TOOLS & BEST PRACTICES FOR BASTION HOSTS

Building SECURE SERVERS *with* LINUX

O'REILLY®

MICHAEL D. BAUER

Building SECURE SERVERS with LINUX

Michael D. Bauer

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

CHAPTER 10
System Log Management and Monitoring

Whatever else you do to secure a Linux system, it must have comprehensive, accurate, and carefully watched logs. Logs serve several purposes. First, they help us troubleshoot virtually all kinds of system and application problems. Second, they provide valuable early-warning signs of system abuse. Third, after all else fails (whether that means a system crash or a system compromise), logs can provide us with crucial forensic data.

This chapter is about making sure your system processes and critical applications log the events and states you're interested in and dealing with this data once it's been logged. The two logging tools we'll cover are syslog and the more powerful Syslog-ng ("syslog new generation"). In the monitoring arena, we'll discuss Swatch (the Simple Watcher), a powerful Perl script that monitors logs in real time and takes action on specified events.

syslog

syslog is the tried-and-true workhorse of Unix logging utilities. It accepts log data from the kernel (by way of *klogd*), from any and all local process, and even from processes on remote systems. It's flexible as well, allowing you to determine what gets logged and where it gets logged to.

A preconfigured syslog installation is part of the base operating system in virtually all variants of Unix and Linux. However, relatively few system administrators customize it to log the things that are important for their environment and disregard the things that aren't. Since, as few would dispute, information overload is one of the major challenges of system administration, this is unfortunate. Therefore, we begin this chapter with a comprehensive discussion of how to customize and use syslog.

Configuring syslog

Whenever *syslogd*, the syslog daemon, receives a log message, it acts based on the message's type (or "facility") and its priority. syslog's mapping of actions to facilities

What About klogd?

One daemon you probably won't need to reconfigure but should still be aware of is *klogd*, Linux's kernel log daemon. This daemon is started automatically at boot time by the same script that starts the general system logger (probably */etc/init.d/syslogd* or */etc/init.d/sysklogd*, depending on which Linux distribution you use).

By default, *klogd* directs log messages from the kernel to the system logger, which is why most people don't need to worry about *klogd*: you can control the handling of kernel messages by editing the configuration file for *syslogd*.

This is also true if you use Syslog-ng instead of syslog, but since Syslog-ng accepts messages from a much wider variety of sources, including */proc/kmsg* (which is where *klogd* receives its messages), some Syslog-ng users prefer to disable *klogd*. Don't do so yourself unless you first configure Syslog-ng to use */proc/kmsg* as a source.

klogd can be invoked as a standalone logger; that is, it can send kernel messages directly to consoles or a log file. In addition, if it isn't already running as a daemon, *klogd* can be used to dump the contents of the kernel log buffers (i.e., the most recent kernel messages) to a file or to the screen. These applications of *klogd* are especially useful to kernel developers.

For most of us, it's enough to know that for normal system operations, *klogd* can be safely left alone (that is, left with default settings and startup options—*not* disabled). Just remember that when you use syslog in Linux, all kernel messages are handled by *klogd* first.

and priorities is specified in */etc/syslog.conf*. Each line in this file specifies one or more facility/priority selectors followed by an action; a selector consists of a facility or facilities and a (single) priority.

In the following *syslog.conf* line in Example 10-1, `mail.notice` is the selector and `/var/log/mail` is the action (i.e., "write messages to */var/log/mail*").

Example 10-1. Sample syslog.conf line

```
mail.notice              /var/log/mail
```

Within the selector, `mail` is the facility (message category) and `notice` is the level of priority.

Facilities

Facilities are simply categories. Supported facilities in Linux are *auth*, *auth-priv*, *cron*, *daemon*, *kern*, *lpr*, *mail*, *mark*, *news*, *syslog*, *user*, *uucp*, and *local0* through *local7*. Some of these are self-explanatory, but the following are of special note:

auth
: Used for many security events.

auth-priv
: Used for access-control-related messages.

daemon
: Used by system processes and other daemons.

kern
: Used for kernel messages.

mark
: Messages generated by *syslogd* itself, which contain only a timestamp and the string `--MARK--`; to specify how many minutes should transpire between marks, invoke *syslogd* with the *-m [minutes]* flag.

user
: The default facility when none is specified by an application or in a selector.

ocal7
: Boot messages.

*
: Wildcard signifying "any facility."

none
: Wildcard signifying "no facility."

Priorities

Unlike facilities, which have no relationship to each other, priorities are hierarchical. Possible priorities in Linux are (in increasing order of urgency): *debug*, *info*, *notice*, *warning*, *err*, *crit*, *alert*, and *emerg*. Note that the "urgency" of a given message is determined by the programmer who wrote it; facility and priority are set by the programs that generate messages, not by syslog.

As with facilities, the wildcards * and *none* may also be used. Only one priority or wildcard may be specified per selector. A priority may be preceded by either or both of the modifiers, = and *!*.

If you specify a single priority in a selector (without modifiers), you're actually specifying that priority *plus* all higher priorities. Thus the selector `mail.notice` translates to "all mail-related messages having a priority of *notice* or higher," i.e., having a priority of *notice*, *warning*, *err*, *crit*, *alert*, or *emerg*.

You can specify a single priority by prefixing a = to it. The selector `mail.=notice` translates to "all mail-related messages having a priority of *notice*." Priorities may also be negated: `mail.!notice` is equivalent to "all mail messages except those with priority of *notice*or higher," and `mail.!=notice` corresponds to "all mail messages except those with the priority *notice*."

Actions

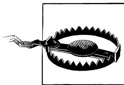
In practice, most log messages are written to files. If you list the full path to a filename as a line's action in *syslog.conf*, messages that match that line will be appended to that file. (If the file doesn't exist, syslog will create it.) In Example 10-1, we instructed syslog to send matched messages to the file */var/log/mail*.

You can send messages other places too. An action can be a file, a named pipe, a device file, a remote host, or a user's screen. Pipes are usually used for debugging. Device files that people use are usually TTYs. Some people also like to send security information to */dev/lp0*—i.e., to a local line printer. Logs that have been printed out can't be erased or altered by an intruder, but they also are subject to mechanical problems (paper jams, ink depletion, etc.) and are harder to parse if you need to find something in a hurry.

Remote logging is one of the most useful features of syslog. If you specify a hostname or IP address preceded by an @ sign as a line's action, messages that match that line will be sent to UDP port 514 on that remote host. For example, the line:

```
*.emerg            @mothership.mydomain.org
```

will send all messages with *emerg* priority to UDP port 514 on the host named *mothership.mydomain.org*. Note that the remote host's (in this example, *mothership's*) *syslogd* process will need to have been started with the `-r` flag for it to accept your log messages. By default, *syslogd* does *not* accept messages from remote systems.



syslog has no access-control mechanism of its own: if you enable the reception of remote messages with the `-r` flag, your host will accept messages on UDP port 514 from any and all remote computers. See the end of this section for some advice on how to mitigate this.

If you run a central log server, which I highly recommend, you'll want to consider some sort of access controls on it for incoming messages. At the very least, you should consider *tcpwrappers*' "hosts access" (source-IP-based) controls or maybe even local firewall rules (*ipchains* or *iptables*).

More sophisticated selectors

You can list multiple facilities separated by commas in a single *syslog.conf* selector. To extend Example 10-1 to include both mail and uucp messages (still with priority *notice* or higher), you could use this line (Example 10-2).

Example 10-2. Multiple facilities in a single selector

```
mail,uucp.notice   /var/log/mail
```

The same is *not* true of priorities. Remember that only one priority or priority wildcard may be specified in a single selector.

Stealth Logging

Lance Spitzner of the Honeynet Project (*http://www.honeynet.org*) suggests a trick that's useful for honey (decoy) nets and maybe even for production DMZs: "stealth logging." This trick allows a host connected to a hub or other shared medium to send its log files to a non-IP-addressed system that sees and captures the log messages but can't be directly accessed over the network, making it much harder for an intruder on your network to tamper with log files.

The idea is simple: suppose you specify a bogus IP address in a *syslog.conf* action (i.e., an IP address that is legitimate for your host's LAN but isn't actually used by any host running *syslogd*). Since syslog messages are sent using the "connectionless" (one-way) UDP protocol, the sending host doesn't expect any reply when it sends a log message.

Furthermore, assuming your DMZ hosts are connected to a shared medium such as a hub, any syslog messages sent over the network will be broadcast on the local LAN. Therefore, it isn't necessary for a central log server on that LAN to have an IP address: the log server can passively "sniff" the log messages via *snort*, *ethereal*, or some other packet sniffer.

Obviously, since an IP-addressless stealth logger won't be accessible via your usual IP-based remote administration tools, you'll need console access to that host to view your logs. Alternatively, you can add a second network interface to the stealth logger, connecting it to a dedicated management network or directly to your management workstation via crossover cable.

In addition to configuring each DMZ host's *syslog.conf* file to log to the bogus IP, you'll also need a bogus ARP entry added to the network startup script on each sending host. If you don't, each system will try in vain to learn the Ethernet address of the host with that IP, and it won't send any log packets.

For example, if you want a given host to pretend to send packets to the bogus IP 192.168.192.168, then in addition to specifying `@192.168.192.168` as the action on one or more lines in */etc/syslog.conf*, you'll need to enter this command from a shell prompt:

```
arp -s 192.168.192.168 03:03:03:31:33:77
```

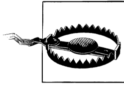
This is not necessary if you send log packets to a "normal" log host (e.g., if 192.168.192.168 is the IP address of a host running *syslogd* with the `-r` flag.)

You may, however, specify multiple selectors separated by semicolons. When a line contains multiple selectors, they're evaluated from left to right: you should list general selectors first, followed by more specific selectors. You can think of selectors as filters: as a message is passed through the line from left to right, it passes first through coarse filters and then through more granular ones.

Continuing our one-line example, suppose we still want important mail and uucp messages to be logged to */var/log/mail*, but we'd like to exclude uucp messages with priority *alert*. Our line then looks like Example 10-3.

Example 10-3. Multiple selectors in a single line

```
mail,uucp.notice;uucp.!=alert    /var/log/mail
```



Actually, *syslogd*'s behavior isn't as predictable as this may imply: listing selectors that contradict each other or that go from specific to general rather than vice versa can yield unexpected results. Therefore, it's more accurate to say "for best results, list general selectors to the left and their exceptions (and/or more-specific selectors) to the right."

Wherever possible, keep things simple. You can use the *logger* command to test your *syslog.conf* rules (see "Testing System Logging with logger" later in this chapter).

Note that in the second selector (`uucp.!=alert`), we used the prefix `!=` before the priority to signify "not equal to." If we wanted to exclude uucp messages with priority *alert* and higher (i.e, *alert* and *emerg*), we could omit the `=` (see Example 10-4).

Example 10-4. Selector list with a less specific exception

```
mail,uucp.notice;uucp.!alert     /var/log/mail
```

You might wonder what will happen to a uucp message of priority *info*: this matches the second selector, so it should be logged to */var/log/mail*, right? Not based on the previous examples. Since the line's first selector matches only mail and uucp messages of priority *notice* and higher, such a message wouldn't be evaluated against the second selector.

There's nothing to stop you from having a different line for dealing with *info*-level uucp messages, though. You can even have more than one line deal with these if you like. Unlike a firewall rule base, each log message is tested against all lines in */etc/syslog.conf* and acted on as many times as it matches.

Suppose we want emergency messages broadcast to all logged-in users, as well as written to their respective application logs. We could use something like Example 10-5.

Example 10-5. A sample syslog.conf file

```
# Sample syslog.conf file that sorts messages by mail, kernel, and "other,"
# and broadcasts emergencies to all logged-in users

# print most sys. events to tty10 and to the xconsole pipe, and emergencies to everyone
kern.warn;*.err;authpriv.none    |/dev/xconsole
*.emerg                           *

# send mail, news (most), & kernel/firewall msgs to their respective logfiles
mail.*                           -/var/log/mail
kern.*                           -/var/log/kernel_n_firewall

# save the rest in one file
*.*;mail.none                    -/var/log/messages
```

Did you notice the - (minus) sign in front of the write-to-file actions? This tells *syslogd* not to synchronize the specified log file after writing a message that matches that line. Skipping synchronization decreases disk utilization and thus improves performance, but it also increases the chances of introducing inconsistencies, such as missing or incomplete log messages, into those files. Use the minus sign, therefore, only in lines that you expect to result in numerous or frequent file writes.

Besides performance optimization, Example 10-5 also contains some useful redundancy. Kernel warnings plus all messages of error-and-higher priority, except *authpriv* messages, are printed to the X-console window. All messages having priority of *emergency* and higher are too, in addition to being written to the screens of all logged-in users.

Furthermore, all mail messages and kernel messages are written to their respective log files. All messages of all priorities (except mail messages of any priority) are written to */var/log/messages*.

Example 10-5 was adapted from the default *syslog.conf* that SuSE 7.1 put on one of my systems. But why shouldn't such a default *syslog.conf* file be fine the way it is? Why change it at all?

Maybe you needn't, but you probably should. In most cases, default *syslog.conf* files either:

- Assign to important messages at least one action that won't effectively bring those messages to your attention (e.g., by sending messages to a TTY console on a system you only access via SSH)
- Handle at least one type of message with too much or too little redundancy to meet your needs

We'll conclude our discussion of *syslog.conf* with Tables 10-1 through 10-4, which summarize *syslog.conf*'s allowed facilities, priorities, and types of actions. Note that numeric codes *should not* be used in *syslog.conf* on Linux systems. They are provided here strictly as a reference, should you need to configure a non-Linux syslog daemon that uses numeric codes (e.g., Cisco IOS), or to send syslog messages to your log server because they're used internally (i.e., in raw syslog packets). You may see them referred to elsewhere.

Table 10-1. syslog.conf's allowed facilities

Facilities	Facility codes
auth	4
auth-priv	10
cron	9
daemon	3
kern	0

Table 10-1. syslog.conf's allowed facilities (continued)

Facilities	Facility codes
lpr	6
mail	2
mark	N/A
news	7
syslog	5
user	1
uucp	8
local{0-7}	16-23
* ("any facility")	N/A

Table 10-2. syslog.conf's priorities

Priorities (in increasing order)	Priority codes
none	N/A
debug	7
info	6
notice	5
warning	4
err	3
crit	2
alert	1
emerg	0
* ("any priority")	N/A

Table 10-3. Use of "!" and "=" as prefixes with priorities

Prefix	Description
*.notice (no prefix)	any event with priority of 'notice' or higher
*.!notice	no event with priority of 'notice' or higher
*.=notice	only events with priority 'notice'
*.!=notice	no events with priority of 'notice'

Table 10-4. Types of actions in syslog.conf

Action	Description
/some/file	Log to specified file
-/some/file	Log to specified file but don't sync afterwards
/some/pipe	Log to specified pipe
/dev/some/tty_or_console	Log to specified console
@remote.hostname.or.IP	Log to specified remote host

Table 10-4. Types of actions in syslog.conf (continued)

Action	Description
username1, username2, etc.	Log to these users' screens
*	Log to all users' screens

Running syslogd

Just as the default *syslog.conf* may or may not meet your needs, the default startup mode of *syslogd* may need tweaking as well. Table 10-5 and subsequent paragraphs touch on some *syslogd* startup flags that are particularly relevant to security. For a complete list, you should refer to the manpage *sysklogd (8)*.

In addition, note that when you're changing and testing *syslog*'s configuration and startup options, it usually makes sense to start and stop *syslogd* and *klogd* in tandem (see the "What About klogd?" sidebar at the beginning of this chapter if you don't know what *klogd* is). Since it also makes sense to start and stop these the same way your system does, I recommend that you use your system's syslog/klogd startup script.

On most Linux systems, both facilities are controlled by the same startup script, named either */etc/init.d/syslog* or */etc/init.d/sysklog* ("sysklog" is shorthand for "syslog and klogd"). See Table 10-5 for a list of some of *syslogd*'s flags.

Table 10-5. Some useful syslogd flags

Flag	Description
`-m` *`minutes_btwn_marks`*	Minutes between "mark" messages (timestamp-only messages that, depending on your viewpoint, either clarify or clutter logs. A value of 0 signifies "no marks").
`-a` *`/additional/socket`*	Used to specify additional sockets, besides */dev/log*, on which *syslogd* should listen for messages.
`-f` *`/path/to/syslog.conf`*	Used to provide the path/name of *syslog.conf*, if different than */etc/syslog.conf*.
`-r`	Listens for *syslog* messages from remote hosts.

The first *syslogd* flag we'll discuss is the only one used by default in Red Hat 7.x in its */etc/init.d/syslog* script. This flag is *-m 0*, which disables *mark* messages. *mark* messages contain only a timestamp and the string `--MARK--`, which some people find useful for navigating lengthy log files. Others find them distracting and redundant, given that each message has its own timestamp anyhow.

To turn *mark* messages on, specify a positive nonzero value after *-m* that tells *syslogd* how many minutes should pass before it sends itself a mark message. Remember that *mark* has its own facility (called, predictably, "mark") and that you must specify at least one selector that matches mark messages (such as *mark.**, which matches all messages sent to the *mark* facility, or **.**, which matches all messages in all facilities).

For example, to make *syslogd* generate *mark* messages every 30 minutes and record them in */var/log/messages*, you would first add a line to */etc/syslog.conf* similar to Example 10-6.

Example 10-6. syslog.conf selector for mark-messages

```
mark.*                                  -/var/log/messages
```

You would then need to start *syslogd*, as shown in Example 10-7.

Example 10-7. Invoking syslogd with 30-minute marks

```
mylinuxbox:/etc/init.d# ./syslogd -m 30
```

Another useful *syslogd* flag is *-a [socket]*. This allows you to specify one or more sockets (in addition to */dev/log* for *syslogd*) from which to accept messages.

In Chapter 6, we used this flag to allow a chrooted *named* process to bounce its messages off of a *dev/log* socket (device-file) in the chroot jail to the nonchrooted *syslogd* process. In that example, BIND was running in a "padded cell" (subset of the full filesystem) and had its own log socket, */var/named/dev/log*. We therefore changed a line in */etc/init.d/syslog* that read as shown in Example 10-8.

Example 10-8. init.d/syslog line invoking syslogd to read messages from a chroot jail

```
daemon syslogd -m 0 -a /var/named/dev/log
```

(Note that the "daemon" function at the beginning of this line is unique to Red Hat's init script functions; the important part here is *syslogd -m 0 -a /var/named/dev/log*.)

More than one *-a* flag may be specified (Example 10-9).

Example 10-9. Invoking syslogd with multiple "additional log device" directives

```
syslogd -a /var/named/dev/log -a /var/otherchroot/dev/log -a /additional/dev/log
```

Continuing down the list of flags in Table 10-5, suppose you need to test a new syslog configuration file named *syslog.conf.test*, but you prefer not to overwrite */etc/syslog.conf*, which is where syslogd looks for its configuration file by default. Use the *-f* flag to tell syslogd to use your new configuration file (Example 10-10).

Example 10-10. Specifying the path to syslogd's configuration file

```
mylinuxbox:/etc/init.d# ./syslogd -f ./syslog.conf.test
```

We've already covered use of the *-r* flag, which tells syslogd to accept log messages from remote hosts, but we haven't talked about the security ramifications of this. On the one hand, security is clearly enhanced when you use a centralized log server or do anything else that makes it easier for you to manage and monitor your logs.

On the other hand, you must take different threat models into account. Are your logs sensitive? If log messages traverse untrusted networks and if the inner workings of the servers that send those messages are best kept secret, then the risks may outweigh the benefit (at least, the specific benefit of syslogd's unauthenticated clear-text remote logging mechanism).

If this is the case for you, skip to this chapter's section on Syslog-ng. Syslog-ng can send remote messages via the TCP protocol and can therefore be used in conjunction with *stunnel*, *ssh*, and other tools that greatly enhance its security. Since syslog uses only the connectionless UDP protocol for remote logging and therefore can't "tunnel" its messages though *stunnel* or *ssh*, syslog is inherently less securable than Syslog-ng.

If your log messages aren't sensitive (at least the ones you send to a remote logger), then there's still the problem of Denial of Service and message forgery attacks. If you invoke *syslogd* with the *-r* flag, it will accept *all* remote messages without performing *any checks whatsoever* on the validity of the messages themselves or on their senders. Again, this risk is most effectively mitigated by using Syslog-ng.

But one tool you *can* use with syslog to partially mitigate the risk of invalid remote messages is TCPwrappers. Specifically, TCPwrappers' "hosts access" authentication mechanism provides a simple means of defining which hosts may connect and via which protocols they may connect to your log server. Hosts-access authentication is easily tricked by source-IP-spoofing (especially since syslog transactions are strictly one way), but it's better than nothing, and it's probably sufficient to prevent mischievous but lazy attackers from interfering with syslog.

If you're willing to bet that it is, obtain and install TCPwrappers and refer to its *hosts_access(5)* manpage for details. Note that despite its name, TCPwrappers' hosts access can be used to control UDP-based applications.

Syslog-ng

As useful and ubiquitous as syslog is, it's beginning to show its age. Modern Unix and Unix-like systems are considerably more complex than they were when syslog was invented, and they have outgrown both syslog's limited facilities and its primitive network-forwarding functionality.

Syslog-ng ("syslog new generation") is an attempt to increase syslog's flexibility by adding better message filtering, better forwarding, and eventually (though not quite yet), message integrity and encryption. In addition, Syslog-ng supports remote logging over both the TCP and UDP protocols. Syslog-ng is the brainchild of and is primarily developed and maintained by Balazs ("Bazsi") Scheidler.

Lest you think Syslog-ng is untested or untrusted, it's already been incorporated into Debian GNU/Linux 2.2 "Potato" as a binary package (in the "admin" section). Syslog-ng is in fact both stable and popular. Furthermore, even though its advanced security features are still works in progress, Syslog-ng can be used in conjunction with TCP "tunneling" tools such as *stunnel* and *ssh* to authenticate or encrypt log messages sent to remote hosts.

Compiling and Installing Syslog-ng from Source Code

The non-Debian users among you may not wish to wait for your distribution of choice to follow suit with its own binary package of Syslog-ng. Let's start, then, with a brief description of how to compile and install Syslog-ng from source.

First, you need to obtain the latest Syslog-ng source code. As of this writing, there are two concurrent branches of Syslog-ng development. Syslog-ng Version 1.4 is the stable branch, so I recommend you use the latest release of Syslog-ng 1.4.

Version 1.5 is the experimental branch, and although it's officially disclaimed as unstable, some people use it on production systems due to its new "field expansion" feature, which allows you to write messages in your own custom formats. If you decide this functionality is worth the risk of running experimental code, be sure to subscribe to the Syslog-ng mailing list (see *http://lists.balabit.hu/mailman/listinfo/syslog-ng* to subscribe).

Speaking of which, it probably behooves you to browse the archives of this mailing list periodically even if you stick to the stable branch of Syslog-ng. Bazsi Scheidler tends to prioritize bug fixes over documentation, so Syslog-ng documentation tends to be incomplete and even out of date.

But Bazsi not only maintains the mailing list, he also very actively participates in it, as do other very knowledgeable and helpful Syslog-ng users and contributors. Thus the mailing list is an excellent source of Syslog-ng assistance. Before posting a question, you may wish to see if anyone else has asked it first. See the Syslog-ng mailing list archives at *http://lists.balabit.hu/pipermail/syslog-ng/*.

Syslog-ng can be downloaded either directly from Bazsi Scheidler's web site at *http://www.balabit.hu* or from its Freshmeat project site at *http://freshmeat.net/projects/syslog-ng/*. In addition to Syslog-ng itself, you'll need the source code for libol, Syslog-ng's support library.

Unzip and untar both archives. Compile and install libol first, then Syslog-ng. For both packages the procedure is the same:

1. Change the working directory to the source's root:

   ```
   cd packagename
   ```

2. Run the source's configure script:

   ```
   ./configure
   ```

3. Build the package:

```
./make
```

4. Install the package:

```
./make install
```

This will install everything in the default locations, which for both libol and Sylog-ng are subdirectories of */usr/local* (e.g., */usr/local/lib*, */usr/local/sbin*, etc.). If you wish to install either package somewhere else—e.g., your home directory (which is not a bad place to test new software)—then in Step 2, pass that directory to *configure* with the *--prefix=* flag as in Example 10-11.

Example 10-11. Telling configure where to install the package

```
mylinuxbox:/usr/src/libol-0.2.23# ./configure --prefix=/your/dir/here
```

After both libol and Syslog-ng have been compiled and installed, you need to set up a few things in Syslog-ng's operating environment. First, create the directory */etc/syslog-ng*. Next, copy one or more of the example *syslog-ng.conf* files into this directory from the source-distribution's *contrib/* and *doc/* directories (unless you intend to create your *syslog-ng.conf* completely from scratch).

Finally, you need to create a startup script for *syslog-ng* in */etc/init.d* and symbolic links to it in the appropriate runlevel directories (for most Linux distributions, */etc/rc2.d*, */etc/rc3.d*, and */etc/rc5.d*). Sample *syslog-ng* init scripts for several Linux distributions are provided in the Syslog-ng source distribution's *contrib/* directory. If you don't find one there that works for you, it's a simple matter to make a copy of your old *syslog* or *sysklogd* init-script and hack it to start *syslog-ng* rather than *syslogd*.

Running syslog-ng

It's premature to start *syslog-ng* before you've created a configuration file. However, since *syslog-ng* has so few startup flags, I'll mention them in brief and spend the remainder of this section on *syslog-ng.conf* use.

The only flags supported by the *syslog-ng* daemon are listed in Table 10-6.

Table 10-6. syslog-ng startup flags

Flag	Description
`-d`	Print debugging messages
`-v`	Print even more debugging messages
`-f filename`	Use *filename* as the configuration file (default=*/etc/syslog-ng/syslog-ng.conf*)
`-V`	Print version number
`-p pidfilename`	Name process-ID-file *pidfilename* (default=*/var/run/syslog-ng.pid*)

In normal use, set these flags in the startup script you installed or created when you installed Syslog-ng, and use that script not only automatically at startup time, but also manually if you need to restart or stop Syslog-ng afterwards.

Configuring Syslog-ng

There's quite a bit more involved in configuring Syslog-ng than with syslog, but that's a symptom of its flexibility. Once you understand how *syslog-ng.conf* works, writing your own configurations is simple, and adapting sample configurations for your own purposes is even simpler. Its main drawback is its sketchy documentation; hopefully, what follows here will mitigate that drawback for you.

By default, Syslog-ng's configuration file is named *syslog-ng.conf* and resides in */etc/syslog-ng/*. Let's dissect a simple example of one in Example 10-12.

Example 10-12. A simple syslog-ng.conf file

```
# Simple syslog-ng.conf file.

options {
    use_fqdn(no);
    sync(0);
    };

source s_sys { unix-stream("/dev/log"); internal(); };
source s_net { udp(); };

destination d_security { file("/var/log/security"); };
destination d_messages { file("/var/log/messages"); };
destination d_console { usertty("root"); };

filter f_authpriv { facility(auth, authpriv); };
filter f_messages { level(info .. emerg)
        and not facility(auth, authpriv); };
filter f_emergency { level(emerg); };

log { source(s_sys); filter(f_authpriv); destination(d_security); };
log { source(s_sys); filter(f_messages); destination(d_messages); };
log { source(s_sys); filter(f_emergency); destination(d_console); };
```

As you can see, a *syslog-ng.conf* file consists of *options{}*, *source{}*, *destination{}*, *filter{}*, and *log{}* statements. Each of these statements may contain additional settings, usually delimited by semicolons.

Syntactically, *syslog-ng.conf* is very similar to C and other structured programming languages. Statements are terminated by semicolons; whitespace is ignored and may therefore be used to enhance readability (e.g., by breaking up and indenting lengthy statements across several lines).

After defining global options, message sources, message destinations, and message filters, combine them to create logging rules.

Global options

Global options are set in *syslog-ng.conf*'s *options{}* section. Some options may be used in the *options{}* section and in one or more other sections. Predictably, options set within *source{}*, *destination{}*, *filter{}*, and *log{}* sections overrule those set in *options{}*. Table 10-7 lists some of the most useful of Syslog-ng's options.

Table 10-7. Syslog-ng options

Option	Description
`schain_hostnames( yes | no )`	After printing the hostname provided by tcp/udp message's sender, show names of all hosts by which a tcp or udp message has been handled (default=yes).
`sskeep_hostname( yes | no )`	Trust hostname provided by tcp/udp message's sender (default=no).
`ssuse_fqdn( yes | no )`	Record full name of tcp/udp message-sender (default=no).
`ssuse_dns( yes | no )`	Resolve IP address of tcp/udp message-sender (default=yes).
`ssuse_time_recvd( yes | no )`	Set message's timestamp equal to time message was received, not time contained in message (default=no).
`sstime_reopen( NUMBER )`	Number of seconds after a tcp connection dies before reconnecting (default=60).
`sstime_reap( NUMBER )`	Number of seconds to wait before closing an inactive file (i.e., an open log file to which no messages have been written for the specified length of time) (default=60).
`sslog_fifo_size( NUMBER )`[a]	Number of messages to queue in memory before processing if *syslog-ng* is busy; note that when queue is full, new messages will be dropped, but the larger the fifo size, the greater *syslog-ng*'s RAM footprint (default=100).
`sssync( NUMBER )`[a]	Number of lines (messages) written to a log file before file is synchronized (default=0).
`ssowner( string )`[a]	Owner of log files *syslog-ng* creates (default=root).
`ssgroup( string )`[a]	Group for log files *syslog-ng* creates (default=root).
`ssperm( NUMBER )`[a]	File-permissions for log files *syslog-ng* creates (default=0600).
`sscreate_dirs( yes | no )`[a]	Whether to create directories specified in destination-file paths if they don't exist (default=no).
`ssdir_owner( string )`[a]	Owner of directories *syslog-ng* creates (default=root).
`ssdir_group( string )`[a]	Group for directories *syslog-ng* creates (default=root).
`ssdir_perm( NUMBER )`[a]	Directory permissions for directories *syslog-ng* creates (default=0700).

[a] These options may also be used in *file()* declarations within *destination{}* statements.

Options that deal with hostnames and their resolution (*chain_hostnames()*, *keep_hostname()*, *use_fqdn()*, and *use_dns*) deal specifically with the hostnames of remote log clients and not with hostnames/IPs referenced in the body of the message.

In other words, if *syslog-ng.conf* on a central log server contains this statement:

```
options { use_dns(yes); };
```

and the remote host *joe-bob*, whose IP address is 10.9.8.7, sends this message:

```
Sep 13 19:56:56 s_sys@10.9.8.7 sshd[13037]: Accepted publickey for ROOT from
10.9.8.254 port 1355 ssh2
```

then the log server will log:

```
Sep 13 19:56:56 s_sys@joebob sshd[13037]: Accepted publickey for ROOT from
10.9.8.254 port 1355 ssh2
```

As you can see, *10.9.8.7* was resolved to *joebob*, but *10.9.8.254* wasn't looked up. (For now you can disregard the `s_sys@` in front of the hostname; I'll explain that shortly.) The *use_dns(yes)* statement applies only to the hostname at the beginning of the message indicating which host sent it; it doesn't apply to other IP addresses that may occur later in the message.

Note also that options related to files and directories may be specified both in the global *options{}* statement and as modifiers to *file()* definitions within *destination{}* statements. *file()* options, when different from their global counterparts, override them. This allows you to create a "rule of thumb" with specific exceptions.

The *chain_hostname()* and *keep_hostname()* options are also worth mentioning. By default, *keep_hostname()* is set to `no`, meaning that *syslog-ng* will not take the hostname supplied by a remote log server at face value; *syslog-ng* will instead resolve the source IPs of packets from that host to determine for itself what that host's name is. This is in contrast to syslog, which takes remote hosts' names at face value.

chain_hostname() determines whether *syslog-ng* should list all hosts through which each message has been relayed. By default, this option is set to `yes`.

Example 10-13 illustrates the effects of *keep_hostname(no)* and *chain_hostname(yes)* (i.e., syslog-ng's default behavior). It shows a log message (in this case, a *syslog-ng* startup notification) being generated locally and then relayed twice. *host1*, who gives its hostname as "linux," generates the message and then sends it to *host2*. *host2* records both "linux" *and* "host1," having double checked that hostname itself via DNS. Finally, the message is relayed to *host3*.

Example 10-13. A log message relayed from one host to two others

```
Original log entry on host1:
Sep 19 22:57:16 s_loc@linux syslog-ng[1656]: syslog-ng version 1.4.13 starting

Entry as sent to and recorded by host2:
Sep 19 22:57:16 s_loc@linux/host1 syslog-ng[1656]: syslog-ng version 1.4.13 starting
```

Example 10-13. A log message relayed from one host to two others (continued)

```
Same log entry as relayed from host2 to host3:
Sep 19 22:57:16 s_loc@linux/host1/host2 syslog-ng[1656]: syslog-ng version 1.4.13 starting
```

There are several interesting things to note in this example. First, you can see that in the second entry (the one logged by *host2*), Syslog-ng does *not* clearly indicate that "linux" is actually *host1*—it simply adds the "real" hostname after the "fake" one in the slash-delimited hostname chain.

Second, the timestamp is *identical* in all three log entries. It's unlikely that three hosts would be in sync to the millisecond *and* be able to relay log messages amongst themselves virtually instantaneously. In fact, the timestamp given to the message by the originating host (*host1* here) is preserved on each host to which the message is relayed, unless a host has its own *use_time_recd()* option set to "yes" (which causes *syslog-ng* to replace message-provided timestamps with the time at which the message was received locally).

Finally, Example 10-13 also shows that when *host1* created the message, it (actually its local *syslog-ng* process) appended *s_loc*, to the message—this is the label of the *source{}* on *host1* from which the local *syslog-ng* process received the message. Example 10-14 lists *host1*'s *syslog-ng.conf* file, the one responsible for the first entry shown in Example 10-13.

Example 10-14. host1's syslog-ng.conf file

```
options { };
source s_loc { unix-stream("/dev/log"); internal(); };
destination d_host2 { udp("host2" port(514)); };
destination d_local { file("/var/log/messages"); };
log { source(s_loc); source(s_net); destination(d_host2); destination(d_local); };
```

Which brings us to the next topic: Syslog-ng message sources.

Sources

The *syslog-ng.conf* file listed in Example 10-14 contains one *source{}* definition, which itself contains two source "drivers" (message-inputs). *syslog-ng.conf* may contain many *source{}* definitions, each of which may, in turn, contain multiple drivers. In other words, the syntax of source definitions is as follows:

```
source sourcelabel { driver1( [options] ); driver2( [options] ); etc. };
```

where *sourcelabel* is an arbitrary string used to identify this group of inputs, and where *driver1()*, *driver2()*, etc. are one or more source drivers that you wish to treat as a single group.

Let's take a closer look at the source definition in Example 10-14:

```
source s_loc { unix-stream("/dev/log"); internal(); };
```

This line creates a source called *s_loc* that refers to messages obtained from */dev/log* (i.e., the local system-log socket) and from the local *syslog-ng* process.

Syslog-ng is quite flexible in the variety of source drivers from which it can accept messages. In addition to Unix sockets (e.g., */dev/log*), *syslog-ng* itself, and UDP streams from remote hosts, Syslog-ng can accept messages from named pipes, TCP connections from remote hosts, and special files (e.g., */proc* files). Table 10-8 lists Syslog-ng's supported source drivers.

Table 10-8. Source drivers for Syslog-ng

Source	Description
`internal()`	Messages from the *syslog-ng* daemon itself.
`file("filename" [options])`	Messages read from a special file such as */proc/kmsg*.
`pipe("filename" )`	Messages received from a named pipe.
`unix_stream("filename" [options])`	Messages received from Unix sockets that can be read from in the connection-oriented stream mode—e.g., */dev/log* under kernels prior to 2.4; the maximum allowed number of concurrent stream connections may be specified (default=100).
`unix_dgram("filename" [options])`	Messages received from Unix sockets that can be read from in the connectionless datagram mode—e.g. *klogd* messages from */dev/log* under kernel 2.4.x.
`tcp([ip(address)] [port(#)] [max-connections(#)] )`	Messages received from remote hosts via the tcp protocol on the specified TCP port (default=514) on the specified local network interface (default=all); the maximum number of concurrent TCP connections may be specified (default=10).
`udp([ip(address)] [port(#)] )`	Messages received from remote hosts via the udp protocol on the specified UDP port (default=514) on the specified local network interface (default=all).

As we just saw in Example 10-14, *internal()* is *syslog-ng* itself: *syslog-ng* sends itself startup messages, errors, and other messages via this source. Therefore, you should include *internal()* in at least one *source{}* definition. *file()* is used to specify special files from which *syslog-ng* should retrieve messages. The special file you'd most likely want *syslog-ng* to read messages from is */proc/kmsg*.

Note, however, that *file()* is *not* intended for use on regular text files. If you wish *syslog-ng* to "tail" dynamic log files written by other applications (e.g., *httpd*), you'll need to write a script that pipes the output from a *tail -f [filename]* command to *logger*. (For instructions on using *logger*, see the section "Testing System Logging with logger" later in this chapter.)

unix_stream() and *unix_dgram()* are important drivers: these read messages from connection-oriented and connectionless Unix sockets, respectively. As noted at the end of "Compiling and Installing Syslog-ng from Source Code," Linux kernels Versions 2.4.1 and higher use Unix datagram sockets; if you specify */dev/log* as a *unix_stream()* source, kernel messages won't be captured. Therefore, use *unix_dgram()* when defining your local-system log source, e.g.:

```
source s_loc { unix-dgram("/dev/log"); internal(); };
```

If your kernel is pre-2.4.0, you should instead use *unix_stream()* for */dev/log*.

tcp() and *udp()* read messages from remote hosts via the connection-oriented TCP protocol and the connectionless UDP protocol, respectively. In both *tcp()* and *udp()*, a listening address and a port number may be specified. By default, *syslog-ng* listens on 0.0.0.0:514—that is, "all interfaces, port 514." (Specifically, the default for *tcp()* is 0.0.0.0:TCP514, and for *udp()*, that is 0.0.0.0:UDP514.)

Example 10-15 shows source statements for *tcp()* and *udp()*, with IP and port options defined.

Example 10-15. tcp() and udp() sources

```
source s_tcpmessages { tcp( ip(192.168.190.190) port(10514) ); };
source s_udpmessages { udp(); };
```

In Example 10-15, we're defining the source *s_tcpmessages* as all messages received on TCP port 10514, but only on the local network interface whose IP address is 192.168.190.190. The source *s_udpmessages*, however, accepts all UDP messages received on UDP port 514 on all local network interfaces.

Besides *ip()* and *port()*, there's one more source option I'd like to cover. *max_connections()*, which can only be used in *tcp()* and *unix_stream()* sources, restricts the number of simultaneous connections from a given source that *syslog-ng* will accept. This is a tradeoff between security and performance: if this number is high, then few messages will be dropped when the server is under load, but at the expense of resources. If this number is low, the chance that logging activity will bog down the server is minimized, but whenever the number of maximum connections is reached, messages will be dropped until a connection is freed up.

The correct syntax for *max-connections()* is simple: specify a positive integer between the parentheses. For example, let's adapt the *tcp()* source from Example 10-15 to accept a maximum of 100 concurrent TCP connections from remote hosts:

```
source s_tcpmessages { tcp( ip(192.168.190.190) port(10514) max-connections(100) );
};
```

By default, *max-connections()* is set to 100 for *unix-stream()* sources and 10 for *tcp()* sources.

By the way, TCP port 514 is the default listening port not only for *syslog-ng*, but also for *rshd*. This isn't a big deal, for the simple reason that *rshd* has no business running on an ostensibly secure Internet-accessible system. If, for example, you wish to use both *syslog-ng* and *rshd* on an intranet server (even then I recommend *sshd* instead), then you should specify a different (unused) port for *syslog-ng* to accept TCP connections on.

Destinations

syslog-ng can be configured to send messages to the same places syslog can: ASCII files, named pipes, remote hosts via UDP, and TTYs. In addition, *syslog-ng* can send

messages to Unix sockets, remote hosts via TCP, and to the standard inputs of programs. Table 10-9 lists the allowed destination types (called "drivers") in Syslog-ng.

Table 10-9. Supported destination drivers in syslog-ng.conf

Driver	Description
`file("`*`filename[$MACROS]`*`" )`	Write messages to standard ASCII-text log file. If file doesn't exist, *syslog-ng* will create it. Macros may be used within or in lieu of a filename; these allow dynamic naming of files (see Table 10-10).
`tcp("`*`address`*`" `*`[`*`port(#);`*`]`*` )`	Transmit messages via TCP to the specified TCP port (default=514) on the specified IP address or hostname. (You must specify an address or name.)
`udp("`*`address`*`" `*`[`*`port(#);`*`]`*` )`	Transmit messages via UDP to the specified UDP port (default=514) on the specified IP address or hostname. (You must specify an address or name.)
`pipe("`*`pipename`*`")`	Send messages to a named pipe such as */dev/xconsole*.
`unix_stream("`*`filename`*`" `*`[options]`*`)`	Send messages in connection-oriented stream mode to a Unix socket such as */dev/log*.
`unix_dgram("`*`filename`*`" `*`[options]`*`)`	Send messages in connectionless datagram mode to a Unix socket such as */dev/log*.
`usertty( `*`username`*` )`	Send messages to specified user's console.
`program("`*`/path/to/program`*`")`	Send messages to standard input of specified program with specified options.

As with ordinary syslog, the most important type of destination is *file()*. Unlike with syslog, Syslog-ng supports filename-expansion macros and a number of options that give one much more granular control over how log files are handled.

When you specify the name of a file for *syslog-ng* to write messages to, you may use macros to create all or part of the filename. For example, to tell *syslog-ng* to write messages to a file whose name includes the current day, you could define a destination like this:

```
destination d_dailylog { file("/var/log/messages.$WEEKDAY"); };
```

When Syslog-ng writes to this particular destination, it will use the filename */var/log/messages.Tues*, */var/log/messages.Wed*, etc., depending on what day it is. See Table 10-10 for a complete list of supported filename macros.

Table 10-10. Macros supported in file() destinations

Macro	Expands to
`PROGRAM`	The name of the program that sent the message
`HOST`	The name of the host that originated the message
`FACILITY`	The facility to which the message was logged

Table 10-10. Macros supported in file() destinations (continued)

Macro	Expands to
PRIORITY *or* LEVEL *(synonyms)*	The designated priority level
YEAR	The current year[a]
MONTH	The current month[a]
DAY	The current day[a]
WEEKDAY	The current day's name (Monday, etc.)[a]
HOUR	The current hour[a]
MIN	The current minute[a]
SEC	The current second[a]

[a] If the global option *use_time_recvd()* is set to `yes`, then this macro's value will be taken from the local system time when the message was received; otherwise, for messages from remote hosts, the timestamp contained in the message will be used.

As with syslog, if a file specified in a *file()* destination doesn't exist, *syslog-ng* will create it. Unlike syslog, Syslog-ng has a number of options that can be implemented both globally and on a per-log-file basis. (Global settings are overridden by per-log-file settings, allowing you to create "general rules" with exceptions.)

For example, whether and how *syslog-ng* creates new directories for its log files is controlled via the options *create_dirs()*, *dir_owner()*, *dir_group()*, and *dir_perm()*. Example 10-16 illustrates the use of these options within a *destination{}* statement.

Example 10-16. Controlling a file() destination's directory-creating behavior

```
destination d_mylog { file("/var/log/ngfiles/mylog" create_dirs(yes) dir_owner(root) \
dir_group(root) dir_perm(0700)); };
```

Example 10-16 also happens to show the default values of the *dir_owner*, *dir_group()*, and *dir_perm()* options. While this may seem unrealistic (why would anyone go to the trouble of setting an option to its default?), it's necessary if nondefaults are specified in a global *options{}* statement and you want the default values used for a specific file—remember, options set in a *destination{}* statement override those set in an *options{}* statement.

Other global/file-specific options can be used to set characteristics of the log file itself: *owner()*, *group()*, and *perm()*, which by default are set to *root*, *root*, and *0600*, respectively. In case you're wondering, there is no *create_file()* option—*syslog-ng* has the irrevocable ability to create files (unless that file's path includes a nonexistent directory and *create_dirs()* is set to `no`). Example 10-17 shows a destination definition that includes these options.

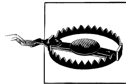
Example 10-17. Options that affect file properties

```
destination d_micklog { file("/var/log/micklog" owner(mick) group(wheel) perm(0640)); };
```

The other *file()* option we'll cover here is *sync()*, which can be used to limit the frequency with which log files are synchronized. This is analogous to syslog's "-" prefix, but much more granular: whereas the "-" merely turns off synchronization, *file()* accepts a numeric value that delays synchronization to as many or as few messages as you like.

The higher the value, the more messages are cached prior to filesystem synchronization and, therefore, the fewer "open for read" actions on the filesystem. The lower the number, the lower the chances of data loss and the lower the delay between a message being processed and written to disk.

By default, *sync()* is set to zero, meaning "synchronize after each message." In general, the default or a low *sync()* value is preferable for low-volume scenarios, but numbers in the 100s or even 1,000s may be necessary in high-volume situations. A good rule of thumb is to set this value to the approximate number of log-message lines per second your system must handle at peak loads.



If you use a log monitor such as Swatch (described later in this chapter) to be alerted of attacks in progress, don't set *sync()* too high. If an intruder deletes a log file, all of Syslog-ng's cached messages will be lost without having been parsed by the log monitor. (Log monitors parse messages as they are written, not beforehand.)

Filters

And now we come to some of the serious magic in Syslog-ng: message filters. Filters, while strictly optional, allow you to route messages based not only on priority/level and facility (which syslog can do), but also on the name of the program that sent the message, the name of the host that forwarded it over the network, a regular expression evaluated against the message itself, or even the name of another filter.

A *filter{}* statement consists of a label (the filter's name) and one or more criteria connected by operators (*and*, *or*, and *not* are supported). Table 10-11 lists the different types of criteria that a *filter{}* statement may contain.

Table 10-11. filter{} functions

Function (criterion)	Description
`facility( `*facility-name*` )`	Facility to which the message was logged (see Table 10-1 for facility names).

`priority( `*priority-name*` )`
`priority( `*priority-name1, priority-name2, etc.*` )`
`priority( `*priority-name1*` .. `*priority-name2*` )`
	Priority assigned to the message (see Table 10-2 for priority-names); a list of priorities separated by commas may be specified, or a range of priorities expressed as two priorities (upper and lower limits) separated by two periods.

`level( `*priority-name*` )`	Same as *priority()*.
`program( `*program-name*` )`	Program that created the message.
`host( `*hostname*` )`	Host from which message was received.

Table 10-11. filter{} functions (continued)

Function (criterion)	Description
match(*regular-expression*)	Regular expression to evaluate against the message's body.
filter(*filter-name*)	Other filter to evaluate.

Example 10-18 shows several *filter{}* statements taken from the default *syslog-ng.conf* file included in Debian 2.2's syslog-ng package.

Example 10-18. Filters

```
filter f_mail { facility(mail); };
filter f_debug { not facility(auth, authpriv, news, mail); };
filter f_messages { level(info .. warn) and not facility(auth, authpriv, cron, daemon,
mail, news); };
filter f_cother { level(debug, info, notice, warn) or facility(daemon, mail); };
```

The first line in Example 10-17, `filter f_mail`, matches all messages logged to the *mail* facility. The second filter, *f_debug*, matches all messages not logged to the *auth*, *authpriv*, *news*, and *mail* facilities.

The third filter, *f_messages*, matches messages of priority levels *info* through *warn*, except those logged to the *auth*, *authpriv*, *cron*, *daemon*, *mail*, and *news* facilities. The last filter, called *f_cother*, matches all messages of priority levels *debug*, *info*, *notice*, and *warn*, and also all messages logged to the *daemon* and *mail* facilities.

When you create your own filters, be sure to test them using the *logger* command. See the section entitled "Testing System Logging with logger" later in this chapter.

Log statements

Now we combine the elements we've just defined (sources, filters, and destinations) into *log{}* statements. Arguably, these are the simplest statements in *syslog-ng.conf*: each consists only of a semicolon-delimited list of *source()*, *destination()*, and, optionally, *filter()* references. (Filters are optional because a *log{}* statement containing only *source()* and *destination()* references will send all messages from the specified sources to all specified destinations.)

Elements from several previous examples are combined in Example 10-19, which culminates in several *log{}* statements.

Example 10-19. Another sample syslog-ng.conf file

```
source s_loc { unix-stream("/dev/log"); internal(); };
source s_tcpmessages { tcp( ip(192.168.190.190); port(10514);); };

destination d_dailylog { file("/var/log/messages.$WEEKDAY"); };
destination d_micklog { file("/var/log/micklog" owner(mick) perm(0600)); };

filter f_mail { facility(mail); };
```

Example 10-19. Another sample syslog-ng.conf file (continued)

```
filter f_messages { level(info .. warn) and not facility(auth, authpriv, cron, daemon,
mail, news); };

log { source(s_tcpmessages); destination(d_micklog); };
log { source(s_loc); filter(f_mail); destination(d_micklog); };
log { source(s_loc); filter(f_messages); destination(d_dailylog); };
```

As you can see in this example, all messages from the host 192.168.190.190 are written to the log file */var/log/micklog*, as are all local mail messages. Messages that match the *f_messages()* filter are written to the log file */var/log/messages.$WEEKDAY*, e.g., */var/log/Sun*, */var/log/Mon*, etc.

Example 10-19 isn't very realistic, though: no nonmail messages with priority-level higher than *warn* are dealt with. This begs the question, "Can I get *syslog-ng* to filter on 'none of the above?'" The answer is yes: to match all messages that haven't yet matched filters in previous *log{}* statements, you can use the built-in filter *DEFAULT*. The following line, if added to the bottom of Example 10-18, will cause all messages not processed by any of the prior three *log{}* statements to be written to the daily log file:

```
log { source(s_loc); filter(DEFAULT); destination(d_dailylog); };
```

Advanced Configurations

As you're hopefully convinced of by this point, Syslog-ng is extremely flexible, so much so that it isn't feasible to illustrate all possible Syslog-ng configurations. I would be remiss, however, if I didn't list at least one advanced *syslog-ng.conf* file.

Example 10-20 shows a setup that causes *syslog-ng* to watch out for login failures and access denials by matching messages against a regular expression and then sending the messages to a shell script (listed in Example 10-21).

Example 10-20. Using syslog-ng as its own log watcher

```
# WARNING: while this syslog-ng.conf file is syntactically correct and complete, it is
# intended for illustrative purposes only -- entire categories of message
# are ignored!

source s_local { unix_stream("dev/log"); internal(); };
filter f_denials { match("[Dd]enied|[Ff]ail"); };
destination d_mailtomick { program("/usr/local/sbin/mailtomick.sh"); };
log { source(s_local); filter(f_denials); destination(d_mailtomick); };
```

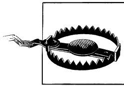
Example 10-21. Script for emailing log messages

```
#!/bin/bash
# mailtomick.sh
# Script which listens for standard input and emails each line to mick
#
```

Example 10-21. Script for emailing log messages (continued)

```
while read line;
do
echo $line | mail -s "Weirdness on that Linux box" mick@pinheads-on-ice.net
done
```

The most important lines in Example 10-20 are the filter *f_denials* and the destination *d_mailtomick*. The filter uses a *match()* directive containing a regular expression that matches the strings "denied," "Denied," "Fail," and "fail."* The destination *d_mailtomick* sends messages via a *program()* declaration to the standard input of a script I wrote called */usr/local/sbin/mailtomick.sh*.

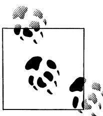


Before we go further in the analysis, here's an important caveat: *program()* opens the specified program once and leaves it open until syslog-ng is stopped or restarted. Keep this in mind when deciding whether to use *pipe()* or *program()* (i.e., *pipe()* doesn't do this), and in choosing what sort of applications you invoke with *program()*.

In some cases, keeping a script open (actually a *bash* process) is a waste of resources and even a security risk (if you run *syslog-ng* as *root*). Furthermore, the particular use of email in Examples 10-19 and 10-20 introduces the possibility of Denial of Service attacks (e.g., filling up the system administrator's mailbox). But under the right circumstances, such as on a non-Internet-accessible host that has a few CPU cycles to spare, this is a legitimate use of Syslog-ng.

The script itself, */usr/local/sbin/mailtomick.sh*, simply reads lines from the standard input and emails each line to *mick@pinheads-on-ice.net*. Since *syslog-ng* needs to keep this script open, the *read* command is contained in an endless loop. This script will run until the *syslog-ng* process that invoked it is restarted or killed.

In the interest of focusing on the most typical uses of Syslog-ng, I've listed some *syslog-ng.conf* options without giving examples of their usage and omitted a couple of other options altogether. Suffice it to say that the global/file option *log_fifo_size()* and the global options *time_reap()*, *time_reopen()*, *gc_idle_threshold()*, and *gc_busy_threshold()* are useful for tuning *syslog-ng*'s performance to fit your particular environment.



The official (maintained) documentation for Syslog-ng is the *Syslog-ng Reference Manual*. PostScript, SGML, HTML, and ASCII text versions of this document are included in the */doc* directory of Syslog-ng's source-code distribution.

For advanced or otherwise unaddressed issues, the best source of Syslog-ng information is the Syslog-ng mailing list and its archives. See *http://lists.balabit.hu/mailman/listinfo/syslog-ng* for subscription information and archives.

* If you're completely new to regular expressions, I highly recommend *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly).

Testing System Logging with logger

Before we leave the topic of system-logger configuration and use, we should cover a tool that can be used to test your new configurations, regardless of whether you use syslog or Syslog-ng: *logger*. *logger* is a command-line application that sends messages to the system logger. In addition to being a good diagnostic tool, *logger* is especially useful for adding logging functionality to shell scripts.

The usage we're interested in here, of course, is diagnostics. It's easiest to explain how to use *logger* with an example.

Suppose you've just reconfigured syslog to send all daemon messages with priority "warn" to */var/log/warnings*. To test the new *syslog.conf* file, you'd first restart *syslogd* and *klogd* and then you'd enter a command like the one in Example 10-22.

Example 10-22. Sending a test message with logger

```
mylinuxbox:~# logger -p daemon.warn "This is only a test."
```

As you can see, logger's syntax is simple. The *-p* parameter allows you to specify a *facility.priority* selector. Everything after this selector (and any other parameters or flags) is taken to be the message.

Because I'm a fast typist, I often use *while…do…done* statements in interactive *bash* sessions to run impromptu scripts (actually, just complex command lines). Example 10-23's sequence of commands works interactively or as a script.

Example 10-23. Generating test messages from a bash prompt

```
mylinuxbox:~# for i in {debug,info,notice,warning,err,crit,alert,emerg}
> do
> logger -p daemon.$i "Test daemon message, level $I"
> done
```

This sends tests messages to the daemon facility for each of all eight priorities.

Example 10-24, presented in the form of an actual script, generates messages for *all* facilities at each priority level.

Example 10-24. Generating even more test messages with a bash script

```
#!/bin/bash
for i in {auth,auth-priv,cron,daemon,kern,lpr,mail,mark,news,syslog,user,uucp,local0,
local1,local2,local3,local4,local5,local6,local7}       # (this is all one line!)

do
for k in {debug,info,notice,warning,err,crit,alert,emerg}
do
logger -p $i.$k "Test daemon message, facility $i priority $k"
done
done
```

Logger works with both syslog and Syslog-ng.

Managing System-Log Files

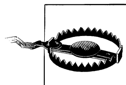
Configuring and fine-tuning your system-logging facilities is extremely important for system security and general diagnostics. But if your logs grow too large and fill up their filesystem, all that work may come to naught.

As with syslog itself, most Linux distributions come with a preconfigured log-rotation scheme. As with syslog, while this scheme tends to work adequately for many users, it's too important a mechanism to take for granted: it behooves you to understand, periodically evaluate, and, if necessary, customize your log-management setup.

Log Management in Red Hat 7 and Debian 2.2: /sbin/logrotate

Both Red Hat 7 and Debian 2.2 use a binary program called *logrotate* to handle system-log growth. In fact, they use very similar implementations of *logrotate*: global options and low-level (system) log files are addressed in */etc/logrotate.conf*, and application-specific configuration scripts are kept in */etc/logrotate.d/*.

When *logrotate* is run, all scripts in */etc/logrotate.d* are included into *logrotate.conf* and parsed as one big script. This makes *logrotate*'s configuration very modular: when you install an RPM or DEB package (of software that creates logs), your package manager automatically installs a script in */etc/logrotate.d*, which will be removed later if you uninstall the package.



Actually, the *include* directive in *logrotate.conf* may be used to specify additional or different directories and files to include. In no event, however, should you remove the statement that includes */etc/logrotate.d* if you use Red Hat or Debian, both of whose package managers depend on this directory for package-specific log-rotation scripts.

Syntax of logrotate.conf and its included scripts

There are really only two types of elements in *logrotate.conf* and its included scripts: directives (i.e., options) and log-file specifications. A directive is simply a parameter or a variable declaration; a log-file specification is a group of directives that apply to a specific log file or group of log files.

In Example 10-25, we see a simple */etc/logrotate.conf* file.

Example 10-25. Simple logrotate.conf file

```
# Very simple logrotate.conf file

# Global options: rotate logs monthly, saving four old copies and sending
# error-messages to root. After "rotating out" a file, touch a new one

monthly
rotate 4
errors root
create

# Keep an eye on /var/log/messages
/var/log/messages {
   size 200k
   create
   postrotate
      /bin/kill -HUP `cat /var/run/syslog-ng.pid 2> /dev/null` 2> /dev/null || true
   endscript
}
```

In Example 10-25, the global options at the top may be thought of as the default log-file specification. Any directive for a specific log file takes precedence over the global options. Accordingly, we see in this example that although by default logs are rotated once a month and that four archives will be kept, the file */var/log/messages* will be rotated not on the basis of time, but on size.

However, the other global directives will still apply: four old copies will be kept; immediately after a log is renamed (which is how they're "rotated"), a newly empty current log file will be created ("touched"); and error messages will be emailed to *root*.

logrotate supports a large number of different directives, but in practice, you'll probably spend more time tweaking the subscripts placed in *logrotate.d* than you will writing scripts from scratch. With that in mind, Table 10-12 lists some commonly encountered *logrotate* directives. A complete list is provided in the manpage *logrotate(8)*.

Table 10-12. Common logrotate directives

Directive	Description

`/path/to/logfile {`
`directive1`
`directive2`
`etc.`
`}`
	Log file specification header/footer (i.e., "apply these directives to the file */path/to/logfile*"). Whitespace is ignored.
	Applicable global directives are also applied to the log file, but when a given directive is specified both globally and locally (within a log file specification), the local setting overrules the global one.

rotate *number*	Tells `logrotate` to retain *number* old versions of the specified log file. Setting this to zero amounts to telling `logrotate` to overwrite the old log file.

Table 10-12. Common logrotate directives (continued)

Directive	Description
daily | weekly | monthly | size=*number_bytes*	The criterion for rotating the specified file: either because one day or week or month has passed since the last rotation, or because the file's size has reached or exceeded *number_bytes* since the last time *logrotate* was run. Note that if *number_bytes* is a number, bytes are assumed; if expressed as a number followed by a lowercase "k," Kilobytes are assumed; if expressed as a number followed by a capital "M," Megabytes are assumed.
mail *[username|mail@address]*	Email old files to the specified local user or email address rather than deleting them.
errors *[username|email@address]*	Email *logrotate* error messages to the specified local user or email address.
`compress`	Use *gzip* to compress old versions of log files.
`copytruncate`	Instead of renaming the current log file and creating a new (empty) one, move most of its data out into an archive file. Accommodates programs that can't interrupt logging (i.e., that need to keep the log file open for writing continuously).
`create` *[octalmode owner group]*	Recreate the (now empty) log file immediately after rotation. If specified, set any or all of these properties: *octalmode* (filemode in octal notation—e.g., 0700), *owner*, and *group* properties.
`ifempty | notifempty`	By default, *logrotate* will rotate a file even if it's empty. *notifempty* cancels this behavior; *ifempty* restores it (e.g., overriding a global *notifempty* setting).
`include` *file_or_directory*	When parsing *logrotate.conf*, include the specified file or the files in the specified directory.
`missingok | nomissingok`	By default, *logrotate* will return a message if a log file doesn't exist. *nomissingok* cancels this behavior (i.e., tells *logrotate* to skip that log file quietly); *missingok* restores the default behavior (e.g., overriding a global *nomissingok* setting).
`olddir` *dir* `| noolddir`	Tells *logrotate* to keep old versions of a log file in *dir*, whereas *noolddir* tells *logrotate* to keep old versions in the same directory as the current version (*noolddir* is the default behavior).

`postrotate`
 line1
 line2
 etc.
`endscript`
	Execute specified *lines* after rotating the log file. Can't be declared globally. Typically used to send a SIGHUP to the application that uses the log file.

`prerotate`
 line1
 line2
 etc.
`endscript`
	Execute specified *lines* before rotating the log file. Can't be declared globally.

Just What Do We Mean By "Rotate?"

All log-management mechanisms involve periodically moving/renaming a log file to an archive copy and creating a new (empty) log file. Rotation is necessary when multiple archive copies are maintained.

In the most common log-rotation scheme, a set of static filenames is maintained. For example, *messages, messages.1, messages.2, messages.3* is a typical three-archive filename set—*messages* being the "current" log file and *messages.3* being the oldest archive.

In this scheme, rotation is achieved by coping the second-to-oldest file over the oldest file (e.g., `mv messages.2 messages.3`). The third-oldest file's name is then changed to that of the second-oldest file's, and so forth, until the current file is renamed and a new (empty) "current" log file is created (e.g., `mv messages messages.1; touch messages`). This is how *logrotate* behaves when its *rotate* parameter is set to a nonzero value.

In the second common mechanism, archive filenames are unique (e.g., *messages*, *messages.20010807*, *messages.20010708*, etc.). In this case, rotation is a simple matter of changing the current file's name and then creating a new (empty) "current" log file (e.g., `mv messages messages.20010928; touch messages`). The final step is to compare the age of the oldest log archive file to a "maximum age" setting and to delete it if it's reached that age.

This second scheme is used by SuSE's *aaa_base_rotate_logs* script (covered later in this chapter).

Running logrotate

In both Red Hat 7 and Debian 2.2, *logrotate* is invoked by the script */etc/cron.daily/logrotate*, which consists of a single command:

```
/usr/sbin/logrotate /etc/logrotate.conf
```

This doesn't necessarily mean that logs are rotated daily; it means that *logrotate* checks each log file daily against its configuration script and rotates or doesn't rotate the log file accordingly.

If you want *logrotate* to be run less frequently, you can move this script to */etc/cron.weekly* or even */etc/cron.monthly* (though the latter is emphatically *not* recommended unless *logrotate* is, for some strange reason, configured to rotate each and every file monthly).

Log Management in SuSE 7

Log rotation in SuSE, as with so much else, is configured at a gross level in */etc/rc.config* (the configuration file for *suseconfig*, which is the primary backend engine of *yast*). This file contains a variable called *MAX_DAYS_FOR_LOG_FILES*, which you

can use to set the maximum number of days system logs are kept (by default, 365). In addition, the log-rotation tools themselves come preconfigured and preactivated.

Chances are, however, that you'll need to tweak SuSE's log-management setup more granularly than *MAX_DAYS_FOR_LOG_FILES*, especially if you install Syslog-ng and disable syslog. As it happens, SuSE's log-rotation scheme is less powerful but also much simpler than Red Hat's and Debian's *logrotate*.

SuSE uses a script called */etc/cron.daily/aaa_base_rotate_logs* for day-to-day log rotation. This script shouldn't be manually edited; its behavior is controlled by the file */etc/logfiles*, which is simply a list of the files you wish to rotate along with the maximum sizes you want them to reach, the permissions and ownerships they should have, and the startup script (if any) that should be restarted after rotation is done.

Example 10-26 is an excerpt from the default */etc/logfiles* from SuSE 7.1.

Example 10-26. Excerpts from /etc/logfiles

```
# /etc/logfiles - This file tells cron.daily, which log files have to be watched
#
# File                      max size   mode     ownership    service
#                                                           (reload if changed)
/var/log/mgetty.*             +1024k   644      root.root
/var/log/messages             +4096k   640      root.root
/var/log/httpd/access_log     +4096k   644      root.root     apache
/var/squid/logs/access.log    +4096k   640      squid.root
```

In the first noncomment line, all log files whose name begins */var/log/mgetty* will be rotated after exceeding 1,024 kilobytes, after which they'll be rotated to new files whose permissions are `-rw-r--r--` and that are owned by user *root* and group *root*.

The third line states that the file */var/log/httpd/access_log* should be rotated after exceeding 4,096 kilobytes, should be recreated with permissions `-rw-r--r--`, owned by user *root* and group *root*, and after rotation is done, the startup script */etc/init.d/apache* should be restarted.

Since the maximum age of all log files is set globally in */etc/rc.config*, take care not to set the maximum size of a frequently written-to file (such as */var/log/messages*) too high. If this happens and if the maximum age is high enough, your logs may fill their volume.

Speaking of which, I highly recommend the use of a dedicated */var* partition on any machine that acts as a server; a full */var* partition is much less likely to cause disruptive system behavior (e.g., crashing) than a full root partition.

Using Swatch for Automated Log Monitoring

Okay, you've painstakingly configured, tested, and fine-tuned your system logger to sort system messages by type and importance and then log them both to their respective files and to a central log server. You've also configured a log-rotation scheme that keeps as much old log data around as you think you'll need.

But who's got the time to actually *read* all those log messages?

swatch (the "Simple WATCHer") does. *swatch*, a free log-monitoring utility written 100% in Perl, monitors logs as they're being written and takes action when it finds something you've told it to look out for. Swatch does for logs what tripwire does for system-file integrity.

Installing Swatch

There are two ways to install *swatch*. First, of course, is via whatever binary package of *swatch* your Linux distribution of choice provides. (I use the term loosely here; "executable package" is more precise.) The current version of Mandrake has an RPM package of *swatch*, but none of the other most popular distributions (i.e., Red Hat, SuSE, Slackware, or Debian) appear to.

This is just as well, though, since the second way to install *swatch* is quite interesting. *swatch*'s source distribution, available from *http://www.stanford.edu/~atkins/swatch*, includes a sophisticated script called *Makefile.PL* that automatically checks for all necessary Perl modules (see "Should We Let Perl Download and Install Its Own Modules?" later in this chapter) and uses Perl 5's CPAN functionality to download and install any modules you need; it then generates a *Makefile* that can be used to build swatch.

After you've installed the required modules, either automatically from *swatch*'s *Makefile.PL* script or manually (and then running perl Makefile.PL), *Makefile.PL* should return the contents of Example 10-27.

Example 10-27. Successful Makefile.PL run

```
 [root@barrelofun swatch-3.0.1]# perl Makefile.PL
Checking for Time::HiRes 1.12 ... ok
Checking for Date::Calc ... ok
Checking for Date::Format ... ok
Checking for File::Tail ... ok
Checking if your kit is complete...
Looks good
Writing Makefile for swatch
[root@barrelofun swatch-3.0.1]#
```

Once *Makefile.PL* has successfully created a *Makefile* for *swatch*, you can execute the following commands to build and install it:

```
make
make test
make install
make realclean
```

The *make test* command is optional but useful: it ensures that *swatch* can properly use the Perl modules we just went to the trouble of installing.

Should We Let Perl Download and Install Its Own Modules?

The Comprehensive Perl Archive Network (CPAN) is a network of Perl software archives from around the world. Perl Version 5.6.x includes modules (CPAN and CPAN::FirstTime, among others) that allow it to fetch, verify the checksums of, and even use gcc to compile Perl modules from CPAN sites on the Internet. In-depth descriptions of CPAN and Perl's CPAN functionality are beyond this chapter's scope, but I have one hint and one warning to offer.

First, the hint. To install the module Example::Module (not a real Perl module), you enter the command:

```
perl -MCPAN -e "install Example::Module"
```

If it's the first time you've used the -MCPAN flag, the module CPAN::FirstTime will be triggered and you'll be asked to choose from various options as to how Perl should fetch and install modules from CPAN. These are well-phrased questions with reasonable defaults. But do pay attention to the output while this command executes: the module you're installing may depend on other modules and may require you to go back and execute, e.g.:

```
perl -MCPAN -e "install Example::PreRequisite"
```

before making a second attempt at installing the first module.

Now for the warning: using CPAN is neither more nor less secure than downloading and installing other software from any other Internet source. On the one hand, before being installed, each downloaded module is automatically checked against a checksum that incorporates a cryptographically strong MD5 hash. On the other hand, this hash is intended to prevent corrupt downloads from going unnoticed, not to provide security per se.

Furthermore, even assuming that a given package's checksum probably won't be replaced along with a tampered-with module (a big assumption), all this protects against is the unauthorized alteration of software after it's been uploaded to CPAN by its author. There's nothing to stop an evil registered CPAN developer (anybody may register as one) from uploading hostile code along with a valid checksum. But of course, there's nothing to stop that evil developer from posting bad stuff to SourceForge or FreshMeat, either.

Thus, if you really want to be thorough, the most secure way to install a given Perl module is to:

1. Identify/locate the module on *http://search.cpan.org*.
2. Follow the link to CPAN's page for the module.
3. Download the module *not* from CPAN, but from its developer's official web site (listed under "Author Information" in the web page referred to earlier in Step 2).
4. If available, also download any checksum or hash provided by the developer for the tarball you just downloaded.

—continued—

5. Use *gpg*, *md5*, etc. to verify that the tarball matches the hash.
6. Unzip and expand the tarball, e.g., `tar -xzvf groovyperlmod.tar.gz`.
7. If you're a Righteously Paranoid Kung-Fu Master or aspire to becoming one, review the source code for sloppiness and shenanigans, report your findings to the developer or the world at large, and bask in the open source community's awe and gratitude. (I'm being flippant, but open source code is truly open only when people bother to examine it!)

Follow the module's building and installing directions, usually contained in a file called *INSTALL* and generally amounting to something like:

```
perl ./Makefile.PL
make
make test
make install
```

Note that if the modules you need are being brought to your attention by swatch's `Makefile.PL` script, then to use the paranoid installation method, you'll want to write down the needed module names and kill that script (via plain old CONTROL-c) before installing the modules and rerunning swatch's *Makefile.PL*.

Before I forget, there's actually a third way to install missing Perl modules: from your Linux distribution's FTP site or CDROM. While none approach CPAN's selection, most Linux distributions have packaged versions of the most popular Perl modules. Following are the modules you need for swatch and the packages that contain them in Red Hat 7 and Debian 2.2:

- Perl ModuleRed Hat 7 RPMDebian "deb" package
- Date::Calcperl-Date-Calclibdate-calc-perl
- Time::HiResperl-Time-HiReslibtime-hires-perl
- Date::Formatperl-TimeDatelibtimedate-perl
- File::Tailperl-File-Taillibfile-tail-perl

None of this may seem terribly specific to swatch, and indeed it isn't, but it *is* important—more and more useful utilities are being released either as Perl modules or as Perl scripts that depend on Perl modules, so the chances are that swatch will not be the last Makefile.PL-based utility you install. Understanding some ramifications of all this module madness is worth the liter of ink I just spent on it, trust me.

swatch Configuration in Brief

Since the whole point of *swatch* is to simplify our lives, configuring *swatch* itself is, well, simple. *swatch* is controlled by a single file, *$HOME/.swatchrc* by default. This file contains text patterns, in the form of regular expressions, that you want *swatch* to watch for. Each regular expression is followed by the action(s) you wish to *swatch* to take whenever it encounters that text.

For example, suppose you've got an Apache-based web server and you want to be alerted any time someone attempts a buffer-overflow attack by requesting an extremely

long filename (URL). By trying this yourself against the web server while tailing its */var/apache/error.log*, you know that Apache will log an entry that includes the string "File name too long." Suppose further that you want to be emailed every time this happens. Example 10-28 shows what you'd need to have in your *.swatchrc* file.

Example 10-28. Simple entry in .swatchrc

```
watchfor /File name too long/
        mail addresses=mick\@visi.com,subject=BufferOverflow_attempt
```

As you can see, the entry begins with a *watchfor* statement, followed by a regular expression. If you aren't yet proficient in the use of regular expressions, don't worry: this can be as simple as a snippet of the text you want *swatch* to look for, spelled out verbatim between two slashes.

Swatch will perform your choice of a number of actions when it matches your regular expression. In this example, we've told swatch to send email to `mick\@visi.com`, with a subject of `BufferOverflow_attempt`. Note the backslash before the `@` sign—without it, Perl will interpret the `@` sign as a special character. Note also that if you want spaces in your subject-line, each space needs to be escaped with a backslash—e.g., `subject=Buffer\ Overflow\ attempt`.

Actions besides sending email include the ones in Table 10-13.

Table 10-13. Some actions swatch can take

Action (keyword)	Description
`echo=`*`normal, underscore, blue, inverse, etc.`*	Print matched line to console, with or without special text mode (default mode is "normal").
`bell` *`N`*	Echo the line to console, with "beep" sounded *N* times (default = 1).
`exec` *`command`*	Execute the command or script *command*.
`pipe` *`command`*	Pipe the line to the command *command*.
`throttle` *`HH:MM:SS`*	Wait for *`HH:MM:SS`* (period of time) after a line triggers a match, before performing actions on another match of the same expression. Helps prevent Denial of Service attacks via swatch (e.g., deliberately triggering huge numbers of swatch events in a short period).

For more details on configuring these and the other actions that swatch supports, see the *swatch(1)* manpage.

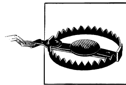
If you use Syslog-ng, you may be able to use some combination of *match()* filters, *program()* destinations, and *pipe()* destinations to achieve most of what *swatch* does.

However, *swatch*'s *throttle* parameter is an important advantage: whereas Syslog-ng acts on every message that matches a given filter, *throttle* gives *swatch* the intelligence to ignore repeated occurrences of a given event, potentially preventing minor events from becoming major annoyances.

Let's take that example a step further. Suppose in addition to being emailed about buffer-overflow attempts, you want to know whenever someone hits a certain web page, but only if you're logged on to a console at the time. In the same *.swatchrc* file, you'd add something like Example 10-29.

Example 10-29. An event that beeps and prints to console

```
watchfor /wuzza.html/
    echo=red
    bell 2
```



You will only see these messages and hear these beeps if you are logged on to the console in the same shell session from which you launched swatch. If you log out to go get a sandwich, when you return and log back in, you will no longer see messages generated by the swatch processes launched in your old session, even though those processes will still be running.

When in doubt, add either a "mail" action or some other non console-specific action (e.g., an "exec" action that triggers a script that pages you, etc.), unless, that is, the pattern in question isn't critical.

Alert readers have no doubt noticed that the scenario in the previous example will work only for Apache installations in which both errors and access messages are logged to the same file. We haven't associated different expressions with different watched files, nor can we. But what if you want swatch to watch more than one log file?

This is no problem. Although each *.swatchrc* file may describe only one watched file, there's nothing to stop you from running multiple instances of swatch, each with its own *.swatchrc* file. In other words, *.swatchrc* is the default, but not the required name for swatch configurations.

To split our two examples into two files, you'd put the lines in Example 10-27 into a file called, for example, *.swatchrc.hterror* and the lines in Example 10-28 into a file called *.swatchrc.htaccess*.

Advanced swatch Configuration

So far we've only considered actions we want triggered every time a given pattern is matched. There are several ways we can control swatch's behavior with greater granularity, however.

The first and most obvious is that search patterns take the form of regular expressions. Regular expressions, which really constitute a text-formatting language of their own, are incredibly powerful and responsible for a good deal of the magic of Perl, sed, vi, and many other Unix utilities.

It behooves you to know at least a couple "regex" tricks. Trick number one is called alternation, and it adds a "logical or" to your regular expression in the form of a "|" sign. Consider this regular expression:

```
/reject|failed/
```

This expression will match any line containing either the word "reject" or the word "failed." Use alternation when you want swatch to take the same action for more than one pattern.

Trick number two is the Perl-specific regular-expression modifier "case-insensitive," also known as "slash-i" since it always follows a regular expression's trailing slash. The regular expression:

```
/reject/i
```

matches any line containing the word "reject" whether it's spelled "Reject," "REJECT," "rEjEcT," etc. Granted, this isn't nearly as useful as alternation, and in the interest of full disclosure, I'm compelled to mention that slash-i is one of the more CPU-intensive Perl modifiers. However, if despite your best efforts at log tailing, self attacking, etc., you aren't 100% sure how a worrisome attack might look in a log file, slash-i helps you make a reasonable guess.

Another way to control swatch more precisely is to specify what time of day a given action may be performed. You can do this by sticking a *when*= option after any action. For example, in Example 10-30, I have a *.swatchrc* entry for a medium-importance event, which I want to know about via console messages during weekdays, but which I'll need email messages to know about during the weekend.

Example 10-30. Actions with when option specified

```
/file system full/
    echo=red
    mail addresses=mick\@visi.com,subject=Volume_Full,when=7-1:1-24
```

The syntax of the *when*= option is `when=`*`range_of_days:range_of_hours`*. Thus, in Example 10-30, we see that any time the message "file system full" is logged, swatch will echo the log entry to the console in red ink. It will also send email, but only if it's Saturday ("7") or Sunday ("1").

Running swatch

Swatch expects *.swatchrc* to live in the home directory of the user who invokes swatch. Swatch also keeps its temporary files there by default. (Each time it's invoked, it creates and runs a script called a "watcher process," whose name ends with a dot followed by the PID of the swatch process that created it).

The `-c` *`path/to/configfile`* and `--script-dir=`*`/path/to/scripts`* flags let you specify alternate locations for swatch's configuration and script files, respectively. Never

keep either in a world-writable directory, however. In fact, only these files' owners should be able to read them.

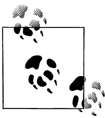
For example, to invoke swatch so that it reads my custom configuration file in */var/log* and also uses that directory for its watcher process script, I'd use the command listed in Example 10-31.

Example 10-31. Specifying nondefault paths

```
mylinuxbox:~# swatch -c /var/log/.swatchrc.access --script-dir=/var/log &
```

I also need to tell swatch which file to tail, and for that I need the *-t filename* flag. If I wanted to use the previous command to have swatch monitor */var/log/apache/access_log*, it would look like this:

```
mylinuxbox:~# swatch -c /var/log/.swatchrc.access --script-dir=/var/log
\  -t /var/log/apache/access_log &
```



swatch generally doesn't clean up after itself very well; it tends to leave watcher-process scripts behind. Keep an eye out and periodically delete these in your home directory or in the script directories you tend to specify with *--script-dir*.

Again, if you want swatch to monitor multiple files, you'll need to run swatch multiple times, with at least a different tailing target (*-t* value) specified each time and probably a different configuration file for each as well.

Fine-Tuning swatch

Once swatch is configured and running, we must turn our attention to the Goldilocks Goal: we want swatch to be running neither "too hot" (alerting us about routine or trivial events) nor "too cold" (never alerting us about anything). But what constitutes "just right?" There are as many answers to this question as there are uses for Unix.

Anyhow, you don't need me to tell you what constitutes nuisance-level reporting: if it happens, you'll know it. You may even experience a scare or two in responding to events that set off alarms appropriately but turn out to be harmless nonetheless. Read the manual, tweak *.swatchrc*, and stay the course.

The other scenario, in which too little is watched for, is much harder to address, especially for the beginning system administrator. By definition, anomalous events don't happen very frequently, so how do you anticipate how they'll manifest themselves in the logs? My first bit of advice is to get in the habit of browsing your system logs often enough to get a feel for what the routine operation of your systems looks like.

Better still, "tail" the logs in real time. If you enter the command *tail -f /var/log/messages*, the last 50 lines of the system log will be printed, plus *all subsequent lines,*

as they're generated, until you kill tail with a *Control-c*. This works for any file, even a log file that changes very rapidly.

Another good thing you can do is to "beat up on" (probe/attack) your system in one virtual console or xterm while tailing various log files in another. nmap and Nessus, which are covered in Chapter 3 (Hardening Linux), are perfect for this.

By now you may be saying, "Hey, I thought the whole reason I installed swatch was so I wouldn't have to watch log files manually!" Wrong. Swatch *minimizes*, but does not eliminate, the need for us to parse log files.

Were you able to quit using your arithmetic skills after you got your first pocket calculator? No. For that matter, can you use a calculator in the first place unless you already know how to add, multiply, etc.? Definitely not. The same goes for log file parsing: you can't tell swatch to look for things you can't identify yourself, no more than you can ask for directions to a town whose name you've forgotten.

Why You Shouldn't Configure swatch Once and Forget About It

In the same vein, I urge you to not be complacent about swatch silence. If swatch's actions don't fire very often, it could be that your system isn't getting probed or misused very much, but it's at least as likely that swatch isn't casting its net wide enough. Continue to periodically scan through your logs manually to see if you're missing anything, and continue to tweak *.swatchrc*.

Don't forget to periodically reconsider the auditing/logging configurations of the daemons that generate log messages in the first place. Swatch won't catch events that aren't logged at all. Refer to the *syslogd(8)* manpage for general instructions on managing your *syslogd* daemon, and the manpages of the various things that log to syslog for specific instructions on changing the way they log events.

Resources

http://www.stanford.edu/~atkins/swatch. swatch home page. (Has links to the latest version, online manpages, etc.)

http://www.stanford.edu/~atkins/swatch/lisa93.html. Hansen, Stephen and Todd Atkins, creators of swatch. "Centralized System Monitoring with Swatch." (Old, but still useful.)

http://www.enteract.com/~lspitz/swatch.html. Spitzner, Lance. "Watching Your Logs." (A brief introduction to swatch.)

Friedl, Jeffrey E. F. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly & Associates, Inc. 1998.