

Syslog (Centralized Logging and Analysis)

Jason Healy, Director of Networks and Systems

Last Updated Mar 18, 2008

Contents

1	Syslog (Centralized Logging and Analysis)	5
1.1	Introduction	5
1.1.1	Logging Software	5
1.1.2	Analysis Software	6
1.1.3	Intended Audience	6
1.2	syslog-ng	6
1.2.1	General Concepts	7
1.2.2	Installation	7
1.2.3	Configuration	7
1.3	Syslog Client Configuration	11
1.4	Logcheck	11
1.4.1	Obtaining the Software	12
1.4.2	Usage	12
1.4.3	Syslog-NG and Logcheck	12
1.4.4	Basic Configuration	13
1.4.5	Custom Rules	14

Chapter 1

Syslog (Centralized Logging and Analysis)

Last updated 2008/03/18

1.1 Introduction

In a perfect world, nothing would ever go wrong. Unfortunately (at least in the network world), nothing is ever perfect, and things do go wrong from time to time. One of the best ways to diagnose problems is through the use of **logging**. Most applications and systems generate log messages, either on a regular basis (status logging) or when something goes wrong (error logging). Collecting and analysing these logs is a great way to track down problems.

Unfortunately, keeping track of all the logs on every piece of equipment can be a daunting task. This document explains how to collect all application logs in a central location for easier analysis. We also discuss software to monitor and report on anomalies found in the log files.

1.1.1 Logging Software

At Suffield, we mainly use machines and appliances that adhere to the **syslog** log format. This format has been used in the UNIX world for quite some time, and it is readily supported by many operating systems, appliances, and software packages.

Syslogs are normally collected by a **syslog daemon**. The daemon we have

chosen to use is called **syslog-ng** (for "Syslog Next Generation"). It allows for complex setups with multiple sources and destinations of logging, along with built-in pattern-matching conditions and custom hooks for external analysis.

1.1.2 Analysis Software

While **syslog-ng** does possess some filtering and reporting capabilities, we prefer to run our logs through an external tool. This allows us to search the logs for unusual behavior (failed logins, warning messages, *etc.*) so we can take corrective action as soon as possible.

1.1.3 Intended Audience

This document assumes that the reader is proficient in the installation and configuration of a UNIX-like operating system. We will be using the Debian flavor of Linux for our examples, though most other unices should operate in a similar manner.

1.2 syslog-ng

We use a drop-in replacement for the standard `syslog` daemon called **syslog-ng**. It receives messages from remote hosts in the standard syslog format, but it provides many nice features that other daemons lack. Of particular interest is its ability to route messages to different files, external databases, or external programs. The system is very flexible, and works well even for large-scale installations.

For more information on **syslog-ng**, including detailed installation instructions, the latest downloads, and other information, please see the main syslog-ng web site:

<http://www.balabit.com/products/syslog-ng/>

Additionally, you may wish to read O'Reilly's *Building Secure Servers with Linux* book. Chapter 12 is available for free online, and deals specifically with syslog-ng (as well as some of the other topics discussed here). The book's website is:

<http://www.oreilly.com/catalog/linuxss2/>

1.2.1 General Concepts

Syslog-ng works with three major components in its configuration:

1. **sources** of log information (e.g., network sockets)
2. **destinations** of log information (e.g., files)
3. **filters** that match certain log information

Syslog-ng combines these three things into one or more log pipelines. This gives you the ability to pipe multiple sources into a certain destination, send one source to many destinations, or send only certain logs to particular destinations based on filters.

1.2.2 Installation

Many systems have pre-built packages of `syslog-ng` available, without the need for source. We use Debian linux, and the install is as simple as:

```
# apt-get install syslog-ng
```

This will create a sample configuration file in `/etc/syslog-ng`, which you can customize.

1.2.3 Configuration

All configuration occurs in the file `/etc/syslog-ng.conf`. The sections below discuss changes to the defaults provided under Debian. Note that some installations may use different defaults for other values; check the documentation for more information about special options.

Global Options

All options belong in a configuration stanza named `options{}` (the configuration values go between the { curly braces }).

We use this section to set global timeouts and retries for logs and network connections. In some cases, these values can be overridden in later sections of the files.

Additionally, this section deals with use of DNS to resolve hostnames. The important values here are:

- `chain_hostnames(no)`: tell the server not to report every hostname between the source and receiver; just use the source's address.
- `keep_hostname(no)`: do not trust the hostname used by the source; replace it with the name queried from DNS. Note that if you do not have proper reverse DNS set up, this can cause `syslog-ng` to block while it waits for DNS queries.
- `use_dns(yes)`: allow the server to make DNS queries for host names.
- `use_fqdn(yes)`: store the full name of the source, rather than just the short hostname.

Other values exist for tweaking the caching and timeouts of DNS lookups; see the configuration file for more information.

Sources

Syslog-ng must be told which sources to listen on to receive messages. The most frequently used ones are:

- `internal()`: This source is generated from within `syslog-ng` itself. Any status messages that `syslog-ng` creates will be sent from this source, so you should include it in at least one log statement to capture its output.
- `file()`: This reads from a special file on the local system. Under Linux, you'll want to use this to read `/proc/kmsg`. Note that this does **not** "tail" a regular file; it only works with special files designed to work as pipes.
- `unix_dgram()`: This reads from a local UNIX socket in a connectionless fashion (use `unix_stream()` for connection-oriented sockets). On Linux, this is used to read the special file `/dev/log`.
- `udp()`: This causes `syslog-ng` to listen to a UDP port on the network interface for messages from other systems. Since we're building a centralized logging host, you'll **definitely** want to use this option. A TCP option also exists, though it is less frequently used (most clients default to UDP).

You'll need to declare one or more sources, each using one or more of these sources. We recommend lumping the sources into broader source categories. For example, `/proc/kmsg`, `internal()`, and `/dev/log` are all local to the machine that is running `syslog-ng`. Therefore, you may wish to merge them all into a single source that tracks local machine messages.

To declare a source, simply use the `source` parameter, give it a name, and enclose all the sources in { curly braces }:

```
source s_local { unix_dgram("/dev/log"); internal() };
source s_remote { udp(); };
```

Filters

Filters determine which messages will be routed to which destinations. Filters can test every aspect of an incoming message (often using regular expressions), making them extremely powerful. You can match source IP, subnet, syslog priority, log facility, or even strings embedded in the messages.

You can combine conditions with "and" and "or", as well as negate them with "not". This allows you to build up a complex filter out of many simple matching rules.

To declare a filter, use a `filter` parameter, give it a name, and enclose the matching rules inside { curly braces }:

```
filter f_simple { match("foobar"); };
filter f_complex { facility(mail) and (level(info .. err) or host("bob")); };
```

Destinations

Once messages have been filtered, they must be sent to a destination. Syslog-ng can use many different types of destinations, including plain files, pipes, other syslog hosts, and external programs.

Files are the most frequently used destination in a typical setup, as they are permanently stored, easy to search, and work in the same way that a traditional syslog daemon does. We will primarily examine files in this section.

However, it is worth noting two other destinations, though we will not discuss them further:

- The `program()` call forks to a new process and sends messages to it via STDIN. This is an easy way to quickly get messages to an external program. However, `syslog-ng` only forks once per startup of the daemon, which means that the program is tied to the execution of `syslog-ng` itself.
- The `pipe()` call sends messages to a named pipe on the system. This can be useful when sending messages to a completely separate process (such as `xconsole` or a custom script that's waiting for input). It is more robust than `program()`, as the log files can be "dropped off" at the pipe and later read by a different program with different privileges or parameters.

Destinations may have permissions, owners, groups, sync times, and other options that override the ones defined in the global `options{}` section of the config file. You may wish to tweak some of these settings (for example, setting a lower sync rate on busy log files).

Destinations may also set a **template**, which defines how the logs are formatted as they are output. Each template contains a formatting string which can contain **macros** that are expanded to their values at the time of output. For example, to format each line with the full date, host, and level of severity:

```
template("$FULLDATE $HOST [$LEVEL]: $MESSAGE");
```

Additionally, for the `file()` destination, you can use template macros in the filename. This allows for simple writing to different files based upon content in the message itself (such as date, time, or host).

Here is a sample output to file using a templated name, and specially formatted output:

```
destination d_file { file("/var/log/$YEAR.$MONTH.$DAY/$HOST"
                        template("$FULLDATE $HOST [$LEVEL]: $MESSAGE")
                      );
};
```

Log Rules

Once you've defined sources, destinations, and filters, you're ready to combine them into **log rules**. A log rule simply ties together a source and destination, and optionally allows for a filter to only allow certain messages to match the rule. In its simplest form, a log rule looks like this:

```
log { source(s_local); filter(f_important); destination(d_varlog); };
```

(Assuming that `s_local`, `f_important`, and `d_varlog` have already been set up earlier in the file.)

You are free to provide multiple sources, filters, and destinations in a single log statement, allowing you to easily create complex log setups.

Finally, newer versions of syslog-ng support a **flags** option in log rules. This special option has three settings:

1. **final** means that once a message matches this rule, it is no longer considered for other log rules.

2. **fallback** means that this rule matches only if the message has not matched any non-fallback rules so far (useful as a "default" rule to catch unmatched messages).
3. **catchall** means that only the filter and destination are used for this rule; all possible message sources are used.

Thus, to make a rule that matches only unmatched messages, we might say:

```
log { source(s_local); destination(d_default); flags(fallback); };
```

You should create log statements that link up all the sources and destinations that you will be using. Note that unless you use flags like "fallback", it is possible for a log message to match more than one log rule, and thus be logged to more than one destination.

1.3 Syslog Client Configuration

To send information to the central server from a machine running `syslog`, we simply instruct the daemon to forward information to a remote host.

The simplest way to do this is to add a catch-all for every message in `/etc/syslog.conf`:

```
*.* @172.30.0.10
```

Then, restart your syslog daemon (using an init script, `kill -HUP`, or whatever method your platform uses).

1.4 Logcheck

Once you have a centralized logging machine in place, its time to do something with those logs. While it's helpful to keep logs files for later review, it's even more helpful to automatically scan log files for "strange" events so they can be reported. What constitutes a "strange" event is up to you; you'll need to decide what sorts of events require further inspection.

At Suffield, we make use of the `logcheck` program to help us with automated analysis. The program scans log files at regular intervals, and e-mails any lines that look suspicious. In this way, we are automatically alerted when certain log messages appear.

1.4.1 Obtaining the Software

We use the Debian flavor of Linux, which includes a prepackaged version of `logcheck`. Simply install the tool from the command line with:

```
apt-get install logcheck
```

If you don't use Debian, your distribution may still include a package for `logcheck`. Consult your distribution's package database.

If you can't find it, you can download the upstream version directly:

<http://sourceforge.net/projects/sentrytools>

1.4.2 Usage

Normally, `logcheck` is run from `cron` (or some other scheduling process) on a regular basis (*e.g.*, once an hour). The script checks a list of log files for specific patterns, and reports any that are found.

`Logcheck` includes three basic levels of reporting functionality: *workstation*, *server*, and *paranoid*. Each level includes everything in the levels below (*i.e.*, *server* includes all of *workstation*). You should choose a level that reports enough information, without flooding you with too many false reports. Our approach has been to select the highest level (*paranoid*), and then selectively disable warnings using custom patterns.

Note that `logcheck` defaults to reporting **all** unrecognized messages. In other words, if you do not have a pattern which explicitly matches a message type, `logcheck` will include it as suspicious. This is good, as it ensures that you receive messages that you haven't necessarily dreamed up a configuration for.

1.4.3 Syslog-NG and Logcheck

`Logcheck` requires a list of log files to check. Unfortunately, our `syslog-ng` setup files logs away in many different files (by date, category, program, *etc.*). To make running `logcheck` easier, we have added an additional destination file to `syslog-ng`. We funnel many different types of logs to this file, so that `logcheck` only needs to inspect a single file.

Because this log file is a copy of data logged elsewhere, we don't need to keep it around when we're done. Thus, we use `syslog-ng`'s ability to name files by day to set up a weekly rotation where old files get overwritten by new ones. This allows for a short amount of history (one week) without wasting too much space.

A sample syslog-ng stanza might look like this:

```
destination df_logcheck {
    file("/var/log/logcheck/$WEEKDAY.log"
        template("$FULLDATE: $HOST ($FACILITY/$LEVEL) [$PROGRAM] $MSGONLY\n")
        template_escape(no)
        remove_if_older(518400) # overwrite if older than 6 days
    );
};
```

You can configure syslog-ng to send any (or even all) data to this file in addition to the regular files you would otherwise use.

1.4.4 Basic Configuration

At the minimum, you must provide a list of log files to check, the patterns you wish to check for, and an e-mail address to send reports to.

Begin by editing the `/etc/logcheck/logcheck.conf` file. Choose a `REPORTLEVEL` value (we use "paranoid"), and customize `SENDMAILTO` to point to a live e-mail address in your organization. You may also wish to enable `SUPPORT_CRACKING_IGNORE` if you're getting cracking-level events that are false positives (if you aren't, leave this at its default value for now).

Next, edit `/etc/logcheck/logfiles` to include the path to the log files you wish to check. If you're using our syslog-ng config above, you'd list each weekday rotation file:

```
/var/log/logcheck/Sun.log
/var/log/logcheck/Mon.log
# ... and so on ...
/var/log/logcheck/Sat.log
```

Finally, feel free to edit `/etc/logcheck/header.txt` to include a custom message at the start of each e-mail.

Depending on your setup, you may need to enable the logcheck script in cron. Under Debian, you must edit the file `/etc/cron.d/logcheck` and uncomment the line that starts with:

```
#2 * * * * logcheck ...
```

Once you've made these edits, logcheck should begin checking your logfiles and mailing you reports. The crontab line above would run logcheck every hour at 2 minutes past the hour. Check your distribution's documentation for more details.

1.4.5 Custom Rules

You should let logcheck run during a period of normal traffic on your network. You'll receive several alerts, and probably some false positives. Once you've verified that a false positive is really false, you can write rules to exclude the message from future reports.

Before we get to writing our own rules, here's a brief introduction to rule processing in logcheck. Each message in a log file is scanned, and then checked against the following conditions (in order):

1. Does the message match a rule in `cracking.d`? If so, the message is marked as an **attack alert**, and processing continues.
2. Does the message match a rule in `cracking.ignore.d`, and is the global `SUPPORT_CRACKING_IGNORE` enabled? If so, the message is **ignored** and **processing stops**. If the message does not match, but has been marked as an **attack alert**, then it is included in the outgoing e-mail, and processing stops. Otherwise, processing continues.
3. Does the message match a rule in `violations.d`? If so, the message is marked as a **security event**, and processing continues.
4. Does the message match a rule in `violations.ignore.d`? If so, the message is **ignored** and **processing stops**. If the message does not match, but has been marked as a **security event**, then it is included in the outgoing e-mail, and processing stops. Otherwise, processing continues.
5. Does the message match a rule in `ignore.d`? If so, the message is **ignored** and **processing stops**. Otherwise, the message is included in the outgoing e-mail.

As you can see, the rules follow an "include/exclude" pattern, with a default "include" at the end for any unmatched messages. When you create your own rules, you must be careful to match rules at the correct level (so as to not mask rules at a lower level), and to only match exactly what you need.

In general, you should write "include" rules wherever you think most appropriate (*e.g.*, **cracking** or **violation**), and write "exclude" rules at the same level as the rule that creates the false positive.

All matching rules in logcheck are written as *extended regular expressions* (as understood by the `egrep` program). A full discussion of regular expressions is far beyond the scope of this document; please consult other internet sources or books for more information. Also, feel free to "steal" from existing rules that are known to work!

To create a rule, simply code up a regular expression and drop it in a file. We suggest that you name the files with a local prefix (*e.g.*, "suffield"). For example, if you are getting log messages similar to the following:

```
[dhcpd] Wrote 0 new dynamic host decls to leases file.
```

You could ignore them by placing ignore rules in a file called `suffield-dhcpd`, and saving the file into the appropriate `ignore.d` directory. The ignore rule itself might look like:

```
\[dhcpd\] Wrote [0-9]+ new dynamic host decls to leases file
```

(The "[0-9]+" is a regular expression pattern; if you don't know what it does, you should consult external resources for more information.)

You may place more than one regular expression in a file, one per line. Remember the order that rules are evaluated in; you may need to create overly-broad violation rules, and then selectively ignore certain cases.