination(d_mesg); };
```

The `log{}` statement blocks also have some potential flag modifiers (see Table 5-9).

Table 5-9. `log{}` *Statement Block Flags*

Flag	Description
`final`	Indicates the processing of log statements ends here. If the messages matches this `log{}` statement, it will be processed here and discarded.
`fallback`	Makes a log statement "fall back." This means that only messages not matching any "nonfallback" log statements will be dispatched.
`catchall`	The source of the message is ignored; only the filters are taken into account when matching messages.

You can add flags to the end of `log{}` statements like this:

```
log { source(s_sys); destination(d_con); flags(final); };
```

This log statement is modified to show that if the message is from the source `s_sys` with destination `d_con`, then log it to that destination and do not match that message against any further `log{}` statements. This does not necessarily mean the message is logged only once. If it was matched to any `log{}` listed in your `syslog-ng.conf` file prior to this one, they will also have logged that message.

Sample syslog-ng.conf File

You have seen what all the possible syslog-NG statement blocks are. Now it is time to combine them into an overall sample configuration. The configuration in Listing 5-24 shows basic host-logging messages on a local system being sent to various destinations. This is a working configuration, and you can further expand on it to enhance its capabilities.

■ **Tip** Do not forget syslog-NG comes with an excellent example `syslog-ng.conf` file, and you can also use the conversion tool, `syslog2ng`, to create a file from your existing `syslog.conf` file!

Listing 5-24. *Starter* `syslog-ng.conf` *File*

```
options {
sync (0);
time_reopen (10);
log_fifo_size (1000);
create_dirs (no);
owner (root);
group (root);
perm (0600);
};
```

```
source s_sys {
pipe ("/proc/kmsg" log_prefix("kernel: "));
unix-dgram ("/dev/log");
internal();
};

filter f_defaultmessages { level(info) and not (facility(mail) ➥
or facility(authpriv) or facility(cron) or facility(local4)); };
filter f_authentication { facility(authpriv) or facility(auth); };
filter f_mail { facility(mail); };
filter f_emerg { level(emerg); };
filter f_bootlog { facility(local7); };
filter f_cron { facility(cron); };

destination d_console { file("/dev/console"); };
destination d_allusers { usertty("*"); };
destination d_defaultmessages { file("/var/log/messages"); };
destination d_authentication { file("/var/log/secure"); };
destination d_mail { file("/var/log/maillog"); };
destination d_bootlog { file("/var/log/boot.log"); };
destination d_cron { file("/var/log/cron"); };

log { source(s_sys); filter(f_defaultmessages); destination(d_defaultmessages); };
log { source(s_sys); filter(f_authentication); destination(d_authentication); };
log { source(s_sys); filter(f_mail); destination(d_mail); };
log { source(s_sys); filter(f_emerg); destination(d_allusers); ➥
destination(d_console); };
log { source(s_sys); filter(f_bootlog); destination(d_bootlog); };
log { source(s_sys); filter(f_cron); destination(d_cron); };
```

To make sure you fully understand what the `syslog-ng.conf` file is doing, let's step through one of the items being logged here. In the `s_sys` source statement, you are collecting from the standard logging device, `/dev/log`.

```
unix-dgram ("/dev/log");
```

Amongst the messages to this device are security-related messages sent to the `auth` and `auth-priv` facilities. In the following line, I have defined a filter statement, `f_authentication`, to pick these up:

```
filter f_authentication { facility(authpriv) or facility(auth); };
```

Next I have defined a destination for the messages, `d_authentication`. This destination that writes to a file in the `/var/log` directory is called `secure`.

```
destination d_authentication { file("/var/log/secure"); };
```

In the global `options{}` block, I have told syslog-NG using the `owner`, `group`, and `perm` options that this file will be owned by the `root` user and group and have permissions of 0600 (which allows only the owner to read and write to it).

Lastly, I have defined a `log{}` statement to actually do the logging itself.

```
log { source(s_sys); filter(f_authentication); destination(d_authentication); };
```

The `log{}` statement combines the previous statements to perform the logging function.

With the steps defined here, you should be able to build your own logging statements using the powerful options available to you with syslog-NG.

Logging to a Database with syslog-NG

So why log to a database? If you need to store logs for any length of time, most probably for some statistical purpose or because of the need for an audit trail, you should look at logging to a database, because it will make the task considerably easier. Querying megabytes of text files containing log messages using tools such as `grep` is cumbersome and prone to error. An SQL database, on the other hand, is designed to be queried via a variety of tools. You can even enable ODBC on your database flavor and query it with third-party tools such as Crystal Reports. This also makes the process of pruning and purging your log entries easier, as you can build SQL queries to perform this task much more simply and with considerably more precision than with file-based log archives.

So if you have the why of it, then how do you do it? I will assume you are using syslog-NG for your logging; however, if you decide to retain `syslogd`, then you can find a link in the "Resources" section to instructions for enabling database logging from `syslogd`.

■ **Note** For the backend database I have chosen to use MySQL, but it is also possible to log to PostgreSQL or even Oracle. This section assumes you have MySQL installed and running on your logging system. See the "Resources" section for more information.

The architecture of database logging is simple. syslog-NG logs the messages you want to store to a pipe, and a script reads those entries from the pipe and writes them to the database. I have used `d_mysql` as the name of the destination in syslog-NG, `mysql.pipe` as the name of the proposed pipe, and `syslog.logs` as the name of the database table. See Figure 5-2.

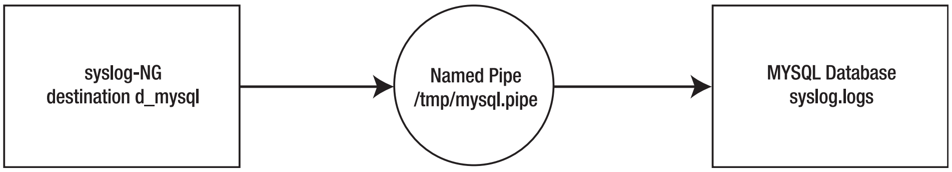

Figure 5-2. *Logging to a database*

So first you need to tell syslog-NG where to send the messages you want to store to a pipe. I will assume you are going to store all messages sent to the `s_sys` source. Listing 5-25 shows the statements needed to log to the database. You can add these to the configuration file in Listing 5-24 to allow you to test database logging.

Listing 5-25. *syslog-NG Statements to Log to a Database*

```
destination d_mysql { pipe("/tmp/mysql.pipe" template("INSERT INTO logs (host,
facility, priority, level, tag, date, time, program, msg) ➥
VALUES( '$HOST','$FACILITY', '$PRIORITY', '$LEVEL', '$TAG', '$YEAR-$MONTH-$DAY',
'$HOUR:$MIN:$SEC', '$PROGRAM', '$MSG' );\n") template-escape(yes)); };

log { source(s_sys); destination(d_mysql); };
```

Tip You may also what to define a filter statement to select only particular messages you want to keep. For example, you could use the `f_authentication` filter from Listing 5-24 to log only security-related messages to the database.

Note the use of the `template-escape(yes)` option to ensure the macros are properly escaped and will be written correctly to the MySQL database.

You then need to create a pipe to store the syslog-NG messages.

```
puppy# mkfifo /tmp/mysql.pipe
```

Now you need to create the MySQL database and a user to connect to it. You can use the `syslog.sql` script shown in Listing 5-26.

Listing 5-26. *The* `syslog.sql` *Script*

```
# Table structure for table `log`
CREATE DATABASE syslog;

USE syslog;

CREATE TABLE logs (
host varchar(32) default NULL,
facility varchar(10) default NULL,
priority varchar(10) default NULL,
level varchar(10) default NULL,
tag varchar(10) default NULL,
date date default NULL,
time time default NULL,
program varchar(15) default NULL,
msg text,
seq int(10) unsigned NOT NULL auto_increment,
PRIMARY KEY (seq),
KEY host (host),
KEY seq (seq),
KEY program (program),
KEY time (time),
KEY date (date),
KEY priority (priority),
KEY facility (facility)
) TYPE=MyISAM;

GRANT ALL PRIVILEGES ON syslog.* TO syslog@localhost identified by 'syslog' ➥
with grant option;
```

This script will create a database called `syslog` with a table called `log` accessible by a user called `syslog` with a password of `syslog`. You should change the grant privileges, user, and password to suit your environment—the `syslog` user needs only `INSERT` privileges to the table.

To run this script, you use the following command:

```
puppy# mysql -u user -p < /path/to/syslog.sql
Enter password:
```

Replace *`user`* with a MySQL user with the authority to create tables and grant privileges, and replace *`/path/to/syslog.sql`* with the location of the script shown in Listing 5-26. You will be prompted to enter the required password for the user specified with the `-u` option.

You can check whether the creation of the database is successful by first connecting to the MySQL server as the `syslog` user and then connecting to the `syslog` database and querying its tables.

```
puppy# mysql -u syslog -p
Enter password:
mysql> connect syslog;
Current database: syslog
mysql> show tables;
Tables_in_syslog
logs
1 row in set (0.00 sec)
```

If the `shows tables` command returns a table called `logs`, then the script has been successful.

You then need a script to read the contents of the `mysql.pipe` pipe and send them to the database. I provide a suitable script in Listing 5-27.

Listing 5-27. *Script to Read* `mysql.pipe`

```
# syslog2mysql script#
#!/bin/bash

if [ -e /tmp/mysql.pipe ]; then
        while [ -e /tmp/mysql.pipe ]
                do
                        mysql -u syslog --password=syslog syslog < /tmp/mysql.pipe
        done
else
        mkfifo /tmp/mysql.pipe
fi
```

Now let's test this script. First, you need to restart syslog-NG.

```
puppy# /etc/rc.d/init.d/syslog-ng restart
```

Second, on the command line, run the script from Listing 5-27 and put it in the background.

```
puppy# /root/syslog2mysql &
```

This script is now monitoring the pipe, `mysql.pipe`, you created in the `/tmp` directory and will redirect any input to that pipe to MySQL.

■ **Tip** I recommend you incorporate the starting and stopping of this script into the `syslog-NG init` script to ensure it gets starts and stops when syslog-NG does.

Now send a log message using the `logger` command. If you have added filtering to the `log{}` block defined in Listing 5-25, then you need to ensure whatever log message you send with `logger` is going to be picked up by that filtering statement and sent to MySQL.

```
logger -p auth.info "Test syslog to MySQL messages from facility auth with ➥
priority info"
```

syslog-NG will write the log message to the `mysql.pipe` script, and the `syslog2mysql` script will direct the log message into MySQL. Now if you connect to your MySQL server and query the content of the `logs` table, you should see the log entry you have sent using `logger`. You can do this with the following commands:

```
puppy# mysql -u syslog -p
Enter password:
mysql> connect syslog;
Current database: syslog
mysql> select * from logs
```

Now your syslog-NG should be logging to the MySQL database.

Secure Logging with syslog-NG

I have discussed in a few places the importance of secure logging and protecting your logging system from both DoS attacks and attempts by intruders to read your logging traffic. To achieve this, you will use Stunnel, the Universal SSL Wrapper, which, as mentioned earlier in the chapter, encapsulates TCP packets with SSL. Stunnel uses certificates and public-key encryption to ensure no one can read the TCP traffic.

■ **Tip** I have discussed Stunnel in considerably more detail in Chapter 3. This is simply a quick-and-dirty explanation of how to get Stunnel working for syslog-NG tunneling. I also discuss OpenSSL and SSL certificates in that chapter, and you may want to create your certificates differently after reading that chapter.

First, you need to install Stunnel. A prerequisite of Stunnel is OpenSSL, which most Linux distributions install by default. You can get Stunnel from `http://www.stunnel.org/` by clicking the Download button in the left menu. Unpack the archive, and change in the resulting directory. The Stunnel install process is simple.

```
puppy# ./configure --prefix=/usr --sysconfdir=/etc
```

The `--prefix` and `--sysconfdir` options place the binaries and related files under `/usr` and the Stunnel configuration files in `/etc/stunnel`.

Second, make and install like this:

```
puppy# make && make install
```

The `make` process will prompt you to input some information for the creation of an OpenSSL certificate. Fill in the details for your environment.

Now you need to create some certificates using OpenSSL for both your syslog-NG server and your clients. On your syslog-NG server, go to `/usr/share/ssl/certs` and create a certificate for your server.

```
puppy# make syslog-ng-servername.pem
```

■ **Tip** The `certs` directory can reside in different places on different distributions. On Red Hat and Mandrake systems it is located in `/usr/share/ssl/certs`. On Debian it is located in `/usr/local/ssl/certs`, and on SuSE it is located in `/usr/ssl/certs`.

Replace the *`servername`* variable with the name of your server. Copy this certificate to the `/etc/stunnel` directory on the system you have designated as the central logging server.

The client certificates are a little different. You need to create a client certificate for each client you want to connect to the syslog-NG server.

```
puppy# make syslog-ng-clientname.pem
```

Your certificates will look something like Listing 5-28.

Listing 5-28. *Your* `syslog-ng` *Certificates*

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDOX34OBdIzsF+vfbWixN54Xfdo73PaUwb+JjoLeF7bu6qKHlgA
RvLiJaambwNCiRJ8jn6GSLiDwOGaffAuQO3YtSrW/NoOsxH6wHEjvOW8d2tWOkbW
o3fOkAeNKCiqBTNDdDRHWnelY5nXgj3jPXOQsuOQq3TlNGy/Dx5YkbprVQIDAQ
4PRxBezKTsaoecCYOIMCQQCw7bOmpJX+DyqLX43STjHt4s7yKio16IOZR1Srsk68
zlOD7HgjNPW8wQEY6yRK7PI+j5o/LNulOXk7JOfOYQUQ
-----END RSA PRIVATE KEY-----
```

```
-----BEGIN CERTIFICATE-----
MIICoDCCAgmgAwIBAgIBADANBgkqhkiG9w0BAQQFADBFMQswCQYDVQQGE
M7Bfr321osTeeF33aO9z2lMG/iY6C3he27uqih5YAEby4iWmpm8DQokSfI5+hki4
g8Dhmn3wLkDt2LUq1vzaNLMR+sBxI7zlvHdrVjpG1qN3zpAHjSgoqgUzQ3QOR1p3
pWOZ14I94z1zkLLjkKt05TRsvw8eWJG6a1UCAwEAAaOBnzCBnDAdBgNVHQ4EF
brNsdA==
-----END CERTIFICATE-----
```

As you can see, in each certificate file are two keys. The first is the private key, which is contained within the `BEGIN RSA PRIVATE` and `END RSA PRIVATE KEY` text. The second is the certificate, which is contained within the `BEGIN CERTIFICATE` and `END CERTIFICATE` text.

To authenticate, your server needs the certificate portion of each of your client certificates. So, copy the newly created client certificate, and remove the private key portion leaving the certificate portion. The file will now look like Listing 5-29.

Listing 5-29. *SSL Certificate*

```
-----BEGIN CERTIFICATE-----
MIICoDCCAgmgAwIBAgIBADANBgkqhkiG9w0BAQQFADBFMQswCQYDVQQGE
M7Bfr321osTeeF33aO9z2lMG/iY6C3he27uqih5YAEby4iWmpm8DQokSfI5+hki4
g8Dhmn3wLkDt2LUq1vzaNLMR+sBxI7zlvHdrVjpG1qN3zpAHjSgoqgUzQ3QOR1p3
pWOZ14I94z1zkLLjkKt05TRsvw8eWJG6a1UCAwEAAaOBnzCBnDAdBgNVHQ4EF
brNsdA==
-----END CERTIFICATE-----
```

For ease of management I recommend storing all these certificates in a single file; I call mine `syslog-ng-clients.pem` and simply append the certificate portion of each new client onto the end of the file.

To authenticate on the client end, the client requires a copy of your certificate with the private key and the certificate in it like the original file shown in Listing 5-28. Copy this into the `/etc/stunnel` directory on the client. You also need the certificate portion of the server certificate. So make a copy of your server certificate. Call it `syslog-ng-`*`servername`*`.pubcert`. From the `syslog-ng-`*`servername`*`.pubcert` file, remove the private key portion file and copy the resulting file to the `/etc/stunnel` directory on the client

The following example shows the steps taken to add a new client:

1. Create a client certificate like this:

   ```
   puppy# cd /usr/share/ssl/certs/
   puppy# make syslog-ng-clientname.pem
   ```

2. Append the certificate portion of the new client certificate to the `syslog-ng-clients.pem` file in `/etc/stunnel`.

3. Copy the `/etc/stunnel/syslog-ng-`*`servername`*`.pubcert` file and the `syslog-ng-`*`clientname`*`.pem` file to the new client.

Now that you have keys on the client and server systems, you need to configure Stunnel to read those keys and set up the connections. You do this by creating and editing `stunnel.conf` files, which you should also locate in `/etc/stunnel` on both the client and server systems.

On the server side, your `stunnel.conf` should look like Listing 5-30.

Listing 5-30. *Server-Side* `stunnel.conf` *Configuration*

```
cert = /etc/stunnel/syslog-ng-servername.pem
pid = /var/run/stunnel.pid
# Some debugging stuff
debug = debug
output = /var/log/stunnel.log
# Service-level configuration
CAfile = /etc/stunnel/syslog-ng-clients.pem
verify = 3
[5140]
accept = 5140
connect = 514
```

■ **Tip** I have enabled a fairly high level of logging in Stunnel (which is useful to help diagnose any errors). If you want a lesser amount of logging, then change `debug` to a higher priority (for example, `info`).

The `cert` option defines the certificate for the local system (you would replace *`servername`* with the name of your syslog-NG server), and the `CAfile` option points to the collection of certificates from which this server authorizes connection. The service-level configuration tells Stunnel to accept connections on port 5140 and redirect those connections to port 514 on the local host.

Your `stunnel.conf` should look like Listing 5-31.

Listing 5-31. *Client-Side Configuration*

```
cert = /etc/stunnel/syslog-ng-clientname.pem
pid = /var/run/stunnel.pid
# Some debugging stuff
debug = debug
output = /var/log/stunnel.log
# Service-level configuration
client = yes
CAfile = /etc/stunnel/syslog-ng-servername.pubcert
verify = 3
[5140]
accept = 127.0.0.1:514
connect = syslogserverIP:5140
```

On the client side, the `cert` file defines the certificate for the local system (you would replace *`clientname`* with the name of your client system), and the `CAfile` option points to the certificate of the server (you would replace *`servername`* with the name of your syslog-NG server) to which you want to connect, in this case `syslog-ng-`*`servername`*`.pubcert`. The additional parameter `client` is set to `yes`. This tells Stunnel that this system is a client of a remote Stunnel system. The service-level configuration tells Stunnel to accept connections on IP `127.0.0.1` (`localhost`) at port 514 and redirect those connections to port 5140 on the syslog-NG server. In Listing 5-31 you would replace *`syslogserverIP`* with the IP address of your syslog-NG server.

This sets up Stunnel, and now you need to make some changes to allow syslog-NG to receive your Stunnel'ed traffic. On the syslog-NG server, ensure your `tcp()` source statement in your `syslog-ng.conf` file looks like Listing 5-32.

Listing 5-32. *Server-Side* `syslog-ng.conf` *for Stunnel*

```
source s_tcp { tcp(ip("127.0.0.1") port(514)); };
```

This ensures syslog-NG is checking port 514 on `localhost` or `127.0.0.1` where Stunnel will direct any incoming syslog-NG traffic coming from port 5140.

On the client side, ensure your `syslog-ng.conf` destination and log statements are also updated, as shown in Listing 5-33.

Listing 5-33. *Client-Side* `syslog-ng.conf` *for Stunnel*

```
destination d_secure { tcp("127.0.0.1" port(514)); };
log { source(s_sys); destination(d_secure); };
```

This ensures syslog-NG is logging to port 514 on `localhost` or `127.0.0.1` where Stunnel will redirect that traffic to port 5140 on your syslog-NG server. The `log{}` statement will log everything from source `s_sys` to that destination.

■ **Tip** If you are using Stunnel for secure logging, you need to ensure the `keep_hostname()` option is set to `yes`; otherwise, all the messages will have `localhost` as their hostname.

Now you are almost ready to go. All you need to do is start Stunnel on both your server and client systems. Starting Stunnel is easy. You do not need any options for the `stunnel` binary; however, it is probably a good idea to be sure Stunnel is pointing at the right configuration file.

```
puppy# stunnel /path/to/conf/file
```

Now restart syslog-NG on both your server and client systems, and your logging traffic should now be secured from prying eyes.

Testing Logging with logger

Present on all Linux distributions, `logger` is a useful command-line tool to test your logging configuration. Listing 5-34 demonstrates `logger`.

Listing 5-34. *Running the* `logger` *Command*

```
puppy# logger -p mail.info "This is a test message for facility mail and ➥
priority info"
```

Listing 5-34 would write the message "This is a test message for facility mail and priority info" to your `syslog` or syslog-NG daemon and into whatever destination you have configured for messages with a facility of `mail` and a priority of `info`. As you can see, the `-p` parameter allows you specify a facility and priority combination and then the test message contained in quotation marks.

I often use `logger` inside `bash` scripts to generate multiple messages for testing purposes. The script in Listing 5-35 generates a `syslog` message for every facility and priority combination.

Listing 5-35. *Log Testing* `bash` *Script*

```
#!/bin/bash
for f in
{auth,authpriv,cron,daemon,kern,lpr,mail,mark,news,syslog,user,uucp,local0,➥
local1,local2,local3,local4,local5,local6,local7}
do
for p in {debug,info,notice,warning,err,crit,alert,emerg}
do
logger -p $f.$p "Test syslog messages from facility $f with priority $p"
done
done
```

You can also use `logger` to pipe a growing file into `syslog` or `syslog-NG`. Try the simple script shown in Listing 5-36.

Listing 5-36. *Piping a Growing File into* `syslog`

```
#!/bin/bash
tail -f logfile | logger -p facility.priority
```

This script simply runs `tail -f` on *`logfile`* (replace this with the name of the file you want to pipe into your choice of `syslog` daemon) and pipes the result into `logger` using a facility and priority of your choice. Of course, this script could obviously be greatly expanded in complexity and purpose, but it should give you a start.

Logger works for both `syslog` and syslog-NG.

Log Analysis and Correlation

Many people think log analysis and correlation are "black" arts—log voodoo. This is not entirely true. It can be a tricky art to master, and you need to be constantly refining that art; however, inherently once you implement a systematic approach to it, then it becomes a simple part of your daily systems' monitoring routine.

The first thing to remember is that analysis and correlation are two very different things. *Analysis* is the study of constituent parts and their interrelationships in making up a whole. It must be said that the best analysis tool available is yourself. System administrators learn the patterns of their machines' operations and can often detect a problem far sooner than automated monitoring or alerting systems have done on the same problem. I have two problems with this model. The first problem is that you cannot be everywhere at once. The second problem is that the growing volume of the data collected by the systems can become overwhelming.

This is where correlation comes in. *Correlation* is best defined as the act of detecting relationships between data. You set up tools to collect your data, filter the "wheat from the chaff," and then correlate that remaining data to put the right pieces of information in front of you so you can provide an accurate analysis. Properly setup and managed tools can sort through the constant stream of data that the daily operations of your systems and any attacks on those systems generate. They can detect the relationships between that data and either put those pieces together into a coherent whole or provide you with the right pieces to allow you to put that analysis together for yourself.

But you have to ensure those tools are the right tools and are configured to look for the right things so you can rely on them to tell you that something is wrong and that you need to intervene. As a result of the importance of those tools to your environment, building and implementing them should be a carefully staged process. I will now cover those stages in brief.

The first stage of building such an automated log monitoring system is to make sure you are collecting the right things and putting them in the right place. Make lists of all your applications, devices, and systems and where they log to. Read carefully through the sections in this chapter discussing `syslog` and syslog-NG, and make sure whatever you set up covers your entire environment. Make sure your logging infrastructure encompasses every piece of data generated that may be vital to protecting your systems.

The second stage is bringing together all that information and working out what you really want to know. Make lists of the critical messages that are important to you and your systems. Throw test attacks and systems failures at your test systems, and record the resulting message traffic; also, port scan your systems and firewalls, even unplugging hardware or deliberately breaking applications in a test environment to record the results. Group those lists into priority listings; some messages you may want to be paged for, others can go via e-mail, and some may trigger automated processes or generate attempts at self-recovery such as restarting a process.

The third stage is implementing your log correlation and analysis, including configuring your correlation tools and designing the required responses. Make sure you carefully document each message, the response to the message, and any special information that relates to this message. Then test them. And test them again. And keep testing them. Your logging environment should not be and almost certainly will never be static. You will always discover something new you want to watch for and respond to. Attackers are constantly finding new ways to penetrate systems that generate different data for your logging systems. Attacks are much like viruses—you need to constantly update your definitions to keep up with them.

So where do you go from here? I will now introduce you to a powerful tool that will help you achieve your logging goals. That tool is called SEC.

SEC is the most powerful open-source log correlation tool available.[3] SEC utilizes Perl regular expressions to find the messages that are important to running your system out of the huge volume of log traffic most Linux systems generate. It can find a single message or match pairs of related messages; for example, it can find matching messages that indicate when a user has logged on and off a system. SEC can also keep count of messages it receives and act only if it receives a number of messages exceeding a threshold that you can define. SEC can also react to the messages it receives by performing actions such as running a shell script. These actions can include the content of the messages. For example, it is possible to run a shell script as a SEC action and use some or all of the message content as a variable to be inputted into that shell script.

■ **Note** As a result of SEC's reliance on Perl regular expressions, you need to be reasonably comfortable with using them. The Perl documentation on regular expressions is excellent. Try `http://www.perldoc.com/perl5.6.1/pod/perlre.html` and `http://www.perldoc.com/perl5.8.0/pod/perlretut.html`. Also, I have listed several excellent books on regular expressions in this chapter's "Resources" section.

Seeing all this functionality you may think SEC is overkill for your requirements, but the ability to expand your event correlation capabilities far outweighs the cost of implementation. It is my experience that it is critical in your logging environment to avoid having to make compromises in your monitoring that could cause you to be exposed to vulnerabilities or a potentially missing vital messages. The functionality richness of SEC should be able to cover all your current and future event correlation needs.

Because of SEC's complexity, it is impossible to completely cover all its features within this chapter, so I will avoid discussing some of the more advanced features of SEC, most notably contexts. SEC's full implementation and variables could easily occupy a book in their own right. I will get you started with SEC by showing you how to install it, how to get it running, how to point your logs to SEC, and how set up some basic message-matching rules; then I will point you to the resources you will need to fully enable SEC within your own environment.

■ **Tip** A good place to start learning more about SEC is the mailing list maintained at the SourceForge site for SEC. You can subscribe to the mailing list and read its archives at `http://lists.sourceforge.net/lists/listinfo/simple-evcorr-users`. SEC's author Risto Vaarandi is a regular, active, and helpful participant to this list, and the archives of the list contain many useful examples of SEC rules to help you.

---

3. SEC is written by Risto Vaarandi and supported by Vaarandi's employer, The Union Bank of Estonia. It is free to download and uses the GNU General Public License.

Installing and Running SEC

You can download SEC from `http://kodu.neti.ee/~risto/sec/` in the download section. Installing SEC is a simple process. SEC is a Perl script. To use it, you will need at least Perl version 5.005 installed on your system. (But a more recent version such as 5.6 is strongly recommended.) SEC also relies on the modules `Getopt`, `POSIX`, `Fcntl`, and `IO::Handle`, but these modules are included in the Perl base installation. Then unpack the SEC archive. Inside the archive is the engine of the SEC tool, a Perl script called `sec.pl`. Copy the `sec.pl` script to a directory of your choice. For these purposes, I have copied the `sec.pl` file into a directory that I created called `/usr/local/sec`. SEC also comes with a comprehensive `man` page that you should also install.

You start SEC from the command line by running the `sec.pl` script. Listing 5-37 shows a command line you can use to start SEC.

Listing 5-37. *Sample SEC Startup Options*

```
puppy# /usr/local/sec/sec.pl -input=/var/log/messages ➥
-conf=/usr/local/sec/sec.conf -log=/var/log/sec.log -debug=6 -detach
```

To start SEC, the first option you need is `-input`. Inputs are where you define the source of the messages SEC will be analyzing. You can have multiple input statements on the command line that gather messages from several sources. Listing 5-37 uses one input, `/var/log/messages`.

The next option, `-conf`, tells SEC where to find its configuration file. The configuration file contains all the rules SEC uses to analyze incoming messages. You probably do not have one of these yet, but you can just create an empty file to get started; SEC will start fine just when you use this empty file.

```
puppy# touch /usr/local/sec/sec.conf
```

You can specify more than one configuration file by adding more `-conf` options to the command line. This allows you to have multiple collections of rules for different situations or for different times of the day.

I have also specified some logging for SEC. In Listing 5-37, SEC is logging to the file `/var/log/sec.log` with a debugging level of 6 (the maximum).

The last option, `-detach`, tells SEC to detach and become a daemon.

If you were to run the command in Listing 5-37, it would result in the following being logged to the `sec.log` file in `/var/log`. The last line indicates the start was successful.

```
Fri Mar  5 17:28:09 2004: Simple Event Correlator version 2.2.5
Fri Mar  5 17:28:09 2004: Changing working directory to /
Fri Mar  5 17:28:09 2004: Reading configuration from /usr/local/sec/sec.conf
Fri Mar  5  17:28:09 2004: No valid rules found in configuration ➥
file /usr/local/sec/sec.conf
Fri Mar  5 17:28:09 2004: Daemonization complete
```

SEC is now running as a daemon on the system and awaiting events to process. If you change a rule or add additional rules, you need to restart the SEC process to reload the configuration files.

```
puppy# killall -HUP sec.pl
```

Tip SEC also comes with a file called `sec.startup` that contains an `init` script you can adjust to start SEC automatically when you start your system; this should allow you to easily control reloading and restarting `sec.pl`.

SEC has some additional command-line options to control its behavior and configuration. Table 5-10 covers the important ones.

Table 5-10. *SEC Command-Line Options*

Option	Description
`-input=file pattern[=context]`	The input sources for SEC that can be files, named pipes, or standard input. You can have multiple input statements on your command line. The optional context option will set up a context. Contexts help you to write rules that match events from specific input sources. Note that I do not cover contexts in this chapter.
`-pid=pidfile`	Specifies a file to store the process ID of SEC. You must use this if you want a PID file.
`-quoting` and `-noquoting`	If quoting is turned on, then all strings provided to external shell commands by SEC will be put inside quotes to escape them. The default is not to quote.
`-tail` and `-notail`	These tell SEC what to do with files. If `-notail` is set, then SEC will read any input sources and then exit when it reaches the end of the file or source. If `-tail` is set, then SEC will jump to the end of the input source and wait for additional input as if you had issued the `tail -f` command. The default is `-tail`.
`-fromstart` and `-nofromstart`	These flags are used in combination with `-tail`. When `-fromstart` is enabled, it will force SEC to process input files from start to finish and then go into "tail" mode and wait for additional input. These options obviously have no effect if `-notail` is set. The default option is `-nofromstart`.
`-detach` and `-nodetach`	If you add `-detach` to the command line, SEC will daemonize. The default is `-nodetach` with SEC running in the controlling terminal.
`-testonly` and `-notestonly`	If the `-testonly` option is specified, then SEC will exit immediately after parsing the configuration file(s) for any errors. If the configuration file(s) do not contain any errors, then SEC will exit with an exit code of `0` and otherwise with an exit code of `1`. The default is `-notestonly`.

You can read about additional SEC options on timeouts in input sources in the SEC `man` page.

Inputting Messages to SEC

The easiest way to get messages into SEC is to go through a named pipe. I recommend setting up either `syslog` or syslog-NG to point to a named pipe and inputting your messages to SEC through that pipe. First let's create a named pipe. Create a pipe in the `/var/log` directory called `sec`, like so:

```
puppy# mkfifo /var/log/sec
```

When I start SEC, I would now use this named pipe as an input on the starting command line by adding the option `-input /var/log/sec`.

Now you need to define this pipe to the respective logging daemons. For syslog-NG this is an easy process, as shown in Listing 5-38.

Listing 5-38. *syslog-NG Configuration for SEC*

```
destination d_sec { pipe("/var/log/sec"); };
log { source(s_sys); destination(d_sec); };
log { source(s_tcp); destination(d_sec); };
```

As you can see from Listing 5-37, you define a named pipe destination in syslog-NG, in this case `/var/log/sec`, and then log all the sources you want to this pipe. You can add the statements in Listing 5-38 to the sample configuration in Listing 5-24 to get this working immediately. You will need to restart syslog-NG to update the configuration.

■ **Tip** If you have an especially busy system or one with performance issues, it may be wise to increase the syslog-NG global option `log_fifo_size (`*`num`*`);` (defined in the `options{}` statement block). This controls the number of lines being held in buffer before they are written to disk. This should help prevent overflows and dropped messages if your pipe is slow to process events.

For `syslogd`, the process of getting messages into SEC requires pointing the facilities and priorities you want to input to SEC to a pipe. See Listing 5-39.

Listing 5-39. `syslogd` *Configuration for SEC*

```
*.info | /var/log/sec
```

This example would send all messages of `info` priority or higher from every facility to the named pipe, `/var/log/sec`.

As SEC can also read from files, you could also log the event messages you want to process with SEC to a file or series of files and use those as input sources. For the purpose of this explanation and for ease of use and configuration, I recommend the named pipe method. This is principally because there is no risk of log data being inputted to SEC twice if you accidentally tell SEC to reprocess a log file (which can happen using the `-fromstart` and `-nofromstart` options). Additionally, if you want to specify that messages go to SEC and to a file or database, you are not required to store the data twice. You keep a copy in a file or database, and the other copy goes into the pipe and directly into SEC without being written to disk and therefore taking up disk space.

Building Your SEC Rules

The SEC configuration file contains a series of rule statements. Each rule statement consists of a series of pairs of keys and values separated by an equals (=) sign. There is one key and value pair per line. You can see an example of a key and value pair on the next line:

```
type=Single
```

For the purposes of this explanation, I will call these *key=value pairs*. You can use the backslash (\) symbol to continue a key=value pair onto the next line. You can specify a comment using the pound (#) symbol. SEC assumes that a blank line or comment is the end of the current rule statement, so only add comments or blank lines at the start or end of a rule statement. Let's look now at an example of a rule statement to better understand how SEC and SEC rules work. See Listing 5-40.

Listing 5-40. *Sample SEC Rule Statement*

```
type=Single
continue=TakeNext
ptype=regexp
pattern=STATS: dropped ([0-9]+)
desc=Dropped $1 messages - go check this out
action=shellcmd /bin/echo '$0' | /bin/mail -s "%s" admin@yourdomain.com
```

Let's discuss this example line by line. The first line indicates the type of SEC rule that is being used. In Listing 5-40 I have used the simplest rule, `Single`, which simply finds a message and then executes an action.

The second line in Listing 5-40 is optional. The `continue` line has two potential options, `TakeNext` and `DontCont`. The first option, `TakeNext`, tells SEC that even if the log entry matches this rule, keep searching through the file for other rules that may match the entry. The second option, `DontCont`, tells SEC that if the log entry matches this rule, then stop here and do not try to match the entry against any additional rules. This means that a log entry will be checked against every single rule in your configuration file until it finds a rule it matches that has a `continue` setting of `DontCont`.

This is useful when some messages may be relevant to more than one rule in your configuration file. An example of when you could use this is if a message has more than one implication or purpose. For example, a user login message may be used to record user login statistics, but you may also want to be e-mailed if the `root` user logs on. You would use one rule to record the user statistics that has a `continue` option of `TakeNext`. After processing this rule, the message would be checked against the other rules in the configuration file and would be picked up by the rule that e-mails you if `root` logged on.

■ **Note** If you omit the `continue` option from the rule statement, SEC defaults to `DontCont`.

The next two lines in the rules statement allow SEC to match particular events. The first is `ptype`, or the pattern type. The pattern type tells SEC how to interpret the information on the next line, the pattern itself. You can use the pattern types shown in Table 5-11.

Table 5-11. *SEC Pattern Types*

Pattern Type	Description
`RegExp[`*`number`*`]`	A Perl regular expression.
`SubStr[`*`number`*`]`	A substring.
`NRegExp[`*`number`*`]`	A negated regular expression; the results of the pattern match are negated.
`NSubStr[`*`number`*`]`	A negated substring; the results of the pattern match are negated.

The *`number`* portion after the pattern type tells SEC to compare the rule against the last *number* of log entries. If you leave *`number`* blank, then SEC defaults to `1`—the last log entry received. Listing 5-40 used a standard `regexp` pattern type that tells SEC to interpret the pattern line as a Perl regular expression.

The third line in Listing 5-40, `pattern`, shows the pattern itself. In this example, it is a regular expression. This regular expression would match on any message that consisted of the text `STATS: dropped` and any number greater than one.

You may notice I have placed part of the regular expression, `[0-9]+`, in parentheses. In SEC the content of anything in the `pattern` line that you place in parentheses becomes a variable available to SEC. In this instance, `([0-9]+)` becomes the variable `$1`; any subsequent data enclosed in parentheses would become `$2`, then `$3`, and so on. So if the message being tested against this rule was `STATS: dropped 123`, then the message would be matched and the variable `$1` would be assigned a content of `123`. Another special variable, `$0`, is reserved for the content of the log entry or entries the rule is being tested against. In this example, the variable `$0` would contain the input line `STATS: dropped 123`.

The fourth line in Listing 5-40 shows the `desc` key=value pair. This is a textual description of the event being matched. Inside this description you can use any variables defined in the pattern. Thus, the `desc` for Listing 5-40 is `Dropped $1 messages - go check this out`. Using the message data in the previous paragraph, this would result in a description of `Dropped 123 messages - go check this out`. You will note that I have used the variables, `$1`, that I defined in the `pattern` line in the `desc` line. The final constructed description is also available to you in SEC as the `%s` variable.

The fifth and last line in Listing 5-40 shows the `action` key=value pair. This line tells SEC what to do with the resulting match, log entry, and/or variables generated as a result of the match. In the `action` line, in addition to any variables defined in the pattern (the `$0` variable and the `%s` variable indicating the `desc` line), you also have access to two other internal variables: `%t`, the textual time stamp that is equivalent to the result of the `date` command, and `%u`, the numeric time stamp that is equivalent to the result of the `time` command.

Now you have seen your first SEC rule. It just scrapes the surface of what SEC is capable of doing. Let's look at another example to show you what else SEC is capable of doing. Listing 5-41 uses the `SingleWithThreshold` rule type to identify repeated `sshd` failed login attempts.

Listing 5-41. *Using the* `SingleWithThreshold` *Rule Type*

```
type=SingleWithThreshold
ptype=regexp
pattern=(\w+)\s+sshd\[\d+\]\:\s+Failed password for (\w+) from ➥
(\d+.\d+.\d+.\d+) port \d+ \w+\d+
desc=User $2 logging in from IP $3 to system $1 failed to enter the correct password
thresh=3
window=60
action=write /var/log/badpassword.log %s
```

With this rule I am looking to match variations on the following log entry:

```
Mar 12 14:10:01 puppy sshd[738]: Failed password for bob ➥
from 10.0.0.10 port 44328 ssh2
```

The rule type I am using to do this is called `SingleWithThreshold`. This rule type matches log entries and keeps counts of how many log entries are matched within a particular window of time. The window is specified using the `window` option and is expressed in seconds. In Listing 5-41 it is set to 60 seconds. The window starts counting when SEC first matches a message against that rule. It then compares the number of matches to a threshold, which you can see defined in Listing 5-41 using the `thresh` option as three matches. If the number of matches reaches the threshold within the window of time, then the `action` line is performed. In Listing 5-41 the action I have specified is to write the contents of the `desc` line to the specified file, `/var/log/badpassword.log`, using the `write` action. The `write` action can write to a file, to a named pipe, or to standard output.

So what other rules types are available to you? Well, SEC has a large collection of possible rules that are capable of complicated event correlation. You can see a list of all the other available rules types in Table 5-12.

Table 5-12. *SEC Rule Types*

Rule Type	Description
`SingleWithScript`	Matches an event, executes a script, and then, depending on the exit value of the script, executes a further action.
`SingleWithSuppress`	Matches an event, executes an action immediately, and then ignores any further matching events for *x* seconds.
`Pair`	Has a paired set of matches. It matches an initial event and executes an action immediately. It ignores any following matching events until it finds the paired event and executes another action.
`PairWithWindow`	Also has a paired set of matches. When it matches an initial event, it waits for *x* seconds for the paired event to arrive. If the paired event arrives within the given window, then it executes an action. If the paired event does not arrive within the given window, then it executes a different action.
`SingleWith2Thresholds`	Counts up matching events during *x1* seconds, and if more than the threshold of *t1* events is exceeded, then it executes an action. It then starts to count matching events again, and if the number during *x2* seconds drops below the threshold of *t2*, then it executes another action.

Rule Type	Description
`Suppress`	Suppresses any matching events. You can use this to exclude any events from being matched by later rules. This is useful for removing high-volume low-informational content messages that would otherwise clog SEC.
`Calendar`	Executes an action at a specific time.

So how do you use some of these other rules types? Let's look at some additional examples. Specifically, Listing 5-42 shows using the `Pair` rule type.

Listing 5-42. *Using the* `Pair` *Rule Type*

```
type=Pair
ptype=regexp
pattern=(\w+\s+\d+\s+\d\d:\d\d:\d\d)\s+(\w+)\s+su\(pam_unix\)➥
(\[\d+\])\:\s+session opened for user root by (\w+)\(\w+\=\d+\)
desc=User $4 has succeeded in an su to root at $1 on system $2. ➥
Do you trust user $4?
action=shellcmd /bin/echo '%s' | /bin/mail -s ➥
"SU Session Open Warning" admin@yourdomain.com
ptype2=regexp
pattern2=(\w+\s+\d+\s+\d\d:\d\d:\d\d)\s+$2\s+su\(pam_unix\)➥
$3\:\s+session closed for user root
desc2=Potentially mischievous user %4 has closed their su session at %1 on system %2
action2=shellcmd /bin/echo '%s' | /bin/mail -s ➥
"SU Session Close Warning" admin@yourdomain.com
```

In this example, I am using `Pair` to detect whenever somebody used the `su` command to become `root` on a system and then monitor the log file for when they closed that `su` session. So, I will be looking to match variations of the following two log entries:

```
Mar  6 09:42:55 puppy su(pam_unix)[17354]: session opened for user ➥
root by bob(uid=500)
Mar  6 10:38:13 puppy su(pam_unix)[17354]: session closed for user root
```

The rule type I will use for this is `Pair`, which is designed to detect a matching pair of log entries. You could also use the `PairWithWindow` rule type, which is designed to find a matching pair of log entries within a particular time window much like the `SingleWithThreshold` rule type you saw in Listing 5-41. With the `Pair` rule types you actually define two sets of pattern type and pattern, description, and action items. This is because you are matching two log entries. The second set of items are suffixed with the number `2` and referred to as `ptype2` and `pattern2`, and so on, to differentiate them from the first set. The first set of items are used when the first log entry is matched; for example, the `action` line is executed when the log entry is matched. The second set of items is used if the second log entry is matched; for example, the `action2` line is executed when the second log entry is matched.

For the first set of pattern type and pattern, I have used a regular expression pattern type. Inside the pattern I have also defined a number of elements of the log entry I am seeking to match as variables: the hostname on which the su session took place, the user who used the su command, the time the session opened and closed, and the process ID that issued the su command. You can then see that I have used some of these variables in the desc and action lines. The action I am using in Listing 5-42 is called shellcmd to execute a shell command when a log entry is matched.

The second pattern type will also be a regular expression. In this pattern, how do you know if the log entry indicating the end of the su session is related to the original log entry opening the su session? Well, SEC can use variables from the first pattern line, pattern, and these variables can form part of the regular expression being matched in the second pattern line, pattern2. In the first pattern line I defined the hostname of the system the su session was taking place on as $2 and the process ID of the session as $3. If you refer to those variables in the pattern2 line, then SEC knows you are referring to variables defined in the first pattern line. You use the host name and process ID to match the incoming log entry against the first log entry.

But this raises another question. How does SEC tell the difference between the variables defined in the two pattern lines when you use them in the desc2 line, for example? Well, variables for the first pattern line if you want to use them again in the desc2 or action2 lines are prefixed by %, and variables from the second pattern line are prefixed with $. You can see I have used the $4 variable defined in the first pattern line in the desc2 line by calling it %4.

Another useful rule type is Suppress. Listing 5-43 shows an example of a Suppress rule.

Listing 5-43. *Using the* Suppress *Rule Type*

```
type=Suppress
ptype=regexp
pattern=\w+\s+syslog-ng\[\d+\]\:\s+STATS: dropped \d+
```

Listing 5-43 is designed to suppress the following log entry:

```
Mar 12 01:05:00 puppy syslog-ng[22565]: STATS: dropped 0
```

The Suppress rule type simply consists of the rule type, a pattern type, and a pattern to match. Event suppression is especially useful for stopping SEC processing events you know have no value. You can specify a series of Suppress rules at the start of your configuration file to stop SEC unnecessarily processing unimportant messages. Be careful to be sure you are not suppressing a useful message, and be especially careful not to make your regular expressions too broad and suppress messages you need to see from getting through.

Suppress rules are also a place where you could use the pattern type of Substr. Let's rewrite Listing 5-43 using a substring instead of a regular expression.

```
type=Suppress
ptype=substr
pattern=This message is to be suppressed.
```

To match a log entry to a substring rule, the content of the pattern line must exactly match the content of the log entry. If required in a substring, you can use the backslash constructs \t, \n, \r, and \s to indicate any tabulation, newlines, carriage returns, or space characters.

■**Tip** As special characters are indicated with a backslash in Perl, if you need to use a backslash in a substring or regular expression, you must escape it. For instance in Perl, `\\` denotes a backslash.

The `Suppress` rule type is not the only type of rule that allows you to suppress messages. You can also use the `SingleWithSuppress` rule type. This rule type is designed to match a single log entry, execute an action, and then suppress any other log entries that match the rule for a fixed period defined using the `window` line. This is designed to allow you to enable message compression. Message compression is useful where multiple instances of a log entry are generated but you need to be notified or have an action performed for only the first matched log entry. You can compress 100 messages to one response or action instead of each of the messages generating 100 individual responses or actions. Listing 5-44 shows an example of the `SingleWithSuppress` rule type.

Listing 5-44. *Using the* `SingleWithSuppress` *Rule Type*

```
type=SingleWithSuppress
ptype=RegExp
pattern=(\S+): Table overflow [0-9]+ of [0-9]+ in Table (\S+)
desc=Please check for a table overflow in $2
action=shellcmd notify.sh "%s"
window=600
```

Listing 5-44 uses a regular expression to check for a table overflow message generated by a database. I know this message can be generated hundreds of times in a short period, so I use the `SingleWithSuppress` rule to match only the first log entry and notify a user about the error message. If additional log entries are matched to this rule within the next 600 seconds (as defined using the `window` line), then they are suppressed and no action is performed. If the log entry appears again more than 600 seconds after the first log entry was matched, then another action is generated and all further matching log entries would be suppressed for another 600 seconds. This, for example, could be because the original problem has not been fixed and another notification is needed.

Within the last few examples, you have seen only a couple of SEC's possible actions, `write` and `shellcmd`. Within SEC additional possible actions are available. Table 5-13 describes some key ones table. These actions you can view in the SEC `man` page.

Table 5-13. *SEC Actions*

Action	Description
`assign %letter [text]`	Assigns the content of *text* to a user-defined *%letter* variable. You can use other % variables in your *text*, like those variables defined in your pattern. If you do not provide any *text*, then the value of the variable %s is used.
`event [time] [event text]`	After *time* seconds, a event with the content of [*event text*] is created. SEC treats the [*event text*] string exactly like a log entry and compares it to all rules. If you do not specify any [*event text*], then the value of the %s variable is used. If you specify 0 as [*time*] or omit the value altogether, then it will be created immediately.
`logonly`	The event description is logged to the SEC log file.
`none`	Takes no action.
`spawn shellcmd`	This is identical to the *shellcmd* action, but any standard output from *shellcmd* is inputted to SEC as if it were a log entry and matched against the rules. This is done by generating an *event 0* [*output line*] to each line from standard output. Be careful that the *shellcmd* command being spawned does not output a large volume of data or an endless loop, as SEC will process these results first and thus become locked.

You can put more than one action on an `action` line by separating them with a semicolon. You can see this in the next line:

```
action=shellcmd notify.sh "%s"; write /var/log/output.log %s
```

Here I have combined the `shellcmd` and `write` actions.

Listing 5-45 shows one final example, the `Calendar` rule type. The `Calendar` rule type is constructed differently than the other rule types are constructed.

Listing 5-45. *Using the* `Calendar` *Rule Type*

```
type=Calendar
time=1-59 * * * *
desc=This is an important message SEC needs to check
action=shellcmd purge.sh
```

The `Calender` rule type uses a special line called `time`. The `time` line uses the standard `crontab` format of five fields, separated by whitespace; those fields are minutes, hours, days of the month, months of the year, and weekdays. You can use the `Calendar` rule type to schedule events or kick off log-related processes. I often use `Calendar` events to schedule the clearing and management of files I use during the logging process.

These examples should have provided you with the grounding to start writing your own SEC rules. For further information and assistance with writing SEC rules, check the SEC FAQ and the example at `http://kodu.neti.ee/~risto/sec/FAQ.html` and `http://kodu.neti.ee/~risto/sec/examples.html`, respectively. Also, as mentioned earlier, the SEC mailing list is an excellent source of assistance and information.

Log Management and Rotation

An important part of managing your logging environment is controlling the volume of your log files and keeping your log files to a manageable size.

Tip If you need to store messages for the long term, I recommend you look at logging to a database. I already discussed earlier in this chapter how to set up logging to a database.

This section will cover the process of automating rotating your logs on a daily, weekly, or monthly basis.

Log rotation can be quite complicated to manually script, so I recommend you use the `logrotate` tool. Most Linux distributions come with the `logrotate` tool. Of the common distributions, it is present on all Red Hat variations, Mandrake, Debian, and SuSE, and an e-build exists for it on Gentoo, which can be installed with the following command:

```
puppy# emerge logrotate
```

`logrotate` is simple to configure and relies on `crontab` to run on a scheduled basis. The base `logrotate` configuration is located in `/etc/logrotate.conf` (see Listing 5-46).

Listing 5-46. `logrotate.conf`

```
#log rotation
weekly
# keep old logs
rotate 4
#create new logs
create
#include .d files
include /etc/logrotate.d
```

This simple file contains the global options that `logrotate` uses to handle log files. In this example, all logs files rotate weekly, logs are rotated four times before they are deleted, new log files are created, and the `logrotate` tool checks the `logrotate.d` directory for any new `logrotate` files. You can use other options you can use, as shown in Table 5-14. You can delve into the `logrotate` `man` file for other options.

Table 5-14. `logrotate.conf` *Options*

Option	Description
`daily`	Logs are rotated on a daily basis.
`weekly`	Logs are rotated on a weekly basis.
`monthly`	Logs are rotated on a monthly basis.
`compress`	Old log files are compressed with `gzip`.

(Continues)

Table 5-14. *Continued*

Option	Description
`create` *`mode owner group`*	Creates new log files with a mode in octal form of 0700 and the owner and group (the opposite is `nocreate`).
`ifempty`	Rotates the log file even if it is empty.
`include` *`directory or filename`*	Includes the contents of the listed file and directory to be processed by `logrotate`.
`mail` *`address`*	When a log is rotated out of existence, mail it to *`address`*.
`nomail`	Do not mail the last log to any address.
`missingok`	If the log file is missing, then skip it and move onto the next without issuing an error message.
`nomissingok`	If the log file is missing, issue an error message (the default behavior).
`rotate` *`count`*	Rotate the log files *`count`* times before they are removed. If *`count`* is `0`, then old log files are removed, not rotated.
`size` *`size`*`[`*`M`*`,`*`k`*`]`	Log files are rotated when they get bigger than the maximum *`size`*; *`M`* indicates size in megabytes, and *`k`* indicates size in kilobytes.
`sharedscripts`	Pre- and post-scripts can be run for each log file being rotated. If a log file definition consists of a collection of log files (for example, `/var/log/samba/*`), and *`sharedscripts`* is set, then the pre/post-scripts are run only once. The opposite is *`nosharedscripts`*.

Listing 5-46 shows the last command, `include`, which principally drives `logrotate`. The `logrotate.d` directory included in that example stores a collection of files that tell `logrotate` how to handle your various log files. You can also define additional directories and files and include them in the `logrotate.conf` file to suit your environment. Most distributions, however, use the `logrotate.d` directory and come with a number of predefined files in this directory to handle common log rotations such as `mail`, `cron`, and `syslog` messages. I recommend adding your own `logrotate` files here also. Listing 5-47 shows you one of those files.

Listing 5-47. *Red Hat* `syslog` `logrotate` *File*

```
/var/log/messages /var/log/secure /var/log/maillog /var/log/spooler ➥
/var/log/boot.log /var/log/cron
{
daily
rotate 7
sharedscripts
postrotate
      /bin/kill -HUP 'cat /var/run/syslog-ng.pid 2> /dev/null' 2> /dev/null || true
endscript
}
```

Inside these files you can override most of the global options in `logrotate.conf` to customize your log rotation for individual files or directories. Listing 5-47 first lists all the files I want to rotate. This could also include directories using the syntax `/path/to/log/files/*`. Then enclosed in { } are any options for this particular set of files. In this example I have overridden the global logging options to rotate these files on a daily basis and keep seven rotations of the log files.

Next you are going to run a script. You can run scripts using the `prerotate` command, which runs the script prior to rotating any logs, or using `postrotate`, which runs the script after rotating the log file(s). Listing 5-47 runs a script that restarts syslog-NG after the log file(s) have been rotated. As the option `sharedscripts` is enabled, the script will be run only once no matter how many individual log files are rotated. The script statement is terminated with the `endscript` option.

So how does `logrotate` run? You can have `cron` run `logrotate` at scheduled times, or you can manually run it on the command line. If running on the command line, `logrotate` defaults to a configuration file of `/etc/logrotate.conf`. You can override this configuration file as you can see on the following line:

```
puppy# logrotate /etc/logrotate2.conf
```

`logrotate` also has several command-line options to use, as shown in Table 5-15.

Table 5-15. `logrotate` *Command-Line Options*

Option	Description
`-d`	Debug mode in which no changes will be made to log files; it will output the results of what it may have rotated. Implies `-v` mode also.
`-v`	Verbose mode.
`-f`	Forces a log rotation even if not required.

By default on most systems `logrotate` is run on a daily basis by `cron`, and this is the model I recommend you should use. Check your `cron.daily` directory in `/etc` for a `logrotate` script that should contain something like Listing 5-48.

Listing 5-48. `logrotate cron` *Script*

```
#!/bin/sh
/usr/sbin/logrotate /etc/logrotate.conf
EXITVALUE=$?
if [ $EXITVALUE != 0 ]; then
    /usr/bin/logger -t logrotate "ALERT exited abnormally with [$EXITVALUE]"
fi
exit 0
```

Resources

The following are resources you can use.

Mailing Lists

- **syslog-NG**: `https://lists.balabit.hu/mailman/listinfo/syslog-ng`
- **SEC**: `http://lists.sourceforge.net/lists/listinfo/simple-evcorr-users`

Sites

- **syslog-NG**: `http://www.balabit.com/products/syslog_ng/`
- **Regular expressions**: `http://www.perldoc.com/perl5.8.0/pod/perlretut.html`
    - `http://www.perldoc.com/perl5.6.1/pod/perlre.html`
    - `http://www.weitz.de/regex-coach/`
- **SEC**: `http://kodu.neti.ee/~risto/sec/`
- **Syslog to MySQL**: `http://www.frasunek.com/sources/security/sqlsyslogd/`

Books

- Friedl, Jeffrey E.F. *Mastering Regular Expressions*, Second Edition. Sebastopol, CA: O'Reilly, 2002.
- Good, Nathan A. *Regular Expression Recipes: A Problem-Solution Approach*. Berkeley, CA: Apress, 2004.
- Stubblebine, Tony. *Regular Expression Pocket Reference*. Sebastopol, CA: O'Reilly, 2003.