Log Filtering with Rsyslog

DAVID LANG

David Lang is a Staff IT Engineer at Intuit, where he has spent more than a decade working in the Security Department for the Banking Division. He was introduced to Linux in 1993 and has been making his living with Linux since 1996. He is an Amateur Extra Class Radio Operator and served on the communications staff of the Civil Air Patrol California Wing, where his duties included managing the statewide digital wireless network. He was awarded the 2012 Chuck Yerkes award for his participation on various open source mailing lists.
david@lang.hm

In my first *;login:* article [1], I provided an overview of how to build an enterprise-class logging system and recommended using rsyslog as the transport. For those who are not familiar with modern syslog daemons, this may seem like a strange recommendation. In this article I will provide an overview of rsyslog's capabilities, with the focus on its filtering capabilities. Where a traditional syslog limited you to filtering on the facility and severity reported by the application writing the logs, rsyslog lets you filter anything in the log message, as well as several things that are not.

Traditional Syslog

Traditional syslog messages have a facility value (the type of log it is) and a severity value (the importance of the message). These are combined to create the priority (PRI) of the message, which is a decimal number: PRI = Facility * 8 + Severity.

The log messages are sent between machines in the format:

```
<PRI>timestamp hostname syslogtag message
```

Normally, when the messages are written to a file, the <PRI> header is left off, so what shows up in the file is:

```
timestamp hostname syslogtag message
```

Syslog filters (located in /etc/syslog.conf) are in the form of:

```
facility.severity[,facility.severity] <whitespace>6
```

The possible actions are

- Write to a file
- Send to a named pipe
- Send to a remote machine via UDP
- Write to a terminal/console
- Send to users

The PRI value for a message is completely determined by the application that's creating the message, with no protection preventing any user from writing a message claiming to be from the kernel with a severity level of "emergency." This allows you to use some of the predefined system facilities for your application (say, news or UUCP), at the cost of confusing newcomers to your environment. Most people expect that all non-system applications are going to use one of the local* facilities.

An example /etc/syslog.conf file:

```
mail.*              /var/log/mail.log
auth,authpriv.*     /var/log/auth.log
*.*                 /var/log/messages
*.*                 @192.168.1.6
```

Almost every program that writes to syslog allows you to specify what facility to use, and almost none of them prevent you from configuring anything you want. With scripts, you can use the /usr/bin/logger command to log whatever you want.

```
$ logger -p kernel.emerg -t kernel -s "The system is on fire!!!"
```

results in a log that looks like the following and is tagged with the facility "kernel" and the severity "emergency":

```
Jul 21 19:55:43 myhostname kernel: The System is on fire!!!
```

Rsyslog

Rsyslog has a rapid development cycle compared to Linux distros. As of the time of writing, most Linux distros ship with rsyslog 5.x, while RHEL versions 6.3 and earlier ship with rsyslog 3.22 as the default. Rsyslog 5.x became an option in RHEL 5.9 and became the default in RHEL 6.4. Meanwhile, the current supported version is rsyslog 7.4. As can be expected, the current upstream versions include many features that are not available in older versions. Adiscon, the primary sponsor of rsyslog, development hosts repositories for the newest versions for both RHEL/CentOS and Ubuntu packages, and several other people maintain current packages for other systems [2].

Among the many changes in rsyslog 6.x there was a new config syntax added. Unless stated otherwise, all examples provided in this article have been tested with rsyslog 3.x or newer.

Rsyslog has a modular design and, in addition to the capabilities of traditional syslog, supports many other modules that offer many additional functions.

Input Modules accept input into rsyslog:

```
im3195, imdiag, imfile, imgssapi, imjournal, imklog, imkmsg,
immark, impstats, imptcp, imrelp, imsolaris, imtcp, imttcp, imudp.
imuxsock, imzmq3
```

Stackable Parser Modules parse or modify the data the input modules accepted:

```
pmrfc3164, pmrfc5424, pmaixforwardedfrom, pmcisconames,
pmlastmsg, pmrfc3164sd, pmsnare
```

Message Modification Modules modify the parsed message or create variables from the message:

```
mmanon, mmaudit, mmcount, mmfields, mmjsonparse, mmnormalize,
mmsnmptrapd
```

Output Modules deliver the message to a destination:

```
omelasticsearch, omgssapi, omhdfs, omhiredis, omjournal,
omlibdbi, ommail, ommongodb, ommysql, omoracle, ompgsql,
omprog, omrabbitmq, omrelp, omruleset, omsnmp, omstdout,
omtesting, omudpspoof, omuxsock, omzmq3, omfwd (tcp/udp
network delivery), omdiscard, omfile, ompipe, omshell, omusrmsg
```

String Generation Modules provide predefined templates such as the following built-in templates:

```
RSYSLOG_DebugFormat, RSYSLOG_FileFormat,
RSYSLOG_ForwardFormat, RSYSLOG_SysklogdFileFormat,
RSYSLOG_SyslogProtocol23Format,
RSYSLOG_TraditionalFileFormat,
RSYSLOG_TraditionalForwardFormat
```

Compatibility with Traditional Syslog

Rsyslog supports the traditional PRI-based filtering syntax, so if your current usage fits within this syntax, you can continue to use it.

At startup, rsyslog needs a little more information in its config file to tell it which input modules to load and how to configure them, but the filtering lines can be identical.

An example /etc/rsyslog.conf equivalent to the /etc/syslog.conf shown earlier would be:

```
$ModLoad imuxsock
$ModLoad imklog
$ModLoad imudp
$UDPServerRun 514
mail.*                  /var/log/mail.log
auth,authpriv.*         /var/log/auth.log
*.*                     /var/log/messages
*.*               @192.168.1.6
```

Because rsyslog has an include function, the /etc/rsyslog.conf could be simplified to:

```
$ModLoad imuxsock
$ModLoad imklog
$ModLoad imudp
$UDPServerRun 514
$IncludeConfig /etc/rsyslog.conf
```

Several Linux distros use the line:

```
$IncludeConfig /etc/rsyslog.d/*.conf
```

to let you manage the configurations for different applications in separate application-specific files, without having all configuration information collected in the same file. There is a bug in rsyslog 6.x and 7.0 (fixed in 7.2) that caused the included files to be processed in reverse order. One caution with included files: rsyslog includes all the files and then evaluates the resulting config. This means that if you set a configuration value in one included file, it will still be in effect for later included files.

Modification of the Outbound Message

Rsyslog also allows you to change the log message that it sends out to any destination. You can create a format template [3] with a config entry like:

```
$template strangelog,"text %hostname:::lowercase% %msg% more\n"
```

where the items between '%' are variables (with formatting options).

Then in your action, you can tell rsyslog to use a specific template instead of the default template:

```
*.*     /var/log/messages;strangelog
```

Rsyslog even lets you create a template for the filename, so you could use a configuration like:

```
$template sortedlogs="/var/log/messages-%fromhost-ip%"
*.*     ?sortedlogs;strangelog
```

to write the messages to different files, with filenames in the format specified by the "sortedlogs" template, based on the source IP address.

Variables Available

Rsyslog provides different flavors of variables for use in config files: message property variables, trusted property variables, message content variables, and user-defined variables.

Message Property Variables

These are items derived from the message or the connection information, such as the timestamp within the message, the timestamp when the message was received on the local system, the hostname in the message, the hostname/IP of the system that delivered the message to the local box, PRI info, etc. [4]. For rsyslog version 5 and earlier, these were the only variables available.

Trusted Properties

Late in the 5.x series, rsyslog implemented the ability to query the kernel to get information about the process on the other end of the /dev/log socket (UID, PID, name of binary, command line, etc.), so that it could log information that normal non-root user processes cannot forge (processes running as root can still forge this information). In rsyslog 5.x, the information could only be appended to the log message, but with 6.x and newer, this information can be turned into variables.

Variables Parsed from Message Content

Rsyslog version 6 introduced "message modification" modules. These modules are allowed to modify a message after it has been parsed, and they can be invoked as the result of a filter test. In addition to modifying the message, these modules can also set variables that can be used the same way that the properties defined above are used.

The two most significant message modification modules for creating variables are mmjsonparse and mmnormalize.

Mmjsonparse will parse a JSON-formatted message and create a tree of variables for you to use.

These rsyslog.conf additions needed to use this module:

```
$ModLoad mmjsonparse
*.* :mmjsonparse:
```

This supports multiple levels of structure: `$!root!level1!level2!` etc. refers to an individual item, `$!` refers to the entire tree, and `$!root!level1` refers to a partial subtree.

Mmnormalize [5] lets you define a rule set for parsing messages, and it will do a very efficient parse of the log message, creating variables.

For example, starting with this example log message:

```
Jul 21 19:55:03 kernel: [1084540.211910] Denied: IN=eth0 OUT=
MAC=00:30:48:90:cc:a6:00:30:48:da:48:e8:08:00 SRC=10.10.10.10
DST=10.10.10.11 LEN=60 TOS=0x10 PREC=0x00 TTL=64 ID=28843
DF PROTO=TCP SPT=44075 DPT=444 WINDOW=14600 RES=0x00 SYN
URGP=0
```

and the rule file normalize.rb:

```
rule=: %kerntime:word% Denied: IN=%in:word% OUT=
MAC=%mac:word% SRC=%src-ip:ipv4% DST=%dst-ip:ipv4%
LEN=%len:number% TOS=%tos:word% PREC=%prec:word%
TTL=%ttl:number% ID=%id:number% %DF:word% PROTO=%proto:word%
SPT=%src-port:number% DPT=%dst-port:number%
WINDOW=%window:number% RES=%res:word% %pkt-type:word%
```

produces this log message:

```
Jul 21 19:55:49 myhostname json_msg: { "urgp": "0", "pkt-
type": "SYN", "res": "0x00", "window": "14600", "dst-port":
"444", "src-port": "51954", "proto": "TCP", "DF": "DF",
"id":"31890", "ttl": "64", "prec": "0x00", "tos": "0x10", "len":
"60", "dst-ip":"10.10.10.10", "src-ip": "10.10.10.11", "mac":
"00:30:48:90:cc:a6:00:30:48:da:48:e8:08:00", "in": "eth0",
"kerntime":"[1152127.460873]" }
```

You do need to add the following lines to rsyslog.conf to use this module:

```
$ModLoad mmnormalize
$mmnormalizeUseRawMSG off
$mmnormalizeRuleBase /rsyslog/rulebase.rb
*.* :mmnormalize:
$template json_fmt,"%timereported% %hostname% json_msg:
%$!%\n"
*.*   /var/log/test;json_fmt
```

User-Defined Variables

Rsyslog versions 7 and later allow you to define your own variables in the config file in addition to the ones created by the message modification modules. In rsyslog 7.6 there will be three flavors of variables that you can create:

- **Normal variables,** which can be created by "message modification modules" or by config statements. These are addressed as "$!name".
- **Local variables,** which cannot be set by message modification modules. These are addressed as "$.name".
- **Global variables,** which cannot be set by message modification modules, but will persist across log messages (other variables are cleared after every message is processed). These are addressed as "$/name".

Here are some examples of defining variables. Unlike other config statements, set and unset require a trailing ';':

```
set $!user!level1!var1="test";
set $!user!level1!var1=$!something + 1;
unset $!user!level1;
```

Using Variables

One common problem that people run into when using variables is the fact that the different types of variables were added to rsyslog at different times, and as a result there are different ways they are named.

The traditional message property variables have just the variable name, such as "timereported" or "fromhost-ip".

Other properties, mostly referring to the runtime environment (rather than the log message), have names like "$myhostname" or "$now":

- Variables parsed from the message with mm modules have names like "$!name".
- Local variables have names like "$.name".
- Global variables have names like "$/name".
- When using variables, the examples usually have the classic properties, so you see things like:
- %msg% in a template
- :msg, in a property-based filter
- $msg in a script-style config

But when you are using the other variable types, you must be aware that the variable prefix ('$' '$!' '$.' '$/') is considered part of the variable name, not a reference to it, so you would use something like:

- * %$!portnumber% in a template
- * :$!portnumber in a property-based filter

But you only use "$!portnumber" not "$$!portnumber" in a script-style filter or new-style config statement.

You can use "$$!portnumber" without syntax errors in some cases, but this results in an indirect reference to something.

New Filtering Capabilities

Use Last Match

The simplest and fastest "filter" to use is the '&' filter; It isn't really a filter because it just tells rsyslog to use the result of the last test. If that last test matched, the '&' will match as well.

This is extremely useful for cases in which you want to do several things if a condition is met.

A common example is when you want to log all messages of a particular type in one place, and send them off to another system.

```
mail.*    /var/log/mail.log
&         @mailanalysis
```

With rsyslog version 6 and later you can use {} to group multiple actions together, and as a result '&' isn't needed as much as it used to be.

```
Mail.*    { /var/log/mail.log
          @mailanalysis }
```

Stop Processing This Log Action

When you know that you don't want to process a log message any longer, you can tell rsyslog to stop and not waste time checking any further rules. This is commonly used in conjunction with the & filter or a block of actions to prevent rsyslog from trying to match any other filter rules after you have done what you want with a message. Without a stop, the message will get sent to every output that has a matching filter:

```
mail.*    /var/log/mail.log
&         @mailanalysis
&         ~
```

Or with the rsyslog version 6+

```
Mail.*    { /var/log/mail.log
        @mailanalysis
stop }
```

and rsyslog will stop processing this message and no other rules will be checked. Be careful—using included config files as a stop in one file may have an unexpected impact on the processing of another file.

"Always" Filter

In rsyslog version 7, the config optimizer is able to identify actions that have no filter in front of them. So instead of writing lines like:

```
*.*   /var/log/messages
*.*   @loghost
```

you can just write:

```
/var/log/messages
@loghost
```

The optimizer will optimize away "always match" filters in any case, so there is no performance penalty to continuing to write things the traditional way.

Property-Based filters

rsyslog has long supported property-based filters [6], which are formatted as:

```
:variable, [!]compare-operation, "value"
```

Examples of the different types are:

```
:programname, isequal, "sendmail" /var/log/mail.log
:msg, contains, "(root) CMD)"     ~
:msg, startswith, "pam_unix"      /var/log/auth.log
```

With property-based filters, you are no longer limited to filtering on the PRI value that was defined in the message. You can now filter based on the program name, or anything else in the log message. As a result, seeing rsyslog config files that have few (if any) PRI-based filters is common, and even those tend to be *.* or *.severity type filters, completely ignoring the facility.

For example, to file different types of logs into different output files, the following type of config is common:

```
:programname, startswith, "%ASA"    /var/log/cisco-messages
& ~
:programname, startswith, "postfix" /var/log/postfix-messages
& ~
:programname, isequal,    "snmpd"   /var/log/snmpd-messages
& ~
:programname, isequal,    "sendmail" /var/log/sendmail-messages
& ~
```

Script-Based Filters

Property filters can only test one thing, so rsyslog also includes script-based filters. These are familiar looking if-then conditions. Prior to the config optimizer that was added in rsyslog version 7, these were slow compared to PRI filters and significantly slower than property-based filters. In rsyslog version 7, the optimizer makes all the different formats equivalent in speed.

Script-based filters look like [7]:

```
IF test THEN action [ELSE action]
```

where test can be an arbitrarily complex expression, with normal precedence of operations, Boolean short-cutting, and built-in functions.

Action can be just about any block of config statements (including nested IF statements). Not all config items can be put into the "then" section of a test. In general, setup type commands (template definitions, input definitions, config parameters that change rsyslog internals) are not allowed. Commands that do some sort of action (set a variable, send the message to an output, invoke message modification modules) are allowed.

The equivalent to the property filter example would be:

```
if $programname startswith("%ASA") then /var/log/cisco-messages
else if $programname startswith("postfix") then
   /var/log/postfix-messages
else if $programname startswith("snmpd") then
   /var/log/snmpd-messages
else if $programname startswith("sendmail") then
   /var/log/sendmail-messages
else {
   <rest of rules>
}
```

Array Matches

Starting in rsyslog 7.2, repeated similar tests can be greatly optimized with "array" matches. Rather than having tests for many possible matches formatted like:

```
if $programname == "postfix" or $programname=="exim"
    or $programname=="sendmail" then /var/log/mail.log
```

rsyslog now supports what it calls Array Matches.

This allows you to write the test as:

```
if $programname == ["postfix","exim","sendmail"] then
   /var/log/mail.log
```

This can be extremely powerful when you combine it with dynamic file templates:

```
$template maillogs,"/var/log/mail-%programname-%severity%"
if $programname == ["postfix","exim","sendmail"] then ?maillogs
```

This will split the log files for mail apps into separate files for each type of program and severity level.

New Config Syntax

The old syntax will continue to be supported, and you can freely mix and match between the different config syntaxes within the same file (or included files) so you don't have to change your config files when you upgrade. For some of the newer functionality, though, you must use the new syntax.

In the old config syntax, you must set up options and then execute the function, while the new format looks like function calls with many parameters.

This example is in the old syntax:

```
$mmnormalizeUseRawMSG off
$mmnormalizeRuleBase /rsyslog/rulebase.rb
*.* :mmnormalize:
```

Here is the equivalent config using the new syntax:

```
action(type="mmnormalize" UseRawMsg="off"
   ruleBase="/etc/rsyslog.d/normalize.rb")
```

Here's another example using the old syntax for a more complex action (sending a SNMP trap):

```
$actionsnmptransport udp
$actionsnmptarget 192.168.1.100
$actionsnmptargetport 162
$actionsnmpversion 1
$actionsnmpcommunity testtest
$actionsnmptrapoid 1.3.6.1.4.1.19406.1.2.1
$actionsnmpsyslogmessageoid 1.3.6.1.4.1.19406.1.1.2.1
$actionsnmpenterpriseoid 1.3.6.1.4.1.3.1.1
$actionsnmptraptype 2
$actionsnmpspecifictype 0
:omsnmp:
```

With the new syntax, the same config appears in a much more compact format:

```
action(type="omsnmp" transport="udp" server="192.168.1.1"
   trapoid="1.3.6.1.4.1.19406.1.2.1" port="162" version="1"
   messageoid="1.3.6.1.4.1.19406.1.1.2.1" community="testtest"
   enterpriseoid="1.3.6.1.4.1.3.1.1" traptype="2" specifictype="0")
```

On the other hand, some things are simpler with the old config.

```
$template strange,"some text %variable% %variable:modifiers%\n"
```

is significantly longer using the new syntax:

```
template(name="strange" type=string
   string="some text %variable% %variable:modifiers%\n")
```

Even this simple rule in the old syntax:

```
*.*    /var/log/messages;templatename
```

becomes longer, although a bit more obvious as to what it does, using the new syntax:

```
*.*    action(type="omfile" File="/var/log/messages"
         Template="templatename")
```

Choosing which syntax to use is completely up to you— use whichever you find easier for the task at hand. Most configurations will include a mix of old and new, but in general, the more complex the configuration, the more likely you are to benefit from the new config syntax. Prior to the config optimizer added in rsyslog v7, PRI-based filters were by far the fastest type of filter to use.

Example

In the first article, I recommended using recent versions of rsyslog on the Aggregator and Analysis farm machines, so that you can take advantage of the greatly expanded capabilities and performance of the newer versions. One of the recommendations that I made was to use JSON-structured messages for the transport so that additional metadata could be added. The following config files are an example of what you may want to use.

Note that in these examples, I mix old and new config styles and make use of the "always" filter.

On all "normal" systems (app-servers, routers, switches, etc.), deliver all messages to the Edge Aggregation servers. On *nix systems, add an entry like the following to /etc/syslog.conf or /etc/rsyslog.conf:

```
*.*@edge-server-for-local-network
```

Here is an example /etc/rsyslog.conf for an Edge Aggregator. Note that rsyslog treats newlines as whitespace, so no line continuation characters are necessary. The exception to this is the $template commands, which need to be on one line (but is split here for printing):

```
module(load="imuxsock" SysSock.Annotate="on"
    SysSock.ParseTrusted="on")
module(load="imklog")
module(load="imudp")
input(type="imudp" port="514")
module(load="imtcp" MaxSessions="1000")
input(type="imtcp" port="514")
module(load="mmjsonparse")
action(type="mmjsonparse")
if $fromhost-ip != "127.0.0.1" then {
   # if the log is being received from another machine,
   #    add metadata to the log
   set $!trusted!origserver = $fromhost-ip;
   set $!trusted!edge!time = $timegenerated;
   set $!trusted!edge!relay = $$myhostname;
   set $!trusted!edge!input = $inputname;
```

```
} else {
    set $!trusted!local!input = $inputname;
}
# set this to reflect the environment that this Edge server is
servicing
set $!trusted!environment = "Dev network";
# note the template must be on a single line
# wrapping is for display only
$template structured_forwarding,
    "<%pri%>%timereported% %hostname% %syslogtag% %$!%\n"
/var/log/messages;structured_forwarding
@@core-server
#send to the core server via TCP consider using RELP instead
```

And here is an example configuration for the Analysis Farm systems:

```
module(load="imuxsock" SysSock.Annotate="on"
    SysSock.ParseTrusted="on")
module(load="imklog")
module(load="imtcp" MaxSessions="1000")
input(type="imtcp" port="514")
module(load="mmjsonparse")
action(type="mmjsonparse")
if $fromhost-ip == "127.0.0.1" then {
    #if this is a local log, send it to an edge relay.
    set $!trusted!local!input = $inputname;
    @edge-server
    stop }
$template std,"%timereported% %hostname% %syslogtag%%$!msg%\n"
/var/log/messages;std
```

To demonstrate how this works, on an app-server I executed:

```
logger testtest
```

which produced this message in /var/log/messages:

```
Jul 24 14:51:42 app-server dlang: testtest
```

On the Edge Aggregator server, the log message is reformatted and metadata is added to produce the following log entry that is sent to the Core Aggregator (which then relays the message to all Analysis Farms):

```
<13>Jul 24 14:51:42 app-server dlang: { "msg": "testtest",
"trusted": { "origserver": "10.1.2.9", "edge": { "time":
"Jul 24 21:51:42", "relay": "edge-server", "input": "imudp" },
"environment": "Dev network" } }
```

Note that with the app-server set to Pacific time and the edge server set to GMT, the timestamp when the log was created doesn't match when it's received.

And, finally, on the Analysis Farm systems, the following message will be produced in /var/log/messages:

```
Jul 24 14:51:42 app-server dlang: testtest
```

This threw away all the metadata, resulting in a message that looks identical to what was originally generated, but the metadata was available for filtering decisions up to this point. And a slightly different format on the Analysis Farm server could make any of the metadata available to the analysis tools.

As a second demonstration, on an Edge Aggregator I again executed:

```
logger testtest
```

Because this adds trusted properties to the message, it sends the following log entry to the Core Aggregator:

```
<13>Jul 24 21:53:39 edge-server dlang: { "pid": 4346, "uid":
1000, "gid": 1000, "appname": "logger", "cmd": "", "msg":
"testtest", "trusted": { "local": { "input": "imuxsock" },
"environment": "sending network" } }
```

Again, the Analysis Farm server will throw away the extra metadata and reformat the log to be:

```
Jul 24 21:53:39 edge-server dlang: testtest
```

But this time there was more metadata available about the process that created the log message available prior to the final output.

In conclusion, rsyslog has tremendous flexibility in processing your log messages. You can filter on just about anything that you care about, and you can modify messages as you send them out to any of the many different supported outputs.

References

[1] David Lang, "Enterprise Logging," *;login:*, vol. 38, no. 4: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/enterprise-logging.

[2] http://www.rsyslog.com/doc/rsyslog_packages.html.

[3] http://www.rsyslog.com/doc/rsyslog_conf_templates.html.

[4] http://www.rsyslog.com/doc/property_replacer.html.

[5] http://www.rsyslog.com/doc/mmnormalize.html.

[6] http://www.rsyslog.com/doc/rsyslog_conf_filter.html.

[7] http://www.rsyslog.com/doc/rainerscript.html.