Rsyslog Doc Documentation

Release 8.29.0.master

foo bar

Aug 26, 2017

Contents

Rsyslog is a **r**ocket-fast **sys**tem for **log** processing. It offers high-performance, great security features and a modular design. While it started as a regular syslogd, rsyslog has evolved into a kind of **swiss army knife of logging**, being able to

- accept inputs from a wide variety of sources,
- transform them,
- and output the results to diverse destinations.

Rsyslog has a strong enterprise focus but also scales down to small systems. It supports, among others, *MySQL*, *PostgreSQL*, *failover log destinations*, ElasticSearch, syslog/tcp transport, fine grain output format control, high precision timestamps, queued operations and the ability to filter on any message part.

CHAPTER 1

Manual

Configuration

Rsyslogd is configured via the rsyslog.conf file, typically found in `/etc`. By default, rsyslogd reads the file `/etc/rsyslog.conf`. This can be changed by a command line option.

Note that **configurations can be built interactively** via the online rsyslog configuration builder tool.

Basic Structure

This section describes how rsyslog configuration basically works. Think of rsyslog as a big logging and event processing toolset. It can be considered a framework with some basic processing that is fixed in the way data flows, but is highly customizable in the details of this message flow. During configuration, this customization is done by defining and customizing the rsyslog objects.

Quick overview of message flow and objects

Messages enter rsyslog with the help of input modules. Then, they are passed to ruleset, where rules are conditionally applied. When a rule matches, the message is transferred to an action, which then does something to the message, e.g. writes it to a file, database or forwards it to a remote host.

Processing Principles

- inputs submit received messages to rulesets
  - if the ruleset is not specifically bound, the default ruleset is used
- by default, there is one ruleset (RSYSLOG_DefaultRuleset)
- additional rulesets can be user-defined
- each ruleset contains zero or more rules

  - while it is permitted to have zero rules inside a ruleset, this obviously makes no sense

- a rule consists of a filter and an action list
- filters provide yes/no decisions and thus control-of-flow capability
- if a filter "matches" (filter says "yes"), the corresponding action list is executed. If it does not match, nothing special happens
- rules are evaluated in sequence from the first to the last rule **inside** the given ruleset. No rules from unrelated rulesets are ever executed.
- all rules are **always** fully evaluated, no matter if a filter matches or not (so we do **not** stop at the first match). If message processing shall stop, the "discard" action (represented by the tilde character or the stop command) must explicitly be executed. If discard is executed, message processing immediately stops, without evaluating any further rules.
- an action list contains one or many actions
- inside an action list no further filters are possible
- to have more than one action inside a list, the ampersand character must be placed in the position of the filter, and this must immediately follow the previous action
- actions consist of the action call itself (e.g. ":omusrmsg:") as well as all action-defining configuration statements ($Action... directives)
- if legacy format is used (see below), $Action... directives **must** be specified in front of the action they are intended to configure
- some config directives automatically refer to their previous values after being applied, others not. See the respective doc for details. Be warned that this is currently not always properly documented.
- in general, rsyslog v5 is heavily outdated and its native config language is a pain. The rsyslog project strongly recommends using at least version 7, where these problems are solved and configuration is much easier.
- legacy configuration statements (those starting with $) do **not** affect RainerScript objects (e.g. actions).

Configuration File

Upon startup, rsyslog reads its configuration from the `rsyslog.conf` file by default. This file may contain references to include other config files.

A different "root" configuration file can be specified via the `-f <file>` rsyslogd command line option. This is usually done within some init script or similar facility.

Statement Types

Rsyslog supports three different types of configuration statements concurrently:

- **sysklogd** - this is the plain old format, taught everywhere and still pretty useful for simple use cases. Note that some constructs are no longer supported because they are incompatible with newer features. These are mentioned in the compatibility docs.
- **legacy rsyslog** - these are statements that begin with a dollar sign. They set some configuration parameters and modify e.g. the way actions operate. This is the only format supported in pre-v6 versions of rsyslog. It is still fully supported in v6 and above. Note that some plugins and features may still only be available through legacy format (because plugins need to be explicitly upgraded to use the new style format, and this hasn't happened to all plugins).

- **RainerScript** - the new style format. This is the best and most precise format to be used for more complex cases. The rest of this page assumes RainerScript based rsyslog.conf.

The rsyslog.conf files consists of statements. For old style (sysklogd & legacy rsyslog), lines do matter. For new style (RainerScript) line spacing is irrelevant. Most importantly, this means with new style actions and all other objects can split across lines as users want to.

Recommended use of Statement Types

In general it is recommended to use RainerScript type statements, as these provide clean and easy to read control-of-flow as well as no doubt about which parameters are active. They also have no side-effects with include files, which can be a major obstacle with legacy rsyslog statements.

For very simple things sysklogd statement types are still suggested, especially if the full config consists of such simple things. The classical sample is writing to files (or forwarding) via priority. In sysklogd, this looks like:

```
mail.info /var/log/mail.log
mail.err @server.example.net
```

This is hard to beat in simplicity, still being taught in courses and a lot of people know this syntax. It is perfectly fine to use these constructs even in newly written config files.

As a rule of thumb, RainerScript config statements should be used when

- configuration parameters are required (e.g. the `Action...` type of legacy statements)
- more elaborate control-of-flow is required (e.g. when multiple actions must be nested under the same condition)

It is usually **not** recommended to use rsyslog legacy config format (those directives starting with a dollar sign). However, a few settings and modules have not yet been converted to RainerScript. In those cases, the legacy syntax must be used.

Comments

There are two types of comments:

- **#-Comments** - start with a hash sign (#) and run to the end of the line
- **C-style Comments** - start with /* and end with */, just like in the C programming language. They can be used to comment out multiple lines at once. Comment nesting is not supported, but #-Comments can be contained inside a C-style comment.

Processing Order

Directives are processed from the top of rsyslog.conf to the bottom. Order matters. For example, if you stop processing of a message, obviously all statements after the stop statement are never evaluated.

Flow Control Statements

Flow control is provided by:

- *Control Structures*
- *Filter Conditions*

Data Manipulation Statements

Data manipulation is achieved by **set**, **unset** and **reset** statements which are *documented here in detail*.

Inputs

Every input requires an input module to be loaded and a listener defined for it. Full details can be found inside the *rsyslog modules* documentation. Once loaded, inputs are defined via the **input()** object.

Outputs

Outputs are also called "actions". A small set of actions is pre-loaded (like the output file writer, which is used in almost every rsyslog.conf), others must be loaded just like inputs.

An action is invoked via the **action(type="type" ...)** object. Type is mandatory and must contain the name of the plugin to be called (e.g. "omfile" or "ommongodb"). Other parameters may be present. Their type and use depends on the output plugin in question.

Rulesets and Rules

Rulesets and rules form the basis of rsyslog processing. In short, a rule is a way how rsyslog shall process a specific message. Usually, there is a type of filter (if-statement) in front of the rule. Complex nesting of rules is possible, much like in a programming language.

Rulesets are containers for rules. A single ruleset can contain many rules. In the programming language analogy, one may think of a ruleset like being a program. A ruleset can be "bound" (assigned) to a specific input. In the analogy, this means that when a message comes in via that input, the "program" (ruleset) bound to it will be executed (but not any other!).

There is detailed documentation available for *rsyslog rulesets*.

For quick reference, rulesets are defined as follows:

```
ruleset(name="rulesetname") {
    action(type="omfile" file="/path/to/file")
    action(type="..." ...)
    /* and so on... */
}
```

Templates

Templates are a key feature of rsyslog. They allow to specify any format a user might want. They are also used for dynamic file name generation. Every output in rsyslog uses templates - this holds true for files, user messages and so on. The database writer expects its template to be a proper SQL statement - so this is highly customizable too. You might ask how does all of this work when no templates at all are specified. Good question ;). The answer is simple, though. Templates are compatible with the stock syslogd formats which are hardcoded into rsyslogd. So if no template is specified, we use one of those hardcoded templates. Search for "template_" in rsconf.c and you will find the hardcoded ones.

Templates are specified by template() statements. They can also be specified via $template legacy statements.

Note: key elements of templates are rsyslog properties. See the *rsyslog properties reference* for a list of which are available.

Template processing

Due to lack of standarization regarding logs formats, when a template is specified it's supposed to include HEADER, as defined in RFC5424

It's very important to have this in mind, and also how to understand how rsyslog parsing works

For example, if MSG field is set to "this:is a message" and no HOSTNAME, neither TAG are specified, outgoing parser will split the message as:

```
TAG:this:
MSG:is a message
```

The template() statement

The template() statement is used to define templates. Note that it is a **static** statement, that means all templates are defined when rsyslog reads the config file. As such, templates are not affected by if-statements or config nesting.

The basic structure of the template statement is as follows:

```
template(parameters)
```

In addition to this simpler syntax, list templates (to be described below) support an extended syntax:

```
template(parameters) { list-descriptions }
```

Each template has a parameter **name**, which specifies the template name, and a parameter **type**, which specifies the template type. The name parameter must be unique, and behaviour is unpredictable if it is not. The **type** parameter specifies different template types. Different types simply enable different ways to specify the template content. The template type **does not** affect what an (output) plugin can do with it. So use the type that best fits your needs (from a config writing point of view!). The following types are available:

- list
- subtree
- string
- plugin

The various types are described below.

list

In this case, the template is generated by a list of constant and variable statements. These follow the template spec in curly braces. This type is also primarily meant for use with structure-aware outputs, like ommongodb. However, it also works perfectly with text-based outputs. We recommend to use this mode if more complex property substitutions need to be done. In that case, the list-based template syntax is much clearer than the simple string-based one.

The list template contains the template header (with **type="list"**) and is followed by **constant** and **property** statements, given in curly braces to signify the template statement they belong to. As the name says, **constant** statements describe constant text and **property** describes property access. There are many options to **property**, described further below. Most of these options are used to extract only partial property contents or to modify the text obtained (like to change its case to upper or lower case, only).

To grasp the idea, an actual sample is:

```
template(name="tpl1" type="list") {
    constant(value="Syslog MSG is: '")
    property(name="msg")
    constant(value="', ")
    property(name="timereported" dateFormat="rfc3339" caseConversion="lower")
    constant(value="\n")
    }
```

This sample is probably primarily targeted at the usual file-based output.

constant statement

This provides a way to specify constant text. The text is used literally. It is primarily intended for text-based output, so that some constant text can be included. For example, if a complex template is built for file output, one usually needs to finish it by a newline, which can be introduced by a constant statement. Here is an actual sample of that use case from the rsylsog testbench:

```
template(name="outfmt" type="list") {
    property(name="$!usr!msgnum")
    constant(value="\n")
}
```

The following escape sequences are recognized inside the constant text:

- \\ - single backslash
- \n - LF
- \ooo - (three octal digits) - represents character with this numerical value (e.g. \101 equals "A"). Note that three octal digits must be given (in contrast to some languages, where between one and three are valid). While we support octal notation, we recommend to use hex notation as this is better known.
- \xhh - (where h is a hex digit) - represents character with this numerical value (e.g. \x41 equals "A"). Note that two hexadecimal digits must be given (in contrast to some languages where one or two are valid).
- ... some others ... list needs to be extended

Note: if an unsupported character follows a backslash, this is treated as an error. Behaviour is unpredictable in this case.

To aid usage of the same template both for text-based outputs and structured ones, constant text without an "outname" parameter will be ignored when creating the name/value tree for structured outputs. So if you want to supply some constant text e.g. to mongodb, you must include an outname, as can be seen here:

```
template(name="outfmt" type="list") {
    property(name="$!usr!msgnum")
    constant(value="\n" outname="IWantThisInMyDB")
}
```

The "constant" statement supports the following parameters:

- value - the constant value to use
- outname - output field name (for structured outputs)

property statement

This statement is used to include property text. It can access all properties. Also, options permit to specify picking only part of a property or modifying it. It supports the following parameters:

- name - the name of the property to access
- outname - output field name (for structured outputs)
- dateformat - date format to use (only for date-related properties)
- date.inUTC - date shall be shown in UTC (please note that this requires a bit more performance due to the necessary conversions) Available since 8.18.0.
- caseconversion - permits to convert case of the text. Supported values are "lower" and "upper"
- controlcharacters - specifies how to handle control characters. Supported values are "escape", which escapes them, "space", which replaces them by a single space, and "drop", which simply removes them from the string.
- securepath - used for creating pathnames suitable for use in dynafile templates
- format - specify format on a field basis. Supported values are:
  - "csv" for use when csv-data is generated
  - "json" which formats proper json content (but without a field header)
  - "jsonf" which formats as a complete json field
  - "jsonr" which avoids double escaping the value but makes it safe for a json field
  - "jsonfr" which is the combination of "jsonf" and "jsonr".
- position.from - obtain substring starting from this position (1 is the first position)
- position.to - obtain substring up to this position
- position.relativeToEnd - the from and to position is relative to the end of the string instead of the usual start of string. (available since rsyslog v7.3.10)
- fixedwidth - changes behaviour of position.to so that it pads the source string with spaces up to the value of position.to if the source string is shorter. "on" or "off" (default) (available since rsyslog v8.13.0)
- **compressspace - compresses multiple spaces (US-ASCII SP character) inside the** string to a single one. This compression happens at a very late stage in processing. Most importantly, it happens after substring extraction, so the **position.from** and **position.to** positions are **NOT** affected by this option. (available since v8.18.0).
- field.number - obtain this field match
- field.delimiter - decimal value of delimiter character for field extraction
- regex.expression - expression to use
- regex.type - either ERE or BRE
- regex.nomatchmode - what to do if we have no match
- regex.match - match to use
- regex.submatch - submatch to use
- droplastlf - drop a trailing LF, if it is present
- mandatory - signifies a field as mandatory. If set to "on", this field will always be present in data passed to structured outputs, even if it is empty. If "off" (the default) empty fields will not be passed to structured outputs. This is especially useful for outputs that support dynamic schemas (like ommongodb).

- spifno1stsp - expert options for RFC3164 template processing

subtree

Available since rsyslog 7.1.4

In this case, the template is generated based on a complete (CEE) subtree. This type of template is most useful for outputs that know how to process hierarchical structure, like ommongodb. With that type, the parameter **subtree** must be specified, which tells which subtree to use. For example template(name="tpl1" type="subtree" subtree="$!") includes all CEE data, while template(name="tpl2" type="subtree" subtree="$!usr!tpl2") includes only the subtree starting at $!usr!tpl2. The core idea when using this type of template is that the actual data is prefabricated via set and unset script statements, and the resulting structure is then used inside the template. This method MUST be used if a complete subtree needs to be placed *directly* into the object's root. With all other template types, only subcontainers can be generated. Note that subtree type can also be used with text-based outputs, like omfile. HOWEVER, you do not have any capability to specify constant text, and as such cannot include line breaks. As a consequence, using this template type for text outputs is usually only useful for debugging or very special cases (e.g. where the text is interpreted by a JSON parser later on).

Use case

A typical use case is to first create a custom subtree and then include it into the template, like in this small example:

```
set $!usr!tpl2!msg = $msg;
set $!usr!tpl2!dataflow = field($msg, 58, 2);
template(name="tpl2" type="subtree" subtree="$!usr!tpl2")
```

Here, we assume that $msg contains various fields, and the data from a field is to be extracted and stored - together with the message - as field content.

string

This closely resembles the legacy template statement. It has a mandatory parameter **string**, which holds the template string to be applied. A template string is a mix of constant text and replacement variables (see property replacer). These variables are taken from message or other dynamic content when the final string to be passed to a plugin is generated. String-based templates are a great way to specify textual content, especially if no complex manipulation to properties is necessary.

This is a sample for a string-based template:

```
template(name="tpl3" type="string"
         string="%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-
↪sp%%msg:::drop-last-lf%\n"
        )
```

The text between percent signs ('%') is interpreted by the rsyslog *property replacer*. In a nutshell, it contains the property to use as well as options for formatting and further processing. This is very similar to what the `property` object in list templates does (it actually is just a different language to express most of the same things).

Everything outside of the percent signs is constant text. In the above case, we have mostly spaces between the property values. At the end of the string, an escape sequence is used.

Escape sequences permit to specify nonprintable characters. They work very similar to escape sequences in C and many other languages. They are initiated by the backslash characters and followed by one or more characters that

specify the actual character. For example \7 is the US-ASCII BEL character and \n is a newline. The set is similar to what C and perl support, but a bit more limited.

plugin

In this case, the template is generated by a plugin (which is then called a "strgen" or "string generator"). The format is fixed as it is coded. While this is inflexible, it provides superior performance, and is often used for that reason (not that "regular" templates are slow - but in very demanding environments that "last bit" can make a difference). Refer to the plugin's documentation for further details. For this type, the parameter **plugin** must be specified and must contain the name of the plugin as it identifies itself. Note that the plugin must be loaded prior to being used inside a template. Config example:

```
template(name="tpl4" type="plugin" plugin="mystrgen")
```

options

The <options> part is optional. It carries options influencing the template as a whole and is a part of the template parameters. See details below. Be sure NOT to mistake template options with property options - the latter ones are processed by the property replacer and apply to a SINGLE property, only (and not the whole template). Template options are case-insensitive. Currently defined are:

option.sql - format the string suitable for a SQL statement in MySQL format. This will replace single quotes (""") and the backslash character by their backslash-escaped counterpart ("\'" and "\\") inside each field. Please note that in MySQL configuration, the `NO_BACKSLASH_ESCAPES` mode must be turned off for this format to work (this is the default).

option.stdsql - format the string suitable for a SQL statement that is to be sent to a standards-compliant sql server. This will replace single quotes (""") by two single quotes ("' ") inside each field. You must use stdsql together with MySQL if in MySQL configuration the `NO_BACKSLASH_ESCAPES` is turned on.

option.json - format the string suitable for a json statement. This will replace single quotes (""") by two single quotes ("' ") inside each field.

option.casesensitive - treat property name references as case sensitive. The default is "off", where all property name references are first converted to lowercase during template definition. With this option turned "on", property names are looked up as defined in the template. Use this option if you have JSON (`$!*`), local (`!.*`), or global (`$!\\*`) properties which container uppercase letters. The normal Rsyslog properties are case-insensitive, so this option is not needed for properly referencing those properties.

The use the options **option.sql**, **option.stdsql**, and **option.json** are mutually exclusive. Using more than one at the same time can cause unpredictable behaviour.

Either the **sql** or **stdsql** option **must** be specified when a template is used for writing to a database, otherwise injection might occur. Please note that due to the unfortunate fact that several vendors have violated the sql standard and introduced their own escape methods, it is impossible to have a single option doing all the work. So you yourself must make sure you are using the right format. **If you choose the wrong one, you are still vulnerable to sql injection.** Please note that the database writer *checks* that the sql option is present in the template. If it is not present, the write database action is disabled. This is to guard you against accidentally forgetting it and then becoming vulnerable to SQL injection. The sql option can also be useful with files - especially if you want to import them into a database on another machine for performance reasons. However, do NOT use it if you do not have a real need for it - among others, it takes some toll on the processing time. Not much, but on a really busy system you might notice it.

The default template for the write to database action has the sql option set. As we currently support only MySQL and the sql option matches the default MySQL configuration, this is a good choice. However, if you have turned on `NO_BACKSLASH_ESCAPES` in your MySQL config, you need to supply a template with the stdsql option. Otherwise you will become vulnerable to SQL injection.

```
template (name="TraditionalFormat" type="string"
string="%timegenerated% %HOSTNAME% %syslogtag%%msg%\\n"
```

Examples

Standard Template for Writing to Files

```
template(name="FileFormat" type="list") {
    property(name="timestamp" dateFormat="rfc3339")
    constant(value=" ")
    property(name="hostname")
    constant(value=" ")
    property(name="syslogtag")
    property(name="msg" spifno1stsp="on" )
    property(name="msg" droplastlf="on" )
    constant(value="\n")
    }
```

The equivalent string template looks like this:

```
template(name="FileFormat" type="string"
         string= "%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%
↪%msg:::drop-last-lf%\n"
        )
```

Note that the template string itself must be on a single line.

Standard Template for Forwarding to a Remote Host (RFC3164 mode)

```
template(name="ForwardFormat" type="list") {
    constant(value="<")
    property(name="pri")
    constant(value=">")
    property(name="timestamp" dateFormat="rfc3339")
    constant(value=" ")
    property(name="hostname")
    constant(value=" ")
    property(name="syslogtag" position.from="1" position.to="32")
    property(name="msg" spifno1stsp="on" )
    property(name="msg")
    }
```

The equivalent string template looks like this:

```
template(name="forwardFormat" type="string"
         string="<%PRI%>%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag:1:32%
↪%msg:::sp-if-no-1st-sp%%msg%"
        )
```

Note that the template string itself must be on a single line.

Standard Template for write to the MySQL database

```
template(name="StdSQLformat" type="list" option.sql="on") {
        constant(value="insert into SystemEvents (Message, Facility, FromHost,␣
↪Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag)")
        constant(value=" values ('")
        property(name="msg")
        constant(value="', ")
        property(name="syslogfacility")
        constant(value=", '")
        property(name="hostname")
        constant(value="', ")
        property(name="syslogpriority")
        constant(value=", '")
        property(name="timereported" dateFormat="mysql")
        constant(value="', '")
        property(name="timegenerated" dateFormat="mysql")
        constant(value="', ")
        property(name="iut")
        constant(value=", '")
        property(name="syslogtag")
        constant(value="')")
        }
```

The equivalent string template looks like this:

```
template(name="stdSQLformat" type="string" option.sql="on"
         string="insert into SystemEvents (Message, Facility, FromHost, Priority,␣
↪DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag) values ('%msg%',
↪%syslogfacility%, '%HOSTNAME%', %syslogpriority%, '%timereported:::date-mysql%', '
↪%timegenerated:::date-mysql%', %iut%, '%syslogtag%')"
        )
```

Note that the template string itself must be on a single line.

Creating Dynamic File Names for omfile

Templates can be used to generate actions with dynamic file names. For example, if you would like to split syslog messages from different hosts to different files (one per host), you can define the following template:

```
template (name="DynFile" type="string" string="/var/log/system-%HOSTNAME%.log")
```

Legacy example:

```
$template DynFile,"/var/log/system-%HOSTNAME%.log"
```

This template can then be used when defining an action. It will result in something like "/var/log/system-localhost.log"

legacy format

In pre v6-versions of rsyslog, you need to use the `$template` statement to configure templates. They provide the equivalent to string- and plugin-based templates. The legacy syntax continuous to work in v7, however we recommend to avoid legacy format for newly written config files. Legacy and current config statements can coexist within the same config file.

The general format is

```
$template name,param[,options]
```

where "name" is the template name and "param" is a single parameter that specifies template content. The optional "options" part is used to set template options.

string

The parameter is the same string that with the current-style format you specify in the **string** parameter, for example:

```
$template strtpl,"PRI: %pri%, MSG: %msg%\n"
```

Note that list templates are not available in legacy format, so you need to use complex property replacer constructs to do complex things.

plugin

This is equivalent to the "plugin"-type template directive. Here, the parameter is the plugin name, with an equal sign prepended. An example is:

```
$template plugintpl,=myplugin
```

Reserved Template Names

Template names beginning with "RSYSLOG_" are reserved for rsyslog use. Do NOT use them if, otherwise you may receive a conflict in the future (and quite unpredictable behaviour). There is a small set of pre-defined templates that you can use without the need to define it:

- **RSYSLOG_TraditionalFileFormat** - the "old style" default log file format with low-precision timestamps
- **RSYSLOG_FileFormat** - a modern-style logfile format similar to TraditionalFileFormat, both with high-precision timestamps and timezone information
- **RSYSLOG_TraditionalForwardFormat** - the traditional forwarding format with low-precision timestamps. Most useful if you send messages to other syslogd's or rsyslogd below version 3.12.5.
- **RSYSLOG_SysklogdFileFormat** - sysklogd compatible log file format. If used with options: `$SpaceLFOnReceive on`, `$EscapeControlCharactersOnReceive off`, `$DropTrailingLFOnReception off`, the log format will conform to sysklogd log format.
- **RSYSLOG_ForwardFormat** - a new high-precision forwarding format very similar to the traditional one, but with high-precision timestamps and timezone information. Recommended to be used when sending messages to rsyslog 3.12.5 or above.
- **RSYSLOG_SyslogProtocol23Format** - the format specified in IETF's internet-draft ietf-syslog-protocol-23, which is very close to the actual syslog standard RFC5424 (we couldn't update this template as things were in production for quite some time when RFC5424 was finally approved). This format includes several improvements. You may use this format with all relatively recent versions of rsyslog or syslogd.
- **RSYSLOG_DebugFormat** - a special format used for troubleshooting property problems. This format is meant to be written to a log file. Do **not** use for production or remote forwarding.

Legacy String-based Template Samples

This section provides some default templates in legacy format, as used in rsyslog previous to version 6. Note that this format is still supported, so there is no hard need to upgrade existing configurations. However, it is strongly recommended that the legacy constructs are not used when crafting new templates. Note that each $template statement is on a **single** line, but probably broken across several lines for display purposes by your browsers. Lines are separated by empty lines. Keep in mind, that line breaks are important in legacy format.

```
$template FileFormat,"%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag%%msg:::sp-if-
↪no-1st-sp%%msg:::drop-last-lf%\n"
$template TraditionalFileFormat,"%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-
↪1st-sp%%msg:::drop-last-lf%\n"
$template ForwardFormat,"<%PRI%>%TIMESTAMP:::date-rfc3339% %HOSTNAME% %syslogtag:1:32%
↪%msg:::sp-if-no-1st-sp%%msg%"
$template TraditionalForwardFormat,"<%PRI%>%TIMESTAMP% %HOSTNAME% %syslogtag:1:32%
↪%msg:::sp-if-no-1st-sp%%msg%"
$template StdSQLFormat,"insert into SystemEvents (Message, Facility, FromHost,␣
↪Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag) values ('%msg%',
↪%syslogfacility%, '%HOSTNAME%', %syslogpriority%, '%timereported:::date-mysql%', '
↪%timegenerated:::date-mysql%', %iut%, '%syslogtag%')",SQL``
```

See Also

- How to bind a template
- Adding the BOM to a message
- How to separate log files by host name of the sending device

rsyslog Properties

Data items in rsyslog are called "properties". They can have different origin. The most important ones are those that stem from received messages. But there are also others. Whenever you want to access data items, you need to access the respective property.

Properties are used in

- *templates*
- conditional statements

The property name is case-insensitive (prior to 3.17.0, they were case-senstive).

Message Properties

These are extracted by rsyslog parsers from the original message. All message properties start with a letter.

The following message properties exist:

msg the MSG part of the message (aka "the message" ;))

rawmsg the message "as is". Should be useful for debugging and also if a message should be forwarded totally unaltered. Please notice *EscapecontrolCharactersOnReceive* is enabled by default, so it may be different from what was received in the socket.

rawmsg-after-pri Almost the same as **rawmsg**, but the syslog PRI is removed. If no PRI was present, **rawmsg-after-pri** is identical to **rawmsg**. Note that the syslog PRI is header field that contains information on syslog facility and severity. It is enclosed in greater-than and less-than characters, e.g. "<191>". This field is often not written to log files, but usually needs to be present for the receiver to properly classify the message. There are some rare cases where one wants the raw message, but not the PRI. You can use this property to obtain that. In general, you should know that you need this format, otherwise stay away from the property.

hostname hostname from the message

source alias for HOSTNAME

fromhost hostname of the system the message was received from (in a relay chain, this is the system immediately in front of us and not necessarily the original sender). This is a DNS-resolved name, except if that is not possible or DNS resolution has been disabled.

fromhost-ip The same as fromhost, but always as an IP address. Local inputs (like imklog) use 127.0.0.1 in this property.

syslogtag TAG from the message

programname the "static" part of the tag, as defined by BSD syslogd. For example, when TAG is "named[12345]", programname is "named".

Precisely, the programname is terminated by either (whichever occurs first):

- end of tag
- nonprintable character
- ':'
- '['
- '/'

The above definition has been taken from the FreeBSD syslogd sources.

Please note that some applications include slashes in the static part of the tag, e.g. "app/foo[1234]". In this case, programname is "app". If they store an absolute path name like in "/app/foo[1234]", programname will become empty (""). If you need to actually store slashes as part of the programname, you can use the global option

global(parser.permitSlashInProgramName="on")

to permit this. Then, a syslogtag of "/app/foo[1234]" will result in programname being "/app/foo". Note: this option is available starting at rsyslogd version 8.25.0.

pri PRI part of the message - undecoded (single value)

pri-text the PRI part of the message in a textual form with the numerical PRI appended in brackes (e.g. "local0.err<133>")

iut the monitorware InfoUnitType - used when talking to a MonitorWare backend (also for Adiscon LogAnalyzer)

syslogfacility the facility from the message - in numerical form

syslogfacility-text the facility from the message - in text form

syslogseverity severity from the message - in numerical form

syslogseverity-text severity from the message - in text form

syslogpriority an alias for syslogseverity - included for historical reasons (be careful: it still is the severity, not PRI!)

syslogpriority-text an alias for syslogseverity-text

timegenerated timestamp when the message was RECEIVED. Always in high resolution

timereported timestamp from the message. Resolution depends on what was provided in the message (in most cases, only seconds)

timestamp alias for timereported

protocol-version The contents of the PROTCOL-VERSION field from IETF draft draft-ietf-syslog-protcol

structured-data The contents of the STRUCTURED-DATA field from IETF draft draft-ietf-syslog-protocol

app-name The contents of the APP-NAME field from IETF draft draft-ietf-syslog-protocol

procid The contents of the PROCID field from IETF draft draft-ietf-syslog-protocol

msgid The contents of the MSGID field from IETF draft draft-ietf-syslog-protocol

inputname The name of the input module that generated the message (e.g. "imuxsock", "imudp"). Note that not all modules necessarily provide this property. If not provided, it is an empty string. Also note that the input module may provide any value of its liking. Most importantly, it is **not** necessarily the module input name. Internal sources can also provide inputnames. Currently, "rsyslogd" is defined as inputname for messages internally generated by rsyslogd, for example startup and shutdown and error messages. This property is considered useful when trying to filter messages based on where they originated - e.g. locally generated messages ("rsyslogd", "imuxsock", "imklog") should go to a different place than messages generated somewhere.

jsonmesg

Available since rsyslog 8.3.0

The whole message object as JSON representation. Note that the JSON string will *not* include and LF and it will contain *all other message properties* specified here as respective JSON containers. It also includes all message variables in the "$!" subtree (this may be null if none are present).

This property is primarily meant as an interface to other systems and tools that want access to the full property set (namely external plugins). Note that it contains the same data items potentially multiple times. For example, parts of the syslog tag will by containened in the rawmsg, syslogtag, and programname properties. As such, this property has some additional overhead. Thus, it is suggested to be used only when there is actual need for it.

System Properties

These properties are provided by the rsyslog core engine. They are **not** related to the message. All system properties start with a dollar-sign.

Special care needs to be taken in regard to time-related system variables:

- `timereported` contains the timestamp that is contained within the message header. Ideally, it resembles the time when the message was created at the original sender. Depending on how long the message was in the relay chain, this can be quite old.
- `timegenerated` contains the timestamp when the message was received by the local system. Here "received" actually means the point in time when the message was handed over from the OS to rsyslog's reception buffers, but before any actual processing takes place. This also means a message is "received" before it is placed into any queue. Note that depending on the input, some minimal processing like extraction of the actual message content from the receive buffer can happen. If multiple messages are received via the same receive buffer (a common scenario for example with TCP-based syslog), they bear the same `timegenerated` stamp because they actually were received at the same time.
- `$now` is **not** from the message. It is the system time when the message is being **processed**. There is always a small difference between `timegenerated` and `$now` because processing always happens after reception. If the message is sitting inside a queue on the local system, the time difference between the two can be some seconds (e.g. due to a message burst and in-memory queueing) up to several hours in extreme cases where a message is sitting inside a disk queue (e.g. due to a database outage). The `timereported` property is

usually older than `timegenerated`, but may be totally different due to differences in time and time zone configuration between systems.

The following system properties exist:

$bom The UTF-8 encoded Unicode byte-order mask (BOM). This may be useful in templates for RFC5424 support, when the character set is know to be Unicode.

$myhostname The name of the current host as it knows itself (probably useful for filtering in a generic way)

Time-Related System Properties

All of these system properties exist in a local time variant (e.g. $now) and a variant that emits UTC (e.g. $now-utc). The UTC variant is always available by appending "-utc". Note that within a single template, only the localtime or UTC variant should be used. It is possible to mix both variants within a single template. However, in this case it is **not** guaranteed that both variants given exactly the same time. The technical reason behind is that rsyslog needs to re-query system time when the variant is changed. So we strongly recommend not mixing both variants in the same template.

Note that use in different templates will generate a consistent timestamp within each template. However, as $now always provides local system time at time of using it, time may advance and consequently different templates may have different time stamp. To avoid this, use *timegenerated* instead.

$now The current date stamp in the format YYYY-MM-DD

$year The current year (4-digit)

$month The current month (2-digit)

$day The current day of the month (2-digit)

$hour The current hour in military (24 hour) time (2-digit)

$hhour The current half hour we are in. From minute 0 to 29, this is always 0 while from 30 to 59 it is always 1.

$qhour The current quarter hour we are in. Much like $HHOUR, but values range from 0 to 3 (for the four quater hours that are in each hour)

$minute The current minute (2-digit)

The Property Replacer

The property replacer is a core component in rsyslogd's string template system. A syslog message has a number of well-defined properties. Each of this properties can be accessed **and** manipulated by the property replacer. With it, it is easy to use only part of a property value or manipulate the value, e.g. by converting all characters to lower case.

Accessing Properties

Syslog message properties are used inside templates. They are accessed by putting them between percent signs. Properties can be modified by the property replacer. The full syntax is as follows:

```
%property:fromChar:toChar:options%
```

Available Properties

The property replacer can use all *rsyslog properties*.

Character Positions

FromChar and **toChar** are used to build substrings. They specify the offset within the string that should be copied. Offset counting starts at 1, so if you need to obtain the first 2 characters of the message text, you can use this syntax: "%msg:1:2%". If you do not whish to specify from and to, but you want to specify options, you still need to include the colons. For example, if you would like to convert the full message text to lower case, use "%msg:::lowercase%". If you would like to extract from a position until the end of the string, you can place a dollar-sign ("$") in toChar (e.g. %msg:10:$%, which will extract from position 10 to the end of the string).

There is also support for **regular expressions**. To use them, you need to place a "R" into FromChar. This tells rsyslog that a regular expression instead of position-based extraction is desired. The actual regular expression must then be provided in toChar. The regular expression **must** be followed by the string "–end". It denotes the end of the regular expression and will not become part of it. If you are using regular expressions, the property replacer will return the part of the property text that matches the regular expression. An example for a property replacer sequence with a regular expression is: "%msg:R:.*Sev:. \(.*\) \[.*–end%"

It is possible to specify some parametes after the "R". These are comma-separated. They are:

R,<regexp-type>,<submatch>,<*nomatch*>,<match-number>

regexp-type is either "BRE" for Posix basic regular expressions or "ERE" for extended ones. The string must be given in upper case. The default is "BRE" to be consistent with earlier versions of rsyslog that did not support ERE. The submatch identifies the submatch to be used with the result. A single digit is supported. Match 0 is the full match, while 1 to 9 are the acutal submatches. The match-number identifies which match to use, if the expression occurs more than once inside the string. Please note that the first match is number 0, the second 1 and so on. Up to 10 matches (up to number 9) are supported. Please note that it would be more natural to have the match-number in front of submatch, but this would break backward-compatibility. So the match-number must be specified after "nomatch".

nomatch specifies what should be used in case no match is found.

The following is a sample of an ERE expression that takes the first submatch from the message string and replaces the expression with the full field if no match is found:

```
%msg:R,ERE,1,FIELD:for (vlan[0-9]\*):--end%
```

and this takes the first submatch of the second match of said expression:

```
%msg:R,ERE,1,FIELD,1:for (vlan[0-9]\*):--end%
```

Please note: there is also a rsyslog regular expression checker/generator **online tool available.** With that tool, you can check your regular expressions and also generate a valid property replacer sequence. Usage of this tool is recommended. Depending on the version offered, the tool may not cover all subleties that can be done with the property replacer. It concentrates on the most often used cases. So it is still useful to hand-craft expressions for demanding environments.

Also, extraction can be done based on so-called "fields". To do so, place a "F" into FromChar. A field in its current definition is anything that is delimited by a delimiter character. The delimiter by default is TAB (US-ASCII value 9). However, if can be changed to any other US-ASCII character by specifying a comma and the **decimal** US-ASCII value of the delimiter immediately after the "F". For example, to use comma (",") as a delimiter, use this field specifier: "F,44". If your syslog data is delimited, this is a quicker way to extract than via regular expressions (actually, a *much* quicker way). Field counting starts at 1. Field zero is accepted, but will always lead to a "field not found" error. The same happens if a field number higher than the number of fields in the property is requested. The field number must be placed in the "ToChar" parameter. An example where the 3rd field (delimited by TAB) from the msg property is extracted is as follows: "%msg:F:3%". The same example with semicolon as delimiter is "%msg:F,59:3%".

The use of fields does not permit to select substrings, what is rather unfortunate. To solve this issue, starting with 6.3.9, fromPos and toPos can be specified for strings as well. However, the syntax is quite ugly, but it was the only way to integrate this functonality into the already-existing system. To do so, use ",fromPos" and ",toPos" during field

extraction. Let's assume you want to extract the substring from position 5 to 9 in the previous example. Then, the syntax is as follows: "%msg:F,59,5:3,9%". As you can see, "F,59" means field-mode, with semicolon delimiter and ",5" means starting at position 5. Then "3,9" means field 3 and string extraction to position 9.

Please note that the special characters "F" and "R" are case-sensitive. Only upper case works, lower case will return an error. There are no white spaces permitted inside the sequence (that will lead to error messages and will NOT provide the intended result).

Each occurence of the field delimiter starts a new field. However, if you add a plus sign ("+") after the field delimiter, multiple delimiters, one immediately after the others, are treated as separate fields. This can be useful in cases where the syslog message contains such sequences. A frequent case may be with code that is written as follows:

```
int n, m;
...
syslog(LOG_ERR, "%d test %6d", n, m);
```

This will result into things like this in syslog messages: "1 test 2", "1 test 23", "1 test 234567"

As you can see, the fields are delimited by space characters, but their exact number is unknown. They can properly be extracted as follows:

```
"%msg:F,32:2%" to "%msg:F,32+:2%".
```

This feature was suggested by Zhuang Yuyao and implemented by him. It is modeled after perl compatible regular expressions.

Property Options

Property options are case-insensitive. Currently, the following options are defined:

uppercase convert property to uppercase only

lowercase convert property text to lowercase only

fixed-width changes behaviour of toChar so that it pads the source string with spaces up to the value of toChar if the source string is shorter. *This feature was introduced in rsyslog 8.13.0*

json encode the value so that it can be used inside a JSON field. This means that several characters (according to the JSON spec) are being escaped, for example US-ASCII LF is replaced by "\n". The json option cannot be used together with either jsonf or csv options.

jsonf[:outname] (available in 6.3.9+) This signifies that the property should be expressed as a JSON field. That means not only the property is written, but rather a complete JSON field in the format

`"fieldname"="value"`

where "fieldname" is given in the *outname* property (or the property name if none was assigned) and value is the end result of property replacer operation. Note that value supports all property replacer options, like substrings, case converson and the like. Values are properly JSON-escaped, however field names are (currently) not, so it is expected that proper field names are configured. The jsonf option cannot be used together with either json or csv options.

For more information you can read this article from Rainer's blog.

csv formats the resulting field (after all modifications) in CSV format as specified in RFC 4180. Rsyslog will always use double quotes. Note that in order to have full CSV-formatted text, you need to define a proper template. An example is this one: $template csvline,"%syslogtag:::csv%,%msg:::csv%" Most importantly, you need to provide the commas between the fields inside the template. *This feature was introduced in rsyslog 4.1.6.*

drop-last-lf The last LF in the message (if any), is dropped. Especially useful for PIX.

date-utc convert data to UTC prior to outputing it (available since 8.18.0)

date-mysql format as mysql date

date-rfc3164 format as RFC 3164 date

date-rfc3164-buggyday similar to date-rfc3164, but emulates a common coding error: RFC 3164 demands that a space is written for single-digit days. With this option, a zero is written instead. This format seems to be used by syslog-ng and the date-rfc3164-buggyday option can be used in migration scenarios where otherwise lots of scripts would need to be adjusted. It is recommended *not* to use this option when forwarding to remote hosts - they may treat the date as invalid (especially when parsing strictly according to RFC 3164).

This feature was introduced in rsyslog 4.6.2 and v4 versions above and 5.5.3 and all versions above.

date-rfc3339 format as RFC 3339 date

date-unixtimestamp Format as a unix timestamp (seconds since epoch)

date-year just the year part (4-digit) of a timestamp

date-month just the month part (2-digit) of a timestamp

date-day just the day part (2-digit) of a timestamp

date-hour just the hour part (2-digit, 24-hour clock) of a timestamp

date-minute just the minute part (2-digit) of a timestamp

date-second just the second part (2-digit) of a timestamp

date-subseconds just the subseconds of a timestamp (always 0 for a low precision timestamp)

date-tzoffshour just the timezone offset hour part (2-digit) of a timestamp

date-tzoffsmin just the timezone offset minute part (2-digit) of a timestamp. Note that this is usually 0, but there are some time zones that have offsets which are not hourly-granular. If so, this is the minute offset.

date-tzoffsdirection just the timezone offset direction part of a timestamp. This specifies if the offsets needs to be added ("+") or subtracted ("-") to the timestamp in order to get UTC.

date-ordinal returns the ordinal for the given day, e.g. it is 2 for January, 2nd

date-week returns the week number

date-wday just the weekday number of the timstamp. This is a single digit, with 0=Sunday, 1=Monday, ..., 6=Saturday.

date-wdayname just the abbreviated english name of the weekday (e.g. "Mon", "Sat") of the timestamp.

escape-cc replace control characters (ASCII value 127 and values less then 32) with an escape sequence. The sequence is "#<charval>" where charval is the 3-digit decimal value of the control character. For example, a tabulator would be replaced by "#009". Note: using this option requires that $EscapeControlCharactersOnReceive is set to off.

space-cc replace control characters by spaces Note: using this option requires that $EscapeControlCharactersOnReceive is set to off.

drop-cc drop control characters - the resulting string will neither contain control characters, escape sequences nor any other replacement character like space. Note: using this option requires that $EscapeControlCharactersOnReceive is set to off.

compressspace compresses multiple spaces (US-ASCII SP character) inside the string to a single one. This compression happens at a very late stage in processing. Most importantly, it happens after substring extraction, so the **FromChar** and **ToChar** positions are **NOT** affected by this option. (available since v8.18.0)

sp-if-no-1st-sp This option looks scary and should probably not be used by a user. For any field given, it returns either a single space character or no character at all. Field content is never returned. A space is returned if (and only if) the first character of the field's content is NOT a space. This option is kind of a hack to solve a problem rooted in RFC 3164: 3164 specifies no delimiter between the syslog tag sequence and the actual message text. Almost all implementation in fact delimit the two by a space. As of RFC 3164, this space is part of the message text itself. This leads to a problem when building the message (e.g. when writing to disk or forwarding). Should a delimiting space be included if the message does not start with one? If not, the tag is immediately followed by another non-space character, which can lead some log parsers to misinterpret what is the tag and what the message. The problem finally surfaced when the klog module was restructured and the tag correctly written. It exists with other message sources, too. The solution was the introduction of this special property replacer option. Now, the default template can contain a conditional space, which exists only if the message does not start with one. While this does not solve all issues, it should work good enough in the far majority of all cases. If you read this text and have no idea of what it is talking about - relax: this is a good indication you will never need this option. Simply forget about it ;)

secpath-drop Drops slashes inside the field (e.g. "a/b" becomes "ab"). Useful for secure pathname generation (with dynafiles).

secpath-replace Replace slashes inside the field by an underscore. (e.g. "a/b" becomes "a_b"). Useful for secure pathname generation (with dynafiles).

To use multiple options, simply place them one after each other with a comma delimiting them. For example "escape-cc,sp-if-no-1st-sp". If you use conflicting options together, the last one will override the previous one. For example, using "escape-cc,drop-cc" will use drop-cc and "drop-cc,escape-cc" will use escape-cc mode.

Further Links

- Article on "*Recording the Priority of Syslog Messages*" (describes use of templates to record severity and facility of a message)
- Configuration file syntax, this is where you actually use the property replacer.

Property Replacer nomatch mode

The "nomatch-Mode" specifies which string the property replacer shall return if a regular expression did not find the search string.. Traditionally, the string "**NO MATCH**" was returned, but many people complained this was almost never useful. Still, this mode is support as "**DFLT**" for legacy configurations.

Three additional and potentially useful modes exist: in one (**BLANK**) a blank string is returned. This is probably useful for inserting values into databases where no value shall be inserted if the expression could not be found.

A similar mode is "**ZERO**" where the string "0" is returned. This is suitable for numerical values. A use case may be that you record a traffic log based on firewall rules and the "bytes transmitted" counter is extracted via a regular expression. If no "bytes transmitted" counter is available in the current message, it is probably a good idea to return an empty string, which the database layer can turn into a zero.

The other mode is "**FIELD**", in which the complete field is returned. This may be useful in cases where absense of a match is considered a failure and the message that triggered it shall be logged.

If in doubt, **it is highly suggested to use the** rsyslog online regular expression checker and generator **to see these options in action**. With that online tool, you can craft regular expressions based on samples and try out the different modes.

Summary of nomatch Modes

Mode	Returned
DFLT	"**NO MATCH**"
BLANK	"" (empty string)
ZERO	"0"
FIELD	full content of original field
	Interactive Tool

Filter Conditions

Rsyslog offers four different types "filter conditions":

- "traditional" severity and facility based selectors
- property-based filters
- expression-based filters
- BSD-style blocks (not upward compatible)

Selectors

Selectors are the traditional way of filtering syslog messages. They have been kept in rsyslog with their original syntax, because it is well-known, highly effective and also needed for compatibility with stock syslogd configuration files. If you just need to filter based on priority and facility, you should do this with selector lines. They are **not** second-class citizens in rsyslog and offer the best performance for this job.

The selector field itself again consists of two parts, a facility and a priority, separated by a period (".'‘). Both parts are case insensitive and can also be specified as decimal numbers, but don't do that, you have been warned. Both facilities and priorities are described in syslog(3). The names mentioned below correspond to the similar LOG_-values in /usr/include/syslog.h.

The facility is one of the following keywords: auth, authpriv, cron, daemon, kern, lpr, mail, mark, news, security (same as auth), syslog, user, uucp and local0 through local7. The keyword security should not be used anymore and mark is only for internal use and therefore should not be used in applications. Anyway, you may want to specify and redirect these messages here. The facility specifies the subsystem that produced the message, i.e. all mail programs log with the mail facility (LOG_MAIL) if they log using syslog.

The priority is one of the following keywords, in ascending order: debug, info, notice, warning, warn (same as warning), err, error (same as err), crit, alert, emerg, panic (same as emerg). The keywords error, warn and panic are deprecated and should not be used anymore. The priority defines the severity of the message.

The behavior of the original BSD syslogd is that all messages of the specified priority and higher are logged according to the given action. Rsyslogd behaves the same, but has some extensions.

In addition to the above mentioned names the rsyslogd(8) understands the following extensions: An asterisk ("*'‘) stands for all facilities or all priorities, depending on where it is used (before or after the period). The keyword none stands for no priority of the given facility. You can specify multiple facilities with the same priority pattern in one statement using the comma (",'‘) operator. You may specify as much facilities as you want. Remember that only the facility part from such a statement is taken, a priority part would be skipped.

Multiple selectors may be specified for a single action using the semicolon (";") separator. Remember that each selector in the selector field is capable to overwrite the preceding ones. Using this behavior you can exclude some priorities from the pattern.

Rsyslogd has a syntax extension to the original BSD source, that makes its use more intuitively. You may precede every priority with an equals sign ("=") to specify only this single priority and not any of the above. You may also (both is valid, too) precede the priority with an exclamation mark ("!") to ignore all that priorities, either exact this one or this and any higher priority. If you use both extensions than the exclamation mark must occur before the equals sign, just use it intuitively.

Property-Based Filters

Property-based filters are unique to rsyslogd. They allow to filter on any property, like HOSTNAME, syslogtag and msg. A list of all currently-supported properties can be found in the property replacer documentation (but keep in mind that only the properties, not the replacer is supported). With this filter, each properties can be checked against a specified value, using a specified compare operation.

A property-based filter must start with a colon **in column 1**. This tells rsyslogd that it is the new filter type. The colon must be followed by the property name, a comma, the name of the compare operation to carry out, another comma and then the value to compare against. This value must be quoted. There can be spaces and tabs between the commas. Property names and compare operations are case-sensitive, so "msg" works, while "MSG" is an invalid property name. In brief, the syntax is as follows:

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

Compare-Operations

The following **compare-operations** are currently supported:

contains Checks if the string provided in value is contained in the property. There must be an exact match, wildcards are not supported.

isequal Compares the "value" string provided and the property contents. These two values must be exactly equal to match. The difference to contains is that contains searches for the value anywhere inside the property value, whereas all characters must be identical for isequal. As such, isequal is most useful for fields like syslogtag or FROMHOST, where you probably know the exact contents.

startswith Checks if the value is found exactly at the beginning of the property value. For example, if you search for "val" with

```
:msg, startswith, "val"
```

it will be a match if msg contains "values are in this message" but it won't match if the msg contains "There are values in this message" (in the later case, *"contains"* would match). Please note that "startswith" is by far faster than regular expressions. So even once they are implemented, it can make very much sense (performance-wise) to use "startswith".

regex Compares the property against the provided POSIX BRE regular expression.

ereregex Compares the property against the provided POSIX ERE regular expression.

You can use the bang-character (!) immediately in front of a compare-operation, the outcome of this operation is negated. For example, if msg contains "This is an informative message", the following sample would not match:

```
:msg, contains, "error"
```

but this one matches:

```
:msg, !contains, "error"
```

Using negation can be useful if you would like to do some generic processing but exclude some specific events. You can use the discard action in conjunction with that. A sample would be:

```
*.* /var/log/allmsgs-including-informational.log
:msg, contains, "informational"  ~
*.* /var/log/allmsgs-but-informational.log
```

Do not overlook the tilde in line 2! In this sample, all messages are written to the file allmsgs-including-informational.log. Then, all messages containing the string "informational" are discarded. That means the config file lines below the "discard line" (number 2 in our sample) will not be applied to this message. Then, all remaining lines will also be written to the file allmsgs-but-informational.log.

Value Part

Value is a quoted string. It supports some escape sequences:

\" - the quote character (e.g. "String with \"Quotes\"") \\ - the backslash character (e.g. "C:\\tmp")

Escape sequences always start with a backslash. Additional escape sequences might be added in the future. Backslash characters **must** be escaped. Any other sequence then those outlined above is invalid and may lead to unpredictable results.

Probably, "msg" is the most prominent use case of property based filters. It is the actual message text. If you would like to filter based on some message content (e.g. the presence of a specific code), this can be done easily by:

```
:msg, contains, "ID-4711"
```

This filter will match when the message contains the string "ID-4711". Please note that the comparison is case-sensitive, so it would not match if "id-4711" would be contained in the message.

```
:msg, regex, "fatal .* error"
```

This filter uses a POSIX regular expression. It matches when the string contains the words "fatal" and "error" with anything in between (e.g. "fatal net error" and "fatal lib error" but not "fatal error" as two spaces are required by the regular expression!).

Getting property-based filters right can sometimes be challenging. In order to help you do it with as minimal effort as possible, rsyslogd spits out debug information for all property-based filters during their evaluation. To enable this, run rsyslogd in foreground and specify the "-d" option.

Boolean operations inside property based filters (like 'message contains "ID17" or message contains "ID18"') are currently not supported (except for "not" as outlined above). Please note that while it is possible to query facility and severity via property-based filters, it is far more advisable to use classic selectors (see above) for those cases.

Expression-Based Filters

Expression based filters allow filtering on arbitrary complex expressions, which can include boolean, arithmetic and string operations. Expression filters will evolve into a full configuration scripting language. Unfortunately, their syntax will slightly change during that process. So if you use them now, you need to be prepared to change your configuration files some time later. However, we try to implement the scripting facility as soon as possible (also in respect to stage work needed). So the window of exposure is probably not too long.

Expression based filters are indicated by the keyword "if" in column 1 of a new line. They have this format:

```
if expr then action-part-of-selector-line
```

"if" and "then" are fixed keywords that mus be present. "expr" is a (potentially quite complex) expression. So the expression documentation for details. "action-part-of-selector-line" is an action, just as you know it (e.g. "/var/log/logfile" to write to that file).

BSD-style Blocks

Note: rsyslog v7+ does no longer support BSD-style blocks for technical reasons. So it is strongly recommended **not** to use them.

Rsyslogd supports BSD-style blocks inside rsyslog.conf. Each block of lines is separated from the previous block by a program or hostname specification. A block will only log messages corresponding to the most recent program and hostname specifications given. Thus, a block which selects 'ppp' as the program, directly followed by a block that selects messages from the hostname 'dialhost', then the second block will only log messages from the ppp program on dialhost.

A program specification is a line beginning with '!prog' and the following blocks will be associated with calls to syslog from that specific program. A program specification for 'foo' will also match any message logged by the kernel with the prefix 'foo: '. Alternatively, a program specification '-foo' causes the following blocks to be applied to messages from any program but the one specified. A hostname specification of the form '+hostname' and the following blocks will be applied to messages received from the specified hostname. Alternatively, a hostname specification '-hostname' causes the following blocks to be applied to messages from any host but the one specified. If the hostname is given as '@', the local hostname will be used. (NOT YET IMPLEMENTED) A program or hostname specification may be reset by giving the program or hostname as '*'.

Please note that the "#!prog", "#+hostname" and "#-hostname" syntax available in BSD syslogd is not supported by rsyslogd. By default, no hostname or program is set.

Examples

```
*.* /var/log/file1 # the traditional way
if $msg contains 'error' then /var/log/errlog # the expression-based way
```

Right now, you need to specify numerical values if you would like to check for facilities and severity. These can be found in **RFC 5424**. If you don't like that, you can of course also use the textual property - just be sure to use the right one. As expression support is enhanced, this will change. For example, if you would like to filter on message that have facility local0, start with "DEVNAME" and have either "error1" or "error0" in their message content, you could use the following filter:

```
if $syslogfacility-text == 'local0' and $msg startswith 'DEVNAME' and ($msg contains
↪'error1' or $msg contains 'error0') then /var/log/somelog
```

Please note that the above must all be on one line! And if you would like to store all messages except those that contain "error1" or "error0", you just need to add a "not":

```
if $syslogfacility-text == 'local0' and $msg startswith 'DEVNAME' and not ($msg␣
↪contains 'error1' or $msg contains 'error0') then /var/log/somelog
```

If you would like to do case-insensitive comparisons, use "contains_i" instead of "contains" and "startswith_i" instead of "startswith". Note that regular expressions are currently NOT supported in expression-based filters. These will be added later when function support is added to the expression engine (the reason is that regular expressions will be a separate loadable module, which requires some more prequisites before it can be implemented).

RainerScript

RainerScript is a scripting language specifically designed and well-suited for processing network events and configuring event processors. It is the prime configuration language used for rsyslog. Please note that RainerScript may not be abreviated as rscript, because that's somebody else's trademark.

Some limited RainerScript support is available since rsyslog 3.12.0 (for expression support). In v5, "if .. then" statements are supported. The first full implementation is available since rsyslog v6.

Data Types

RainerScript is a typeless language. That doesn't imply you don't need to care about types. Of course, expressions like "A" + "B" will not return a valid result, as you can't really add two letters (to concatenate them, use the concatenation operator &). However, all type conversions are automatically done by the script interpreter when there is need to do so.

Expressions

The language supports arbitrary complex expressions. All usual operators are supported. The precedence of operations is as follows (with operations being higher in the list being carried out before those lower in the list, e.g. multiplications are done before additions.

- expressions in parenthesis
- not, unary minus
- *, /, % (modulus, as in C)
- +, -, & (string concatenation)
- ==, !=, <>, <, >, <=, >=, contains (strings!), startswith (strings!)
- and
- or

For example, "not a == b" probably returns not what you intended. The script processor will first evaluate "not a" and then compare the resulting boolean to the value of b. What you probably intended to do is "not (a == b)". And if you just want to test for inequality, we highly suggest to use "!=" or "<>". Both are exactly the same and are provided so that you can pick whichever you like best. So inquality of a and b should be tested as "a <> b". The "not" operator should be reserved to cases where it actually is needed to form a complex boolean expression. In those cases, parenthesis are highly recommended.

Functions

RainerScript supports a currently quite limited set of functions:

getenv(str)

like the OS call, returns the value of the environment variable, if it exists. Returns an empty string if it does not exist.

The following example can be used to build a dynamic filter based on some environment variable:

```
if $msg contains getenv('TRIGGERVAR') then /path/to/errfile
```

strlen(str)

returns the length of the provided string

tolower(str)

converts the provided string into lowercase

cstr(expr)

converts expr to a string value

cnum(expr)

converts expr to a number (integer) Note: if the expression does not contain a numerical value, behaviour is undefined.

wrap(str, wrapper_str)

returns the str wrapped with wrapper_str. Eg.

```
wrap("foo bar", "##")
```

produces

```
"##foo bar##"
```

wrap(str, wrapper_str, escaper_str)

returns the str wrapped with wrapper_str. But additionally, any instances of wrapper_str appearing in str would be replaced by the escaper_str. Eg.

```
wrap("foo'bar", "'", "_")
```

produces

```
"'foo_bar'"
```

replace(str, substr_to_replace, replace_with)

returns new string with all instances of substr_to_replace replaced by replace_with. Eg.

```
replace("foo bar baz", " b", ", B")
```

produces

```
"foo, Bar, Baz".
```

re_match(expr, re)

returns 1, if expr matches re, 0 otherwise. Uses POSIX ERE.

re_extract(expr, re, match, submatch, no-found)

extracts data from a string (property) via a regular expression match. POSIX ERE regular expressions are used. The variable "match" contains the number of the match to use. This permits to pick up more than the first expression match. Submatch is the submatch to match (max 50 supported). The "no-found" parameter specifies which string is to be returned in case when the regular expression is not found. Note that match and submatch start with zero. It currently is not possible to extract more than one submatch with a single call.

field(str, delim, matchnbr)

returns a field-based substring. str is the string to search, delim is the delimiter and matchnbr is the match to search for (the first match starts at 1). This works similar as the field based property-replacer option. Versions prior to 7.3.7 only support a single character as delimiter character. Starting with version 7.3.7, a full string can be used as delimiter. If a single character is being used as delimiter, delim is the numerical ascii value of the field delimiter character (so that non-printable characters can by specified). If a string is used as delmiter, a multi-character string (e.g. "#011") is to be specified.

Note that when a single character is specified as string `field($msg, ",", 3)` a string-based extraction is done, which is more performance intense than the equivalent single-character `field($msg, 44 ,3)` extraction. Eg.

```
set $!usr!field = field($msg, 32, 3);  -- the third field, delimited by space

set $!usr!field = field($msg, "#011", 2); -- the second field, delmited by "#011"
```

exec_template

Sets a variable through the execution of a template. Basically this permits to easily extract some part of a property and use it later as any other variable.

```
template(name="extract" type="string" string="%msg:F:5%")
set $!xyz = exec_template("extract");
```

the variable xyz can now be used to apply some filtering :

```
if $!xyz contains 'abc' then {action()}
```

or to build dynamically a file path :

```
template(name="DynaFile" type="string" string="/var/log/%$!xyz%-data/%timereported%-%
↪$!xyz%.log")
```

Read more about it here : http://www.rsyslog.com/how-to-use-set-variable-and-exec_template

prifilt(constant)

mimics a traditional PRI-based filter (like "*.*" or "mail.info"). The traditional filter string must be given as a **constant string**. Dynamic string evaluation is not permitted (for performance reasons).

dyn_inc(bucket_name_litteral_string, str)

Increments counter identified by `str` in dyn-stats bucket identified by `bucket_name_litteral_string`. Returns 0 when increment is successful, any other return value indicates increment failed.

Counters updated here are reported by **impstats**.

Except for special circumstances (such as memory allocation failing etc), increment may fail due to metric-name cardinality being under-estimated. Bucket is configured to support a maximum cardinality (to prevent abuse) and it rejects increment-operation if it encounters a new(previously unseen) metric-name(`str`) when full.

Read more about it here *Dynamic Stats*

lookup(table_name_litteral_string, key)

Lookup tables are a powerful construct to obtain *class* information based on message content. It works on top of a data-file which maps key (to be looked up) to value (the result of lookup).

The idea is to use a message properties (or derivatives of it) as an index into a table which then returns another value. For example, $fromhost-ip could be used as an index, with the table value representing the type of server or the department or remote office it is located in.

Read more about it here *Lookup Tables*

num2ipv4

Converts an integer into an IPv4-address and returns the address as string. Input is an integer with a value between 0 and 4294967295. The output format is '>decimal<.>decimal<.>decimal<.>decimal<' and '-1' if the integer input is invalid ot the function encounters a problem.

ipv42num

Converts an IPv4-address into an integer and returns the integer. Input is a string; the expected address format may include spaces in the beginning and end, but must not contain any other characters in between (except dots). If the format does include these, the function results in an error and returns -1.

ltrim

Removes any spaces at the start of a given string. Input is a string, output is the same string starting with the first non-space charakter.

rtrim

Removes any spaces at the end of a given string. Input is a string, output is the same string ending with the last non-space charakter.

Control Structures

Control structures in RainerScript are similar in semantics to a lot of other mainstream languages such as C, Java, Javascript, Ruby, Bash etc. So this section assumes the reader is familier with semantics of such structures, and goes about describing RainerScript implementation in usage-example form rather than by formal-definition and detailed semantics documentation.

RainerScript supports following control structures:

if

```
if ($msg contains "important") then {
   if ( $. foo != "" ) then set $.foo = $.bar & $.baz;
   action(type="omfile" file="/var/log/important.log" template="outfmt")
}
```

if/else-if/else

```
if ($msg contains "important") then {
   set $.foo = $.bar & $.baz;
   action(type="omfile" file="/var/log/important.log" template="outfmt")
} else if ($msg startswith "slow-query:") then {
   action(type="omfile" file="/var/log/slow_log.log" template="outfmt")
} else {
   set $.foo = $.quux;
   action(type="omfile" file="/var/log/general.log" template="outfmt")
}
```

foreach

Foreach can iterate both arrays and objects. As opposed to array-iteration (which is ordered), object-iteration accesses key-values in arbitrary order (is unordered).

For the foreach invocation below:

```
foreach ($.i in $.collection) do {
   ...
}
```

Say `$.collection` holds an array `[1, "2", {"a": "b"}, 4]`, value of `$.i` across invocations would be `1`, `"2"`, `{"a" : "b"}` and `4`.

When `$.collection` holds an object `{"a": "b", "c" : [1, 2, 3], "d" : {"foo": "bar"}}`, value of `$.i` across invocations would be `{"key" : "a", "value" : "b"}`, `{"key" : "c", "value" : [1, 2, 3]}` and `{"key" : "d", "value" : {"foo" : "bar"}}` (not

necessarily in the that order). In this case key and value will need to be accessed as `$.i!key` and `$.i!value` respectively.

Here is an example of a nested foreach statement:

```
foreach ($.quux in $!foo) do {
   action(type="omfile" file="./rsyslog.out.log" template="quux")
   foreach ($.corge in $.quux!bar) do {
      reset $.grault = $.corge;
      action(type="omfile" file="./rsyslog.out.log" template="grault")
      if ($.garply != "") then
          set $.garply = $.garply & ", ";
      reset $.garply = $.garply & $.grault!baz;
   }
}
```

Please note that asynchronous-action calls in foreach-statement body should almost always set `action.copyMsg` to `on`. This is because action calls within foreach usually want to work with the variable loop populates(in the above example, `$.quux` and `$.corge`) which causes message-mutation and async-action must see message as it was in a certain invocation of loop-body, so they must make a copy to keep it safe from further modification as iteration continues. For instance, an async-action invocation with linked-list based queue would look like:

```
foreach ($.quux in $!foo) do {
   action(type="omfile" file="./rsyslog.out.log" template="quux" queue.type=
↪"linkedlist" action.copyMsg="on")
}
```

call

Details here: *The rsyslog "call" statement*

continue

a NOP, useful e.g. inside the then part of an if

configuration objects

action()

The *action* object is the primary means of describing actions to be carried out.

global()

This is used to set global configuration parameters. For details, please see the rsyslog global configuration object.

input()

The *input* object is the primary means of describing inputs, which are used to gather messages for rsyslog processing.

module()

The module object is used to load plugins.

parser()

The *parser* object is used to define custom parser objects.

timezone()

The *timezone* object is used to define timezone settings.

Constant Strings

String constants are necessary in many places: comparisons, configuration parameter values and function arguments, to name a few important ones.

In constant strings, special characters are escape by prepending a backslash in front of them – just in the same way this is done in the C programming language or PHP.

If in doubt how to properly escape, use the RainerScript String Escape Online Tool.

Variable (Property) types

All rsyslog properties (see the *properties* page for a list) can be used in RainerScript by prefixing them with "$", for example :

```
set $.x!host = $hostname;
```

In addition, it also supports local variables. Local variables are local to the current message, but are NOT message properties (e.g. the "$!" all JSON property does not contain them).

Only message json (CEE/Lumberjack) properties can be modified by the **set**, **unset** and **reset** statements, not any other message property. Obviously, local variables are also modifieable.

Message JSON property names start with "$!" where the bang character represents the root.

Local variables names start with "$.", where the dot denotes the root.

Both JSON properties as well as local variables may contain an arbitrary deep path before the final element. The bang character is always used as path separator, no matter if it is a message property or a local variable. For example "$!path1!path2!varname" is a three-level deep message property where as the very similar looking "$.path1!path2!varname" specifies a three-level deep local variable. The bang or dot character immediately following the dollar sign is used by rsyslog to separate the different types.

Note that the trailing semicolon is needed to indicate the end of expression. If it is not given, config load will fail with a syntax error message.

Check the following usage examples to understand how these statements behave:

set

sets the value of a local-variable or json property, but the addressed variable already contains a value its behaviour differs as follows:

merges the value if both existing and new value are objects, but merges the new value to *root* rather than with value of the given key. Eg.

```
set $.x!one = "val_1";
# results in $. = { "x": { "one": "val_1" } }
set $.y!two = "val_2";
# results in $. = { "x": { "one": "val_1" }, "y": { "two": "val_2" } }

set $.z!var = $.x;
# results in $. = { "x": { "one": "val_1" }, "y": { "two": "val_2" }, "z": { "var": {
↪"one": "val_1" } } }

set $.z!var = $.y;
# results in $. = { "x": { "one": "val_1" }, "y": { "two": "val_2" }, "z": { "var": {
↪"one": "val_1" } }, "two": "val_2" }
# note that the key *two* is at root level and not  under *$.z!var*.
```

ignores the new value if old value was an object, but new value is a not an object (Eg. string, number etc). Eg:

```
set $.x!one = "val_1";
set $.x = "quux";
# results in $. = { "x": { "one": "val_1" } }
# note that "quux" was ignored
```

resets variable, if old value was not an object.

```
set $.x!val = "val_1";
set $.x!val = "quux";
# results in $. = { "x": { "val": "quux" } }
```

unset

removes the key. Eg:

```
set $.x!val = "val_1";
unset $.x!val;
# results in $. = { "x": { } }
```

reset

force sets the new value regardless of what the variable originally contained or if it was even set. Eg.

```
# to contrast with the set example above, here is how results would look with reset
set $.x!one = "val_1";
set $.y!two = "val_2";
set $.z!var = $.x;
# results in $. = { "x": { "one": "val_1" }, "y": { "two": "val_2" }, "z": { "var": {
↪"one": "val_1" } } }
# 'set' or 'reset' can be used interchangably above(3 lines), they both have the same␣
↪behaviour, as variable doesn't have an existing value
```

```
reset $.z!var = $.y;
# results in $. = { "x": { "one": "val_1" }, "y": { "two": "val_2" }, "z": { "var": {
↪"two": "val_2" } } }
# note how the value of $.z!var was replaced

reset $.x = "quux";
# results in $. = { "x": "quux", "y": { "two": "val_2" }, "z": { "var": { "two": "val_
↪2" } } }
```

Lookup Tables

Lookup tables are a powerful construct to obtain "class" information based on message content (e.g. to build log file names for different server types, departments or remote offices).

General Queue Parameters

Queue parameters can be used together with the following statements:

- *action()*
- ruleset()
- main_queue()

Queues need to be configured in the action or ruleset it should affect. If nothing is configured, default values will be used. Thus, the default ruleset has only the default main queue. Specific Action queues are not set up by default.

To fully understand queue parameters and how they interact, be sure to read the *queues* documentation.

- **queue.filename** name File name to be used for the queue files. Please note that this is actually just the file name. A directory can NOT be specified in this parameter. If the files shall be created in a specific directory, specify queue.spoolDirectory for this. The filename is used to build to complete path for queue files.
- **queue.spoolDirectory** name This is the directory into which queue files will be stored. Note that the directory must exist, it is NOT automatically created by rsyslog. If no spoolDirectory is specified, the work directory is used.
- **queue.size** number This is the maximum size of the queue in number of messages. Note that setting the queue size to very small values (roughly below 100 messages) is not supported and can lead to unpredictable results. For more information on the current status of this restriction see the rsyslog FAQ: "lower bound for queue sizes".
- **queue.dequeuebatchsize** number default 128
- **queue.maxdiskspace** number The maximum size that all queue files together will use on disk. Note that the actual size may be slightly larger than the configured max, as rsyslog never writes partial queue records.
- **queue.highwatermark** number This applies to disk-assisted queues, only. When the queue fills up to this number of messages, the queue begins to spool messages to disk. Please note that this should not happen as part of usual processing, because disk queue mode is very considerably slower than in-memory queue mode. Going to disk should be reserved for cases where an output action destination is offline for some period.
- **queue.lowwatermark** number default 2000
- **queue.fulldelaymark** number Number of messages when the queue should block delayable messages. Messages are NO LONGER PROCESSED until the queue has sufficient space again. If a message is delayable depends on the input. For example, messages received via imtcp are delayable (because TCP can push back), but those received via imudp are not (as UDP does not permit a push back). The intent behind this setting is to leave some space in an almost-full queue for non-delayable messages, which would be lost if the queue runs out

of space. Please note that if you use a DA queue, setting the fulldelaymark BELOW the highwatermark makes the queue never activate disk mode for delayable inputs. So this is probably not what you want.

- **queue.lightdelaymark** number
- **queue.discardmark** number default 9750
- **queue.discardseverity** number *numerical* severity! default 8 (nothing discarded)
- **queue.checkpointinterval** number Disk queues by default do not update housekeeping structures every time the queue writes to disk. This is for performance reasons. In the event of failure, data will still be lost (except when data is mangled via the file structures). However, disk queues can be set to write bookkeeping information on checkpoints (every n records), so that this can be made ultra-reliable, too. If the checkpoint interval is set to one, no data can be lost, but the queue is exceptionally slow.
- **queue.syncqueuefiles** on/off (default "off")

  Disk-based queues can be made very reliable by issuing a (f)sync after each write operation. This happens when you set the parameter to "on". Activating this option has a performance penalty, so it should not be turned on without a good reason. Note that the penalty also depends on *queue.checkpointInterval* frequency.
- **queue.samplinginterval** number

  This option allows queues to be populated by events produced at a specific interval. It provides a way to sample data each N events, instead of processing all, in order to reduce resources usage (disk, bandwidth...) This feature is available for version 8.23 and above.
- **queue.type** [FixedArray/LinkedList/**Direct**/Disk]
- **queue.workerthreads** number number of worker threads, default 1, recommended 1
- **queue.timeoutshutdown** number number is timeout in ms (1000ms is 1sec!), default 0 (indefinite)
- **queue.timeoutactioncompletion** number number is timeout in ms (1000ms is 1sec!), default 1000, 0 means immediate!
- **queue.timeoutenqueue** number number is timeout in ms (1000ms is 1sec!), default 2000, 0 means discard immediate.

  This timeout value is used when the queue is full. If rsyslog cannot enqueue a message within the timeout period, the message is discarded. Note that this is setting of last resort (assuming defaults are used for the queue settings or proper parameters are set): all delayable inputs (like imtcp or imfile) have already been pushed back at this stage. Also, discarding of lower priority messages (if configured) has already happened. So we run into one of these situations if we do not timeout quickly enough:

  - if using imuxsock and no systemd journal is involved, the system would become unresponsive and most probably a hard reset would be required.
  - if using imuxsock with imjournal forwarding is active, messages are lost because the journal discards them (more agressive than rsyslog does)
  - if using imjournal, the journal will buffer messages. If journal runs out of configured space, messages will be discarded. So in this mode discarding is moved to a bit later place.
  - other non-delayable sources like imudp will also loose messages

  So this setting is provided in order to guard against problematic situations, which always will result either in message loss or system hang. For action queues, one may debate if it would be better to overflow rapidly to the main queue. If so desired, this is easy to acomplish by setting a very large timeout value. The same, of course, is true for the main queue, but you have been warned if you do so!

  In some other words, you can consider this scenario, using default values. With all progress blocked (unable to deliver a message):

  - all delayable inputs (tcp, relp, imfile, imjournal, etc) will block indefinantly (assuming queue.lightdelaymark and queue.fulldelaymark are set sensible, which they are by default).
  - imudp will be loosing messages because the OS will be dropping them
  - messages arriving via UDP or imuxsock that do make it to rsyslog, and that are a severity high enough to not be filtered by discardseverity, will block for 2 seconds trying to put the message in the queue (in the hope that something happens to make space in the queue) and then be dropped to avoid blocking the machine permanently.

    Then the next message to be processed will also be tried for 2 seconds, etc.
  - If this is going into an action queue, the log message will remain in the main queue during these 2 seconds, and additional logs that arrive will accumulate behind this in the main queue.

- **queue.timeoutworkerthreadshutdown** number number is timeout in ms (1000ms is 1sec!), default 60000 (1 minute)
- **queue.workerthreadminimummessages** number default 100
- **queue.maxfilesize** size_nbr default 1m
- **queue.saveonshutdown** on/**off**
- **queue.dequeueslowdown** number number is timeout in microseconds (1000000us is 1sec!), default 0 (no delay). Simple rate-limiting!
- **queue.dequeuetimebegin** number
- **queue.dequeuetimeend** number
- **queue.samplinginterval** number Sampling interval for action queue. This parameter specifies how many line of logs will be dropped before one enqueued. default 0.

Sample:

The following is a sample of a TCP forwarding action with its own queue.

```
action(type="omfwd" target="192.168.2.11" port="10514" protocol="tcp"
       queue.filename="forwarding" queue.size="1000000" queue.type="LinkedList"
      )
```


The rsyslog "call" statement

The rsyslog "call" statement is used to tie rulesets together. It is modelled after the usual programming language "call" statement. Think of a ruleset as a subroutine (what it really is!) and you get the picture.

The "call" statement can be used to call into any type of rulesets. If a rule set has a queue assigned, the message will be posted to that queue and processed asynchronously. Otherwise, the ruleset will be executed synchronously and control returns to right after the call when the rule set has finished execution.

Note that there is an important difference between asynchronous and synchronous execution in regard to the "stop" statement. It will not affect processing of the original message when run asynchronously.

The "call" statement replaces the deprecated omruleset module. It offers all capabilities omruleset has, but works in a much more efficient way. Note that omruleset was a hack that made calling rulesets possible within the constraints of the pre-v7 engine. "call" is the clean solution for the new engine. Especially for rulesets without associated queues (synchronous operation), it has zero overhead (really!). omruleset always needs to duplicate messages, which usually means at least ~250 bytes of memory writes, some allocs and frees - and even more performance-intense operations.

syntax

```
call rulesetname
```

Where "rulesetname" is the name of a ruleset that is defined elsewhere inside the configration. If the call is synchronous or asynchronous depends on the ruleset parameters. This cannot be overriden by the "call" statement.

related links

- Blog posting announcing "call" statement (with sample)

The rsyslog "call_indirect" statement

The rsyslog "call_indirect" statement is equivalent to *"call" statement* except that the name of the to be called ruleset is not constant but an expression and so can be computed at runtime.

If the ruleset name cannot be found when call_indirect is used, an error message as emitted and the call_indirect statement is ignored. Execution continues with the next statement.

syntax

```
call_indirect expression;
```

Where "expression" is any valid expression. See *expressions* for more information. Note that the trailing semicolon is needed to indicate the end of expression. If it is not given, config load will fail with a syntax error message.

examples

The potentially most useful use-case for "call_indirect" is calling a ruleset based on a message variable. Let us assume that you have named your rulesets according to syslog tags expected. Then you can use

```
call_indirect $syslogtag;
```

To call these rulesets. Note, however, that this may be misused by a malicious attacker, who injects invalid syslog tags. This could especially be used to redirect message flow to known standard rulesets. To somewhat mitigate against this, the ruleset name can be slightly mangled by creating a **unique** prefix (do **not** use the one from this sample). Let us assume the prefix "changeme-" is used, then all your rulesets should start with that string. Then, the following call can be used:

```
call_indirect "changeme-" & $syslogtag;
```

While it is possible to call a ruleset via a constant name:

```
call_indirect "my_ruleset";
```

It is advised to use the "call" statement for this, as it offers superior performance in this case.

additional information

We need to have two different statements, "call" and "call_indirect" because "call" already existed at the time "call_indirect" was added. We could not extend "call" to support expressions, as that would have broken existing configs. In that case `call ruleset` would have become invalid and `call "ruleset"` would have to be used instead. Thus we decided to add the additional "call_indirect" statement for this use case.

global() configuration object

The global configuration object permits to set global parameters. Note that each parameter can only be set once and cannot be re-set thereafter. If a parameter is set multiple times, the behaviour is unpredictable.

The following parameters can be set:

- **action.reportSuspension** - binary, default "on", v7.5.8+

  If enabled ("on") action will log message under *syslog.** when an action suspends or resumes itself. This usually happens when there are problems connecting to backend systems. If disabled ("off"), these messages are not generated. These messages can be useful in detecting problems with backend systems. Most importantly, frequent suspension and resumption points to a problem area.

- **action.reportSuspensionContinuation** - binary, default "off", v7.6.1+, v8.2.0+

  If enabled ("on") the action will not only report the first suspension but each time the suspension is prolonged. Otherwise, the follow-up messages are not logged. If this setting is set to "on", action.reportSuspension is also automaticaly turned "on".

- **workDirectory**
- **dropMsgsWithMaliciousDNSPtrRecords**
- **localHostname**
- **preserveFQDN**
- **defaultNetstreamDriverCAFile**

  For TLS syslog, the CA certificate that can verify the machine keys and certs (see below)

- **defaultNetstreamDriverKeyFile**

  Machine private key

- **defaultNetstreamDriverCertFile**

  Machine public key (certificate)

- **debug.gnutls** (0-10; default:0)

  Any other parameter than 0 enables the debug messages of GnuTLS. the amount of messages given depends on the height of the parameter, 0 being nothing and 10 being very much. Caution! higher parameters may give out way more information than needed. We advise you to first use small parameters to prevent that from happening. **This parameter only has an effect if general debugging is enabled.**

- **processInternalMessages** binary (on/off)

  This tells rsyslog if it shall process internal messages itself. The default mode of operations ("off") makes rsyslog send messages to the system log sink (and if it is the only instance, receive them back from there). This also works with systemd journal and will make rsyslog messages show up in the systemd status control information.

If this (instance) of rsyslog is not the main instance and there is another main logging system, rsyslog internal messages will be inserted into the main instance's syslog stream. In this case, setting to ("on") will let you receive the internal messages in the instance they originate from.

Note that earlier versions of rsyslog worked the opposite way. More information about the change can be found in rsyslog-error-reporting-improved.

- **stdlog.channelspec**

  Permits to set the liblogging-stdlog channel specifier string. This in turn permits to send rsyslog log messages to a destination different from the system default. Note that this parameter has only effect if *processInternalMessages* is set to "off". Otherwise it is silently ignored.

- **defaultNetstreamDriver**

  Set it to "gtls" to enable TLS for TLS syslog

- **maxMessageSize**

  The maximum message size rsyslog can process. Default is 8K. Anything above the maximum size will be truncated.

- **janitor.interval** [minutes], available since 8.3.3

  Sets the interval at which the *janitor process* runs.

- **debug.onShutdown** available in 7.5.8+

  If enabled ("on"), rsyslog will log debug messages when a system shutdown is requested. This can be used to track issues that happen only during shutdown. During normal operations, system performance is NOT affected. Note that for this option to be useful, the debug.logFile parameter must also be set (or the respective environment variable).

- **debug.logFile** available in 7.5.8+

  This is used to specify the debug log file name. It is used for all debug output. Please note that the RSYSLOG_DEBUGLOG environment variable always **overrides** the value of debug.logFile.

- **net.ipprotocol** available in 8.6.0+

  This permits to instruct rsyslog to use IPv4 or IPv6 only. Possible values are "unspecified", in which case both protocols are used, "ipv4-only", and "ipv6-only", which restrict usage to the specified protocol. The default is "unspecified".

  Note: this replaces the former *-4* and *-6* rsyslogd command line options.

- **net.aclAddHostnameOnFail** available in 8.6.0+

  If "on", during ACL processing, hostnames are resolved to IP addresses for performance reasons. If DNS fails during that process, the hostname is added as wildcard text, which results in proper, but somewhat slower operation once DNS is up again.

  The default is "off".

- **net.aclResolveHostname** available in 8.6.0+

  If "off", do not resolve hostnames to IP addresses during ACL processing.

  The default is "on".

- **net.enableDNS** [on/off] available in 8.6.0+

  Default: on

  Can be used to turn DNS name resolution on or off.

- **net.permitACLWarning** [on/off] available in 8.6.0+

  Default: on

  If "off", suppress warnings issued when messages are received from non-authorized machines (those, that are in no AllowedSender list).

- **parser.parseHostnameAndTag** [on/off] available in 8.6.0+

  Default: on

  This controls wheter the parsers try to parse HOSTNAME and TAG fields from messages. The default is "on", in which case parsing occurs. If set to "off", the fields are not parsed. Note that this usually is **not** what you want to have.

  It is highly suggested to change this setting to "off" only if you know exactly why you are doing this.

- **parser.permitSlashInHostname** [on/off] available in 8.25.0+

  Default: off

  This controls whether slashes in the "programname" property are permitted or not. This property bases on a BSD concept, and by BSD syslogd sources, slashes are NOT permitted inside the program name. However, some Linux tools (including most importantly the journal) store slashes as part of the program name inside the syslogtag. In those cases, the programname is truncated at the first slash. If this setting is changed to "on", slashes are permitted and will not terminate programname parsing.

- **senders.keepTrack** [on/off] available 8.17.0+

  Default: off

  If turned on, rsyslog keeps track of known senders and also reports statistical data for them via the impstats mechanism.

  A list of active senders is kept. When a new sender is detected, an informational message is emitted. Senders are purged from the list only after a timeout (see *senders.timoutAfter* parameter). Note that we do not intentionally remove a sender when a connection is closed. The whole point of this sender-tracking is to have the ability to provide longer-duration data. As such, we would not like to drop information just because the sender has disconnected for a short period of time (e.g. for a reboot).

  Senders are tracked by their hostname (taken at connection establishment).

  Note: currently only imptcp and imtcp support sender tracking.

- **senders.timeoutAfter** [seconds] available 8.17.0+

  Default: 12 hours (12*60*60 seconds)

  Specifies after which period a sender is considered to "have gone away". For each sender, rsyslog keeps track of the time it least received messages from it. When it has not received a message during that interval, rsyslog considers the sender to be no longer present. It will then a) emit a warning message (if configured) and b) purge it from the active senders list. As such, the sender will no longer be reported in impstats data once it has timed out.

- **senders.reportGoneAway** [on/off] available 8.17.0+

  Default: off

  Emit a warning message when now data has been received from a sender within the *senders.timeoutAfter* interval.

- **senders.reportNew** [on/off] available 8.17.0+

  Default: off

If sender tracking is active, report a sender that is not yet inside the cache. Note that this means that senders which have been timed out due to prolonged inactivity are also reported once they connect again.

- **debug.unloadModules** [on/off] available 8.17.0+

  Default: on

  This is primarily a debug setting. If set to "off", rsyslog will never unload any modules (including plugins). This usually causes no operational problems, but may in extreme cases. The core benefit of this setting is that it makes valgrind stack traces readable. In previous versions, the same functionality was only available via a special build option.

- **debug.files** [ARRAY of filenames] available 8.29.0+

  Default: none

  This can be used to configure rsyslog to only show debug-output generated in certain files. If the option is set, but no filename is given, the debug-output will behave as if the option is turned off.

  Do note however that due to the way the configuration works, this might not effect the first few debug-outputs, while rsyslog is reading in the configuration. For optimal results we recommend to put this parameter at the very start of your configuration to minmize unwanted output.

  See debug.whitelist for more information.

- **debug.whitelist** [on/off] available 8.29.0+

  Default: on

  This parameter is an assisting parameter of debug.files. If debug.files is used in the configuration, debug.whitelist is a switch for the files named to be either white- or blacklisted from displaying debug-output. If it is set to on, the listed files will generate debug-output, but no other files will. The reverse principle applies if the parameter is set to off.

  See debug.files for more information.

- **environment** [ARRAY of environment variable=value strings] available 8.23.0+

  Default: none

  This permits to set environment variables via rsyslog.conf. The prime motivation for having this is that for many libraries, defaults can be set via environment variables, **but** setting them via operating system service startup files is cumbersome and different on different platforms. So the *environment* parameter provides a handy way to set those variables.

  A common example is to set the *http_proxy* variable, e.g. for use with KSI signing or ElasticSearch. This can be done as follows:

```
global(environment="http_proxy=http://myproxy.example.net")
```

  Note that an environment variable set this way must contain an equal sign, and the variable name must not be longer than 127 characters.

  It is possible to set multiple environment variables in a single global statement. This is done in regular array syntax as follows:

```
global(environment=["http_proxy=http://myproxy.example.net",
                    "another_one=this string is=ok!"
      )
```

  As usual, whitespace is irrelevant in regard to parameter placing. So the above sample could also have been written on a single line.

- **internalmsg.ratelimit.interval** [positive integer] available 8.29.0+

  Default: 5

  Specifies the interval in seconds onto which rate-limiting is to be applied to internal messgaes generated by rsyslog(i.e. error messages). If more than internalmsg.ratelimit.burst messages are read during that interval, further messages up to the end of the interval are discarded.

- **internalmsg.ratelimit.burst** [positive integer] available 8.29.0+

  Default: 500

  Specifies the maximum number of internal messages that can be emitted within the ratelimit.interval interval. For futher information, see description there.

  Caution: Environment variables are set immediately when the corresponding statement is encountered. Likewise, modules are loaded when the module load statement is encountered. This may create **sequence dependencies** inside rsyslog.conf. To avoid this, it is highly suggested that environment variables are set **right at the top of rsyslog.conf**. Also, rsyslog-related environment variables may not apply even when set right at the top. It is safest to still set them in operating system start files. Note that rsyslog environment variables are usually intended only for developers so there should hardly be a need to set them for a regular user. Also, many settings (e.g. debug) are also available as configuration objects.

Actions

The Action object describe what is to be done with a message. They are implemented via *output modules*.

The action object has different parameters:

- those that apply to all actions and are action specific. These are documented below.
- parameters for the action queue. While they also apply to all parameters, they are queue-specific, not action-specific (they are the same that are used in rulesets, for example). The are documented separately under *queue parameters*.
- action-specific parameters. These are specific to a certain type of actions. They are documented by the *output modules* in question.

General Action Parameters

- **name** word

  This names the action. The name is used for statistics gathering and documentation. If no name is given, one is dynamically generated based on the occurence of this action inside the rsyslog configuration. Actions are sequentially numbered from 1 to n.

- **type** string Mandatory parameter for every action. The name of the module that should be used.

- **action.writeAllMarkMessages** *on*/off This setting tells if mark messages are always written ("on", the default) or only if the action was not recently executed ("off"). By default, recently means within the past 20 minutes. If this setting is "on", mark messages are always sent to actions, no matter how recently they have been executed. In this mode, mark messages can be used as a kind of heartbeat. This mode also enables faster processing inside the rule engine. So it should be set to "off" only when there is a good reason to do so.

- **action.execOnlyEveryNthTime** integer If configured, the next action will only be executed every n-th time. For example, if configured to 3, the first two messages that go into the action will be dropped, the 3rd will actually cause the action to execute, the 4th and 5th will be dropped, the 6th executed under the action, ... and so on.

- **action.execOnlyEveryNthTimeout** integer Has a meaning only if Action.ExecOnlyEveryNthTime is also configured for the same action. If so, the timeout setting specifies after which period the counting of "previous actions" expires and a new action count is begun. Specify 0 (the default) to disable timeouts. Why is this option needed? Consider this case: a message comes in at, eg., 10am. That's count 1. Then, nothing happens for the next 10 hours. At 8pm, the next one occurs. That's count 2. Another 5 hours later, the next message occurs, bringing the total count to 3. Thus, this message now triggers the rule. The question is if this is desired behavior? Or should the rule only be triggered if the messages occur within an e.g. 20 minute window? If the later is the case, you need a Action.ExecOnlyEveryNthTimeTimeout="1200" This directive will timeout previous messages seen if they are older than 20 minutes. In the example above, the count would now be always 1 and consequently no rule would ever be triggered.
- **action.execOnlyOnceEveryInterval** integer Execute action only if the last execute is at last seconds in the past (more info in ommail, but may be used with any action)
- **action.execOnlyWhenPreviousIsSuspended** on/off This directive allows to specify if actions should always be executed ("off," the default) or only if the previous action is suspended ("on"). This directive works hand-in-hand with the multiple actions per selector feature. It can be used, for example, to create rules that automatically switch destination servers or databases to a (set of) backup(s), if the primary server fails. Note that this feature depends on proper implementation of the suspend feature in the output module. All built-in output modules properly support it (most importantly the database write and the syslog message forwarder). Note, however, that a failed action may not immediately be detected. For more information, see the rsyslog execOnlyWhenPreviousIsSpuspended preciseness FAQ article.
- **action.repeatedmsgcontainsoriginalmsg** on/off "last message repeated n times" messages, if generated, have a different format that contains the message that is being repeated. Note that only the first "n" characters are included, with n to be at least 80 characters, most probably more (this may change from version to version, thus no specific limit is given). The bottom line is that n is large enough to get a good idea which message was repeated but it is not necessarily large enough for the whole message. (Introduced with 4.1.5).
- **action.resumeRetryCount** integer [default 0, -1 means eternal]
- **action.resumeInterval** integer Sets the ActionResumeInterval for the action. The interval provided is always in seconds. Thus, multiply by 60 if you need minutes and 3,600 if you need hours (not recommended). When an action is suspended (e.g. destination can not be connected), the action is resumed for the configured interval. Thereafter, it is retried. If multiple retries fail, the interval is automatically extended. This is to prevent excessive resource use for retries. After each 10 retries, the interval is extended by itself. To be precise, the actual interval is (numRetries / 10 + 1) * Action.ResumeInterval. so after the 10th try, it by default is 60 and after the 100th try it is 330.
- **action.reportSuspension** on/off Configures rsyslog to report suspension and reactivation of the action. This is useful to note which actions have problems (e.g. connecting to a remote system) and when. The default for this setting is the equally-named global parameter.
- **action.reportSuspensionContinuation** on/off Configures rsyslog to report continuation of action suspension. This emits new messages whenever an action is to be retried, but continues to fail. If set to "on", *action.reportSuspension* is also automatically set to "on". The default for this setting is the equally-named global parameter.
- **action.copyMsg** on/*off* Configures action to *copy* the message if *on*. Defaults to *off* (which is how actions have worked traditionally), which causes queue to refer to the original message object, with reference-counting. (Introduced with 8.10.0).

Useful Links

- Rainer's blog posting on the performance of main and action queue worker threads

Legacy Format

Be warned that legacy action format is hard to get right. It is recommended to use RainerScript-Style action format whenever possible! A key problem with legacy format is that a single action is defined via multiple configurations lines, which may be spread all across rsyslog.conf. Even the definition of multiple actions may be intermixed (often not intentional!). If legacy actions format needs to be used (e.g. some modules may not yet implement the RainerScript format), it is strongly recommended to place all configuration statements pertaining to a single action closely together.

Please also note that legacy action parameters **do not** affect RainerScript action objects. So if you define for example:

```
$actionResumeRetryCount 10
action(type="omfwd" target="server1.example.net")
@@server2.example.net
```

server1's "action.resumeRetryCount" parameter is **not** set, instead server2's is!

A goal of the new RainerScript action format was to avoid confusion which parameters are actually used. As such, it would be counter-productive to honor legacy action parameters inside a RainerScript definition. As result, both types of action definitions are strictly (and nicely) separated from each other. The bottom line is that if RainerScript actions are used, one does not need to care about which legacy action parameters may (still...) be in effect.

Note that not all modules necessarily support legacy action format. Especially newer modules are recommended to NOT support it.

Legacy Description

Templates can be used with many actions. If used, the specified template is used to generate the message content (instead of the default template). To specify a template, write a semicolon after the action value immediately followed by the template name. Beware: templates MUST be defined BEFORE they are used. It is OK to define some templates, then use them in selector lines, define more templates and use use them in the following selector lines. But it is NOT permitted to use a template in a selector line that is above its definition. If you do this, the action will be ignored.

You can have multiple actions for a single selector (or more precisely a single filter of such a selector line). Each action must be on its own line and the line must start with an ampersand ('&') character and have no filters. An example would be

```
*.=crit :omusrmsg:rger
& root
& /var/log/critmsgs
```

These three lines send critical messages to the user rger and root and also store them in /var/log/critmsgs. **Using multiple actions per selector is** convenient and also **offers a performance benefit**. As the filter needs to be evaluated only once, there is less computation required to process the directive compared to the otherwise-equal config directives below:

```
*.=crit :omusrmsg:rger
*.=crit root
*.=crit /var/log/critmsgs
```

Regular File

Typically messages are logged to real files. The file usually is specified by full pathname, beginning with a slash "/". Starting with version 4.6.2 and 5.4.1 (previous v5 version do NOT support this) relative file names can also be specified. To do so, these must begin with a dot. For example, use "./file-in-current-dir.log" to specify a file in the

current directory. Please note that rsyslogd usually changes its working directory to the root, so relative file names must be tested with care (they were introduced primarily as a debugging vehicle, but may have useful other applications as well). You may prefix each entry with the minus "-'' sign to omit syncing the file after every logging. Note that you might lose information if the system crashes right behind a write attempt. Nevertheless this might give you back some performance, especially if you run programs that use logging in a very verbose manner.

If your system is connected to a reliable UPS and you receive lots of log data (e.g. firewall logs), it might be a very good idea to turn of syncing by specifying the "-" in front of the file name.

The filename can be either static(always the same) or **dynamic** (different based on message received). The later is useful if you would automatically split messages into different files based on some message criteria. For example, dynamic file name selectors allow you to split messages into different files based on the host that sent them. With dynamic file names, everything is automatic and you do not need any filters.

It works via the template system. First, you define a template for the file name. An example can be seen above in the description of template. We will use the "DynFile" template defined there. Dynamic filenames are indicated by specifying a questions mark "?" instead of a slash, followed by the template name. Thus, the selector line for our dynamic file name would look as follows:

```
*.* ?DynFile
```

That's all you need to do. Rsyslog will now automatically generate file names for you and store the right messages into the right files. Please note that the minus sign also works with dynamic file name selectors. Thus, to avoid syncing, you may use

```
*.* -?DynFile
```

And of course you can use templates to specify the output format:

```
*.* ?DynFile;MyTemplate
```

A word of caution: rsyslog creates files as needed. So if a new host is using your syslog server, rsyslog will automatically create a new file for it.

Creating directories is also supported. For example you can use the hostname as directory and the program name as file name:

```
$template DynFile,"/var/log/%HOSTNAME%/%programname%.log"
```

Named Pipes

This version of rsyslogd(8) has support for logging output to named pipes (fifos). A fifo or named pipe can be used as a destination for log messages by prepending a pipe symbol ("|'') to the name of the file. This is handy for debugging. Note that the fifo must be created with the mkfifo(1) command before rsyslogd(8) is started.

Terminal and Console

If the file you specified is a tty, special tty-handling is done, same with /dev/console.

Remote Machine

Rsyslogd provides full remote logging, i.e. is able to send messages to a remote host running rsyslogd(8) and to receive messages from remote hosts. Using this feature you're able to control all syslog messages on one host, if all other machines will log remotely to that. This tears down administration needs.

To forward messages to another host, prepend the hostname with the at sign ("@"). A single at sign means that messages will be forwarded via UDP protocol (the standard for syslog). If you prepend two at signs ("@@"), the

messages will be transmitted via TCP. Please note that plain TCP based syslog is not officially standardized, but most major syslogds support it (e.g. syslog-ng or WinSyslog). The forwarding action indicator (at-sign) can be followed by one or more options. If they are given, they must be immediately (without a space) following the final at sign and be enclosed in parenthesis. The individual options must be separated by commas. The following options are right now defined:

z<number>

Enable zlib-compression for the message. The <number> is the compression level. It can be 1 (lowest gain, lowest CPU overhead) to 9 (maximum compression, highest CPU overhead). The level can also be 0, which means "no compression". If given, the "z" option is ignored. So this does not make an awful lot of sense. There is hardly a difference between level 1 and 9 for typical syslog messages. You can expect a compression gain between 0% and 30% for typical messages. Very chatty messages may compress up to 50%, but this is seldom seen with typically traffic. Please note that rsyslogd checks the compression gain. Messages with 60 bytes or less will never be compressed. This is because compression gain is pretty unlikely and we prefer to save CPU cycles. Messages over that size are always compressed. However, it is checked if there is a gain in compression and only if there is, the compressed message is transmitted. Otherwise, the uncompressed messages is transmitted. This saves the receiver CPU cycles for decompression. It also prevents small message to actually become larger in compressed form.

Please note that when a TCP transport is used, compression will also turn on syslog-transport-tls framing. See the "o" option for important information on the implications.

Compressed messages are automatically detected and decompressed by the receiver. There is nothing that needs to be configured on the receiver side.

o

This option is experimental. Use at your own risk and only if you know why you need it! If in doubt, do NOT turn it on.

This option is only valid for plain TCP based transports. It selects a different framing based on IETF internet draft syslog-transport-tls-06. This framing offers some benefits over traditional LF-based framing. However, the standardization effort is not yet complete. There may be changes in upcoming versions of this standard. Rsyslog will be kept in line with the standard. There is some chance that upcoming changes will be incompatible to the current specification. In this case, all systems using -transport-tls framing must be upgraded. There will be no effort made to retain compatibility between different versions of rsyslog. The primary reason for that is that it seems technically impossible to provide compatibility between some of those changes. So you should take this note very serious. It is not something we do not *like* to do (and may change our mind if enough people beg...), it is something we most probably *can not* do for technical reasons (aka: you can beg as much as you like, it won't change anything...).

The most important implication is that compressed syslog messages via TCP must be considered with care. Unfortunately, it is technically impossible to transfer compressed records over traditional syslog plain tcp transports, so you are left with two evil choices...

The hostname may be followed by a colon and the destination port.

The following is an example selector line with forwarding:

. @@(o,z9)192.168.0.1:1470

In this example, messages are forwarded via plain TCP with experimental framing and maximum compression to the host 192.168.0.1 at port 1470.

. @192.168.0.1

In the example above, messages are forwarded via UDP to the machine 192.168.0.1, the destination port defaults to 514. Messages will not be compressed.

Note that IPv6 addresses contain colons. So if an IPv6 address is specified in the hostname part, rsyslogd could not detect where the IP address ends and where the port starts. There is a syntax extension to support this: put squary

brackets around the address (e.g. "[2001::1]"). Square brackets also work with real host names and IPv4 addresses, too.

A valid sample to send messages to the IPv6 host 2001::1 at port 515 is as follows:

. @[2001::1]:515

This works with TCP, too.

Note to sysklogd users: sysklogd does **not** support RFC 3164 format, which is the default forwarding template in rsyslog. As such, you will experience duplicate hostnames if rsyslog is the sender and sysklogd is the receiver. The fix is simple: you need to use a different template. Use that one:

$template sysklogd,"<%PRI%>%TIMESTAMP% %syslogtag%%msg%\n" *.* @192.168.0.1;sysklogd

List of Users

Usually critical messages are also directed to "root" on that machine. You can specify a list of users that shall get the message by simply writing ":omusrmsg: followed by the login name. For example, the send messages to root, use ":omusrmsg:root". You may specify more than one user by separating them with commas (",'). Do not repeat the ":omusrmsg:" prefix in this case. For example, to send data to users root and rger, use ":omusrmsg:root,rger" (do not use ":omusrmsg:root,:omusrmsg:rger", this is invalid). If they're logged in they get the message.

Everyone logged on

Emergency messages often go to all users currently online to notify them that something strange is happening with the system. To specify this wall(1)-feature use an asterisk as the user message destination(":omusrmsg:*').

Call Plugin

This is a generic way to call an output plugin. The plugin must support this functionality. Actual parameters depend on the module, so see the module's doc on what to supply. The general syntax is as follows:

:modname:params;template

Currently, the ommysql database output module supports this syntax (in addtion to the ">" syntax it traditionally supported). For ommysql, the module name is "ommysql" and the params are the traditional ones. The ;template part is not module specific, it is generic rsyslog functionality available to all modules.

As an example, the ommysql module may be called as follows:

:ommysql:dbhost,dbname,dbuser,dbpassword;dbtemplate

For details, please see the "Database Table" section of this documentation.

Note: as of this writing, the ":modname:" part is hardcoded into the module. So the name to use is not necessarily the name the module's plugin file is called.

Database Table

This allows logging of the message to a database table. Currently, only MySQL databases are supported. However, other database drivers will most probably be developed as plugins. By default, a MonitorWare-compatible schema is required for this to work. You can create that schema with the createDB.SQL file that came with the rsyslog package. You can also use any other schema of your liking - you just need to define a proper template and assign this template to the action. The database writer is called by specifying a greater-then sign (">") in front of the

database connect information. Immediately after that sign the database host name must be given, a comma, the database name, another comma, the database user, a comma and then the user's password. If a specific template is to be used, a semicolon followed by the template name can follow the connect information. This is as follows: >dbhost,dbname,dbuser,dbpassword;dbtemplate

Important: to use the database functionality, the MySQL output module must be loaded in the config file BEFORE the first database table action is used. This is done by placing the

```
$ModLoad ommysql
```

directive some place above the first use of the database write (we recommend doing at the the beginning of the config file).

Discard

If the discard action is carried out, the received message is immediately discarded. No further processing of it occurs. Discard has primarily been added to filter out messages before carrying on any further processing. For obvious reasons, the results of "discard" are depending on where in the configuration file it is being used. Please note that once a message has been discarded there is no way to retrieve it in later configuration file lines.

Discard can be highly effective if you want to filter out some annoying messages that otherwise would fill your log files. To do that, place the discard actions early in your log files. This often plays well with property-based filters, giving you great freedom in specifying what you do not want.

Discard is just the single tilde character with no further parameters:

~

For example,

. ~

discards everything (ok, you can achive the same by not running rsyslogd at all...).

Output Channel

Binds an output channel definition (see there for details) to this action. Output channel actions must start with a $-sign, e.g. if you would like to bind your output channel definition "mychannel" to the action, use "$mychannel". Output channels support template definitions like all all other actions.

Shell Execute

NOTE: This action is only supported for backwards compatibility. For new configs, use *omprog* instead. It provides a more solid and secure solution with higher performance.

This executes a program in a subshell. The program is passed the template-generated message as the only command line parameter. Rsyslog waits until the program terminates and only then continues to run.

^program-to-execute;template

The program-to-execute can be any valid executable. It receives the template string as a single parameter (argv[1]).

WARNING: The Shell Execute action was added to serve an urgent need. While it is considered reasonable save when used with some thinking, its implications must be considered. The current implementation uses a system() call to execute the command. This is not the best way to do it (and will hopefully changed in further releases). Also, proper escaping of special characters is done to prevent command injection. However, attackers always find smart ways to circumvent escaping, so we can not say if the escaping applied will really safe you from all hassles. Lastly, rsyslog

will wait until the shell command terminates. Thus, a program error in it (e.g. an infinite loop) can actually disable rsyslog. Even without that, during the programs run-time no messages are processed by rsyslog. As the IP stacks buffers are quickly overflowed, this bears an increased risk of message loss. You must be aware of these implications. Even though they are severe, there are several cases where the "shell execute" action is very useful. This is the reason why we have included it in its current form. To mitigate its risks, always a) test your program thoroughly, b) make sure its runtime is as short as possible (if it requires a longer run-time, you might want to spawn your own sub-shell asynchronously), c) apply proper firewalling so that only known senders can send syslog messages to rsyslog. Point c) is especially important: if rsyslog is accepting message from any hosts, chances are much higher that an attacker might try to exploit the "shell execute" action.

Template Name

Every ACTION can be followed by a template name. If so, that template is used for message formatting. If no name is given, a hard-coded default template is used for the action. There can only be one template name for each given action. The default template is specific to each action. For a description of what a template is and what you can do with it, see the *template* documentation.

Input

The `input` object, as its name suggests, describes message input sources. Without input, no processing happens at all, because no messages enter the rsyslog system. Inputs are implemented via *input modules*.

The input object has different parameters:

- those that apply to all input and are generally available for all inputs. These are documented below.
- input-specific parameters. These are specific to a certain type of input. They are documented by the *input module* in question.

General Input Parameters

type <type-string>
Mandatory

The `<type-string>` is a string identifying the input module as given it each module's documentation. For example, the *UDP syslog input* is named "imudp".

Parser

The `parser` object, as its name suggests, describes message parsers. Message parsers have a standard parser name, which can be used by simply loading the parser module. Only when specific parameters need to be set the parser object is needed.

In that case, it is used to define a new parser name (aka "parser definition") which configures this name to use the parser module with set parameters. This is important as the ruleset() object does not support to set parser parameters. Instead, if parameters are needed, a proper parser name must be defined using the parser() object. A parser name defined via the parser() object can be used whereever a parser name can occur.

Note that not all message parser modules are supported in the parser() object. The reason is that many do not have any user-selectable parameters and as such, there is no point in issuing a parser() object for them.

The parser object has different parameters:

- those that apply to all parser and are generally available for all of them. These are documented below.
- parser-specific parameters. These are specific to a certain parser module. They are documented by the *parser module* in question.

General Parser Parameters

name <name-string>
Mandatory

This names the parser. Names starting with "rsyslog." are reserved for rsyslog use and must not be used. It is suggested to replace "rsyslog." with "custom." and keep the rest of the name descriptive. However, this is not enforced and just good practice.

type <type-string>
Mandatory

The `<type-string>` is a string identifying the parser module as given it each module's documentation. Do not mistake the parser module name with its default parser name. For example, the *Cisco IOS message parser module* parser module name is "pmciscoios", whereas it's default parser name is "rsyslog.pmciscoios".

Samples

The following example creates a custom parser definition and uses it within a ruleset:

```
module(load="pmciscoios")
parser(name="custom.pmciscoios.with_origin" type="pmciscoios")

ruleset(name="myRuleset" parser="custom.pmciscoios.with_origin") {
   ... do something here ...
}
```

The following example uses multiple parsers within a ruleset without a parser object (the order is important):

```
module(load="pmaixforwardedfrom")
module(load="pmlastmsg")

ruleset(name="myRuleset" parser=["rsyslog.lastline","rsyslog.aixforwardedfrom",
↪"rsyslog.rfc5424","rsyslog.rfc3164"]) {
   ... do something here ...
}
```

A more elaborate example can also be found in the *Cisco IOS message parser module* documentation.

timezone

The `timezone` object, as its name suggests, describes timezones. Currently, they are used by message parser modules to interpret timestamps that contain timezone information via a timezone string (but not an offset, e.g. "CET" but not "-01:00"). The object describes an UTC offset for a given timezone ID.

Each timestamp object adds the zone definition to a global table with timezone information. Duplicate IDs are forbidden, but the same offset may be used with multiple IDs.

Parameters

id <name-string>
Mandatory

This identifies the timezone. Note that this id must match the zone name as reported within the timestamps. Different devices and vendors use different, often non-standard, names and so it is important to use the actual ids that messages contain. For multiple devices, this may mean that you may need to include multiple definitions, each one with a different id, for the same time zone. For example, it is seen that some devices report "CEST" for central European daylight savings time while others report "METDST" for it.

offset <[+/-]><hh>:<mm>
Mandatory

This defines the timezone offset over UTC. It must always be 6 characters and start with a "+" (east of UTC) or "-" (west uf UTC) followed by a two-digit hour offset, a colon and a two-digit minute offset. Hour offsets can be in the range from zero to twelve, minute offsets in the range from zero to 59. Any other format is invalid.

Sample

The following sample defines UTC time. From rsyslog PoV, it doesn't matter if a plus or minus offset prefix is used. For consistency, plus is suggested.

```
timezone(id="UTC" offset="+00:00")
```

The next sample defines some common timezones:

```
timezone(id="CET" offset="+01:00")
timezone(id="CEST" offset="+02:00")
timezone(id="METDST" offset="+02:00") # duplicate to support differnt formats
timezone(id="EST" offset="-05:00")
timezone(id="EDT" offset="-04:00")
timezone(id="PST" offset="-08:00")
timezone(id="PDT" offset="-07:00")
```


Examples

Below are example for templates and selector lines. I hope they are self-explanatory.

Templates

Please note that the samples are split across multiple lines. A template MUST NOT actually be split across multiple lines.

A template that resembles traditional syslogd file output: $template TraditionalFormat,"%timegenerated% %HOSTNAME% %syslogtag%%msg:::drop-last-lf%\n"

A template that tells you a little more about the message: $template precise,"%syslogpriority%,%syslogfacility%,%timegenerated% %syslogtag%,%msg%\n"

A template for RFC 3164 format: $template RFC3164fmt,"<%PRI%>%TIMESTAMP% %HOSTNAME% %syslogtag%%msg%"

A template for the format traditonally used for user messages: $template usermsg," XXXX%syslogtag%%msg%\n\r"

And a template with the traditonal wall-message format: $template wallmsg,"\r\n\7Message from syslogd@%HOSTNAME% at %timegenerated%

A template that can be used for the database write (please note the SQL template option) $template MySQLInsert,"insert iut, message, receivedat values ('%iut%', '%msg:::UPPERCASE%', '%timegenerated:::date-mysql%') into systemevents\r\n", SQL

The following template emulates WinSyslog format (it's an Adiscon format, you do not feel bad if you don't know it ;)). It's interesting to see how it takes different parts out of the date stamps. What happens is that the date stamp is split into the actual date and time and the these two are combined with just a comma in between them.

```
$template WinSyslogFmt,"%HOSTNAME%,%timegenerated:1:10:date-rfc3339%,
%timegenerated:12:19:date-rfc3339%,%timegenerated:1:10:date-rfc3339%,
%timegenerated:12:19:date-rfc3339%,%syslogfacility%,%syslogpriority%,
%syslogtag%%msg%\\n"
```

Selector lines

```
# Store critical stuff in critical
#
*.=crit;kern.none /var/adm/critical
```

This will store all messages with the priority crit in the file /var/adm/critical, except for any kernel message.

```
# Kernel messages are first, stored in the kernel
# file, critical messages and higher ones also go
# to another host and to the console. Messages to
# the host server.example.net are forwarded in RFC 3164
# format (using the template defined above).
#
kern.* /var/adm/kernel
kern.crit @server.example.net;RFC3164fmt
kern.crit /dev/console
kern.info;kern.!err /var/adm/kernel-info
```

The first rule direct any message that has the kernel facility to the file /var/adm/kernel.

The second statement directs all kernel messages of the priority crit and higher to the remote host server.example.net. This is useful, because if the host crashes and the disks get irreparable errors you might not be able to read the stored messages. If they're on a remote host, too, you still can try to find out the reason for the crash.

The third rule directs these messages to the actual console, so the person who works on the machine will get them, too.

The fourth line tells rsyslogd to save all kernel messages that come with priorities from info up to warning in the file /var/adm/kernel-info. Everything from err and higher is excluded.

```
# The tcp wrapper loggs with mail.info, we display
# all the connections on tty12
#
mail.=info /dev/tty12
```

This directs all messages that uses mail.info (in source LOG_MAIL | LOG_INFO) to /dev/tty12, the 12th console. For example the tcpwrapper tcpd(8) uses this as it's default.

```
# Store all mail concerning stuff in a file
#
mail.\*;mail.!=info /var/adm/mail
```

This pattern matches all messages that come with the mail facility, except for the info priority. These will be stored in the file /var/adm/mail.

```
# Log all mail.info and news.info messages to info
#
mail,news.=info /var/adm/info
```

This will extract all messages that come either with mail.info or with news.info and store them in the file /var/adm/info.

```
# Log info and notice messages to messages file
#
*.=info;*.=notice;\
mail.none /var/log/messages
```

This lets rsyslogd log all messages that come with either the info or the notice facility into the file /var/log/messages, except for all messages that use the mail facility.

```
# Log info messages to messages file
#
*.=info;\
mail,news.none /var/log/messages
```

This statement causes rsyslogd to log all messages that come with the info priority to the file /var/log/messages. But any message coming either with the mail or the news facility will not be stored.

```
# Emergency messages will be displayed to all users
#
*.=emerg :omusrmsg:*
```

This rule tells rsyslogd to write all emergency messages to all currently logged in users.

```
# Messages of the priority alert will be directed
# to the operator
#
*.alert root,rgerhards
```

This rule directs all messages with a priority of alert or higher to the terminals of the operator, i.e. of the users "root" and "rgerhards" if they're logged in.

```
*.* @server.example.net
```

This rule would redirect all messages to a remote host called server.example.net. This is useful especially in a cluster of machines where all syslog messages will be stored on only one machine.

In the format shown above, UDP is used for transmitting the message. The destination port is set to the default auf 514. Rsyslog is also capable of using much more secure and reliable TCP sessions for message forwarding. Also, the destination port can be specified. To select TCP, simply add one additional @ in front of the host name (that is, @host is UPD, @@host is TCP). For example:

```
*.* @@server.example.net
```

To specify the destination port on the remote machine, use a colon followed by the port number after the machine name. The following forwards to port 1514 on server.example.net:

```
*.* @@server.example.net:1514
```

This syntax works both with TCP and UDP based syslog. However, you will probably primarily need it for TCP, as there is no well-accepted port for this transport (it is non-standard). For UDP, you can usually stick with the default auf 514, but might want to modify it for security reasons. If you would like to do that, it's quite easy:

```
*.* @server.example.net:1514
*.* >dbhost,dbname,dbuser,dbpassword;dbtemplate
```

This rule writes all message to the database "dbname" hosted on "dbhost". The login is done with user "dbuser" and password "dbpassword". The actual table that is updated is specified within the template (which contains the insert statement). The template is called "dbtemplate" in this case.

```
:msg,contains,"error" @server.example.net
```

This rule forwards all messages that contain the word "error" in the msg part to the server "errorServer". Forwarding is via UDP. Please note the colon in fron

Legacy Configuration Directives

All legacy configuration directives need to be specified on a line by their own and must start with a dollar-sign.

Note that legacy configuration directives that set object options (e.g. for inputs or actions) only affect those objects that are defined via legacy constructs. Objects defined via new-style RainerScript objects (e.g. action(), input()) are **not** affected by legacy directives. The reason is that otherwise we would again have the ability to mess up a configuration file with hard to understand constructs. This is avoided by not permitting to mix and match the way object values are set.

Configuration Parameter Types

Configuration parameter values have different data types. Unfortunately, the type currently must be guessed from the description (consider contributing to the doc to help improve it). In general, the following types are used:

- **numbers**

  The traditional integer format. Numbers may include '.' and ',' for readability. So you can for example specify either "1000" or "1,000" with the same result. Please note that rsyslogd simply *ignores* the punctuation. From it's point of view, "1,,0.0.,.,0" also has the value 1000.

- **sizes**

  Used for things like file size, main message queue sizes and the like. These are integers, but support modifier after the number part. For example, 1k means 1024. Supported are k(ilo), m(ega), g(iga), t(era), p(eta) and e(xa). Lower case letters refer to the traditional binary defintion (e.g. 1m equals 1,048,576) whereas upper case letters refer to their new 1000-based definition (e.g 1M equals 1,000,000).

- **complete line**

  A string consisting of multiple characters. This is relatively seldom used and sometimes looks confusing (rsyslog v7+ has a much better approach at these types of values).

- **single word**

  This is used when only a single word can be provided. A "single word" is a string without spaces in it. **No** quoting is necessary nor permitted (the quotes would become part of the word).

- **character**

  A single (printable) character. Must **not** be quoted.

- **boolean**

  The traditional boolean type, specified as "on" (1) or "off" (0).

Note that some other value types are occasionally used. However, the majority of types is one of those listed above. The list is updated as need arises and time permits.

Legacy Global Configuration Statements

Global configuration statements, as their name implies, usually affect some global features. However, some also affect main queues, which are "global" to a ruleset.

True Global Directives

rsyslog.conf configuration directive

$AbortOnUncleanConfig

Type: global configuration directive

Parameter Values: boolean (on/off, yes/no)

Available since: 5.3.1+

Default: off

Description:

This directive permits to prevent rsyslog from running when the configuration file is not clean. "Not Clean" means there are errors or some other annoyances that rsyslgod reports on startup. This is a user-requested feature to have a strict startup mode. Note that with the current code base it is not always possible to differentiate between an real error and a warning-like condition. As such, the startup will also prevented if warnings are present. I consider this a good thing in being "strict", but I admit there also currently is no other way of doing it.

Caveats:

Note that the consequences of a failed rsyslogd startup can be much more serious than a startup with only partial configuration. For example, log data may be lost or systems that depend on the log server in question will not be able to send logs, what in the ultimate result could result in a system hang on those systems. Also, the local system may hang when the local log socket has become full and is not read. There exist many such scenarios. As such, it is strongly recommended not to turn on this directive.

[rsyslog site]

This documentation is part of the rsyslog project.

$DebugPrintCFSyslineHandlerList

Type: global configuration directive

Default: on

Description:

Specifies whether or not the configuration file sysline handler list should be written to the debug log. Possible values: on/off. Default is on. Does not affect operation if debugging is disabled.

Sample:

$DebugPrintModuleList

Type: global configuration directive

Default: on

Description:

Specifies whether or not the module list should be written to the debug log. Possible values: on/off. Default is on. Does not affect operation if debugging is disabled.

Sample:

$DebugPrintTemplateList

Type: global configuration directive

Default: on

Description:

Specifies whether or not the template list should be written to the debug log. Possible values: on/off. Default is on. Does not affect operation if debugging is disabled..

Sample:

$FailOnChownFailure

Type: global configuration directive

Default: on

Description:

This option modifies behaviour of dynaFile creation. If different owners or groups are specified for new files or directories and rsyslogd fails to set these new owners or groups, it will log an error and NOT write to the file in question if that option is set to "on". If it is set to "off", the error will be ignored and processing continues. Keep in mind, that the files in this case may be (in)accessible by people who should not have permission. The default is "on".

Sample:

```
$FailOnChownFailure off
```


$GenerateConfigGraph

Type: global configuration directive

Default:

Available Since: 4.3.1 **CURRENTLY NOT AVAILABLE**

Description:

This directive is currently not supported. We had to disable it when we improved the rule engine. It is considerable effort to re-enable it. On the other hand, we are about to add a new config system, which will make yet another config graph method necessary. As such we have decided to currently disable this functionality and re-introduce it when the new config system has been instantiated.

This directive permits to create (hopefully) good-looking visualizations of rsyslogd's configuration. It does not affect rsyslog operation. If the directive is specified multiple times, all but the last are ignored. If it is specified, a graph is created. This happens both during a regular startup as well a config check run. It is recommended to include this directive only for documentation purposes and remove it from a production configuraton.

The graph is not drawn by rsyslog itself. Instead, it uses the great open source tool Graphviz to do the actual drawing. This has at least two advantages:

- the graph drawing support code in rsyslog is extremly slim and without overhead
- the user may change or further annotate the generated file, thus potentially improving his documentation

The drawback, of course, is that you need to run Graphviz once you have generated the control file with rsyslog. Fortunately, the process to do so is rather easy:

1. add "$GenerateConfigGraph /path/to/file.dot" to rsyslog.conf (from now on, I will call the file just file.dot). Optionally, add "$ActionName" statement **in front of** those actions that you like to use friendly names with. If you do this, keep the names short.
2. run rsyslog at least once (either in regular or configuration check mode)
3. remember to remove the $GenerateConfigGraph directive when you no longer need it (or comment it out)
4. change your working directory to where you place the dot file
5. if you would like to edit the rsyslog-generated file, now is the time to do so
6. do "dot -Tpng file.dot > file.png"

7. remember that you can use "convert -resize 50% file.png resized.png" if dot's output is too large (likely) or too small. Resizing can be especially useful if you intend to get a rough overview over your configuration.

After completing these steps, you should have a nice graph of your configuration. Details are missing, but that is exactly the point. At the start of the graph is always (at least in this version, could be improved) a node called "inputs" in a tripple hexagon shape. This represents all inputs active in the system (assuming you have defined some, what the current version does not check). Next comes the main queue. It is given in a hexagon shape. That shape indicates that a queue is peresent and used to de-couple the inbound from the outbound part of the graph. In technical terms, here is a threading boundary. Action with "real" queues (other than in direct mode) also utilize this shape. For actions, notice that a "hexagon action" creates a deep copy of the message. As such, a "discard hexagon action" actually does nothing, because it duplicates the message and then discards **the duplicate**. At the end of the diagram, you always see a "discard" action. This indicates that rsyslog discards messages which have been run through all available rules.

Edges are labeled with information about when they are taken. For filters, the type of filter, but not any specifics, are given. It is also indicated if no filter is applied in the configuration file (by using a "*.*" selector). Edges without labels are unconditionally taken. The actions themselfs are labeled with the name of the output module that handles them. If provided, the name given via "ActionName" is used instead. No further details are provided.

If there is anything in red, this should draw your attention. In this case, rsyslogd has detected something that does not look quite right. A typical example is a discard action which is followed by some other actions in an action unit. Even though something may be red, it can be valid - rsyslogd's graph generator does not yet check each and every speciality, so the configuration may just cover a very uncommon case.

Now let's look at some examples. The graph below was generated on a fairly standard Fedora rsyslog.conf file. It had only the usually commented-out last forwarding action activated:

This is the typical structure for a simple rsyslog configuration. There are a couple of actions, each guarded by a filter. Messages run from top to bottom and control branches whenever a filter evaluates to true. As there is no discard action, all messages will run through all filters and discarded in the system default discard action right after all configured actions.

A more complex example can be seen in the next graph. This is a configuration I created for testing the graph-creation features, so it contains a little bit of everything. However, real-world configurations can look quite complex, too (and I wouldn't say this one is very complex):

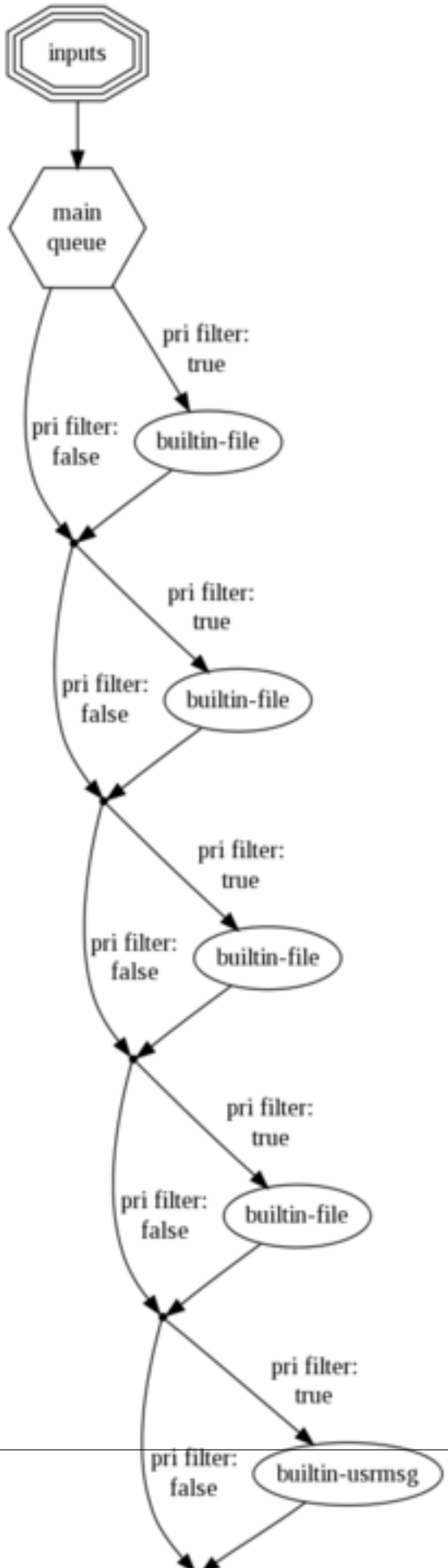
Here, we have a user-defined discard action. You can immediately see this because processing branches after the first "builtin-file" action. Those messages where the filter evaluates to true for will never run through the left-hand action branch. However, there is also a configuration error present: there are two more actions (now shown red) after the discard action. As the message is discarded, these will never be executed. Note that the discard branch contains no further filters. This is because these actions are all part of the same action unit, which is guarded only by an entry filter. The same is present a bit further down at the node labeled "write_system_log_2". This note has one more special feature, that is label was set via "ActionName", thus is does not have standard form (the same happened to the node named "Forward" right at the top of the diagram. Inside this diagram, the "Forward" node is executed asynchonously on its own queue. All others are executed synchronously.

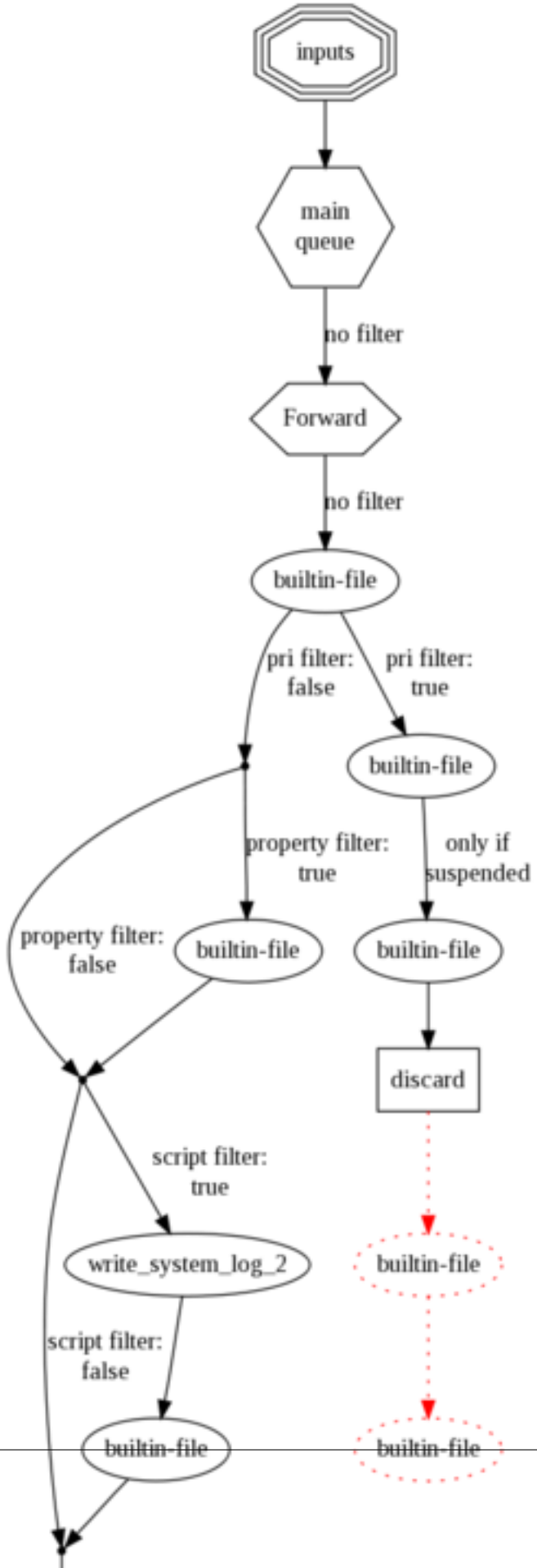
Configuration graphs are useful for documenting a setup, but are also a great troubleshooting resource. It is important to remember that **these graphs are generated from rsyslogd's in-memory action processing structures**. You can not get closer to understanding on how rsyslog interpreted its configuration files. So if the graph does not look what you intended to do, there is probably something worng in rsyslog.conf.

If something is not working as expected, but you do not spot the error immediately, I recommend to generate a graph and zoom it so that you see all of it in one great picture. You may not be able to read anything, but the structure should look good to you and so you can zoom into those areas that draw your attention.

Sample:

```
$DirOwner /path/to/graphfile-file.dot
```


inputs
main queue
pri filter: true
pri filter: false
builtin-file
pri filter: true
pri filter: false
builtin-file
pri filter: true
pri filter: false
builtin-file
pri filter: true
pri filter: false
builtin-file
pri filter: true
pri filter: false
builtin-usrmsg


inputs
main queue
no filter
Forward
no filter
builtin-file
pri filter: false
pri filter: true
builtin-file
property filter: true
only if suspended
property filter: false
builtin-file
builtin-file
discard
script filter: true
write_system_log_2
builtin-file
script filter: false
builtin-file
builtin-file

$IncludeConfig

Type: global configuration directive

Default:

Description:

This directive allows to include other files into the main configuration file. As soon as an IncludeConfig directive is found, the contents of the new file is processed. IncludeConfigs can be nested. Please note that from a logical point of view the files are merged. Thus, if the include modifies some parameters (e.g. $DynaFileChacheSize), these new parameters are in place for the "calling" configuration file when the include is completed. To avoid any side effects, do a $ResetConfigVariables after the $IncludeConfig. It may also be a good idea to do a $ResetConfigVariables right at the start of the include, so that the module knows exactly what it does. Of course, one might specifically NOT do this to inherit parameters from the main file. As always, use it as it best fits...

Note: if multiple files are included, they are processed in ascending sort order of the file name. We use the "glob()" C library function for obtaining the sorted list. On most platforms, especially Linux, this means the the sort order is the same as for bash.

If all regular files in the /etc/rsyslog.d directory are included, then files starting with "." are ignored - so you can use them to place comments into the dir (e.g. "/etc/rsyslog.d/.mycomment" will be ignored). Michael Biebl had the idea to this functionality. Let me quote him:

> **Say you can add an option** $IncludeConfig /etc/rsyslog.d/ (which probably would make a good default) to /etc/rsyslog.conf, which would then merge and include all
>
> ***.conf files** in /etc/rsyslog.d/. This way, a distribution can modify its packages easily to drop a
>
> **simple** config file into this directory upon installation. As an example, the network-manager package could install a simple
>
> **config** file /etc/rsyslog.d/network-manager.conf which would contain. :programname, contains, "NetworkManager"
>
> **-/var/log/NetworkManager.log** Upon uninstallation, the file could be easily removed again. This
>
> **approach** would be much cleaner and less error prone, than having to munge
>
> **around** with the /etc/rsyslog.conf file directly.

Sample:

```
$IncludeConfig /etc/some-included-file.conf
```

Directories can also be included. To do so, the name must end on a slash:

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

And finally, only specific files matching a wildcard my be included from a directory:

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


$MainMsgQueueSize

Type: global configuration directive

Default: 10000

Description:

This allows to specify the maximum size of the message queue. This directive is only available when rsyslogd has been compiled with multithreading support. In this mode, receiver and output modules are de-coupled via an in-memory queue. This queue buffers messages when the output modules are not capable to process them as fast as they are received. Once the queue size is exhausted, messages will be dropped. The slower the output (e.g. MySQL), the larger the queue should be. Buffer space for the actual queue entries is allocated on an as-needed basis. Please keep in mind that a very large queue may exhaust available system memory and swap space. Keep this in mind when configuring the max size. The actual size of a message depends largely on its content and the originator. As a rule of thumb, typically messages should not take up more then roughly 1k (this is the memory structure, not what you see in a network dump!). For typical linux messages, 512 bytes should be a good bet. Please also note that there is a minimal amount of memory taken for each queue entry, no matter if it is used or not. This is one pointer value, so on 32bit systems, it should typically be 4 bytes and on 64bit systems it should typically be 8 bytes. For example, the default queue size of 10,000 entries needs roughly 40k fixed overhead on a 32 bit system.

Sample:

```
$MainMsgQueueSize 100000 # 100,000 may be a value to handle burst traffic
```


$MaxOpenFiles

Available Since: 4.3.0

Type: global configuration directive

Default: *operating system default*

Description:

Set the maximum number of files that the rsyslog process can have open at any given time. Note that this includes open tcp sockets, so this setting is the upper limit for the number of open TCP connections as well. If you expect a large nubmer of concurrent connections, it is suggested that the number is set to the max number connected plus 1000. Please note that each dynafile also requires up to 100 open file handles.

The setting is similar to running "ulimit -n number-of-files".

Please note that depending on permissions and operating system configuration, the setrlimit() request issued by rsyslog may fail, in which case the previous limit is kept in effect. Rsyslog will emit a warning message in this case.

Sample:

```
$MaxOpenFiles 2000
```

Bugs:

For some reason, this settings seems not to work on all platforms. If you experience problems, please let us know so that we can (hopefully) narrow down the issue.

$ModDir

Type: global configuration directive

Default: system default for user libraries, e.g. /usr/local/lib/rsyslog/

Description:

Provides the default directory in which loadable modules reside. This may be used to specify an alternate location that is not based on the system default. If the system default is used, there is no need to specify this directive. Please note that it is vitally important to end the path name with a slash, else module loads will fail.

Sample:

```
$ModDir /usr/rsyslog/libs/  # note the trailing slash!
```

This documentation is part of the rsyslog project.

$ModLoad

Type: global configuration directive

Default:

Description:

Dynamically loads a plug-in into rsyslog's address space and activates it. The plug-in must obey the rsyslog module API. Currently, only MySQL and Postgres output modules are available as a plugins, but users may create their own. A plug-in must be loaded BEFORE any configuration file lines that reference it.

Modules must be present in the system default destination for rsyslog modules. You can also set the directory via the $ModDir directive.

If a full path name is specified, the module is loaded from that path. The default module directory is ignored in that case.

Sample:

```
$ModLoad ommysql # load MySQL functionality $ModLoad /rsyslog/modules/ompgsql.
so # load the postgres module via absolute path
```

This documentation is part of the rsyslog project.

$UMASK

Type: global configuration directive

Default:

Description:

The $umask directive allows to specify the rsyslogd processes' umask. If not specified, the system-provided default is used. The value given must always be a 4-digit octal number, with the initial digit being zero.

If $umask is specified multiple times in the configuration file, results may be somewhat unpredictable. It is recommended to specify it only once.

Sample:

```
$umask 0000
```

This sample removes all restrictions.

[rsyslog site]

This documentation is part of the rsyslog project.

$ResetConfigVariables

Type: global configuration directive

Default:

Description:

Resets all configuration variables to their default value. Any settings made will not be applied to configuration lines following the $ResetConfigVariables. This is a good method to make sure no side-effects exists from previous directives. This directive has no parameters.

Sample:

```
$ResetConfigVariables
```

This documentation is part of the rsyslog project.

- **$MaxMessageSize** <size_nbr>, default 8k - allows to specify maximum supported message size (both for sending and receiving). The default should be sufficient for almost all cases. Do not set this below 1k, as it would cause interoperability problems with other syslog implementations.

  Important: In order for this directive to work correctly, it **must** be placed right at the top of `rsyslog.conf` (before any input is defined).

  Change the setting to e.g. 32768 if you would like to support large message sizes for IHE (32k is the current maximum needed for IHE). I was initially tempted to set the default to 32k, but there is a some memory footprint with the current implementation in rsyslog. If you intend to receive Windows Event Log data (e.g. via EventReporter), you might want to increase this number to an even higher value, as event log messages can be very lengthy ("$MaxMessageSize 64k" is not a bad idea). Note: testing showed that 4k seems to be the typical maximum for **UDP** based syslog. This is an IP stack restriction. Not always ... but very often. If you go beyond that value, be sure to test that rsyslogd actually does what you think it should do ;) It is highly suggested to use a TCP based transport instead of UDP (plain TCP syslog, RELP). This resolves the UDP stack size restrictions. Note that 2k, is the smallest size that must be supported in order to be compliant to the upcoming new syslog RFC series.

- **$LocalHostName** [name] - this directive permits to overwrite the system hostname with the one specified in the directive. If the directive is given multiple times, all but the last one will be ignored. Please note that startup error messages may be issued with the real hostname. This is by design and not a bug (but one may argue if the design should be changed ;)). Available since 4.7.4+, 5.7.3+, 6.1.3+.

- **$LogRSyslogStatusMessages** [**on**/off] - If set to on (the default), rsyslog emits message on startup and shutdown as well as when it is HUPed. This information might be needed by some log analyzers. If set to off, no such status messages are logged, what may be useful for other scenarios. [available since 4.7.0 and 5.3.0]

- **$DefaultRuleset** [name] - changes the default ruleset for unbound inputs to the provided *name* (the default default ruleset is named "RSYSLOG_DefaultRuleset"). It is advised to also read our paper on *using multiple rule sets in rsyslog*.

- **$DefaultNetstreamDriver** <drivername>, the default *network stream driver* to use. Defaults to ptcp.

- **$DefaultNetstreamDriverCAFile** </path/to/cafile.pem>

- **$DefaultNetstreamDriverCertFile** </path/to/certfile.pem>

- **$DefaultNetstreamDriverKeyFile** </path/to/keyfile.pem>

- **$RepeatedMsgContainsOriginalMsg** [on/**off**] - "last message repeated n times" messages, if generated, have a different format that contains the message that is being repeated. Note that only the first "n" characters are included, with n to be at least 80 characters, most probably more (this may change from version to version, thus no specific limit is given). The bottom line is that n is large enough to get a good idea which message

was repeated but it is not necessarily large enough for the whole message. (Introduced with 4.1.5). Once set, it affects all following actions.

- **$OptimizeForUniprocessor** [on/**off**] - turns on optimizatons which lead to better performance on uniprocessors. If you run on multicore-machiens, turning this off lessens CPU load. The default may change as uniprocessor systems become less common. [available since 4.1.0]
- **$PreserveFQDN** [on/**off**) - if set to off (legacy default to remain compatible to sysklogd), the domain part from a name that is within the same domain as the receiving system is stripped. If set to on, full names are always used.
- **$WorkDirectory** <name> (directory for spool and other work files. Do **not** use trailing slashes)
- $PrivDropToGroup
- $PrivDropToGroupID
- $PrivDropToUser
- $PrivDropToUserID
- **$Sleep** <seconds> - puts the rsyslog main thread to sleep for the specified number of seconds immediately when the directive is encountered. You should have a good reason for using this directive!
- **$LocalHostIPIF** <interface name> - (available since 5.9.6) - if provided, the IP of the specified interface (e.g. "eth0") shall be used as fromhost-ip for locall-originating messages. If this directive is not given OR the interface cannot be found (or has no IP address), the default of "127.0.0.1" is used. Note that this directive can be given only once. Trying to reset will result in an error message and the new value will be ignored. Please note that modules must have support for obtaining the local IP address set via this directive. While this is the case for rsyslog-provided modules, it may not always be the case for contributed plugins. **Important:** This directive shall be placed **right at the top of rsyslog.conf**. Otherwise, if error messages are triggered before this directive is processed, rsyslog will fix the local host IP to "127.0.0.1", what than can not be reset.
- **$ErrorMessagesToStderr** [**on**|off] - direct rsyslogd error message to stderr (in addition to other targets)
- **$SpaceLFOnReceive** [on/**off**] - instructs rsyslogd to replace LF with spaces during message reception (sysklogd compatibility aid). This is applied at the beginning of the parser stage and cannot be overridden (neither at the input nor parser level). Consequently, it affects all inputs and parsers.

main queue specific Directives

Note that these directives modify ruleset main message queues. This includes the default ruleset's main message queue, the one that is always present. To do this, specify directives outside of a ruleset definition.

To understand queue parameters, read *queues in rsyslog*.

- **$MainMsgQueueCheckpointInterval** <number>
- **$MainMsgQueueDequeueBatchSize** <number> [default 32]
- **$MainMsgQueueDequeueSlowdown** <number> [number is timeout in *micro*seconds (1000000us is 1sec!), default 0 (no delay). Simple rate-limiting!]
- **$MainMsgQueueDiscardMark** <number> [default 9750]
- **$MainMsgQueueDiscardSeverity** <severity> [either a textual or numerical severity! default 4 (warning)]
- **$MainMsgQueueFileName** <name>
- **$MainMsgQueueHighWaterMark** <number> [default 8000]
- **$MainMsgQueueImmediateShutdown** [on/**off**]

- **$MainMsgQueueLowWaterMark** <number> [default 2000]
- **$MainMsgQueueMaxFileSize** <size_nbr>, default 1m
- **$MainMsgQueueTimeoutActionCompletion** <number> [number is timeout in ms (1000ms is 1sec!), default 1000, 0 means immediate!]
- **$MainMsgQueueTimeoutEnqueue** <number> [number is timeout in ms (1000ms is 1sec!), default 2000, 0 means discard immediately]
- **$MainMsgQueueTimeoutShutdown** <number> [number is timeout in ms (1000ms is 1sec!), default 0 (indefinite)]
- **$MainMsgQueueWorkerTimeoutThreadShutdown** <number> [number is timeout in ms (1000ms is 1sec!), default 60000 (1 minute)]
- **$MainMsgQueueType** [**FixedArray**/LinkedList/Direct/Disk]
- **$MainMsgQueueSaveOnShutdown** [on/**off**]
- **$MainMsgQueueWorkerThreads** <number>, num worker threads, default 1, recommended 1
- **$MainMsgQueueWorkerThreadMinumumMessages** <number>, default 100

Legacy Directives affecting Input Modules

Legacy Directives affecting multiple Input Modules

While these directives only affect input modules, they are global in the sense that they cannot be overwritten for specific input instances. So they apply globally for all inputs that support these directives.

$AllowedSender

Type: input configuration directive

Default: all allowed

Description:

Note: this feature is supported for backward-compatibility, only. The rsyslog team recommends to use proper firewalling instead of this feature.

Allowed sender lists can be used to specify which remote systems are allowed to send syslog messages to rsyslogd. With them, further hurdles can be placed between an attacker and rsyslogd. If a message from a system not in the allowed sender list is received, that message is discarded. A diagnostic message is logged, so that the fact is recorded (this message can be turned off with the "-w" rsyslogd command line option).

Allowed sender lists can be defined for UDP and TCP senders separately. There can be as many allowed senders as needed. The syntax to specify them is:

```
$AllowedSender <type>, ip[/bits], ip[/bits]
```

"$AllowedSender" is the directive - it must be written exactly as shown and the $ must start at the first column of the line. "<type>" is either "UDP" or "TCP" (or "GSS", if this is enabled during compilation). It must immediately be followed by the comma, else you will receive an error message. "ip[/bits]" is a machine or network ip address as in "192.0.2.0/24" or "127.0.0.1". If the "/bits" part is omitted, a single host is assumed (32 bits or mask 255.255.255.255). "/0" is not allowed, because that would match any sending system. If you intend to do that, just remove all $AllowedSender directives. If more than 32 bits are requested with IPv4, they are adjusted to 32. For IPv6, the limit is 128 for obvious reasons. Hostnames, with and without wildcards, may also be provided. If so, the result of revers DNS

resolution is used for filtering. Multiple allowed senders can be specified in a comma-delimited list. Also, multiple $AllowedSender lines can be given. They are all combined into one UDP and one TCP list. Performance-wise, it is good to specify those allowed senders with high traffic volume before those with lower volume. As soon as a match is found, no further evaluation is necessary and so you can save CPU cycles.

Rsyslogd handles allowed sender detection very early in the code, nearly as the first action after receiving a message. This keeps the access to potential vulnerable code in rsyslog at a minimum. However, it is still a good idea to impose allowed sender limitations via firewalling.

WARNING: by UDP design, rsyslogd can not identify a spoofed sender address in UDP syslog packets. As such, a malicious person could spoof the address of an allowed sender, send such packets to rsyslogd and rsyslogd would accept them as being from the faked sender. To prevent this, use syslog via TCP exclusively. If you need to use UDP-based syslog, make sure that you do proper egress and ingress filtering at the firewall and router level.

Rsyslog also detects some kind of malicious reverse DNS entries. In any case, using DNS names adds an extra layer of vulnerability. We recommend to stick with hard-coded IP addresses wherever possible.

Sample:

```
$AllowedSender UDP, 127.0.0.1, 192.0.2.0/24, [::1]/128, *.example.net, somehost.
↪example.com
```


$DropMsgsWithMaliciousDnsPTRRecords

Type: global configuration directive

Default: off

Description:

Rsyslog contains code to detect malicious DNS PTR records (reverse name resolution). An attacker might use specially-crafted DNS entries to make you think that a message might have originated on another IP address. Rsyslog can detect those cases. It will log an error message in any case. If this option here is set to "on", the malicious message will be completely dropped from your logs. If the option is set to "off", the message will be logged, but the original IP will be used instead of the DNS name.

Sample:

```
$DropMsgsWithMaliciousDnsPTRRecords on
```


$ControlCharacterEscapePrefix

Type: global configuration directive

Default: \

Description:

This option specifies the prefix character to be used for control character escaping (see option $EscapeControlCharactersOnReceive). By default, it is '\', which is backwards-compatible with sysklogd. Change it to '#' in order to be compliant to the value that is somewhat suggested by Internet-Draft syslog-protocol.

IMPORTANT: do not use the ‘ character. This is reserved and will most probably be used in the future as a character delimiter. For the same reason, the syntax of this directive will probably change in future releases.

Sample:

```
$EscapeControlCharactersOnReceive #  # as of syslog-protocol
```


$DropTrailingLFOnReception

Type: global configuration directive

Default: on

Description:

Syslog messages frequently have the line feed character (LF) as the last character of the message. In almost all cases, this LF should not really become part of the message. However, recent IETF syslog standardization recommends against modifying syslog messages (e.g. to keep digital signatures valid). This option allows to specify if trailing LFs should be dropped or not. The default is to drop them, which is consistent with what sysklogd does.

Sample:

```
$DropTrailingLFOnRecption on
```


$Escape8BitCharactersOnReceive

Type: global configuration directive

Default: off

Available Since: 5.5.2

Description:

This directive instructs rsyslogd to replace non US-ASCII characters (those that have the 8th bit set) during reception of the message. This may be useful for some systems. Please note that this escaping breaks Unicode and many other encodings. Most importantly, it can be assumed that Asian and European characters will be rendered hardly readable by this settings. However, it may still be useful when the logs themself are primarily in English and only occasionally contain local script. If this option is turned on, all control-characters are converted to a 3-digit octal number and be prefixed with the $ControlCharacterEscapePrefix character (being ‘#’ by default).

Warning:

- turning on this option most probably destroys non-western character sets (like Japanese, Chinese and Korean) as well as European character sets.
- turning on this option destroys digital signatures if such exists inside the message
- if turned on, the drop-cc, space-cc and escape-cc property replacer options do not work as expected because control characters are already removed upon message reception. If you intend to use these property replacer options, you must turn off $Escape8BitCharactersOnReceive.

Sample:

```
$Escape8BitCharactersOnReceive on
```

$EscapeControlCharactersOnReceive

Type: global configuration directive

Default: on

Description:

This directive instructs rsyslogd to replace control characters during reception of the message. The intent is to provide a way to stop non-printable messages from entering the syslog system as whole. If this option is turned on, all control-characters are converted to a 3-digit octal number and be prefixed with the $ControlCharacterEscapePrefix character (being '\' by default). For example, if the BEL character (ctrl-g) is included in the message, it would be converted to "\007". To be compatible to sysklogd, this option must be turned on.

Warning:

- turning on this option most probably destroys non-western character sets (like Japanese, Chinese and Korean)
- turning on this option destroys digital signatures if such exists inside the message
- if turned on, the drop-cc, space-cc and escape-cc property replacer options do not work as expected because control characters are already removed upon message reception. If you intend to use these property replacer options, you must turn off $EscapeControlCharactersOnReceive.

Sample:

```
$EscapeControlCharactersOnReceive on
```


immark-specific Directives

$MarkMessagePeriod

Type: specific to immark input module

Default: 1200 (20 minutes)

Description:

This specifies when mark messages are to be written to output modules. The time specified is in seconds. Specifying 0 is possible and disables mark messages. In that case, however, it is more efficient to NOT load the immark input module.

So far, there is only one mark message process and any subsequent $MarkMessagePeriod overwrites the previous.

This directive is only available after the immark input module has been loaded.

Sample:

```
$MarkMessagePeriod  600 # mark messages appear every 10 Minutes
```

Available since: rsyslog 3.0.0

Legacy Action-Specific Configuration Statements

Statements modify the next action(s) that is/are defined **via legacy syntax** after the respective statement. Actions defined via the action() object are **not** affected by the legacy statements listed here. Use the action() object properties instead.

Generic action configuration Statements

These statements can be used with all types of actions.

$ActionExecOnlyWhenPreviousIsSuspended

Type: global configuration directive

Default: off

Description:

This directive allows to specify if actions should always be executed ("off," the default) or only if the previous action is suspended ("on"). This directive works hand-in-hand with the multiple actions per selector feature. It can be used, for example, to create rules that automatically switch destination servers or databases to a (set of) backup(s), if the primary server fails. Note that this feature depends on proper implementation of the suspend feature in the output module. All built-in output modules properly support it (most importantly the database write and the syslog message forwarder).

This selector processes all messages it receives (*.*). It tries to forward every message to primary-syslog.example.com (via tcp). If it can not reach that server, it tries secondary-1-syslog.example.com, if that fails too, it tries secondary-2-syslog.example.com. If neither of these servers can be connected, the data is stored in /var/log/localbuffer. Please note that the secondaries and the local log buffer are only used if the one before them does not work. So ideally, /var/log/localbuffer will never receive a message. If one of the servers resumes operation, it automatically takes over processing again.

We strongly advise not to use repeated line reduction together with ActionExecOnlyWhenPreviousIsSuspended. It may lead to "interesting" and undesired results (but you can try it if you like).

Sample:

. @@primary-syslog.example.com $ActionExecOnlyWhenPreviousIsSuspended on & @@secondary-1-syslog.example.com # & is used to have more than one action for & @@secondary-2-syslog.example.com # the same selector - the mult-action feature & /var/log/localbuffer $ActionExecOnlyWhenPreviousIsSuspended off # to re-set it for the next selector

This documentation is part of the rsyslog project.

$ActionResumeInterval

Type: global configuration directive

Default: 30

Description:

Sets the ActionResumeInterval for all following actions. The interval provided is always in seconds. Thus, multiply by 60 if you need minutes and 3,600 if you need hours (not recommended).

When an action is suspended (e.g. destination can not be connected), the action is resumed for the configured interval. Thereafter, it is retried. If multiple retries fail, the interval is automatically extended. This is to prevent excessive resource use for retries. After each 10 retries, the interval is extended by itself. To be precise, the actual interval is (numRetries / 10 + 1) * $ActionResumeInterval. so after the 10th try, it by default is 60 and after the 100th try it is 330.

Sample:

$ActionResumeInterval 30

This documentation is part of the rsyslog project.

$RepeatedMsgReduction

Type: global configuration directive

Default: off

Description

This directive models old sysklogd legacy. **Note that many people, including the rsyslog authors, consider this to be a misfeature.** See *Discussion* below to learn why.

This directive specifies whether or not repeated messages should be reduced (this is the "Last line repeated n times" feature). If set to *on*, repeated messages are reduced. If kept at *off*, every message is logged. In very early versions of rsyslog, this was controlled by the *-e* command line option.

What is a repeated message

For a message to be classified as repeated, the following properties must be **identical**:

- msg
- hostname
- procid
- appname

Note that rate-limiters are usually applied to specific input sources or processes. So first and foremost the input source must be the same to classify a messages as a duplicated.

You may want to check out testing rsyslog ratelimiting for some extra information on the per-process ratelimiting.

Discussion

- Very old versions of rsyslog did not have the ability to include the repeated message itself within the repeat message.
- Versions before 7.3.2 applied repeat message reduction to the output side. This had some implications:
  - they did not account for the actual message origin, so two processes emitting an equally-looking messsage triggered the repeated message reduction code
  - repeat message processing could be set on a per-action basis, which has switched to per-input basis for 7.3.2 and above

- While turning this feature on can save some space in logs, most log analysis tools need to see the repeated messages, they can't handle the "last message repeated" format.
- This is a feature that worked decades ago when logs were small and reviewed by a human, it fails badly on high volume logs processed by tools.

Sample

This turns on repeated message reduction (**not** recommended):

```
$RepeatedMsgReduction on    # do not log repeated messages
```


- **$ActionName** <a_single_word> - used primarily for documentation, e.g. when generating a configuration graph. Available sice 4.3.1.
- **$ActionExecOnlyOnceEveryInterval** <seconds> - execute action only if the last execute is at last <seconds> seconds in the past (more info in ommail, but may be used with any action). To disable this setting, use value 0.
- **$ActionExecOnlyEveryNthTime** <number> - If configured, the next action will only be executed every n-th time. For example, if configured to 3, the first two messages that go into the action will be dropped, the 3rd will actually cause the action to execute, the 4th and 5th will be dropped, the 6th executed under the action, ... and so on. Note: this setting is automatically re-set when the actual action is defined.
- **$ActionExecOnlyEveryNthTimeTimeout** <number-of-seconds> - has a meaning only if $ActionExecOnlyEveryNthTime is also configured for the same action. If so, the timeout setting specifies after which period the counting of "previous actions" expires and a new action count is begun. Specify 0 (the default) to disable timeouts. *Why is this option needed?* Consider this case: a message comes in at, eg., 10am. That's count 1. Then, nothing happens for the next 10 hours. At 8pm, the next one occurs. That's count 2. Another 5 hours later, the next message occurs, bringing the total count to 3. Thus, this message now triggers the rule. The question is if this is desired behavior? Or should the rule only be triggered if the messages occur within an e.g. 20 minute window? If the later is the case, you need a $ActionExecOnlyEveryNthTimeTimeout 1200 This directive will timeout previous messages seen if they are older than 20 minutes. In the example above, the count would now be always 1 and consequently no rule would ever be triggered.
- **$ActionResumeRetryCount** <number> [default 0, -1 means eternal]
- **$ActionWriteAllMarkMessages** [on/**off**]- [available since 5.1.5] - normally, mark messages are written to actions only if the action was not recently executed (by default, recently means within the past 20 minutes). If this setting is switched to "on", mark messages are always sent to actions, no matter how recently they have been executed. In this mode, mark messages can be used as a kind of heartbeat. Note that this option auto-resets to "off", so if you intend to use it with multiple actions, it must be specified in front off **all** selector lines that should provide this functionality.

omfile-specific Configuration Statements

These statements are specific to omfile-based actions.

$omfileForceChown

Type: global configuration directive

Parameter Values: boolean (on/off, yes/no)

Available: 4.7.0+, 5.3.0-5.8.x, **NOT** available in 5.9.x or higher

Note: this directive has been removed and is no longer available. The documentation is currently being retained for historical reaons. Expect it to go away at some later stage as well.

Default: off

Description:

Forces rsyslogd to change the ownership for output files that already exist. Please note that this tries to fix a potential problem that exists outside the scope of rsyslog. Actually, it tries to fix invalid ownership/permission settings set by the original file creator.

Rsyslog changes the ownership during initial execution with root privileges. When a privelege drop is configured, privileges are dropped after the file owner ship is changed. Not that this currently is a limitation in rsyslog's privilege drop code, which is on the TODO list to be removed. See Caveats section below for the important implications.

Caveats:

This directive tries to fix a problem that actually is outside the scope of rsyslog. As such, there are a couple of restrictions and situations in which it will not work. **Users are strongly encouraged to fix their system instead of turning this directive on** - it should only be used as a last resort.

At least in the following scenario, this directive will fail expectedly:

It does not address the situation that someone changes the ownership *after* rsyslogd has started. Let's, for example, consider a log rotation script.

- rsyslog is started
- ownership is changed
- privileges dropped
- log rotation (lr) script starts
- lr removes files
- lr creates new files with root:adm (or whatever else)
- lr HUPs rsyslogd
- rsyslogd closes files
- rsyslogd tries to open files
- rsyslogd tries to change ownership –> fail as we are non-root now
- file open fails

Please note that once the privilege drop code is refactored, this directive will no longer work, because then privileges will be dropped before any action is performed, and thus we will no longer be able to chown files that do not belong to the user rsyslogd is configured to run under.

So **expect the directive to go away**. It will not be removed in version 4, but may disappear at any time for any version greater than 4.

Sample:

```
$FileOwner loguser $omfileForceChown on
```

$DirGroup

Type: global configuration directive

Default:

Description:

Set the group for directories newly created. Please note that this setting does not affect the group of directories already existing. The parameter is a group name, for which the groupid is obtained by rsyslogd on during startup processing. Interim changes to the user mapping are not detected.

Sample:

```
$DirGroup loggroup
```


$DirOwner

Type: global configuration directive

Default:

Description:

Set the file owner for directories newly created. Please note that this setting does not affect the owner of directories already existing. The parameter is a user name, for which the userid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

Sample:

```
$DirOwner loguser
```


$DynaFileCacheSize

Type: global configuration directive

Default: 10

Description:

This directive specifies the maximum size of the cache for dynamically-generated file names. Selector lines with dynamic files names ('?' indicator) support writing to multiple files with a single selector line. This setting specifies how many open file handles should be cached. If, for example, the file name is generated with the hostname in it and you have 100 different hosts, a cache size of 100 would ensure that files are opened once and then stay open. This can be a great way to increase performance. If the cache size is lower than the number of different files, the least recently used one is discarded (and the file closed). The hardcoded maximum is 10,000 - a value that we assume should already be very extreme. Please note that if you expect to run with a very large number of files, you probably need to reconfigure the kernel to support such a large number. In practice, we do NOT recommend to use a cache of more than 1,000 entries. The cache lookup would probably require more time than the open and close operations. The minimum value is 1.

Numbers are always in decimal. Leading zeros should be avoided (in some later version, they may be mis-interpreted as being octal). Multiple directives may be given. They are applied to selector lines based on order of appearance.

Sample:

```
$DynaFileCacheSize 100    # a cache of 100 files at most
```


$FileCreateMode

Type: global configuration directive

Default: 0644

Description:

The $FileCreateMode directive allows to specify the creation mode with which rsyslogd creates new files. If not specified, the value 0644 is used (which retains backward-compatibility with earlier releases). The value given must always be a 4-digit octal number, with the initial digit being zero.

Please note that the actual permission depend on rsyslogd's process umask. If in doubt, use "$umask 0000" right at the beginning of the configuration file to remove any restrictions.

$FileCreateMode may be specified multiple times. If so, it specifies the creation mode for all selector lines that follow until the next $FileCreateMode directive. Order of lines is vitally important.

Sample:

```
$FileCreateMode 0600
```

This sample lets rsyslog create files with read and write access only for the users it runs under.

The following sample is deemed to be a complete rsyslog.conf:

```
$umask 0000 # make sure nothing interferes with the following definitions
*.* /var/log/file-with-0644-default $FileCreateMode 0600 *.* /var/log/
file-with-0600 $FileCreateMode 0644 *.* /var/log/file-with-0644
```

As you can see, open modes depend on position in the config file. Note the first line, which is created with the hardcoded default creation mode.

$FileGroup

Type: global configuration directive

Default:

Description:

Set the group for dynaFiles newly created. Please note that this setting does not affect the group of files already existing. The parameter is a group name, for which the groupid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

Sample:

```
$FileGroup loggroup
```

$FileOwner

Type: global configuration directive

Default:

Description:

Set the file owner for dynaFiles newly created. Please note that this setting does not affect the owner of files already existing. The parameter is a user name, for which the userid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

Sample:

```
$FileOwner loguser
```


- **$CreateDirs** [**on**/off] - create directories on an as-needed basis
- **$ActionFileDefaultTemplate** [templateName] - sets a new default template for file actions
- **$ActionFileEnableSync [on/off]** - enables file syncing capability of omfile
- **$OMFileAsyncWriting** [on/**off**], if turned on, the files will be written in asynchronous mode via a separate thread. In that case, double buffers will be used so that one buffer can be filled while the other buffer is being written. Note that in order to enable $OMFileFlushInterval, $OMFileAsyncWriting must be set to "on". Otherwise, the flush interval will be ignored. Also note that when $OMFileFlushOnTXEnd is "on" but $OMFileAsyncWriting is off, output will only be written when the buffer is full. This may take several hours, or even require a rsyslog shutdown. However, a buffer flush can be forced in that case by sending rsyslogd a HUP signal.
- **$OMFileZipLevel** 0..9 [default 0] - if greater 0, turns on gzip compression of the output file. The higher the number, the better the compression, but also the more CPU is required for zipping.
- **$OMFileIOBufferSize** <size_nbr>, default 4k, size of the buffer used to writing output data. The larger the buffer, the potentially better performance is. The default of 4k is quite conservative, it is useful to go up to 64k, and 128K if you used gzip compression (then, even higher sizes may make sense)
- **$OMFileFlushOnTXEnd** <[**on**/off]>, default on. Omfile has the capability to write output using a buffered writer. Disk writes are only done when the buffer is full. So if an error happens during that write, data is potentially lost. In cases where this is unacceptable, set $OMFileFlushOnTXEnd to on. Then, data is written at the end of each transaction (for pre-v5 this means after **each** log message) and the usual error recovery thus can handle write errors without data loss. Note that this option severely reduces the effect of zip compression and should be switched to off for that use case. Note that the default -on- is primarily an aid to preserve the traditional syslogd behaviour.

omfwd-specific Configuration Statements

These statements are specific to omfwd-based actions.

- **$ActionForwardDefaultTemplate** [templateName] - sets a new default template for UDP and plain TCP forwarding action
- **$ActionSendResendLastMsgOnReconnect** <[on/**off**]> specifies if the last message is to be resend when a connecition breaks and has been reconnected. May increase reliability, but comes at the risk of message duplication.
- **$ActionSendStreamDriver** <driver basename> just like $DefaultNetstreamDriver, but for the specific action
- **$ActionSendStreamDriverMode** <mode>, default 0, mode to use with the stream driver (driver-specific)

- **$ActionSendStreamDriverAuthMode** <mode>, authentication mode to use with the stream driver. Note that this directive requires TLS netstream drivers. For all others, it will be ignored. (driver-specific)
- **$ActionSendStreamDriverPermittedPeer** <ID>, accepted fingerprint (SHA1) or name of remote peer. Note that this directive requires TLS netstream drivers. For all others, it will be ignored. (driver-specific) - directive may go away!
- **$ActionSendTCPRebindInterval** nbr- [available since 4.5.1] - instructs the TCP send action to close and re-open the connection to the remote host every nbr of messages sent. Zero, the default, means that no such processing is done. This directive is useful for use with load-balancers. Note that there is some performance overhead associated with it, so it is advisable to not too often "rebind" the connection (what "too often" actually means depends on your configuration, a rule of thumb is that it should be not be much more often than once per second).
- **$ActionSendUDPRebindInterval** nbr- [available since 4.3.2] - instructs the UDP send action to rebind the send socket every nbr of messages sent. Zero, the default, means that no rebind is done. This directive is useful for use with load-balancers.

omgssapi-specific Configuration Statements

These statements are specific to omgssapi actions.

$GssForwardServiceName

Type: global configuration directive

Default: host

Provided by: *omgssapi*

Description:

Specifies the service name used by the client when forwarding GSS-API wrapped messages.

The GSS-API service names are constructed by appending '@' and a hostname following "@@" in each selector.

Sample:

```
$GssForwardServiceName rsyslog
```


$GssMode

Type: global configuration directive

Default: encryption

Provided by: *omgssapi*

Description:

Specifies GSS-API mode to use, which can be "**integrity**" - clients are authenticated and messages are checked for integrity, "**encryption**" - same as "integrity", but messages are also encrypted if both sides support it.

Sample:

```
$GssMode Encryption
```

action-queue specific Configuration Statements

The following statements specify parameters for the action queue. To understand queue parameters, read *queues in rsyslog*.

Action queue parameters usually affect the next action and auto-reset to defaults thereafter. Most importantly, this means that when a "real" (non-direct) queue type is defined, this affects the immediately following action, only. The next and all other actions will be in "direct" mode (no real queue) if not explicitely specified otherwise.

- **$ActionQueueCheckpointInterval** <number>
- **$ActionQueueDequeueBatchSize** <number> [default 16]
- **$ActionQueueDequeueSlowdown** <number> [number is timeout in *micro*seconds (1000000us is 1sec!), default 0 (no delay). Simple rate-limiting!]
- **$ActionQueueDiscardMark** <number> [default 9750]
- **$ActionQueueDiscardSeverity** <number> [*numerical* severity! default 4 (warning)]
- **$ActionQueueFileName** <name>
- **$ActionQueueHighWaterMark** <number> [default 8000]
- **$ActionQueueImmediateShutdown** [on/**off**]
- **$ActionQueueSize** <number>
- **$ActionQueueLowWaterMark** <number> [default 2000]
- **$ActionQueueMaxFileSize** <size_nbr>, default 1m
- **$ActionQueueTimeoutActionCompletion** <number> [number is timeout in ms (1000ms is 1sec!), default 1000, 0 means immediate!]
- **$ActionQueueTimeoutEnqueue** <number> [number is timeout in ms (1000ms is 1sec!), default 2000, 0 means discard immediately]
- **$ActionQueueTimeoutShutdown** <number> [number is timeout in ms (1000ms is 1sec!), default 0 (indefinite)]
- **$ActionQueueWorkerTimeoutThreadShutdown** <number> [number is timeout in ms (1000ms is 1sec!), default 60000 (1 minute)]
- **$ActionQueueType** [FixedArray/LinkedList/**Direct**/Disk]
- **$ActionQueueSaveOnShutdown** [on/**off**]
- **$ActionQueueWorkerThreads** <number>, num worker threads, default 1, recommended 1
- $ActionQueueWorkerThreadMinumumMessages <number>, default 100
- **$ActionGSSForwardDefaultTemplate** [templateName] - sets a new default template for GSS-API forwarding action

Ruleset-Specific Legacy Configuration Statements

These statements can be used to set ruleset parameters. To set these parameters, first use *$Ruleset*, **then** use the other configuration directives. Please keep in mind that each ruleset has a *main* queue. To specify parameter for these ruleset (main) queues, use the main queue configuration directives.

rsyslog.conf configuration directive

$RulesetCreateMainQueue

Type: ruleset-specific configuration directive

Parameter Values: boolean (on/off, yes/no)

Available since: 5.3.5+

Default: off

Description:

Rulesets may use their own "main" message queue for message submission. Specifying this directive, **inside a ruleset definition**, turns this on. This is both a performance enhancement and also permits different rulesets (and thus different inputs within the same rsyslogd instance) to use different types of main message queues.

The ruleset queue is created with the parameters that are specified for the main message queue at the time the directive is given. If different queue configurations are desired, different main message queue directives must be used **in front of** the $RulesetCreateMainQueue directive. Note that this directive may only be given once per ruleset. If multiple statements are specified, only the first is used and for the others error messages are emitted.

Note that the final set of ruleset configuration directives specifies the parameters for the default main message queue.

To learn more about this feature, please be sure to read about multi-ruleset support in rsyslog.

Caveats:

The configuration statement "$RulesetCreateMainQueue off" has no effect at all. The capability to specify this is an artifact of the legacy configuration language.

Example:

This example sets up a tcp server with three listeners. Each of these three listener is bound to a specific ruleset. As a performance optimization, the rulesets all receive their own private queue. The result is that received messages can be independently processed. With only a single main message queue, we would have some lock contention between the messages. This does not happen here. Note that in this example, we use different processing. Of course, all messages could also have been processed in the same way ($IncludeConfig may be useful in that case!).

```
$ModLoad imtcp
# at first, this is a copy of the unmodified rsyslog.conf
#define rulesets first
$RuleSet remote10514
$RulesetCreateMainQueue on # create ruleset-specific queue
*.*     /var/log/remote10514

$RuleSet remote10515
$RulesetCreateMainQueue on # create ruleset-specific queue
*.*     /var/log/remote10515

$RuleSet remote10516
$RulesetCreateMainQueue on # create ruleset-specific queue
mail.*        /var/log/mail10516
&       ~
# note that the discard-action will prevent this messag from
# being written to the remote10516 file - as usual...
*.*     /var/log/remote10516

# and now define listeners bound to the relevant ruleset
$InputTCPServerBindRuleset remote10514
```

```
$InputTCPServerRun 10514

$InputTCPServerBindRuleset remote10515
$InputTCPServerRun 10515

$InputTCPServerBindRuleset remote10516
$InputTCPServerRun 10516
```

Note the positions of the directives. With the legacy language, position is very important. It is highly suggested to use the *ruleset()* object in RainerScript config language if you intend to use ruleset queues. The configuration is much more straightforward in that language and less error-prone.

$RulesetParser

Type: ruleset-specific configuration directive

Parameter Values: string

Available since: 5.3.4+

Default: rsyslog.rfc5424 followed by rsyslog.rfc3164

Description:

This directive permits to specify which message parsers should be used for the ruleset in question. It no ruleset is explicitely specified, the default ruleset is used. Message parsers are contained in (loadable) parser modules with the most common cases (RFC3164 and RFC5424) being build-in into rsyslogd.

When this directive is specified the first time for a ruleset, it will not only add the parser to the ruleset's parser chain, it will also wipe out the default parser chain. So if you need to have them in addition to the custom parser, you need to specify those as well.

Order of directives is important. Parsers are tried one after another, in the order they are specified inside the config. As soon as a parser is able to parse the message, it will do so and no other parsers will be executed. If no matching parser can be found, the message will be discarded and a warning message be issued (but only for the first 1,000 instances of this problem, to prevent message generation loops).

Note that the rfc3164 parser will **always** be able to parse a message - it may just not be the format that you like. This has two important implications: 1) always place that parser at the END of the parser list, or the other parsers after it will never be tried and 2) if you would like to make sure no message is lost, placing the rfc3164 parser at the end of the parser list ensures that.

Multiple parser modules are very useful if you have various devices that emit messages that are malformed in various ways. The route to take then is

- make sure you find a custom parser for that device; if there is no one, you may consider writing one yourself (it is not that hard) or getting one written as part of Adiscon's professional services for rsyslog.
- load your custom parsers via $ModLoad
- create a ruleset for each malformed format; assign the custom parser to it
- create a specific listening port for all devices that emit the same malformed format
- bind the listener to the ruleset with the required parser

Note that it may be cumbersome to add all rules to all rulesets. To avoid this, you can either use $Include or omruleset (what probably provides the best solution).

More information about rulesets in general can be found in *multi-ruleset support in rsyslog*.

Caveats:

currently none known

Example:

This example assumes there are two devices emiting malformed messages via UDP. We have two custom parsers for them, named "device1.parser" and "device2.parser". In addition to that, we have a number of other devices sending wellformed messages, also via UDP.

The solution is to listen for data from the two devices on two special ports (10514 and 10515 in this example), create a ruleset for each and assign the custom parsers to them. The rest of the messages are received via port 514 using the regular parsers. Processing shall be equal for all messages. So we simply forward the malformed messages to the regular queue once they are parsed (keep in mind that a message is never again parsed once any parser properly processed it).

```
$ModLoad imudp
$ModLoad pmdevice1 # load parser "device1.parser" for device 1
$ModLoad pmdevice2 # load parser "device2.parser" for device 2

# define ruleset for the first device sending malformed data
$Ruleset maldev1
$RulesetCreateMainQueue on # create ruleset-specific queue
$RulesetParser "device1.parser" # note: this deactivates the default parsers
# forward all messages to default ruleset:
$ActionOmrulesetRulesetName RSYSLOG\_DefaultRuleset
\*.\* :omruleset:

# define ruleset for the second device sending malformed data
$Ruleset maldev2 $RulesetCreateMainQueue on # create ruleset-specific queue
$RulesetParser "device2.parser" # note: this deactivates the default parsers
# forward all messages to default ruleset:
$ActionOmrulesetRulesetName RSYSLOG\_DefaultRuleset
\*.\* :omruleset:

# switch back to default ruleset
$Ruleset RSYSLOG\_DefaultRuleset
\*.\* /path/to/file
auth.info @authlogger.example.net
# whatever else you usually do...

# now define the inputs and bind them to the rulesets
# first the default listener (utilizing the default ruleset)
$UDPServerRun 514

# now the one with the parser for device type 1:
$InputUDPServerBindRuleset maldev1
$UDPServerRun 10514

# and finally the one for device type 2:
$InputUDPServerBindRuleset maldev2
$UDPServerRun 10515
```

For an example of how multiple parser can be chained (and an actual use case), please see the example section on the pmlastmsg parser module.

Note the positions of the directives. With the current config language, **sequence of statements is very important**. This is ugly, but unfortunately the way it currently works.

- **$Ruleset** *name* - starts a new ruleset or switches back to one already defined. All following actions belong to that new rule set. the *name* does not yet exist, it is created. To switch back to rsyslog's default ruleset, specify "RSYSLOG_DefaultRuleset") as the name. All following actions belong to that new rule set. It is advised to also read our paper on *using multiple rule sets in rsyslog*.

rsyslog statistic counter

Rsyslog supports statistic counters via the *impstats* module. It is important to know that impstats and friends only provides an infrastructure where core components and plugins can register statistics counter. This FAQ entry tries to describe all counters available, but please keep in mind that there may exist that we do not know about.

When interpreting rsyslog statistics, please keep in mind that statistics records are processed as regular syslog messages. As such, the statistics messages themselves increment counters when they are emitted via the regular syslog stream, which is the default (and so counters keep slowly increasing even if there is absolutely no other traffic). Also keep in mind that a busy rsyslog system is very dynamic. Most importantly, this means that the counters may not be 100% consistent, but some slight differences may exist. Avoiding such inconsistencies would be possible only at the price of a very tight locking discipline, which would cause serious performance bottlenecks. Thus, this is not done. Finally, though extremely unlikely, some counters may experience an overflow and restart at 0 for that reasons. However, most counters are 64-bit, so this is extremely unlikely. Those which are not 64 bit are typically taken from some internal data structure that uses lower bits for performance reasons and guards against overflow.

The listing starts with the core component or plugin that creates the counters and than specifies various counters that exist for the sub-entities. The listing below is extended as new counters are added. Some counters probably do not exist in older releases of rsyslog.

Queue

For each queue inside the system its own set of statistics counters is created. If there are multiple action (or main) queues, this can become a rather lengthy list. The stats record begins with the queue name (e.g. "main Q" for the main queue; ruleset queues have the name of the ruleset they are associated to, action queues the name of the action).

- **size** - currently active messages in queue
- **enqueued** - total number of messages enqueued into this queue since startup
- **maxsize** - maximum number of active messages the queue ever held
- **full** - number of times the queue was actually full and could not accept additional messages
- **discarded.full** - number of messages discarded because the queue was full
- **discarded.nf** - number of messages discarded because the queue was nearly full. Starting at this point, messages of lower-than-configured severity are discarded to save space for higher severity ones.

Actions

- **processed** - total number of messages processed by this action. This includes those messages that failed actual execution (so it is a total count of messages ever seen, but not necessarily successfully processed)
- **failed** - total number of messages that failed during processing. These are actually lost if they have not been processed by some other action. Most importantly in a failover chain the messages are flagged as "failed" in the failing actions even though they are forwarded to the failover action (the failover action's "processed" count should equal to failing actions "fail" count in this scenario)a

- **suspended** - (7.5.8+) – total number of times this action suspended itself. Note that this counts the number of times the action transitioned from active to suspended state. The counter is no indication of how long the action was suspended or how often it was retried. This is intentional, as the counter as it currently is permits to tell how often the action ran into a failure condition.
- **suspended.duration** - (7.5.8+) – the total number of seconds this action was disabled. Note that the second count is incremented as soon as the action is suspended for another interval. As such, the time may be higher than it should be at the reporting point of time. If the pstats interval is shorter than the suspension timeout, the same suspended.duration value may be reported for successive pstats outputs. For a long-running system, this is considered a minimal difference. In general, note that this setting is not totally accurate, especially when running with multiple worker threads. In rsyslog v8, this is the total suspended time for all worker instances of this action.
- **resumed** - (7.5.8+) – total number of times this action resumed itself. A resumption occurs after the action has detected that a failure condition does no longer exist.

Plugins

imuxsock

imudp

imtcp

imptcp

imrelp

impstats

omfile

omelasticsearch

Modules

Rsyslog has a modular design. This enables functionality to be dynamically loaded from modules, which may also be written by any third party. Rsyslog itself offers all non-core functionality as modules. Consequently, there is a growing number of modules. Here is the entry point to their documentation and what they do (list is currently not complete)

Please note that each module provides configuration directives, which are NOT necessarily being listed below. Also remember, that a modules configuration directive (and functionality) is only available if it has been loaded (using $ModLoad).

It is relatively easy to write a rsyslog module. If none of the provided modules solve your need, you may consider writing one or have one written for you by Adiscon's professional services for rsyslog (this often is a very cost-effective and efficient way of getting what you need).

There exist different classes of loadable modules:

Output Modules

Output modules process messages. With them, message formats can be transformed and messages be transmitted to various different targets. They are generally defined via *action* configuration objects.

omamqp1: AMQP 1.0 Messaging Output Module

Module Name:	omamqp1
Available Since:	8.17.0
Original Author:	Ken Giusti <kgiusti@gmail.com>

Description

This module provides the ability to send logging via an AMQP 1.0 compliant message bus. It puts the log messages into an AMQP message and sends the message to a destination on the bus.

Dependencies

- libqpid-proton

Configure

```
./configure --enable-omamqp1
```

Message Format

Messages sent from this module to the message bus contain an AMQP List in the message body. This list contains one or more log messages as AMQP String types. Each string entry is a single log message. The list is ordered such that the oldest log appears at the front of the list (e.g. list index 0), whilst the most recent log is at the end of the list.

Interoperability

The output plugin has been tested against the following messaging systems:

- QPID C++ Message Broker
- QPID Dispatch Message Router

Action Parameters

- **host** *(Required)* The address of the message bus in *host[:port]* format. The port defaults to 5672 if absent. Examples: *"localhost"*, *"127.0.0.1:9999"*, *"bus.someplace.org"*
- **target** *(Required)* The destination for the generated messages. This can be the name of a queue or topic. On some messages buses it may be necessary to create this target manually. Example: *"amq.topic"*
- **username** *(Optional)* Used by SASL to authenticate with the message bus.
- **password** *(Optional)* Used by SASL to authenticate with the message bus.
- **template** *(Optional)* Format for the log messages. Default: *RSYSLOG_FileFormat*
- **idleTimeout** *(Optional)* The idle timeout in seconds. This enables connection heartbeats and is used to detect a failed connection to the message bus. Set to zero to disable. Default: *0*
- **reconnectDelay** *(Optional)* The time in seconds this module will delay before attempting to re-established a failed connection to the message bus. Default: *5*
- **maxRetries** *(Optional)* The number of times an undeliverable message is re-sent to the message bus before it is dropped. This is unrelated to rsyslog's action.resumeRetryCount. Once the connection to the message bus is active this module is ready to receive log messages from rsyslog (i.e. the module has 'resumed'). Even though the connection is active, any particular message may be rejected by the message bus (e.g. 'unrouteable'). The module will retry (e.g. 'suspend') for up to *maxRetries* attempts before discarding the message as undeliverable. Setting this to zero disables the limit and unrouteable messages will be retried as long as the connection stays up. You probably do not want that to happen. Default: *10*
- **disableSASL** *(Optional)* Setting this to a non-zero value will disable SASL negotiation. Only necessary if the message bus does not offer SASL support.

TODO

- Add support for SSL connections.

Examples

This example shows a minimal configuration. The module will attempt to connect to a QPID broker at *broker.amqp.org*. Messages are sent to the *amq.topic* topic, which exists on the broker by default:

```
module(load="omamqp1")
action(type="omamqp1"
       host="broker.amqp.org"
       target="amq.topic")
```

This example forces rsyslogd to authenticate with the message bus. The message bus must be provisioned such that user *joe* is allowed to send to the message bus. All messages are sent to *log-queue*. It is assumed that *log-queue* has already been provisioned:

```
module(load="omamqp1")

action(type="omamqp1"
       host="bus.amqp.org"
       target="log-queue"
       username="joe"
       password="trustno1")
```

Notes on use with the QPID C++ broker (qpidd)

Note well: These notes assume use of version 0.34 of the QPID C++ broker. Previous versions may not be fully compatible.

To use the Apache QPID C++ broker **qpidd** as the message bus, a version of qpidd that supports the AMQP 1.0 protocol must be used.

Since qpidd can be packaged without AMQP 1.0 support you should verify AMQP 1.0 has been enabled by checking for AMQP 1.0 related options in the qpidd help text. For example:

```
qpidd --help

...

AMQP 1.0 Options:
   --domain DOMAIN           Domain of this broker
   --queue-patterns PATTERN  Pattern for on-demand queues
   --topic-patterns PATTERN  Pattern for on-demand topics
```

If no AMQP 1.0 related options appear in the help output then your instance of qpidd does not support AMQP 1.0 and cannot be used with this output module.

The destination for message (target) *must* be created before log messages arrive. This can be done using the qpid-config tool.

Example:

```
qpid-config add queue rsyslogd
```

Alternatively the target can be created on demand by configuring a queue-pattern (or topic-pattern) that matches the target. To do this, add a *queue-patterns* or *topic_patterns* configuration directive to the qpidd configuration file /etc/qpid/qpidd.conf.

For example to have qpidd automatically create a queue named *rsyslogd* add the following to the qpidd configuration file:

```
queue-patterns=rsyslogd
```

or, if a topic behavior is desired instead of a queue:

```
topic-patterns=rsyslogd
```

These dynamic targets are auto-delete and will be destroyed once there are no longer any subscribers or queue-bound messages.

Versions of qpidd <= 0.34 also need to have the SASL service name set to *"amqp"* if SASL authentication is used. Add this to the qpidd.conf file:

```
sasl-service-name=amqp
```

Notes on use with the QPID Dispatch Router (qdrouterd)

Note well: These notes assume use of version 0.5 of the QPID Dispatch Router **qdrouterd**. Previous versions may not be fully compatible.

The default qdrouterd configuration does not have SASL authentication turned on. If SASL authentication is required you must configure SASL in the qdrouter configuration file /etc/qpid-dispatch/qdrouterd.conf

First create a SASL configuration file for qdrouterd. This configuration file is usually /etc/sasl2/qdrouterd.conf, but its default location may vary depending on your platform's configuration.

This document assumes you understand how to properly configure Cyrus SASL.

Here is an example qdrouterd SASL configuration file that allows the client to use either the **DIGEST-MD5** or **PLAIN** authentication mechanisims and specifies the path to the SASL user credentials database:

```
pwcheck_method: auxprop
auxprop_plugin: sasldb
sasldb_path: /var/lib/qdrouterd/qdrouterd.sasldb
mech_list: DIGEST-MD5 PLAIN
```

Once a SASL configuration file has been set up for qdrouterd the path to the directory holding the configuration file and the name of the configuration file itself **without the '.conf' suffix** must be added to the /etc/qpid-dispatch/qdrouterd.conf configuration file. This is done by adding *saslConfigPath* and *saslConfigName* to the *container* section of the configuration file. For example, assuming the file /etc/sasl2/qdrouterd.conf holds the qdrouterd SASL configuration:

```
container {
   workerThreads: 4
   containerName: Qpid.Dispatch.Router.A
   saslConfigPath: /etc/sasl2
   saslConfigName: qdrouterd
}
```

In addition the address used by the omamqp1 module to connect to qdrouterd must have SASL authentication turned on. This is done by adding the *authenticatePeer* attribute set to 'yes' to the corresponding *listener* entry:

```
listener {
   addr: 0.0.0.0
   port: amqp
```

```
   authenticatePeer: yes
}
```

This should complete the SASL setup needed by qdrouterd.

The target address used as the destination for the log messages must be picked with care. qdrouterd uses the prefix of the target address to determine the forwarding pattern used for messages sent to that target address. Addresses starting with the prefix *queue* are distributed to only one message receiver. If there are multiple message consumers listening to that target address only one listener will receive the message - mimicking the behavior of a queue with competing subscribers. For example: *queue/rsyslogd*

If a multicast pattern is desired - where all active listeners receive their own copy of the message - the target address prefix *multicast* may be used. For example: *multicast/rsyslogd*

Note well: if there are no active receivers for the log messages the messages will be rejected by qdrouterd since the messages are undeliverable. In this case the omamqp1 module will return a **SUSPENDED** status to the rsyslogd main task. rsyslogd may then re-submit the rejected log messages to the module which will attempt to send them again. This retry option is configured via rsyslogd - it is not part of this module. Refer to the rsyslogd actions documentation.

Using qdrouterd in combination with qpidd

A qdrouterd-based message bus can use a broker as a message storage mechanism for those that require broker-based message services (such as a message store). This section explains how to configure qdrouterd and qpidd for this type of deployment. Please read the above notes for deploying qpidd and qdrouterd first.

Each qdrouterd instance that is to connect the broker to the message bus must define a *connector* section in the qdrouterd.conf file. This connector contains the addressing information necessary to have the message bus set up a connection to the broker. For example, if a broker is available on host broker.host.com at port 5672:

```
connector {
   name: mybroker
   role: on-demand
   addr: broker.host.com
   port: 5672
}
```

In order to route messages to and from the broker, a static *link route* must be configured on qdrouterd. This link route contains a target address prefix and the name of the connector to use for forwarding matching messages.

For example, to have qdrouterd forward messages that have a target address prefixed by “Broker” to the connector defined above, the following link pattern must be added to the qdrouterd.conf configuration:

```
linkRoutePattern {
   prefix: /Broker/
   connector: mybroker
}
```

A queue must then be created on the broker. The name of the queue must be prefixed by the same prefix specified in the linkRoutePattern entry. For example:

```
$ qpid-config add queue Broker/rsyslogd
```

Lastly use the name of the queue for the target address for the omamqp module action. For example, assuming qdrouterd is listening on local port 5672:

```
action(type="omamqp1"
       host="localhost:5672"
       target="Broker/rsyslogd")
```

This documentation is part of the rsyslog project.

omelasticsearch: Elasticsearch Output Module

Module Name: **omelasticsearch**

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available since:6.4.0+

Description:

This module provides native support for logging to Elasticsearch.

Action Parameters

- **server** [`http://` | `https://` |] *<hostname | ip>* [`:` *<port>*] An array of Elasticsearch servers in the specified format. If no scheme is specified, it will be chosen according to *usehttps*. If no port is specified, *serverport* will be used. Defaults to "localhost".

  Requests to Elasticsearch will be load-balanced between all servers in round-robin fashion.

```
Examples:
     server="localhost:9200"
     server=["elasticsearch1", "elasticsearch2"]
```

- **serverport** Default HTTP port to use to connect to Elasticsearch if none is specified on a *server*. Defaults to 9200
- **healthchecktimeout** Specifies the number of milliseconds to wait for a successful health check on a *server*. Before trying to submit events to Elasticsearch, rsyslog will execute an *HTTP HEAD* to `/_cat/health` and expect an *HTTP OK* within this timeframe. Defaults to 3500.

  Note, the health check is verifying connectivity only, not the state of the Elasticsearch cluster.
- **searchIndex** Elasticsearch index to send your logs to. Defaults to "system"
- **dynSearchIndex**<on/**off**> Whether the string provided for *searchIndex* should be taken as a rsyslog template. Defaults to "off", which means the index name will be taken literally. Otherwise, it will look for a template with that name, and the resulting string will be the index name. For example, let's assume you define a template named "date-days" containing "%timereported:1:10:date-rfc3339%". Then, with dynSearchIndex="on", if you say searchIndex="date-days", each log will be sent to and index named after the first 10 characters of the timestamp, like "2013-03-22".
- **searchType** Elasticsearch type to send your index to. Defaults to "events"
- **dynSearchType** <on/**off**> Like *dynSearchIndex*, it allows you to specify a rsyslog template for *searchType*, instead of a static string.
- **asyncrepl**<on/**off**> No longer supported as ElasticSearch no longer supports it.

- **usehttps**<on/**off**> Default scheme to use when sending events to Elasticsearch if none is specified on a *server*. Good for when you have Elasticsearch behind Apache or something else that can add HTTPS. Note that if you have a self-signed certificate, you'd need to install it first. This is done by copying the certificate to a trusted path and then running *update-ca-certificates*. That trusted path is typically */usr/local/share/ca-certificates* but check the man page of *update-ca-certificates* for the default path of your distro
- **timeout** How long Elasticsearch will wait for a primary shard to be available for indexing your log before sending back an error. Defaults to "1m".
- **template** This is the JSON document that will be indexed in Elasticsearch. The resulting string needs to be a valid JSON, otherwise Elasticsearch will return an error. Defaults to:

```
$template JSONDefault, "{\"message\":\"%msg:::json%\",\"fromhost\":\"%HOSTNAME:::json
↪%\",\"facility\":\"%syslogfacility-text%\",\"priority\":\"%syslogpriority-text%\",\
↪"timereported\":\"%timereported:::date-rfc3339%\",\"timegenerated\":\"
↪%timegenerated:::date-rfc3339%\"}"
```

Which will produce this sort of documents (pretty-printed here for readability):

```
{
    "message": " this is a test message",
    "fromhost": "test-host",
    "facility": "user",
    "priority": "info",
    "timereported": "2013-03-12T18:05:01.344864+02:00",
    "timegenerated": "2013-03-12T18:05:01.344864+02:00"
}
```

- **bulkmode**<on/**off**> The default "off" setting means logs are shipped one by one. Each in its own HTTP request, using the Index API. Set it to "on" and it will use Elasticsearch's Bulk API to send multiple logs in the same request. The maximum number of logs sent in a single bulk request depends on your *maxbytes* and queue settings - usually limited by the dequeue batch size. More information about queues can be found here.
- **maxbytes** *(since v8.23.0)* When shipping logs with *bulkmode* **on**, maxbytes specifies the maximum size of the request body sent to Elasticsearch. Logs are batched until either the buffer reaches maxbytes or the the dequeue batch size is reached. In order to ensure Elasticsearch does not reject requests due to content length, verify this value is set accoring to the http.max_content_length setting in Elasticsearch. Defaults to 100m.
- **parent** Specifying a string here will index your logs with that string the parent ID of those logs. Please note that you need to define the parent field in your mapping for that to work. By default, logs are indexed without a parent.
- **dynParent**<on/**off**> Using the same parent for all the logs sent in the same action is quite unlikely. So you'd probably want to turn this "on" and specify a rsyslog template that will provide meaningful parent IDs for your logs.
- **uid** If you have basic HTTP authentication deployed (eg through the elasticsearch-basic plugin), you can specify your user-name here.
- **pwd** Password for basic authentication.
- **errorfile** <filename> (optional)

  If specified, records failed in bulk mode are written to this file, including their error cause. Rsyslog itself does not process the file any more, but the idea behind that mechanism is that the user can create a script to periodically inspect the error file and react appropriately. As the complete request is included, it is possible to simply resubmit messages from that script.

  Please note: when rsyslog has problems connecting to elasticsearch, a general error is assumed and the submit is retried. However, if we receive negative responses during batch processing, we assume an error in the data

itself (like a mandatory field is not filled in, a format error or something along those lines). Such errors cannot be solved by simpy resubmitting the record. As such, they are written to the error file so that the user (script) can examine them and act appropriately. Note that e.g. after search index reconfiguration (e.g. dropping the mandatory attribute) a resubmit may be succesful.

Statistic Counter

This plugin maintains global *statistics*, which accumulate all action instances. The statistic is named "omelasticsearch". Parameters are:

- **submitted** - number of messages submitted for processing (with both success and error result)
- **fail.httprequests** - the number of times a http request failed. Note that a single http request may be used to submit multiple messages, so this number may be (much) lower than fail.http.
- **fail.http** - number of message failures due to connection like-problems (things like remote server down, broken link etc)
- **fail.es** - number of failures due to elasticsearch error reply; Note that this counter does NOT count the number of failed messages but the number of times a failure occured (a potentially much smaller number). Counting messages would be quite performance-intense and is thus not done.

The fail.httprequests and fail.http counters reflect only failures that omelasticsearch detected. Once it detects problems, it (usually, depends on circumstances) tell the rsyslog core that it wants to be suspended until the situation clears (this is a requirement for rsyslog output modules). Once it is suspended, it does NOT receive any further messages. Depending on the user configuration, messages will be lost during this period. Those lost messages will NOT be counted by impstats (as it does not see them).

Note that some previous (pre 7.4.5) versions of this plugin had different counters. These were experimental and confusing. The only ones really used were "submits", which were the number of successfully processed messages and "connfail" which were equivalent to "failed.http".

Samples

The following sample does the following:

- loads the omelasticsearch module
- outputs all logs to Elasticsearch using the default settings

```
module(load="omelasticsearch")
*.*     action(type="omelasticsearch")
```

The following sample does the following:

- loads the omelasticsearch module
- defines a template that will make the JSON contain the following properties
  - RFC-3339 timestamp when the event was generated
  - the message part of the event
  - hostname of the system that generated the message
  - severity of the event, as a string
  - facility, as a string
  - the tag of the event

- outputs to Elasticsearch with the following settings
  - host name of the server is myserver.local
  - port is 9200
  - JSON docs will look as defined in the template above
  - index will be "test-index"
  - type will be "test-type"
  - activate bulk mode. For that to work effectively, we use an in-memory queue that can hold up to 5000 events. The maximum bulk size will be 300
  - retry indefinitely if the HTTP request failed (eg: if the target server is down)

```
module(load="omelasticsearch")
template(name="testTemplate"
         type="list"
         option.json="on") {
           constant(value="{")
             constant(value="\"timestamp\":\"")       property(name="timereported"␣
↪dateFormat="rfc3339")
             constant(value="\",\"message\":\"")      property(name="msg")
             constant(value="\",\"host\":\"")         property(name="hostname")
             constant(value="\",\"severity\":\"")     property(name="syslogseverity-
↪text")
             constant(value="\",\"facility\":\"")     property(name="syslogfacility-
↪text")
             constant(value="\",\"syslogtag\":\"")    property(name="syslogtag")
           constant(value="\"}")
         }
action(type="omelasticsearch"
       server="myserver.local"
       serverport="9200"
       template="testTemplate"
       searchIndex="test-index"
       searchType="test-type"
       bulkmode="on"
       maxbytes="100m"
       queue.type="linkedlist"
       queue.size="5000"
       queue.dequeuebatchsize="300"
       action.resumeretrycount="-1")
```


omfile: File Output Module

Module Name:	omfile
Author:	Rainer Gerhards <rgerhards@adiscon.com>

The omfile plug-in provides the core functionality of writing messages to files residing inside the local file system (which may actually be remote if methods like NFS are used). Both files named with static names as well files with names based on message content are supported by this module.

Configuration Parameters

Omfile is a built-in module that does not need to be loaded. In order to specify module parameters, use

```
module(load="builtin:omfile" ...parameters...)
```

Note that legacy directives **do not** affect new-style RainerScript configuration objects. See *basic configuration structure doc* to learn about different configuration languages in use by rsyslog.

General Notes

As can be seen in the parameters below, owner and groups can be set either by name or by direct id (uid, gid). While using a name is more convenient, using the id is more robust. There may be some situations where the OS is not able to do the name-to-id resolution, and these cases the owner information will be set to the process default. This seems to be uncommon and depends on the authentication provider and service start order. In general, using names is fine.

Module Parameters

template [templateName]
Default: RSYSLOG_FileFormat

Set the default template to be used if an action is not configured to use a specific template.

dirCreateMode [octalNumber]
Default: 0700

Sets the default dirCreateMode to be used for an action if no explicit one is specified.

fileCreateMode [default 0644] [octalNumber]
Default: 0644

Sets the default fileCreateMode to be used for an action if no explicit one is specified.

fileOwner [userName]
Default: process user

Sets the default fileOwner to be used for an action if no explicit one is specified.

fileOwnerNum [uid]
Default: process user

Sets the default fileOwnerNum to be used for an action if no explicit one is specified.

fileGroup [groupName]
Default: process user's primary group

Sets the default fileGroup to be used for an action if no explicit one is specified.

fileGroupNum [gid]
Default: process user's primary group

Sets the default fileGroupNum to be used for an action if no explicit one is specified.

dirOwner [userName]
Default: process user

Sets the default dirOwner to be used for an action if no explicit one is specified.

dirOwnerNum [uid]
Default: process user

Sets the default dirOwnerNum to be used for an action if no explicit one is specified.

dirGroup [groupName]
Default: process user's primary group

Sets the default dirGroup to be used for an action if no explicit one is specified.

dirGroupNum [gid]
Default: process user's primary group

Sets the default dirGroupNum to be used for an action if no explicit one is specified.

Action Parameters

Note that **one** of the parameters *file* or *dynaFile* must be specified. This selects whether a static or dynamic file (name) shall be written to.

file [fileName]
Default: none

This creates a static file output, always writing into the same file. If the file already exists, new data is appended to it. Existing data is not truncated. If the file does not already exist, it is created. Files are kept open as long as rsyslogd is active. This conflicts with external log file rotation. In order to close a file after rotation, send rsyslogd a HUP signal after the file has been rotated away. Either file or dynaFile can be used, but not both. If both are given, dynaFile will be used.

dynaFile [templateName]
Default: none

For each message, the file name is generated based on the given template. Then, this file is opened. As with the *file* property, data is appended if the file already exists. If the file does not exist, a new file is created. The template given in "templateName" is just a regular *rsyslog template*, so all you have full control over how to format the file name. Either file or dynaFile can be used, but not both. If both are given, dynaFile will be used.

A cache of recent files is kept. Note that this cache can consume quite some memory (especially if large buffer sizes are used). Files are kept open as long as they stay inside the cache. Files are removed from the cache when a HUP signal is sent, the *closeTimeout* occurs, or the cache runs out of space, in which case the least recently used entry is evicted.

template [templateName]
Default: template set via "template" module parameter

Sets the template to be used for this action.

closeTimeout [minutes]
Default: for static files: 0; for dynamic files: 10

Available since: 8.3.3

Specifies after how many minutes of inactivity a file is automatically closed. Note that this functionality is implemented based on the *janitor process*. See its doc to understand why and how janitor-based times are approximate.

dynaFileCacheSize [size]
Default: 10

Applies only if dynamic filenames are used. Specifies the number of DynaFiles that will be kept open. Note that this is a per-action value, so if you have multiple dynafile actions, each of them have their individual caches

(which means the numbers sum up). Ideally, the cache size exactly matches the need. You can use *impstats* to tune this value. Note that a too-low cache size can be a very considerable performance bottleneck.

zipLevel [level]
Default: 0

if greater 0, turns on gzip compression of the output file. The higher the number, the better the compression, but also the more CPU is required for zipping.

veryRobustZip [switch]
Default: off

Available since: 7.3.0

if *zipLevel* is greater 0, then this setting controls if extra headers are written to make the resulting file extra hardened against malfunction. If set to off, data appended to previously unclean closed files may not be accessible without extra tools. Note that this risk is usually expected to be bearable, and thus "off" is the default mode. The extra headers considerably degrade compression, files with this option set to "on" may be four to five times as large as files processed in "off" mode.

flushInterval [interval]
Default: 1

Defines, in seconds, the interval after which unwritten data is flushed.

asyncWriting [switch]
Default: off

if turned on, the files will be written in asynchronous mode via a separate thread. In that case, double buffers will be used so that one buffer can be filled while the other buffer is being written. Note that in order to enable FlushInterval, AsyncWriting must be set to "on". Otherwise, the flush interval will be ignored.

flushOnTXEnd [switch]
Default: on

Omfile has the capability to write output using a buffered writer. Disk writes are only done when the buffer is full. So if an error happens during that write, data is potentially lost. Bear in mind that the buffer may become full only after several hours or a rsyslog shutdown (however a buffer flush can still be forced by sending rsyslogd a HUP signal). In cases where this is unacceptable, set FlushOnTXEnd to "on". Then, data is written at the end of each transaction (for pre-v5 this means after each log message) and the usual error recovery thus can handle write errors without data loss. Note that this option severely reduces the effect of zip compression and should be switched to "off" for that use case. Also note that the default -on- is primarily an aid to preserve the traditional syslogd behaviour.

ioBufferSize [size]
Default: 4 KiB

size of the buffer used to writing output data. The larger the buffer, the potentially better performance is. The default of 4k is quite conservative, it is useful to go up to 64k, and 128K if you used gzip compression (then, even higher sizes may make sense)

dirOwner [userName]
Default: system default

Set the file owner for directories newly created. Please note that this setting does not affect the owner of directories already existing. The parameter is a user name, for which the userid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

dirOwnerNum [uid]
Default: system default

Available since: 7.5.8, 8.1.4

Set the file owner for directories newly created. Please note that this setting does not affect the owner of directories already existing. The parameter is a numerical ID, which is used regardless of whether the user actually exists. This can be useful if the user mapping is not available to rsyslog during startup.

dirGroup [groupName]
Default: system default

Set the group for directories newly created. Please note that this setting does not affect the group of directories already existing. The parameter is a group name, for which the groupid is obtained by rsyslogd on during startup processing. Interim changes to the user mapping are not detected.

dirGroupNum [gid]
Default: system default

Set the group for directories newly created. Please note that this setting does not affect the group of directories already existing. The parameter is a numerical ID, which is used regardless of whether the group actually exists. This can be useful if the group mapping is not available to rsyslog during startup.

fileOwner [userName]
Default: system default

Set the file owner for files newly created. Please note that this setting does not affect the owner of files already existing. The parameter is a user name, for which the userid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are *not* detected.

fileOwnerNum [uid]
Default: system default

Available since: 7.5.8, 8.1.4

Set the file owner for files newly created. Please note that this setting does not affect the owner of files already existing. The parameter is a numerical ID, which which is used regardless of whether the user actually exists. This can be useful if the user mapping is not available to rsyslog during startup.

fileGroup [groupName]
Default: system default

Set the group for files newly created. Please note that this setting does not affect the group of files already existing. The parameter is a group name, for which the groupid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

fileGroupNum [gid]
Default: system default

Available since: 7.5.8, 8.1.4

Set the group for files newly created. Please note that this setting does not affect the group of files already existing. The parameter is a numerical ID, which is used regardless of whether the group actually exists. This can be useful if the group mapping is not available to rsyslog during startup.

fileCreateMode [octalNumber]
Default: equally-named module parameter

The FileCreateMode directive allows to specify the creation mode with which rsyslogd creates new files. If not specified, the value 0644 is used (which retains backward-compatibility with earlier releases). The value given must always be a 4-digit octal number, with the initial digit being zero. Please note that the actual permission depend on rsyslogd's process umask. If in doubt, use "$umask 0000" right at the beginning of the configuration file to remove any restrictions.

dirCreateMode [octalNumber]
Default: equally-named module parameter

This is the same as FileCreateMode, but for directories automatically generated.

failOnChOwnFailure [switch]
Default: on

This option modifies behaviour of file creation. If different owners or groups are specified for new files or directories and rsyslogd fails to set these new owners or groups, it will log an error and NOT write to the file in question if that option is set to "on". If it is set to "off", the error will be ignored and processing continues. Keep in mind, that the files in this case may be (in)accessible by people who should not have permission. The default is "on".

createDirs [switch]
Default: on

create directories on an as-needed basis

sync [switch]
Default: off

enables file syncing capability of omfile.

When enabled, rsyslog does a sync to the data file as well as the directory it resides after processing each batch. There currently is no way to sync only after each n-th batch.

Enabling sync causes a severe performance hit. Actually, it slows omfile so much down, that the probability of loosing messages **increases**. In short, you should enable syncing only if you know exactly what you do, and fully understand how the rest of the engine works, and have tuned the rest of the engine to lossless operations.

sig.provider [providerName]
Default: no signature provider

Selects a signature provider for log signing. By selecting a provider, the signature feature is turned on.

Currently there is one signature provider available: "*ksi_ls12*".

Previous signature providers "*gt*" and "*ksi*" are depricated.

cry.provider [providerName]
Default: no crypto provider

Selects a crypto provider for log encryption. By selecting a provider, the encryption feature is turned on.

Currently, there only is one provider called "*gcry*".

Statistic Counter

This plugin maintains *statistics* for each dynafile cache. Dynafile cache performance is critical for overall system performance, so reviewing these counters on a busy system (especially one experiencing performance problems) is advisable. The statistic is named "dynafile cache", followed by the template name used for this dynafile action.

The following properties are maintained for each dynafile:

- **request** - total number of requests made to obtain a dynafile
- **level0** - requests for the current active file, so no real cache lookup needed to be done. These are extremely good.
- **missed** - cache misses, where the required file did not reside in cache. Even with a perfect cache, there will be at least one miss per file. That happens when the file is being accessed for the first time and brought into cache. So "missed" will always be at least as large as the number of different files processed.
- **evicted** - the number of times a file needed to be evicted from the cache as it ran out of space. These can simply happen when date-based files are used, and the previous date files are being removed from the cache as time progresses. It is better, though, to set an appropriate "closeTimeout" (counter described below), so that files are

removed from the cache after they become no longer accessed. It is bad if active files need to be evicted from the cache. This is a very costly operation as an evict requires to close the file (thus a full flush, no matter of its buffer state) and a later access requires a re-open – and the eviction of another file, as the cache obviously has run out of free entries. If this happens frequently, it can severely affect performance. So a high eviction rate is a sign that the dynafile cache size should be increased. If it is already very high, it is recommended to re-think about the design of the file store, at least if the eviction process causes real performance problems.

- **maxused** - the maximum number of cache entries ever used. This can be used to trim the cache down to a value that's actually useful but does not waste resources. Note that when date-based files are used and rsyslog is run for an extended period of time, the cache gradually fills up to the max configured value as older files are migrated out of it. This will make "maxused" questionable after some time. Frequently enough purging the cache can prevent this (usually, once a day is sufficient).
- **closetimeouts** - available since 8.3.3 – tells how often a file was closed due to timeout settings ("closeTimeout" action parameter). These are cases where dynafiles or static files have been closed by rsyslog due to inactivity. Note that if no "closeTimeout" is specified for the action, this counter always is zero. A high or low number in itself doesn't mean anything good or bad. It totally depends on the use case, so no general advise can be given.

Caveats/Known Bugs

- people often report problems that dynafiles are not properly created. The common cause for this problem is SELinux rules, which do not permit the create of those files (check generated file names and pathes!). The same happens for generic permission issues (this is often a problem under Ubuntu where permissions are dropped by default)
- One needs to be careful with log rotation if signatures and/or encryption are being used. These create side-files, which form a set and must be kept together. For signatures, the ".sigstate" file must NOT be rotated away if signature chains are to be build across multiple files. This is because .sigstate contains just global information for the whole file set. However, all other files need to be rotated together. The proper sequence is to

  1. move all files inside the file set
  2. only AFTER this is completely done, HUP rsyslog

  This sequence will ensure that all files inside the set are atomically closed and in sync. HUPing only after a subset of files have been moved results in inconsistencies and will most probably render the file set unusable.

Example

The following command writes all syslog messages into a file.

```
action(type="omfile" dirCreateMode="0700" FileCreateMode="0644"
       File="/var/log/messages")
```

Legacy Configuration Directives

Note that the legacy configuration parameters do **not** affect new-style action definitions via the action() object. This is by design. To set default for action() objects, use module parameters in the

```
module(load="builtin:omfile" ...)
```

object.

Read about *the importance of order in legacy configuration* to understand how to use these configuration directives. **Legacy directives should NOT be used when writing new configuration files.**

- **$DynaFileCacheSize** equivalent to the "dynaFileCacheSize" parameter
- **$OMFileZipLevel** equivalent to the "zipLevel" parameter
- **$OMFileFlushInterval** equivalent to the "flushInterval" parameter
- **$OMFileASyncWriting** equivalent to the "asyncWriting" parameter
- **$OMFileFlushOnTXEnd** equivalent to the "flushOnTXEnd" parameter
- **$OMFileIOBufferSize** equivalent to the "IOBufferSize" parameter
- **$DirOwner** equivalent to the "dirOwner" parameter
- **$DirGroup** equivalent to the "dirGroup" parameter
- **$FileOwner** equivalent to the "fileOwner" parameter
- **$FileGroup** equivalent to the "fileGroup" parameter
- **$DirCreateMode** equivalent to the "dirCreateMode" parameter
- **$FileCreateMode** equivalent to the "fileCreateMode" parameter
- **$FailOnCHOwnFailure** equivalent to the "failOnChOwnFailure" parameter
- **FOMFileForceCHOwn** equivalent to the "ForceChOwn" parameter
- **$CreateDirs** equivalent to the "createDirs" parameter
- **$ActionFileEnableSync** equivalent to the "enableSync" parameter
- **$ActionFileDefaultTemplate** equivalent to the "template" module parameter
- **$ResetConfigVariables** Resets all configuration variables to their default value.

Legacy Sample

The following command writes all syslog messages into a file.

```
$DirCreateMode 0700
$FileCreateMode 0644
*.* /var/log/messages
```


omfwd: syslog Forwarding Output Module

Module Name: omfwd

Author: Rainer Gerhards <rgerhards@adiscon.com>

The omfwd plug-in provides the core functionality of traditional message forwarding via UDP and plain TCP. It is a built-in module that does not need to be loaded.

Note: this documentation describes features present in v7+ of rsyslog. If you use an older version, scroll down to "legacy parameters". If you prefer, you can also obtain a specific version of the rsyslog documentation.

Module Parameters

- **Template** [templateName]

  sets a non-standard default template for this module.

Action Parameters

- **Target** string

  Name or IP-Address of the system that shall receive messages. Any resolvable name is fine.

- **Port**

  Name or numerical value of port to use when connecting to target.

- **Protocol** udp/tcp [default udp]

  Type of protocol to use for forwarding. Note that ‘‘tcp” means both legacy plain tcp syslog as well as RFC5425-based TCL-encrypted syslog. Which one is selected depends on the protocol drivers set before the action commend. Note that as of 6.3.6, there is no way to specify this within the action itself.

- **NetworkNamespace** [default none]

  Name of a network namespace as in /var/run/netns/ to use for forwarding.

  If the setns() system call is not available on the system (e.g. BSD kernel, linux kernel before v2.6.24) the given namespace will be ignored.

- **Device** [default none]

  Bind socket to given device (e.g., eth0)

  For Linux with VRF support, the Device option can be used to specify the VRF for the Target address.

- **TCP_Framing** “traditional” or “octet-counted” [default traditional]

  Framing-Mode to be for forwarding. This affects only TCP-based protocols. It is ignored for UDP. In protocol engineering, “framing” means how multiple messages over the same connection are separated. Usually, this is transparent to users. Unfortunately, the early syslog protocol evolved, and so there are cases where users need to specify the framing. The traditional framing is nontransparent. With it, messages are end when a LF (aka “line break”, “return”) is encountered, and the next message starts immediately after the LF. If multi-line messages are received, these are essentially broken up into multiple message, usually with all but the first message segment being incorrectly formatted. The octet-counting framing solves this issue. With it, each message is prefixed with the actual message length, so that a receivers knows exactly where the message ends. Multi-line messages cause no problem here. This mode is very close to the method described in RFC5425 for TLS-enabled syslog. Unfortunately, only few syslogd implementations support octet-counted framing. As such, the traditional framing is set as default, even though it has defects. If it is known that the receiver supports octet-counted framing, it is suggested to use that framing mode.

- **TCP_FrameDelimiter** [default 10]

  Sets a custom frame delimiter for TCP transmission when running TCP_Framing in “traditional” mode. The delimiter has to be a number between 0 and 255 (representing the ASCII-code of said character). The default value for this parameter is 10, representing a ‘\n’. When using Graylog, the parameter must be set to 0.

- **ZipLevel** 0..9 [default 0]

  Compression level for messages.

  Up until rsyslog 7.5.1, this was the only compression setting that rsyslog understood. Starting with 7.5.1, we have different compression modes. All of them are affected by the ziplevel. If, however, no mode is explicitely

set, setting ziplevel also turns on "single" compression mode, so pre 7.5.1 configuration will continue to work as expected.

The compression level is specified via the usual factor of 0 to 9, with 9 being the strongest compression (taking up most processing time) and 0 being no compression at all (taking up no extra processing time).

- **maxErrorMessages** depricated in 8.29.0, do not use

  This was used to do some very rough "rate limiting" in versions prioer to 8.29.0 and actually stemed back to when there were no real rate-limiting capabilities in rsyslog core. Starting with 8.29.0 this setting is ignored and the rsyslog internal message rate limiter is used instead.

- **compression.mode** *mode*

  mode is one of "none", "single", or "stream:always". The default is "none", in which no compression happens at all. In "single" compression mode, Rsyslog implements a proprietary capability to zip transmitted messages. That compression happens on a message-per-message basis. As such, there is a performance gain only for larger messages. Before compressing a message, rsyslog checks if there is some gain by compression. If so, the message is sent compressed. If not, it is sent uncompressed. As such, it is totally valid that compressed and uncompressed messages are intermixed within a conversation.

  In "stream:always" compression mode the full stream is being compressed. This also uses non-standard protocol and is compatible only with receives that have the same abilities. This mode offers potentially very high compression ratios. With typical syslog messages, it can be as high as 95+% compression (so only one twentieth of data is actually transmitted!). Note that this mode introduces extra latency, as data is only sent when the compressor emits new compressed data. For typical syslog messages, this can mean that some hundered messages may be held in local buffers before they are actually sent. This mode has been introduced in 7.5.1.

  Note: currently only imptcp supports receiving stream-compressed data.

- **compression.stream.flushOnTXEnd** *[**on**/off*] (requires 7.5.3+)

  This setting affects stream compression mode, only. If enabled (the default), the compression buffer will by emptied at the end of a rsyslog batch. If set to "off", end of batch will not affect compression at all.

  While setting it to "off" can potentially greatly improve compression ratio, it will also introduce severe delay between when a message is being processed by rsyslog and actually sent out to the network. We have seen cases where for several thousand message not a single byte was sent. This is good in the sense that it can happen only if we have a great compression ratio. This is most probably a very good mode for busy machines which will process several thousand messages per second and te resulting short delay will not pose any problems. However, the default is more conservative, while it works more "naturally" with even low message traffic. Even in flush mode, notable compression should be achivable (but we do not yet have practice reports on actual compression ratios).

- **RebindInterval** integer

  Permits to specify an interval at which the current connection is broken and re-established. This setting is primarily an aid to load balancers. After the configured number of messages has been transmitted, the current connection is terminated and a new one started. Note that this setting applies to both TCP and UDP traffic. For UDP, the new ''connection" uses a different source port (ports are cycled and not reused too frequently). This usually is perceived as a ''new connection" by load balancers, which in turn forward messages to another physical target system.

- **KeepAlive** *[**on**/off*]

  Enable or disable keep-alive packets at the tcp socket layer. The default is to disable them.

- **KeepAlive.Probes** integer

  The number of unacknowledged probes to send before considering the connection dead and notifying the application layer. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

- **KeepAlive.Interval** integer

  The interval between subsequential keepalive probes, regardless of what the connection has exchanged in the meantime. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

- **KeepAlive.Time** integer

  The interval between the last data packet sent (simple ACKs are not considered data) and the first keepalive probe; after the connection is marked to need keepalive, this counter is not used any further. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

- **StreamDriver** string

  Set the file owner for directories newly created. Please note that this setting does not affect the owner of directories already existing. The parameter is a user name, for which the userid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

- **StreamDriverMode** integer [default 0]

  mode to use with the stream driver (driver-specific)

- **StreamDriverAuthMode** string

  authentication mode to use with the stream driver. Note that this directive requires TLS netstream drivers. For all others, it will be ignored. (driver-specific).

- **StreamDriverPermittedPeers** string

  accepted fingerprint (SHA1) or name of remote peer. Note that this directive requires TLS netstream drivers. For all others, it will be ignored. (driver-specific)

- **ResendLastMSGOnReconnect** on/off

  Permits to resend the last message when a connection is reconnected. This setting affects TCP-based syslog, only. It is most useful for traditional, plain TCP syslog. Using this protocol, it is not always possible to know which messages were successfully transmitted to the receiver when a connection breaks. In many cases, the last message sent is lost. By switching this setting to "yes", rsyslog will always retransmit the last message when a connection is reestablished. This reduces potential message loss, but comes at the price that some messages may be duplicated (what usually is more acceptable).

  Please note that busy systems probably loose more than a single message in such cases. This is caused by an inherant unreliability in plain tcp syslog and there is no way rsyslog could prevent this from happening (if you read the detail description, be sure to follow the link to the follow-up posting). In order to prevent these problems, we recommend the use of *omrelp*.

- **udp.sendToAll** Boolean [on/off]

  Default: off

  When sending UDP messages, there are potentially multiple paths to the target destination. By default, rsyslogd only sends to the first target it can successfully send to. If this option is set to "on", messages are sent to all targets. This may improve reliability, but may also cause message duplication. This option should be enabled only if it is fully understood.

  Note: this option replaces the former -A command line option. In contrast to the -A option, this option must be set once per input() definition.

- **udp.sendDelay Integer**

  Default: 0

  Available since: 8.7.0

This is an **expert option**, do only use it if you know very well why you are using it!

This options permits to introduce a small delay after *each* send operation. The integer specifies the delay in microseconds. This option can be used in cases where too-quick sending of UDP messages causes message loss (UDP is permitted to drop packets if e.g. a device runs out of buffers). Usually, you do not want this delay. The parameter was introduced in order to support some testbench tests. Be sure to think twice before you use it in producetion.

- **gnutlsPriorityString** string

  Default: NULL

  Available since: 8.29.0

  The GnuTLS priority strings specify the TLS session's handshake algorithms and options. These strings are intended as a user-specified override of the library defaults. If this parameter is NULL, the default settings are used. More information about priority Strings here.

See Also

- Encrypted Disk Queues

Caveats/Known Bugs

Currently none.

Sample

The following command sends all syslog messages to a remote server via TCP port 10514.

```
action(type="omfwd" Target="192.168.2.11" Port="10514" Protocol="tcp" Device="eth0")
```

In case the system in use has multiple (maybe virtual) network interfaces network namespaces come in handy, each with its own routing table. To be able to distribute syslogs to remote servers in different namespaces specify them as separate actions.

```
action(type="omfwd" Target="192.168.1.13" Port="10514" Protocol="tcp"␣
↪NetworkNamespace="ns_eth0.0")
action(type="omfwd" Target="192.168.2.24" Port="10514" Protocol="tcp"␣
↪NetworkNamespace="ns_eth0.1")
action(type="omfwd" Target="192.168.3.38" Port="10514" Protocol="tcp"␣
↪NetworkNamespace="ns_eth0.2")
```

Legacy Configuration Directives

- **$ActionForwardDefaultTemplateName**string [templatename] sets a new default template for UDP and plain TCP forwarding action
- **$ActionSendTCPRebindInterval**integer instructs the TCP send action to close and re-open the connection to the remote host every nbr of messages sent. Zero, the default, means that no such processing is done. This directive is useful for use with load-balancers. Note that there is some performance overhead associated with it, so it is advisable to not too often "rebind" the connection (what "too often" actually means depends on your configuration, a rule of thumb is that it should be not be much more often than once per second).

- **$ActionSendUDPRebindInterval**integer instructs the UDP send action to rebind the send socket every nbr of messages sent. Zero, the default, means that no rebind is done. This directive is useful for use with load-balancers.
- **$ActionSendStreamDriver**<driver basename> just like $DefaultNetstreamDriver, but for the specific action
- **$ActionSendStreamDriverMode**<mode> [default 0] mode to use with the stream driver (driver-specific)
- **$ActionSendStreamDriverAuthMode**<mode> authentication mode to use with the stream driver. Note that this directive requires TLS netstream drivers. For all others, it will be ignored. (driver-specific))
- **$ActionSendStreamDriverPermittedPeers**<ID> accepted fingerprint (SHA1) or name of remote peer. Note that this directive requires TLS netstream drivers. For all others, it will be ignored. (driver-specific)
- **$ActionSendResendLastMsgOnReconnect**on/off [default off] specifies if the last message is to be resend when a connecition breaks and has been reconnected. May increase reliability, but comes at the risk of message duplication.
- **$ResetConfigVariables** Resets all configuration variables to their default value. Any settings made will not be applied to configuration lines following the $ResetConfigVariables. This is a good method to make sure no side-effects exists from previous directives. This directive has no parameters.

Legacy Sample

The following command sends all syslog messages to a remote server via TCP port 10514.

```
$ModLoad omfwd
*.* @@192.168.2.11:10514
```


omhdfs: Hadoop Filesystem Output Module

Module Name: omhdfs

Available since:: 5.7.1

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module supports writing message into files on Hadoop's HDFS file system.

Configuration Directives:

- **$OMHDFSFileName [name]** The name of the file to which the output data shall be written.
- **$OMHDFSHost [name]** Name or IP address of the HDFS host to connect to.
- **$OMHDFSPort** [name] Port on which to connect to the HDFS host.
- **$OMHDFSDefaultTemplate** [name] Default template to be used when none is specified. This saves the work of specifying the same template ever and ever again. Of course, the default template can be overwritten via the usual method.

Caveats/Known Bugs:

Building omhdfs is a challenge because we could not yet find out how to integrate Java properly into the autotools build process. The issue is that HDFS is written in Java and libhdfs uses JNI to talk to it. That requires that various

system-specific environment options and pathes be set correctly. At this point, we leave this to the user. If someone know how to do it better, please drop us a line!

- In order to build, you need to set these environment variables BEFORE running ./configure:
  - JAVA_INCLUDES - must have all include pathes that are needed to build JNI C programms, including the -I options necessary for gcc. An example is # export JAVA_INCLUDES="-I/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/include -I/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/include/linux"
  - JAVA_LIBS - must have all library pathes that are needed to build JNI C programms, including the -l/-L options necessary for gcc. An example is # export export JAVA_LIBS="-L/usr/java/jdk1.6.0_21/jre/lib/amd64 -L/usr/java/jdk1.6.0_21/jre/lib/amd64/server -ljava -ljvm -lverify"
- As of HDFS architecture, you must make sure that all relevant environment variables (the usual Java stuff and HADOOP's home directory) are properly set.
- As it looks, libhdfs makes Java throw exceptions to stdout. There is no known work-around for this (and it usually should not case any troubles.

Sample:

```
$ModLoad omhdfs $OMHDFSFileName /var/log/logfile \*.\* :omhdfs:
```

This documentation is part of the rsyslog project.

omhiredis: Redis Output Module

Module Name: omhiredis

Original Author: Brian Knox <bknox@digitalocean.com>

Description

This module provides native support for writing to Redis, using the hiredis client library.

Action Parameters

- **server** Name or address of the Redis server
- **serverport** Port of the Redis server if the server is not listening on the default port.
- **serverpassword** Password to support authenticated redis database server to push messages across networks and datacenters. Parameter is optional if not provided AUTH command wont be sent to the server.
- **mode** Mode to run the output action in: "queue" or "publish". If not supplied, the original "template" mode is used. Note due to a config parsing bug in 8.13, explicitly setting this to "template" mode will result in a config parsing error.
- **template** Template is required if using "template" mode.
- **key** Key is required if using "publish" or "queue" mode.

Examples

Mode: template

In "template" mode, the string constructed by the template is sent to Redis as a command. Note this mode has problems with strings with spaces in them - full message won't work correctly. In this mode, the template argument is required, and the key argument is meaningless.

:: module(load="omhiredis")

template(name="program_count_tmpl" type="string" string="HINCRBY progcount %programname% 1")

action(name="count_programs" server="my-redis-server.example.com" serverport="6379" type="omhiredis" mode="template" template="program_count_tmpl")

Here's an example redis-cli session where we HGETALL the counts:

:: > redis-cli 127.0.0.1:6379> HGETALL progcount 1) "rsyslogd" 2) "35" 3) "rsyslogd-pstats" 4) "4302"

Mode: queue

In "queue" mode, the syslog message is pushed into a Redis list at "key", using the LPUSH command. If a template is not supplied, the plugin will default to the RSYSLOG_ForwardFormat template.

:: module(load="omhiredis")

action(name="push_redis" server="my-redis-server.example.com" serverport="6379" type="omhiredis" mode="queue" key="my_queue")

Here's an example redis-cli session where we RPOP from the queue:

:: > redis-cli 127.0.0.1:6379> RPOP my_queue

"<46>2015-09-17T10:54:50.080252-04:00 myhost rsyslogd: [origin software="rsyslogd" swVersion="8.13.0.master" x-pid="6452" x-info="http://www.rsyslog.com"] start"

127.0.0.1:6379>

Mode: publish

In "publish" mode, the syslog message is published to a Redis topic set by "key". If a template is not supplied, the plugin will default to the RSYSLOG_ForwardFormat template.

:: module(load="omhiredis")

action(name="publish_redis" server="my-redis-server.example.com" serverport="6379" type="omhiredis" mode="publish" key="my_channel")

Here's an example redis-cli session where we SUBSCRIBE to the topic:

:: > redis-cli

127.0.0.1:6379> subscribe my_channel

Reading messages... (press Ctrl-C to quit)

1. "subscribe"
2. "my_channel"
3. (integer) 1
1. "message"
2. "my_channel"
3. "<46>2015-09-17T10:55:44.486416-04:00 myhost rsyslogd-pstats: {"name":"imuxsock","origin":"imuxsock","submitted":

This documentation is part of the rsyslog project.

omhttpfs: Hadoop HTTPFS Output Module

Module Name:	omhttpfs
Available Since:	8.10.0
Author:	sskaje <sskaje@gmail.com>

This module is an alternative to omhdfs via Hadoop HDFS over HTTP.

Dependencies

- libcurl

Configure

```
./configure --enable-omhttpfs
```

Config options

Legacy config **NOT** supported.

- **host** HttpFS server host. Default: *127.0.0.1*
- **port** HttpFS server port. Default: *14000*
- **user** HttpFS auth user. Default: *hdfs*
- **https <on/off>** Turn on if your HttpFS runs on HTTPS. Default: *off*
- **file** File to write, or a template name.
- **isdynfile <on/off>** Turn this on if your **file** is a template name.

  See examples below.
- **template** Format your message when writing to **file**. Default: *RSYSLOG_FileFormat*

Examples

```
module(load="omhttpfs")
template(name="hdfs_tmp_file" type="string" string="/tmp/%$YEAR%/test.log")
template(name="hdfs_tmp_filecontent" type="string" string="%$YEAR%-%$MONTH%-%$DAY%
↪%MSG% ==\n")
local4.*    action(type="omhttpfs" host="10.1.1.161" port="14000" https="off" file=
↪"hdfs_tmp_file" isDynFile="on")
local5.*    action(type="omhttpfs" host="10.1.1.161" port="14000" https="off" file=
↪"hdfs_tmp_file" isDynFile="on" template="hdfs_tmp_filecontent")
```

omjournal: Systemd Journal Output

Module Name: omjournal

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module provides native support for logging to the systemd journal.

Action Parameters:

- **template** Template to use when submitting messages.

By default, rsyslog will use the incoming %msg% as the MESSAGE field of the journald entry, and include the syslog tag and priority.

You can override the default formatting of the message, and include custom fields with a template. Complex fields in the template (eg. json entries) will be added to the journal as json text. Other fields will be coerced to strings.

Journald requires that you include a template parameter named MESSAGE.

Sample:

The following sample writes all syslog messages to the journal with a custom EVENT_TYPE field.

```
module(load="omjournal")

template(name="journal" type="list") {
  constant(value="Something happened" outname="MESSAGE")
  property(name="$!event-type" outname="EVENT_TYPE")
}

action(type="omjournal" template="journal")
```


omkafka: write to Apache Kafka

Module Name:	omkafka
Author:	Rainer Gerhards <rgerhards@adiscon.com>
Available since:	v8.7.0

The omkafka plug-in implements an Apache Kafka producer, permitting rsyslog to write data to Kafka.

Configuration Parameters

Module Parameters

Currently none.

Action Parameters

Note that omkafka supports some *Array*-type parameters. While the parameter name can only be set once, it is possible to set multiple values with that single parameter.

For example, to select "snappy" compression, you can use

```
action(type="omkafka" topic="mytopic" confParam="compression.codec=snappy")
```

which is equivalent to

```
action(type="omkafka" topic="mytopic" confParam=["compression.codec=snappy"])
```

To specify multiple values, just use the bracket notation and create a comma-delimited list of values as shown here:

```
action(type="omkafka" topic="mytopic"
       confParam=["compression.codec=snappy",
                  "socket.timeout.ms=5",
                  "socket.keepalive.enable=true"]
      )
```

broker [brokers]
Type: Array

Default: "localhost:9092"

Specifies the broker(s) to use.

topic [topicName]
Mandatory

Specifies the topic to produce to.

key [key]
Default: none

Kafka key to be used for all messages.

dynatopic [boolean]
Default: off

If set, the topic parameter becomes a template for which topic to produce messages to. The cache is cleared on HUP.

dynatopic.cachesize [positiveInteger]
Default: 50

If set, defines the number of topics that will be kept in the dynatopic cache.

partitions.auto [boolean]
Default: off

librdkafka provides an automatic partitioning function that will evenly split the produced messages into all partitions configured for that topic.

To use, set partitions.auto="on". This is instead of specifying the number of partitions on the producer side, where it would be easier to change the kafka configuration on the cluster for number of partitions/topic vs on every machine talking to Kafka via rsyslog.

If set, it will override any other partitioning scheme configured.

partitions.number [positiveInteger]
Default: none

If set, specifies how many partitions exists **and** activates load-balancing among them. Messages are distributed more or less evenly between the partitions. Note that the number specified must be correct. Otherwise, some errors may occur or some partitions may never receive data.

partitions.usedFixed [positiveInteger]
Default: none

If set, specifies the partition to which data is produced. All data goes to this partition, no other partition is ever involved for this action.

errorFile [filename]
Default: none

If set, messages that could not be sent and caused an error messages are written to the file specified. This file is in JSON format, with a single record being written for each message in error. The entry contains the full message, as well as Kafka error number and reason string.

The idea behind the error file is that the admin can periodically run a script that reads the error file and reacts on it. Note that the error file is kept open from when the first error occured up until rsyslog is terminated or received a HUP signal.

confParam [parameter]

Type: Array

Default: none

Permits to specify Kafka options. Rather than offering a myriad of config settings to match the Kafka parameters, we provide this setting here as a vehicle to set any Kafka parameter. This has the big advantage that Kafka parameters that come up in new releases can immediately be used.

Note that we use librdkafka for the Kafka connection, so the parameters are actually those that librdkafka supports. As of our understanding, this is a superset of the native Kafka parameters.

topicConfParam [parameter]

Type: Array

Default: none

In essence the same as *confParam*, but for the Kafka topic.

template [templateName]

Default: template set via "template" module parameter

Sets the template to be used for this action.

closeTimeout [positiveInteger]

Default: 2000

Sets the time to wait in ms (milliseconds) for draining messages submitted to kafka-handle (provided by librdkafka) before closing it.

The maximum value of closeTimeout used across all omkafka action instances is used as librdkafka unload-timeout while unloading the module (for shutdown, for instance).

resubmitOnFailure [boolean]

Default: off

Available since: 8.28.0

If enabled, failed messages will be resubmit automatically when kafka is able to send messages again. To prevent message loss, this option should be enabled.

keepFailedMessages [boolean]

Default: off

Available since: 8.28.0

If enabled, failed messages will be saved and loaded on shutdown/startup and resend after startup if the kafka server is able to receive messages again. This setting requires resubmitOnFailure to be enabled as well.

failedMsgFile [filename]

Default: none

Available since: 8.28.0

Filename where the failed messages should be stored into. Needs to be set when keepFailedMessages is enabled, otherwise failed messages won't be saved.

Caveats/Known Bugs

- currently none

Example

To be added, see intro to action parameters.

omlibdbi: Generic Database Output Module

Module Name: omlibdbi

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This modules supports a large number of database systems via libdbi. Libdbi abstracts the database layer and provides drivers for many systems. Drivers are available via the libdbi-drivers project. As of this writing, the following drivers are available:

- Firebird/Interbase
- FreeTDS (provides access to MS SQL Server and Sybase)
- MySQL (also supported via the native ommysql plugin in rsyslog)
- PostgreSQL(also supported via the native ommysql plugin in rsyslog)
- SQLite/SQLite3

The following drivers are in various stages of completion:

- Ingres
- mSQL
- Oracle

These drivers seem to be quite usable, at least from an rsyslog point of view.

Libdbi provides a slim layer between rsyslog and the actual database engine. We have not yet done any performance testing (e.g. omlibdbi vs. ommysql) but honestly believe that the performance impact should be irrelevant, if at all measurable. Part of that assumption is that rsyslog just does the "insert" and most of the time is spent either in the database engine or rsyslog itself. It's hard to think of any considerable time spent in the libdbi abstraction layer.

Setup

In order for this plugin to work, you need to have libdbi, the libdbi driver for your database backend and the client software for your database backend installed. There are libdbi packages for many distributions. Please note that rsyslogd requires a quite recent version (0.8.3) of libdbi. It may work with older versions, but these need some special ./configure options to support being called from a dlopen()ed plugin (as omlibdbi is). So in short, you probably save you a lot of headache if you make sure you have at least libdbi version 0.8.3 on your system.

Configuration Directives:

- **$ActionLibdbiDriverDirectory** /path/to/dbd/drivers

  This is a global setting. It points libdbi to its driver directory. Usually, you do not need to set it. If you installed libdbi-driver's at a non-standard location, you may need to specify the directory here. If you are unsure, do not use this configuration directive. Usually, everything works just fine.

- **$ActionLibdbiDriver** drivername

  Name of the dbidriver to use, see libdbi-drivers documentation. As a quick excerpt, at least those were available at the time of this writiting "mysql" (suggest to use ommysql instead), "firebird" (Firbird and InterBase), "ingres", "msql", "Oracle", "sqlite", "sqlite3", "freetds" (for Microsoft SQL and Sybase) and "pgsql" (suggest to use ompgsql instead).

- **$ActionLibdbiHost** hostname

  The host to connect to.

- **$ActionLibdbiUserName** user

  The user used to connect to the database.

- **$ActionlibdbiPassword**

  That user's password.

- **$ActionlibdbiDBName** db

  The database that shall be written to.

- **selector line:** :omlibdbi:;template

  executes the recently configured omlibdbi action. The ;template part is optional. If no template is provided, a default template is used (which is currently optimized for MySQL - sorry, folks...)

Caveats/Known Bugs:

You must make sure that any templates used for omlibdbi properly escape strings. This is usually done by supplying the SQL (or STDSQL) option to the template. Omlibdbi rejects templates without this option for security reasons. However, omlibdbi does not detect if you used the right option for your backend. Future versions of rsyslog (with full expression support) will provide advanced ways of handling this situation. So far, you must be careful. The default template provided by rsyslog is suitable for MySQL, but not necessarily for your database backend. Be careful!

If you receive the rsyslog error message "libdbi or libdbi drivers not present on this system" you may either not have libdbi and its drivers installed or (very probably) the version is earlier than 0.8.3. In this case, you need to make sure you have at least 0.8.3 and the libdbi driver for your database backend present on your system.

I do not have most of the database supported by omlibdbi in my lab. So it received limited cross-platform tests. If you run into troubles, be sure the let us know at http://www.rsyslog.com.

Sample:

The following sample writes all syslog messages to the database "syslog_db" on mysqlserver.example.com. The server is MySQL and being accessed under the account of "user" with password "pwd" (if you have empty passwords, just remove the $ActionLibdbiPassword line).

```
$ModLoad omlibdbi $ActionLibdbiDriver mysql $ActionLibdbiHost
mysqlserver.example.com $ActionLibdbiUserName user $ActionLibdbiPassword
pwd $ActionLibdbiDBName syslog\_db \*.\* :omlibdbi:
```

This documentation is part of the rsyslog project.

ommail: Mail Output Module

Module Name:	ommail
Available Since:	3.17.0
Author:	Rainer Gerhards <rgerhards@adiscon.com>

This module supports sending syslog messages via mail. Each syslog message is sent via its own mail. Obviously, you will want to apply rigorous filtering, otherwise your mailbox (and mail server) will be heavily spammed. The ommail plugin is primarily meant for alerting users. As such, it is assumed that mails will only be sent in an extremely limited number of cases.

Ommail uses up to two templates, one for the mail body and optionally one for the subject line. Note that the subject line can also be set to a constant text. If neither the subject nor the mail body is provided, a quite meaningless subject line is used and the mail body will be a syslog message just as if it were written to a file. It is expected that the users customizes both messages. In an effort to support cell phones (including SMS gateways), there is an option to turn off the body part at all. This is considered to be useful to send a short alert to a pager-like device. It is highly recommended to use the

```
action.execonlyonceeveryinterval="<seconds>"
```

parameter to limit the amount of mails that potentially be generated. With it, mails are sent at most in a <seconds> interval. This may be your life safer. And remember that an hour has 3,600 seconds, so if you would like to receive mails at most once every two hours, include a

```
action.execonlyonceeveryinterval="7200"
```

in the action definition. Messages sent more frequently are simpy discarded.

Configuration Parameters

Configuration parameters are supported starting with v8.5.0. Earlier v7 and v8 versions did only support legacy parameters.

Action Parameters

server <server-name>
Mandatory

Name or IP address of the SMTP server to be used. Must currently be set. The default is 127.0.0.1, the SMTP server on the local machine. Obviously it is not good to expect one to be present on each machine, so this value should be specified.

port <port-number-or-name>
Mandatory

Port number or name of the SMTP port to be used. The default is 25, the standard SMTP port.

mailfrom <sender-address>
Mandatory

The email address used as the senders address.

mailto <recipient-addresses>
Mandatory

The recipient email address(es). Note that this is an array parameter. See samples below on how to specify multiple recipients.

subject.template <template-name>
Default: none, but may be left out

The name of the template to be used as the mail subject.

If you want to include some information from the message inside the template, you need to use *subject.template* with an appropriate template. If you just need a constant text, you can simply use *subject.text* instead, which doesn't require a template definition.

subject.text <subject-string>
Default: none, but may be left out

This is used to set a **constant** subject text.

body.enable <boolean>
Default: on

Setting this to "off" permits to exclude the actual message body. This may be useful for pager-like devices or cell phone SMS messages. The default is "on", which is appropriate for allmost all cases. Turn it off only if you know exactly what you do!

template <template-name>
Default: RSYSLOG_FileFormat

Template to be used for the mail body (if enabled).

The *template.subject* and *template.text* parameters cannot be given together inside a single action definition. Use either one of them. If none is used, a more or less meaningless mail subject is generated (we don't tell you the exact text because that can change - if you want to have something specific, configure it!).

Caveats/Known Bugs

The current ommail implementation supports SMTP-direct mode only. In that mode, the plugin talks to the mail server via SMTP protocol. No other process is involved. This mode offers best reliability as it is not depending on any external entity except the mail server. Mail server downtime is acceptable if the action is put onto its own action queue, so that it may wait for the SMTP server to come back online. However, the module implements only the bare SMTP essentials. Most importantly, it does not provide any authentication capabilities. So your mail server must be configured to accept incoming mail from ommail without any authentication needs (this may be change in the future as need arises, but you may also be referred to sendmail-mode).

In theory, ommail should also offer a mode where it uses the sendmail utility to send its mail (sendmail-mode). This is somewhat less reliable (because we depend on an entity we do not have close control over - sendmail). It also requires dramatically more system ressources, as we need to load the external process (but that should be no problem given the expected infrequent number of calls into this plugin). The big advantage of sendmail mode is that it supports all the bells and whistles of a full-blown SMTP implementation and may even work for local delivery without a SMTP server being present. Sendmail mode will be implemented as need arises. So if you need it, please drop us a line (I nobody does, sendmail mode will probably never be implemented).

Examples

The following example alerts the operator if the string "hard disk fatal failure" is present inside a syslog message. The mail server at mail.example.net is used and the subject shall be "disk problem on <hostname>". Note how \r\n is included inside the body text to create line breaks. A message is sent at most once every 6 hours (21600 seconds), any other messages are silently discarded (or, to be precise, not being forwarded - they are still being processed by the rest of the configuration file).

```
module(load="ommail")

template (name="mailBody"  type="string" string="RSYSLOG Alert\\r\\nmsg='%msg%'")
template (name="mailSubject" type="string" string="disk problem on %hostname%")
```

```
if $msg contains "hard disk fatal failure" then {
   action(type="ommail" server="mail.example.net" port="25"
          mailfrom="rsyslog@example.net"
          mailto="operator@example.net"
          subject.template="mailSubject"
          action.execonlyonceeveryinterval="21600")
}
```

The following example is exactly like the first one, but it sends the mails to two different email addresses:

```
module(load="ommail")

template (name="mailBody"  type="string" string="RSYSLOG Alert\\r\\nmsg='%msg%'")
template (name="mailSubject" type="string" string="disk problem on %hostname%")

if $msg contains "hard disk fatal failure" then {
   action(type="ommail" server="mail.example.net" port="25"
          mailfrom="rsyslog@example.net"
          mailto=["operator@example.net", "admin@example.net"]
          subject.template="mailSubject"
          action.execonlyonceeveryinterval="21600")
}
```

Note the array syntax to specify email addresses. Note that while rsyslog permits you to specify as many recipients as you like, your mail server may limit their number. It is usually a bad idea to use more than 50 recipients, and some servers may have lower limits. If you hit such a limit, you could either create additional actions or (recommended) create an email distribution list.

The next example is again mostly equivalent to the previous one, but it uses a constant subject line, so no subject template is required:

```
module(load="ommail")

template (name="mailBody"  type="string" string="RSYSLOG Alert\\r\\nmsg='%msg%'")

if $msg contains "hard disk fatal failure" then {
   action(type="ommail" server="mail.example.net" port="25"
          mailfrom="rsyslog@example.net"
          mailto=["operator@example.net", "admin@example.net"]
          subject.text="rsyslog detected disk problem"
          action.execonlyonceeveryinterval="21600")
}
```

Legacy Configuration Directives

Note that the legacy configuration parameters do **not** affect new-style action definitions via the action() object. This is by design. To set default for action() objects, use module parameters in the

```
module(load="builtin:ommail" ...)
```

object.

Read about *the importance of order in legacy configuration* to understand how to use these configuration directives. **Legacy directives should NOT be used when writing new configuration files.**

- $ActionMailSMTPServer

equivalent to the *server* action parameter.

- $ActionMailSMTPPort

  equivalent to the *port* action parameter.

- $ActionMailFrom

  equivalent to the *mailfrom* action parameter.

- $ActionMailTo

  mostly equivalent to the *mailto* action parameter. However, to specify multiple recpients, repeat this directive as often as needed. Note: **This directive must be specified for each new action and is automatically reset.** [Multiple recipients are supported for 3.21.2 and above.]

- $ActionMailSubject

  equivalent to the *subject.template* action parameter.

- $ActionMailEnableBody

  equivalent to the *body.enable* action parameter.

Legacy Examples

The following sample alerts the operator if the string "hard disk fatal failure" is present inside a syslog message. The mail server at mail.example.net is used and the subject shall be "disk problem on <hostname>". Note how \r\n is included inside the body text to create line breaks. A message is sent at most once every 6 hours, any other messages are silently discarded (or, to be precise, not being forwarded - they are still being processed by the rest of the configuration file).

```
$ModLoad ommail
$ActionMailSMTPServer mail.example.net
$ActionMailFrom rsyslog@example.net
$ActionMailTo operator@example.net
$template mailSubject,"disk problem on %hostname%"
$template mailBody,"RSYSLOG Alert\\r\\nmsg='%msg%'"
$ActionMailSubject mailSubject
# make sure we receive a mail only once in six
# hours (21,600 seconds ;))
$ActionExecOnlyOnceEveryInterval 21600
# the if ... then ... mailBody must be on one line!
if $msg contains 'hard disk fatal failure' then :ommail:;mailBody
# re-set interval so that other actions are not affected
$ActionExecOnlyOnceEveryInterval 0
```

The sample below is the same, but sends mail to two recipients:

```
$ModLoad ommail
$ActionMailSMTPServer mail.example.net
$ActionMailFrom rsyslog@example.net
$ActionMailTo operator@example.net
$ActionMailTo admin@example.net
$template mailSubject,"disk problem on %hostname%"
$template mailBody,"RSYSLOG Alert\\r\\nmsg='%msg%'"
$ActionMailSubject mailSubject
# make sure we receive a mail only once in six
# hours (21,600 seconds ;))
$ActionExecOnlyOnceEveryInterval 21600
```

```
# the if ...  then ... mailBody mus be on one line!
if $msg contains 'hard disk fatal failure' then :ommail:;mailBody
# re-set interval so that other actions are not affected
$ActionExecOnlyOnceEveryInterval 0
```

A more advanced example plus a discussion on using the email feature inside a reliable system can be found in Rainer's blogpost "Why is native email capability an advantage for a syslogd?"

ommongodb: MongoDB Output Module

Module Name: ommongodb

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module provides native support for logging to MongoDB.

Action Parameters:

- **server** Name or address of the MongoDB server
- **serverport** Permits to select a non-standard port for the MongoDB server. The default is 0, which means the system default port is used. There is no need to specify this parameter unless you know the server is running on a non-standard listen port.
- **db** Database to use
- **collection** Collection to use
- **uid** logon userid used to connect to server. Must have proper permissions.
- **pwd** the user's password
- **template** Template to use when submitting messages.

Note rsyslog contains a canned default template to write to the MongoDB. It will be used automatically if no other template is specified to be used. This template is:

```
template(name="BSON" type="string" string="\\"sys\\" : \\"%hostname%\\",
\\"time\\" : \\"%timereported:::rfc3339%\\", \\"time\_rcvd\\" :
\\"%timegenerated:::rfc3339%\\", \\"msg\\" : \\"%msg%\\",
\\"syslog\_fac\\" : \\"%syslogfacility%\\", \\"syslog\_server\\" :
\\"%syslogseverity%\\", \\"syslog\_tag\\" : \\"%syslogtag%\\",
\\"procid\\" : \\"%programname%\\", \\"pid\\" : \\"%procid%\\",
\\"level\\" : \\"%syslogpriority-text%\\"")
```

This creates the BSON document needed for MongoDB if no template is specified. The default schema is aligned to CEE and project lumberjack. As such, the field names are standard lumberjack field names, and **not** rsyslog property names. When specifying templates, be sure to use rsyslog property names as given in the table. If you would like to use lumberjack-based field names inside MongoDB (which probably is useful depending on the use case), you need to select fields names based on the lumberjack schema. If you just want to use a subset of the fields, but with lumberjack names, you can look up the mapping in the default template. For example, the lumberjack field "level" contains the rsyslog property "syslogpriority-text".

Sample:

The following sample writes all syslog messages to the database "syslog" and into the collection "log" on mongoserver.example.com. The server is being accessed under the account of "user" with password "pwd".

```
module(load="ommongodb")
action(type="ommongodb"
       server="mongoserver.example.com" db="syslog" collection="log"
       uid="user" pwd="pwd")
```


ommysql: MySQL Database Output Module

Module Name: ommysql

Author: Michael Meckelein (Initial Author) / Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module provides native support for logging to MySQL databases. It offers superior performance over the more generic omlibdbi module.

Configuration Directives:

ommysql mostly uses the "old style" configuration, with almost everything on the action line itself. A few newer features are being migrated to the new style-config directive configuration system.

- **$ActionOmmysqlServerPort <port>**

  Permits to select a non-standard port for the MySQL server. The default is 0, which means the system default port is used. There is no need to specify this directive unless you know the server is running on a non-standard listen port.

- **$OmMySQLConfigFile <file name>**

  Permits the selection of an optional MySQL Client Library configuration file (my.cnf) for extended configuration functionality. The use of this configuration directive is necessary only if you have a non-standard environment or if fine-grained control over the database connection is desired.

- **$OmMySQLConfigSection <string>**

  Permits the selection of the section within the configuration file specified by the **$OmMySQLConfigFile** directive. This will likely only be used where the database administrator provides a single configuration file with multiple profiles. This configuration directive is ignored unless **$OmMySQLConfigFile** is also used in the rsyslog configration file. If omitted, the MySQL Client Library default of "client" will be used.

- Action parameters: **:ommysql:database-server,database-name,database-userid,database-password** All parameters should be filled in for a successful connect.

Note rsyslog contains a canned default template to write to the MySQL database. It works on the MonitorWare schema. This template is:

```
$template tpl,"insert into SystemEvents (Message, Facility, FromHost,
Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag) values
('%msg%', %syslogfacility%, '%HOSTNAME%', %syslogpriority%,
'%timereported:::date-mysql%', '%timegenerated:::date-mysql%', %iut%,
'%syslogtag%')",SQL
```

As you can see, the template is an actual SQL statement. Note the ",SQL" option: it tells the template processor that the template is used for SQL processing, thus quote characters are quoted to prevent security issues. You can not assign a template without ",SQL" to a MySQL output action.

If you would like to change fields contents or add or delete your own fields, you can simply do so by modifying the schema (if required) and creating your own custom template.

Sample:

The following sample writes all syslog messages to the database "syslog_db" on mysqlserver.example.com. The server is being accessed under the account of "user" with password "pwd".

```
$ModLoad ommysql $ActionOmmysqlServerPort 1234 # use non-standard port
\*.\*      :ommysql:mysqlserver.example.com,syslog\_db,user,pwd
```

This documentation is part of the rsyslog project.

omoracle: Oracle Database Output Module

Module Name: omoracle

Author: Luis Fernando Muñoz Mejías <Luis.Fernando.Munoz.Mejias@cern.ch> - this module is currently orphaned, the original author does no longer support it.

Available since:: 4.3.0, **does not work with recent rsyslog versions (v7 and up). Use** *omlibdbi* **instead.** An upgrade to the new interfaces is needed. If you would like to contribute, please send us a patch or open a guithub pull request.

Status:: contributed module, not maitained by rsyslog core authors

Description:

This module provides native support for logging to Oracle databases. It offers superior performance over the more generic omlibdbi module. It also includes a number of enhancements, most importantly prepared statements and batching, what provides a big performance improvement.

Note that this module is maintained by its original author. If you need assistance with it, it is suggested to post questions to the rsyslog mailing list.

From the header comments of this module:

```
This is an output module feeding directly to an Oracle
database. It uses Oracle Call Interface, a propietary module
provided by Oracle.

Selector lines to be used are of this form:

:omoracle:;TemplateName

The module gets its configuration via rsyslog $... directives,
namely:

$OmoracleDBUser: user name to log in on the database.

$OmoracleDBPassword: password to log in on the database.

$OmoracleDB: connection string (an Oracle easy connect or a db
name as specified by tnsnames.ora)

$OmoracleBatchSize: Number of elements to send to the DB on each
transaction.

$OmoracleStatement: Statement to be prepared and executed in
```

```
batches. Please note that Oracle's prepared statements have their
placeholders as ':identifier', and this module uses the colon to
guess how many placeholders there will be.

All these directives are mandatory. The dbstring can be an Oracle
easystring or a DB name, as present in the tnsnames.ora file.

The form of the template is just a list of strings you want
inserted to the DB, for instance:

$template TestStmt,"%hostname%%msg%"

Will provide the arguments to a statement like

$OmoracleStatement \
    insert into foo(hostname,message)values(:host,:message)

Also note that identifiers to placeholders are arbitrary. You
need to define the properties on the template in the correct order
you want them passed to the statement!
```

Some additional documentation contributed by Ronny Egner:

```
REQUIREMENTS:
--------------

- Oracle Instantclient 10g (NOT 11g) Base + Devel
  (if you´re on 64-bit linux you should choose the 64-bit libs!)
- JDK 1.6 (not neccessary for oracle plugin but "make" didd not finsished␣
↪successfully without it)

- "oracle-instantclient-config" script
  (seems to shipped with instantclient 10g Release 1 but i was unable to find it for␣
↪10g Release 2 so here it is)


======================  /usr/local/bin/oracle-instantclient-config␣
↪======================
#!/bin/sh
#
# Oracle InstantClient SDK config file
# Jean-Christophe Duberga - Bordeaux 2 University
#

# just adapt it to your environment
incdirs="-I/usr/include/oracle/10.2.0.4/client64"
libdirs="-L/usr/lib/oracle/10.2.0.4/client64/lib"

usage="\
Usage: oracle-instantclient-config [--prefix[=DIR]] [--exec-prefix[=DIR]] [--version]␣
↪[--cflags] [--libs] [--static-libs]"

if test $# -eq 0; then
      echo "${usage}" 1>&2
      exit 1
fi

while test $# -gt 0; do
```

```
  case "$1" in
  -*=*) optarg=`echo "$1" | sed 's/[-_a-zA-Z0-9]*=//'` ;;
  *) optarg= ;;
  esac

  case $1 in
    --prefix=*)
      prefix=$optarg
      if test $exec_prefix_set = no ; then
        exec_prefix=$optarg
      fi
      ;;
    --prefix)
      echo $prefix
      ;;
    --exec-prefix=*)
      exec_prefix=$optarg
      exec_prefix_set=yes
      ;;
    --exec-prefix)
      echo ${exec_prefix}
      ;;
    --version)
      echo ${version}
      ;;
    --cflags)
      echo ${incdirs}
      ;;
    --libs)
      echo $libdirs -lclntsh -lnnz10 -locci -lociei -locijdbc10
      ;;
    --static-libs)
      echo "No static libs" 1>&2
      exit 1
      ;;
    *)
      echo "${usage}" 1>&2
      exit 1
      ;;
  esac
  shift
done

===============   END ==============



COMPILING RSYSLOGD
--------------------


./configure --enable-oracle



RUNNING
```

```
-------

- make sure rsyslogd is able to locate the oracle libs (either via LD_LIBRARY_PATH or␣
↪/etc/ld.so.conf)
- set TNS_ADMIN to point to your tnsnames.ora
- create a tnsnames.ora and test you are able to connect to the database

- create user in oracle as shown in the following example:
        create user syslog identified by syslog default tablespace users quota␣
↪unlimited on users;
        grant create session to syslog;
        create role syslog_role;
        grant syslog_role to syslog;
        grant create table to syslog_role;
        grant create sequence to syslog_role;

- create tables as needed

- configure rsyslog as shown in the following example
        $ModLoad omoracle

        $OmoracleDBUser syslog
        $OmoracleDBPassword syslog
        $OmoracleDB syslog
        $OmoracleBatchSize 1
        $OmoracleBatchItemSize 4096

        $OmoracleStatementTemplate OmoracleStatement
        $template OmoracleStatement,"insert into foo(hostname,message) values (:host,
↪:message)"
        $template TestStmt,"%hostname%%msg%"
        *.*                     :omoracle:;TestStmt
    (you guess it: username = password = database = "syslog".... see $rsyslogd_source/
↪plugins/omoracle/omoracle.c for me info)
```


ompgsql: PostgreSQL Database Output Module

Module Name: ompgsql

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module provides native support for logging to PostgreSQL databases. It's an alternative (with potentially superior performance) to the more generic omlibdbi module.

Configuration Directives:

ompgsql uses the "old style" configuration, with everything on the action line itself

Action parameters

:ompgsql:database-server,database-name,database-userid,database-password

All parameters should be filled in for a successful connect.

Note rsyslog contains a canned default template to write to the Postgres database. This template is:

```
$template StdPgSQLFmt,"insert into SystemEvents (Message, Facility, FromHost,␣
↪Priority, DeviceReportedTime, ReceivedAt, InfoUnitID, SysLogTag) values ('%msg%',
↪%syslogfacility%, '%HOSTNAME%', %syslogpriority%, '%timereported:::date-pgsql%', '
↪%timegenerated:::date-pgsql%', %iut%, '%syslogtag%')",STDSQL
```

As you can see, the template is an actual SQL statement. Note the **STDSQL** option: it tells the template processor that the template is used for SQL processing, thus quote characters are quoted to prevent security issues. You can not assign a template without **STDSQL** to a PostgreSQL output action.

If you would like to change fields contents or add or delete your own fields, you can simply do so by modifying the schema (if required) and creating your own custom template:

```
$template mytemplate,"insert into SystemEvents (Message) values ('%msg%')",STDSQL
:ompgsql:database-server,database-name,database-userid,database-password;mytemplate
```

This documentation is part of the rsyslog project.

ompipe: Pipe Output Module

Module Name: **ompipe**

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

The ompipe plug-in provides the core functionality for logging output to named pipes (fifos). It is a built-in module that does not need to be loaded. **Global Configuration Directives:**

- Template: [templateName] sets a new default template for file actions.

Action specific Configuration Directives:

- Pipe: string a fifo or named pipe can be used as a destination for log messages.

Caveats/Known Bugs: None

Sample: The following command sends all syslog messages to a remote server via TCP port 10514.

```
Module (path="builtin:ompipe")
*.* action(type="ompipe"
Pipe="NameofPipe")
```

Legacy Configuration Directives:

rsyslog has support for logging output to named pipes (fifos). A fifo or named pipe can be used as a destination for log messages by prepending a pipe symbol ("|") to the name of the file. This is handy for debugging. Note that the fifo must be created with the mkfifo(1) command before rsyslogd is started.

Legacy Sample:

The following command sends all syslog messages to a remote server via TCP port 10514.

```
$ModLoad ompipe
*.* |/var/log/pipe
```

omprog: Program integration Output module

Module Name: omprog

Available since: 4.3.0

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module permits to integrate arbitrary external programs into rsyslog's logging. It is similar to the "execute program (^)" action, but offers better security and much higher performance. While "execute program (^)" can be a useful tool for executing programs if rare events occur, omprog can be used to provide massive amounts of log data to a program.

Executes the configured program and feeds log messages to that binary via stdin. The binary is free to do whatever it wants with the supplied data. If the program terminates, it is re-started. If rsyslog terminates, the program's stdin will see EOF. The program must then terminate. The message format passed to the program can, as usual, be modified by defining rsyslog templates.

Note that each time an omprog action is defined, the corresponding program is invoked. A single instance is **not** being re-used. There are arguments pro and con for re-using existing binaries. For the time being, it simply is not done. In the future, we may add an option for such pooling, provided that some demand for that is voiced. You can also mimic the same effect by defining multiple rulesets and including them.

Note that in order to execute the given program, rsyslog needs to have sufficient permissions on the binary file. This is especially true if not running as root. Also, keep in mind that default SELinux policies most probably do not permit rsyslogd to execute arbitrary binaries. As such, permissions must be appropriately added. Note that SELinux restrictions also apply if rsyslogd runs under root. To check if a problem is SELinux-related, you can temporarily disable SELinux and retry. If it then works, you know for sure you have a SELinux issue.

Starting with 8.4.0, rsyslogd emits an error message via the `syslog()` API call when there is a problem executing the binary. This can be extremely valuable in troubleshooting. For those technically savy: when we execute a binary, we need to fork, and we do not have full access to rsyslog's usual error-reporting capabilities after the fork. As the actual execution must happen after the fork, we cannot use the default error logger to emit the error message. As such, we use `syslog()`. In most cases, there is no real difference between both methods. However, if you run multiple rsyslog instances, the messae shows up in that instance that processes the default log socket, which may be different from the one where the error occured. Also, if you redirected the log destination, that redirection may not work as expected.

Module Parameters:

- **Template[templateName]** sets a new default template for file actions.

Action Parameters:

- **binary** Mostly equivalent to the "binary" action parameter, but must contain the binary name only. In legacy config, it is **not possible** to specify command line parameters.
- **hup.signal** [v8.9.0+] Specifies which signal, if any, is to be forwarded to the executed program. Currently, HUP, USR1, USR2, INT, and TERM are supported. If unset, no signal is sent on HUP. This is the default and what pre 8.9.0 versions did.
- **signalOnClose** [switch] [v8.23.0+] *Default: off*

  Signal the child process when worker-instance is stopped or Rsyslog is about to shutdown. To signal shutdown, SIGTERM is issued to child and Rsyslog reaps the process before proceeding.

  No signal is issued if this switch is set to 'off' (default). The child-process can still detect shutdown because 'read' from stdin would EOF. However its possible to have process-leak due to careless error-handling around read. Rsyslog won't try to reap the child process in this case.

Additionaly, this controls the following **GNU/Linux specific behavior**: If 'on', Rsyslog waits for upto 5 seconds for child process to terminate after issuing SIGTERM, after which a SIGKILL is issued ensuring child-death. This ensures even an unresponsive child is reaped before shutdown.

Caveats/Known Bugs:

- None.

Sample:

The following command writes all syslog messages into a file.

```
module(load="omprog")
action(type="omprog"
       binary="/pathto/omprog.py --parm1=\"value 1\" --parm2=\"value2\""
       template="RSYSLOG_TraditionalFileFormat")
```

Legacy Configuration Directives:

- **$ActionOMProgBinary** <binary> The binary program to be executed.

Caveats/Known Bugs:

Currently none known.

omrelp: RELP Output Module

Module Name: omrelp

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module supports sending syslog messages over the reliable RELP protocol. For RELP's advantages over plain tcp syslog, please see the documentation for *imrelp* (the server counterpart).

Setup

Please note that librelp is required for imrelp (it provides the core relp protocol implementation).

Action Configuration Parameters:

This module supports RainerScript configuration starting with rsyslog 7.3.10. For older versions, legacy configuration directives must be used.

- **target** (mandatory) The target server to connect to.
- **template** (not mandatory, default "RSYSLOG_ForwardFormat") Defines the template to be used for the output.
- **timeout** (not mandatory, default 90) Timeout for relp sessions. If set too low, valid sessions may be considered dead and tried to recover.
- **conn.timeout** (not mandatory, default 10) Timeout for the socket connection.
- **windowSize** (not mandatory, default 0) This is an **expert parameter**. It permits to override the RELP window size being used by the client. Changing the window size has both an effect on performance as well as potential message duplication in failure case. A larger window size means more performance, but also potentially more duplicated messages - and vice versa. The default 0 means that librelp's default window size is being used, which is considered a compromise between goals reached. For your information: at the time of this writing, the librelp default window size is 128 messages, but this may change at any time. Note that there is no equivalent server parameter, as the client proposes and manages the window size in RELP protocol.

- **tls** (not mandatory, values "on","off", default "off") If set to "on", the RELP connection will be encrypted by TLS, so that the data is protected against observers. Please note that both the client and the server must have set TLS to either "on" or "off". Other combinations lead to unpredictable results.

  Attention when using GnuTLS 2.10.x or older

  Versions older than GnuTLS 2.10.x may cause a crash (Segfault) under certain circumstances. Most likely when an imrelp inputs and an omrelp output is configured. The crash may happen when you are receiving/sending messages at the same time. Upgrade to a newer version like GnuTLS 2.12.21 to solve the problem.

- **tls.compression** (not mandatory, values "on","off", default "off") The controls if the TLS stream should be compressed (zipped). While this increases CPU use, the network bandwidth should be reduced. Note that typical text-based log records usually compress rather well.

- **tls.permittedPeer** peer Places access restrictions on this forwarder. Only peers which have been listed in this parameter may be connected to. This guards against rouge servers and man-in-the-middle attacks. The validation bases on the certficate the remote peer presents.

  The *peer* parameter lists permitted certificate fingerprints. Note that it is an array parameter, so either a single or multiple fingerprints can be listed. When a non-permitted peer is connected to, the refusal is logged together with it's fingerprint. So if the administrator knows this was a valid request, he can simple add the fingerprint by copy and paste from the logfile to rsyslog.conf. It must be noted, though, that this situation should usually not happen after initial client setup and administrators should be alert in this case.

  Note that usually a single remote peer should be all that is ever needed. Support for multiple peers is primarily included in support of load balancing scenarios. If the connection goes to a specific server, only one specific certificate is ever expected (just like when connecting to a specific ssh server). To specify multiple fingerprints, just enclose them in braces like this:

```
tls.permittedPeer=["SHA1:...1", "SHA1:....2"]
```

  To specify just a single peer, you can either specify the string directly or enclose it in braces.

- **tls.authMode** mode Sets the mode used for mutual authentication. Supported values are either "*fingerprint*" or "*name*". Fingerprint mode basically is what SSH does. It does not require a full PKI to be present, instead self-signed certs can be used on all peers. Even if a CA certificate is given, the validity of the peer cert is NOT verified against it. Only the certificate fingerprint counts. In "name" mode, certificate validation happens. Here, the matching is done against the certificate's subjectAltName and, as a fallback, the subject common name. If the certificate contains multiple names, a match on any one of these names is considered good and permits the peer to talk to rsyslog.

- **tls.cacert** the CA certificate that can verify the machine certs

- **tls.mycert** the machine public certiificate

- **tls.myprivkey** the machine private key

- **tls.prioritystring** (not mandatory, string) This parameter permits to specify the so-called "priority string" to GnuTLS. This string gives complete control over all crypto parameters, including compression setting. For this reason, when the prioritystring is specified, the "tls.compression" parameter has no effect and is ignored. Full information about how to construct a priority string can be found in the GnuTLS manual. At the time of this writing, this information was contained in section 6.10 of the GnuTLS manual. **Note: this is an expert parameter.** Do not use if you do not exactly know what you are doing.

- **localclientip** ip_address (not mandatory, string) omrelp uses ip_address as local client address while connecting to remote logserver.

Sample:

The following sample sends all messages to the central server "centralserv" at port 2514 (note that that server must run imrelp on port 2514).

```
module(load="omrelp")
action(type="omrelp" target="centralserv" port="2514")
```

Legacy Configuration Directives:

This module uses old-style action configuration to keep consistent with the forwarding rule. So far, no additional configuration directives can be specified. To send a message via RELP, use

```
*.*  :omrelp:<server>:<port>;<template>
```

just as you use

```
*.*  @@<server>:<port>;<template>
```

to forward a message via plain tcp syslog.

Caveats/Known Bugs:

See *imrelp*, which documents them.

Legacy Sample:

The following sample sends all messages to the central server "centralserv" at port 2514 (note that that server must run imrelp on port 2514).

```
$ModLoad omrelp
*.* :omrelp:centralserv:2514
```

Note: to use IPv6 addresses, encode them in [::1] format.

omruleset: ruleset output/including module

Module Name: omruleset

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available Since: 5.3.4

Deprecated in: 7.2.0+

Deprecation note

This module exists only for backwards-compatibility reasons. **Do no longer use it in new configurations.** It has been replaced by the much more efficient "call" RainerScript statement. The "call" statement supports everything omruleset does, but in an easier to use way.

Description:

This is a very special "output" module. It permits to pass a message object to another rule set. While this is a very simple action, it enables very complex configurations, e.g. it supports high-speed "and" conditions, sending data to the same file in a non-racy way, include-ruleset functionality as well as some high-performance optimizations (in case the rule sets have the necessary queue definitions).

While it leads to a lot of power, this output module offers seamingly easy functionaltiy. The complexity (and capabilities) arise from how everthing can be combined.

With this module, a message can be sent to processing to another ruleset. This is somewhat similar to a "#include" in the C programming language. However, one needs to keep on the mind that a ruleset can contain its own queue and that a queue can run in various modes.

Note that if no queue is defined in the ruleset, the message is enqueued into the main message queue. This most often is not optimal and means that message processing may be severely defered. Also note that when the ruleset's target queue is full and no free space can be aquired within the usual timeout, the message object may actually be lost. This is an extreme scenario, but users building an audit-grade system need to know this restriction. For regular installations, it should not really be relevant.

At minimum, be sure you understand the *$RulesetCreateMainQueue* directive as well as the importance of statement order in rsyslog.conf before using omruleset!

Recommended Use:

- create rulesets specifically for omruleset
- create these rulesets with their own main queue
- decent queueing parameters (sizes, threads, etc) should be used for the ruleset main queue. If in doubt, use the same parameters as for the overall main queue.
- if you use multiple levels of ruleset nesting, double check for endless loops - the rsyslog engine does not detect these

Configuration Directives:

- **$ActionOmrulesetRulesetName** ruleset-to-submit-to This directive specifies the name of the ruleset that the message provided to omruleset should be submitted to. This ruleset must already have been defined. Note that the directive is automatically reset after each :omruleset: action and there is no default. This is done to prevent accidential loops in ruleset definition, what can happen very quickly. The :omruleset: action will NOT be honored if no ruleset name has been defined. As usual, the ruleset name must be specified in front of the action that it modifies.

Examples:

This example creates a ruleset for a write-to-file action. The idea here is that the same file is written based on multiple filters, problems occur if the file is used together with a buffer. That is because file buffers are action-specific, and so some partial buffers would be written. With omruleset, we create a single action inside its own ruleset and then pass all messages to it whenever we need to do so. Of course, such a simple situation could also be solved by a more complex filter, but the method used here can also be utilized in more complex scenarios (e.g. with multiple listeners). The example tries to keep it simple. Note that we create a ruleset-specific main queue (for simplicity with the default main queue parameters) in order to avoid re-queueing messages back into the main queue.

```
$ModLoad omruleset # define ruleset for commonly written file
$RuleSet CommonAction
$RulesetCreateMainQueue on
*.* /path/to/file.log

#switch back to default ruleset
$ruleset RSYSLOG_DefaultRuleset

# begin first action
# note that we must first specify which ruleset to use for omruleset:
$ActionOmrulesetRulesetName CommonAction
mail.info :omruleset:
# end first action

# begin second action
# note that we must first specify which ruleset to use for omruleset:
$ActionOmrulesetRulesetName CommonAction
```

```
:FROMHOST, isequal, "myhost.example.com" :omruleset:
#end second action

# of course, we can have "regular" actions alongside :omrulset: actions
*.* /path/to/general-message-file.log
```

The next example is used to creat a high-performance nested and filter condition. Here, it is first checked if the message contains a string "error". If so, the message is forwarded to another ruleset which then applies some filters. The advantage of this is that we can use high-performance filters where we otherwise would need to use the (much slower) expression-based filters. Also, this enables pipeline processing, in that second ruleset is executed in parallel to the first one.

```
$ModLoad omruleset
# define "second" ruleset
$RuleSet nested
$RulesetCreateMainQueue on
# again, we use our own queue
mail.* /path/to/mailerr.log
kernel.* /path/to/kernelerr.log
auth.* /path/to/autherr.log

#switch back to default ruleset
$ruleset RSYSLOG_DefaultRuleset

# begin first action - here we filter on "error"
# note that we must first specify which ruleset to use for omruleset:
$ActionOmrulesetRulesetName nested
:msg, contains, "error" :omruleset:
#end first action

# begin second action - as an example we can do anything else in
# this processing. Note that these actions are processed concurrently
# to the ruleset "nested"
:FROMHOST, isequal, "myhost.example.com" /path/to/host.log
#end second action

# of course, we can have "regular" actions alongside :omrulset: actions
*.* /path/to/general-message-file.log
```

Caveats/Known Bugs:

The current configuration file language is not really adequate for a complex construct like omruleset. Unfortunately, more important work is currently preventing me from redoing the config language. So use extreme care when nesting rulesets and be sure to test-run your config before putting it into production, ensuring you have a suffciently large probe of the traffic run over it. If problems arise, the rsyslog debug log is your friend.

This documentation is part of the rsyslog project.

omsnmp: SNMP Trap Output Module

Module Name: omsnmp

Author: Andre Lorbach <alorbach@adiscon.com>

Description:

Provides the ability to send syslog messages as an SNMPv1 & v2c traps. By default, SNMPv2c is preferred. The syslog message is wrapped into a OCTED STRING variable. This module uses the NET-SNMP library. In order to compile this module, you will need to have the NET-SNMP developer (headers) package installed.

Action Line:

%omsnmp% without any further parameters.

Configuration Directives:

- **$actionsnmptransport**(This parameter is optional, the default value is "udp")

  Defines the transport type you wish to use. Technically we can support all transport types which are supported by NET-SNMP. To name a few possible values: udp, tcp, udp6, tcp6, icmp, icmp6 ...

  Example: `$actionsnmptransport udp`

- **$actionsnmptarget**

  This can be a hostname or ip address, and is our snmp target host. This parameter is required, if the snmptarget is not defined, nothing will be send.

  Example: `$actionsnmptarget server.domain.xxx`

- **$actionsnmptargetport** (This parameter is optional, the default value is "162")

  The port which will be used, common values are port 162 or 161.

  Example: `$actionsnmptargetport 162`

- **$actionsnmpversion** (This parameter is optional, the default value is "1")

  There can only be two choices for this parameter for now. 0 means SNMPv1 will be used. 1 means SNMPv2c will be used. Any other value will default to 1.

  Example: `$actionsnmpversion 1`

- **$actionsnmpcommunity** (This parameter is optional, the default value is "public")

  This sets the used SNMP Community.

  Example: `$actionsnmpcommunity public`

- **$actionsnmptrapoid** (This parameter is optional, the default value is "1.3.6.1.4.1.19406.1.2.1" which means "ADISCON-MONITORWARE-MIB::syslogtrap")

  This configuration parameter is used for **SNMPv2** only. This is the OID which defines the trap-type, or notifcation-type rsyslog uses to send the trap. In order to decode this OID, you will need to have the ADISCON-MONITORWARE-MIB and ADISCON-MIB mibs installed on the receiver side. Downloads of these mib files can be found here:

  http://www.adiscon.org/download/ADISCON-MIB.txt

  http://www.adiscon.org/download/ADISCON-MONITORWARE-MIB.txt Thanks to the net-snmp mailinglist for the help and the recommendations ;).

  Example: `$actionsnmptrapoid 1.3.6.1.4.1.19406.1.2.1` If you have this MIBS installed, you can also configured with the OID Name: `$actionsnmptrapoid ADISCON-MONITORWARE-MIB::syslogtrap`

- **$actionsnmpsyslogmessageoid** (This parameter is optional, the default value is "1.3.6.1.4.1.19406.1.1.2.1" which means "ADISCON-MONITORWARE-MIB::syslogMsg")

  This OID will be used as a variable, type "OCTET STRING". This variable will contain up to 255 characters of the original syslog message including syslog header. It is recommend to use the default OID. In order to decode

this OID, you will need to have the ADISCON-MONITORWARE-MIB and ADISCON-MIB mibs installed on the receiver side. To download these custom mibs, see the description of **$actionsnmptrapoid.**

Example: `$actionsnmpsyslogmessageoid 1.3.6.1.4.1.19406.1.1.2.1` If you have this MIBS installed, you can also configured with the OID Name: `$actionsnmpsyslogmessageoid` ADISCON-MONITORWARE-MIB::syslogMsg

- **$actionsnmpenterpriseoid** (This parameter is optional, the default value is "1.3.6.1.4.1.3.1.1" which means "enterprises.cmu.1.1")

  Customize this value if needed. I recommend to use the default value unless you require to use a different OID. This configuration parameter is used for **SNMPv1** only. It has no effect if **SNMPv2** is used.

  Example: `$actionsnmpenterpriseoid 1.3.6.1.4.1.3.1.1`

- **$actionsnmpspecifictype** (This parameter is optional, the default value is "0")

  This is the specific trap number. This configuration parameter is used for **SNMPv1** only. It has no effect if **SNMPv2** is used.

  Example: `$actionsnmpspecifictype 0`

- **$actionsnmptraptype** (This parameter is optional, the default value is "6" which means SNMP_TRAP_ENTERPRISESPECIFIC)

  There are only 7 Possible trap types defined which can be used here. These trap types are:

```
0 = SNMP_TRAP_COLDSTART
1 = SNMP_TRAP_WARMSTART
2 = SNMP_TRAP_LINKDOWN
3 = SNMP_TRAP_LINKUP
4 = SNMP_TRAP_AUTHFAIL
5 = SNMP_TRAP_EGPNEIGHBORLOSS
6 = SNMP_TRAP_ENTERPRISESPECIFIC
```

  Any other value will default to 6 automatically. This configuration parameter is used for **SNMPv1** only. It has no effect if **SNMPv2** is used.

  Example: `$actionsnmptraptype 6`

Caveats/Known Bugs:

- In order to decode the custom OIDs, you will need to have the adiscon mibs installed.

Sample:

The following commands send every message as a snmp trap.

```
$ModLoad omsnmp
$actionsnmptransport udp
$actionsnmptarget localhost
$actionsnmptargetport 162
$actionsnmpversion 1
$actionsnmpcommunity public
*.* :omsnmp:
```

This documentation is part of the rsyslog project.

omstdout: stdout output module (testbench tool)

Module Name: omstdout

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available Since: 4.1.6

Description:

This module writes any messages that are passed to it to stdout. It was developed for the rsyslog test suite. However, there may (limited) other uses exists. Please not that we do not put too much effort into the quality of this module as we do not expect it to be used in real deployments. If you do, please drop us a note so that we can enhance its priority!

Configuration Directives:

- **$ActionOMStdoutArrayInterface** [on|**off** This setting instructs omstdout to use the alternate array based method of parameter passing. If used, the values will be output with commas between the values but no other padding bytes. This is a test aid for the alternate calling interface.
- **$ActionOMStdoutEnsureLFEnding** [**on**|off Makes sure that each message is written with a terminating LF. This is needed for the automatted tests. If the message contains a trailing LF, none is added.

Caveats/Known Bugs:

Currently none known.

This documentation is part of the rsyslog project.

omudpspoof: UDP spoofing output module

Module Name: omstdout

Author: David Lang <david@lang.hm> and Rainer Gerhards <rgerhards@adiscon.com>

Available Since: 5.1.3

Description:

This module is similar to the regular UDP forwarder, but permits to spoof the sender address. Also, it enables to circle through a number of source ports.

Module Parameters

- **template** <templatename>

  This setting instructs omudpspoof to use a template different from the default template for all of its actions that do not have a template specified explicitely.

Action Parameters

- **target** <hostname> or <ip>

  Host that the messages shall be sent to.

- **port** <port>

  Remote port that the messages shall be sent to. Default is 514.

- **sourcetemplate** <templatename>

  This is the name of the template that contains a numerical IP address that is to be used as the source system IP address. While it may often be a constant value, it can be generated as usual via the property replacer, as long as it is a valid IPv4 address. If not specified, the build-in default template RSYSLOG_omudpspoofDfltSourceTpl is used. This template is defined as follows: $template RSYSLOG_omudpspoofDfltSourceTpl,"%fromhost-ip%" So in essence, the default template spoofs the address of the system the message was received from. This is considered the most important use case.

- **sourceport.start** <port>

- **sourceport.end** <port>

  Specify the start and end value for circling the source ports. Start must be less than or equal to the end value. Defaults are 32000 and 42000.

- **template** <templatename>

  This setting instructs omudpspoof to use a template different from the default template for all of its actions that do not have a template specified explicitely.

- **mtu** <mtu>

  Maximum packet length to send, default is 1500.

Legacy Configuration Directives

- **$ActionOMOMUDPSpoofSourceNameTemplate** <templatename> This is the name of the template that contains a numerical IP address that is to be used as the source system IP address. While it may often be a constant value, it can be generated as usual via the property replacer, as long as it is a valid IPv4 address. If not specified, the build-in default template RSYSLOG_omudpspoofDfltSourceTpl is used. This template is defined as follows: $template RSYSLOG_omudpspoofDfltSourceTpl,"%fromhost-ip%" So in essence, the default template spoofs the address of the system the message was received from. This is considered the most important use case.
- **$ActionOMUDPSpoofTargetHost** <hostname> Host that the messages shall be sent to.
- **$ActionOMUDPSpoofTargetPort** <port> Remote port that the messages shall be sent to.
- **$ActionOMUDPSpoofDefaultTemplate** <templatename> This setting instructs omudpspoof to use a template different from the default template for all of its actions that do not have a template specified explicitely.
- **$ActionOMUDPSpoofSourcePortStart** <number> Specifies the start value for circeling the source ports. Must be less than or equal to the end value. Default is 32000.
- **$ActionOMUDPSpoofSourcePortEnd** <number> Specifies the ending value for circeling the source ports. Must be less than or equal to the start value. Default is 42000.

Caveats/Known Bugs

- **IPv6** is currently not supported. If you need this capability, please let us know via the rsyslog mailing list.
- Throughput is MUCH smaller than when using omfwd module.

Sample

Forward the message to 192.168.1.1, using original source and port between 10000 and 19999.

```
Action (
  type="omudpspoof"
  target="192.168.1.1"
  sourceport.start="10000"
  sourceport.end="19999"
)
```

Forward the message to 192.168.1.1, using source address 192.168.111.111 and default ports.

```
Module (
  load="omudpspoof"
)
Template (
  name="spoofaddr"
  type="string"
  string="192.168.111.111"
)
Action (
  type="omudpspoof"
  target="192.168.1.1"
  sourcetemplate="spoofaddr"
)
```

Legacy Sample

The following sample forwards all syslog messages in standard form to the remote server server.example.com. The original sender's address is used. We do not care about the source port. This example is considered the typical use case for omudpspoof.

```
$ModLoad omudpspoof $ActionOMUDPSpoofTargetHost server.example.com
*.*      :omudpspoof:
```

The following sample forwards all syslog messages in unmodified form to the remote server server.example.com. The sender address 192.0.2.1 with fixed source port 514 is used.

```
$ModLoad omudpspoof $template spoofaddr,"192.0.2.1" $template
spooftemplate,"%rawmsg%" $ActionOMUDPSpoofSourceNameTemplate spoofaddr
$ActionOMUDPSpoofTargetHost server.example.com
$ActionOMUDPSpoofSourcePortStart 514 $ActionOMUDPSpoofSourcePortEnd 514
*.*      :omudpspoof:;spooftemplate
```

The following sample is similar to the previous, but uses as many defaults as possible. In that sample, a source port in the range 32000..42000 is used. The message is formatted according to rsyslog's canned default forwarding format. Note that if any parameters have been changed, the previously set defaults will be used!

```
$ModLoad omudpspoof $template spoofaddr,"192.0.2.1"
$ActionOMUDPSpoofSourceNameTemplate spoofaddr
$ActionOMUDPSpoofTargetHost server.example.com
*.*      :omudpspoof:
```

This documentation is part of the rsyslog project.

omusrmsg: notify users

Module Name: omusrmsg

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module permits to send log messages to the user terminal.

Module Parameters:

none

Action Parameters:

- **users** the name of the users to send data to.
- **template** template to user for the message.

Caveats/Known Bugs:

- None.

Sample:

The following command writes emergency messages to all users:

```
action(type="omusrmsg" users="*")
```

omuxsock: Unix sockets Output Module

Module Name: omuxsock

Available since: 4.7.3, 5.5.7

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module supports sending syslog messages to local Unix sockets. Thus it provided a fast message-passing interface between different rsyslog instances. The counterpart to omuxsock is imuxsock. Note that the template used together with omuxsock must be suitable to be processed by the receiver.

Configuration Directives:

- **$OMUxSockSocket** Name of the socket to send data to. This has no default and **must** be set.
- **$OMUxSockDefaultTemplate** This can be used to override the default template to be used together with omuxsock. This is primarily useful if there are many forwarding actions and each of them should use the same template.

Caveats/Known Bugs:

Currently, only datagram sockets are supported.

Sample:

The following sample writes all messages to the "/tmp/socksample" socket.

```
$ModLoad omuxsock
$OMUxSockSocket /tmp/socksample
*.* :omuxsock:
```

GuardTime Log Signature Provider (gt)

Signature Provider Name: gt

Author: Rainer Gerhards <rgerhards@adiscon.com>

Supported: from 7.3.9 to 8.26.0

Description:

Provides the ability to sign syslog messages via the GuardTime signature services.

Configuration Parameters:

Signature providers are loaded by omfile, when the provider is selected in its "sig.providerName" parameter. Parameters for the provider are given in the omfile action instance line.

This provider creates a signature file with the same base name but the extension ".gtsig" for each log file (both for fixed-name files as well as dynafiles). Both files together form a set. So you need to archive both in order to prove integrity.

- **sig.hashFunction** <Hash Algorithm> The following hash algorithms are currently supported:
  - SHA1
  - RIPEMD-160
  - SHA2-224
  - SHA2-256
  - SHA2-384
  - SHA2-512
- **sig.timestampService** <timestamper URL> This provides the URL of the timestamper service. If not selected, a default server is selected. This may not necessarily be a good one for your region.

  Note: If you need to supply user credentials, you can add them to the timestamper URL. If, for example, you have a user "user" with password "pass", you can do so as follows:

  http://user:pass@timestamper.example.net
- **sig.block.sizeLimit** <nbr-records> The maximum number of records inside a single signature block. By default, there is no size limit, so the signature is only written on file closure. Note that a signature request typically takes between one and two seconds. So signing to frequently is probably not a good idea.
- **sig.keepRecordHashes** <on/**off**> Controls if record hashes are written to the .gtsig file. This enhances the ability to spot the location of a signature breach, but costs considerable disk space (65 bytes for each log record for SHA2-512 hashes, for example).
- **sig.keepTreeHashes** <on/**off**> Controls if tree (intermediate) hashes are written to the .gtsig file. This enhances the ability to spot the location of a signature breach, but costs considerable disk space (a bit mire than the amount sig.keepRecordHashes requries). Note that both Tree and Record hashes can be kept inside the signature file.

See Also

- How to sign log messages through signature provider Guardtime

Caveats/Known Bugs:

- currently none known

Samples:

This writes a log file with it's associated signature file. Default parameters are used.

```
action(type="omfile" file="/var/log/somelog" sig.provider="gt")
```

In the next sample, we use the more secure SHA2-512 hash function, sign every 10,000 records and Tree and Record hashes are kept.

```
action(type="omfile" file="/var/log/somelog" sig.provider="gt"
sig.hashfunction="SHA2-512" sig.block.sizelimit="10000"
sig.keepTreeHashes="on" sig.keepRecordHashes="on")
```

Keyless Signature Infrastructure Provider (ksi)

Signature Provider Name: ksi

Author: Rainer Gerhards <rgerhards@adiscon.com>

Supported: from 8.11.0 to 8.26.0

Description:

Provides the ability to sign syslog messages via the GuardTime KSI signature services.

Configuration Parameters:

Signature providers are loaded by omfile, when the provider is selected in its "sig.providerName" parameter. Parameters for the provider are given in the omfile action instance line.

This provider creates a signature file with the same base name but the extension ".ksisig" for each log file (both for fixed-name files as well as dynafiles). Both files together form a set. So you need to archive both in order to prove integrity.

- **sig.hashFunction** <Hash Algorithm> The following hash algorithms are currently supported:
  - SHA1
  - SHA2-256
  - RIPEMD-160
  - SHA2-224
  - SHA2-384
  - SHA2-512
  - RIPEMD-256
  - SHA3-244
  - SHA3-256
  - SHA3-384
  - SHA3-512
  - SM3
- **sig.aggregator.uri** <KSI Aggregator URL> This provides the URL of the KSI Aggregator service provided by guardtime and looks like this:

ksi+tcp://[ip/dnsname]:3332

- **sig.aggregator.user** <KSI UserID> Set your username provided by Guardtime here.
- **sig.aggregator.key** <KSI Key / Password> Set your key provided by Guardtime here.
- **sig.block.sizeLimit** <nbr-records> The maximum number of records inside a single signature block. By default, there is no size limit, so the signature is only written on file closure. Note that a signature request typically takes between one and two seconds. So signing to frequently is probably not a good idea.
- **sig.keepRecordHashes** <on/**off**> Controls if record hashes are written to the .gtsig file. This enhances the ability to spot the location of a signature breach, but costs considerable disk space (65 bytes for each log record for SHA2-512 hashes, for example).
- **sig.keepTreeHashes** <on/**off**> Controls if tree (intermediate) hashes are written to the .gtsig file. This enhances the ability to spot the location of a signature breach, but costs considerable disk space (a bit mire than the amount sig.keepRecordHashes requries). Note that both Tree and Record hashes can be kept inside the signature file.

See Also

Caveats/Known Bugs:

- currently none known

Samples:

This writes a log file with it's associated signature file. Default parameters are used.

```
action(type="omfile" file="/var/log/somelog" sig.provider="ksi")
```

In the next sample, we use the more secure SHA2-512 hash function, sign every 10,000 records and Tree and Record hashes are kept.

```
action(type="omfile" file="/var/log/somelog" sig.provider="ksi"
sig.hashfunction="SHA2-512" sig.block.sizelimit="10000"
sig.keepTreeHashes="on" sig.keepRecordHashes="on")
```

Keyless Signature Infrastructure Provider (rsyslog-ksi-ls12)

Module Name: rsyslog-ksi-ls12

Available Since: 8.27

Author: Guardtime & Adiscon

Description

The `rsyslog-ksi-ls12` module enables record level log signing with Guardtime KSI blockchain. KSI keyless signatures provide long-term log integrity and prove the time of log records cryptographically using independent verification.

Main features of the `rsyslog-ksi-ls12` module are:

- Automated online signing of file output log.
- Efficient block-based signing with record-level verification.
- Log records removal detection.

For best results use the `rsyslog-ksi-ls12` module together with Guardtime `logksi` tool, which will become handy in:

- Signing recovery.
- Extension of KSI signatures inside the log signature file.
- Verification of the log using log signatures.
- Extraction of record-level signatures.
- Integration of log signature files (necessary when signing in async mode).

Getting Started

To get started with log signing:

- Sign up to the Guardtime tryout service to be able to connect to KSI blockchain: guardtime.com/technology/blockchain-developers
- Install the `libksi` library (v3.13 or later) from Guardtime repository either for RHEL/CentOS 6 http://download.guardtime.com/ksi/configuration/guardtime.el6.repo or RHEL/CentOS 7 http://download.guardtime.com/ksi/configuration/guardtime.el7.repo
- Install the `rsyslog-ksi-ls12` module (same version as rsyslog) from Adiscon repository.
- Install the accompanying `logksi` tool (v1.0 or later) from Guardtime repository either for RHEL/CentOS 6 http://download.guardtime.com/ksi/configuration/guardtime.el6.repo or RHEL/CentOS 7 http://download.guardtime.com/ksi/configuration/guardtime.el7.repo

Currently the log signing is only supported by the file output module, thus the action type must be `omfile`. To activate signing, add the following parameters to the action of interest in your rsyslog configuration file:

Mandatory parameters (no default value defined):

- **sig.provider** specifies the signature provider; in case of `rsyslog-ksi-ls12` package this is `"ksi_ls12"`.
- **sig.block.levelLimit** defines the maximum level of the root of the local aggregation tree per one block.
- **sig.aggregator.url** defines the endpoint of the KSI signing service in KSI Gateway. Supported URI schemes are:
  - *http://*
  - *ksi+http://*
  - *ksi+tcp://*
- **sig.aggregator.user** specifies the login name for the KSI signing service.
- **sig.aggregator.key** specifies the key for the login name.

Optional parameters (if not defined, default value is used):

- **sig.syncmode** defines the signing mode: `"sync"` (default) or `"async"`.
- **sig.hashFunction** defines the hash function to be used for hashing, default is `"SHA2-256"`. Other SHA-2, as well as RIPEMED-160 functions are supported.
- **sig.block.timeLimit** defines the maximum duration of one block in seconds. Default value `"0"` indicates that no time limit is set.
- **sig.keepTreeHashes** turns on/off the storing of the hashes that were used as leaves for building the Merkle tree, default is `"off"`.
- **sig.keepRecordHashes** turns on/off the storing of the hashes of the log records, default is `"on"`.

The log signature file, which stores the KSI signatures and information about the signed blocks, appears in the same directory as the log file itself; it is named `<logfile>.logsig`.

Sample

To sign the logs in `/var/log/secure` with KSI:

```
# The authpriv file has restricted access and is signed with KSI
authpriv.*    action(type="omfile" file="/var/log/secure"
  sig.provider="ksi_ls12"
  sig.syncmode="sync"
  sig.hashFunction="SHA2-256"
  sig.block.levelLimit="8"
  sig.block.timeLimit="0"
  sig.aggregator.url=
    "http://tryout.guardtime.net:8080/gt-signingservice"
  sig.aggregator.user="rsmith"
  sig.aggregator.key="secret"
  sig.keepTreeHashes="off"
  sig.keepRecordHashes="on")
```

Note that all parameter values must be between quotation marks!

See Also

To better understand the log signing mechanism and the module's possibilities it is advised to consult with:

- KSI Rsyslog Integration User Guide
- KSI Developer Guide

Access for both of these documents requires Guardtime tryout service credentials, available from https://guardtime.com/technology/blockchain-developers

Input Modules

Input modules are used to gather messages from various sources. They interface to message generators. They are generally defined via the *input* configuration object.

im3195: RFC3195 Input Module

Receives syslog messages via RFC 3195. The RAW profile is fully implemented and the COOKED profile is provided in an experimental state. This module uses liblogging for the actual protocol handling.

Author: Rainer Gerhards <rgerhards@adiscon.com>

Configuration Directives

`$Input3195ListenPort <port>`
The port on which imklog listens for RFC 3195 messages. The default port is 601 (the IANA-assigned port)

Caveats/Known Bugs

Due to no demand at all for RFC3195, we have converted rfc3195d to this input module, but we have NOT conducted any testing. Also, the module does not yet properly handle the recovery case. If someone intends to put this module into production, good testing should be conducted. It also is a good idea to notify the rsyslog project that you intend to use it in production. In this case, we'll probably give the module another cleanup. We don't do this now because so far it looks just like a big waste of time.

Currently only a single listener can be defined. That one binds to all interfaces.

Example

The following sample accepts syslog messages via RFC 3195 on port 1601.

```
$ModLoad im3195 $Input3195ListenPort 1601
```

imfile: Text File Input Module

Module Name:	**imfile**
Author:	Rainer Gerhards <rgerhards@adiscon.com>

This module provides the ability to convert any standard text file into a syslog message. A standard text file is a file consisting of printable characters with lines being delimited by LF.

The file is read line-by-line and any line read is passed to rsyslog's rule engine. The rule engine applies filter conditions and selects which actions needs to be carried out. Empty lines are **not** processed, as they would result in empty syslog records. They are simply ignored.

As new lines are written they are taken from the file and processed. Depending on the selected mode, this happens via inotify or based on a polling interval. Especially in polling mode, file reading doesn't happen immediately. But there are also slight delays (due to process scheduling and internal processing) in inotify mode.

The file monitor supports file rotation. To fully work, rsyslogd must run while the file is rotated. Then, any remaining lines from the old file are read and processed and when done with that, the new file is being processed from the beginning. If rsyslogd is stopped during rotation, the new file is read, but any not-yet-reported lines from the previous file can no longer be obtained.

When rsyslogd is stopped while monitoring a text file, it records the last processed location and continues to work from there upon restart. So no data is lost during a restart (except, as noted above, if the file is rotated just in this very moment).

See Also

- presentation on using wildcards with imfile

Metadata

The imfile module supports message metadata. It supports the following data items

- filename

  Name of the file where the message originated from. This is most useful when using wildcards inside file monitors, because it then is the only way to know which file the message originated from. The value can be accessed using the %$!metadata!filename% property.

Metadata is only present if enabled. By default it is enabled for input() statements that contain wildcards. For all others, it is disabled by default. It can explicitly be turned on or off via the *addMetadata* input() parameter, which always overrides the default.

State Files

Rsyslog must keep track of which parts of the monitored file are already processed. This is done in so-called "state files". These files are always created in the rsyslog working directory (configurable via $WorkDirectory).

To avoid problems with duplicate state files, rsyslog automatically generates state file names according to the following scheme:

- the string "imfile-state:" is added before the actual file name, which includes the full path
- the full name is prepended after that string, but all occurrences of "/" are replaced by "-" to facilitate handling of these files

As a concrete example, consider file `/var/log/applog` is being monitored. The corresponding state file will be named `imfile-state:-var-log-applog`.

Note that it is possible to set a fixed state file name via the deprecated "stateFile" parameter. It is suggested to avoid this, as the user must take care of name clashes. Most importantly, if "stateFile" is set for file monitors with wildcards, the **same** state file is used for all occurrences of these files. In short, this will usually not work and cause confusion. Upon startup, rsyslog tries to detect these cases and emit warning messages. However, the detection simply checks for the presence of "*" and as such it will not cover more complex cases.

Note that when $WorkDirectory is not set or set to a non-writable location, the state file **will not be generated**. In those cases, the file content will always be completely re-sent by imfile, because the module does not know that it already processed parts of that file.

Module Parameters

mode ["inotify"/"polling"]

Default: "inotify"

Available since: 8.1.5

This specifies if imfile is shall run in inotify ("inotify") or polling ("polling") mode. Traditionally, imfile used polling mode, which is much more resource-intense (and slower) than inotify mode. It is suggested that users turn on "polling" mode only if they experience strange problems in inotify mode. In theory, there should never be a reason to enable "polling" mode and later versions will most probably remove it.

Note: if a legacy "$ModLoad" statement is used, the default is *polling*. This default was kept to prevent problems with old configurations. It might change in the future.

readTimeout [seconds]

Default: 0 (no timeout)

Available since: 8.23.0

This sets the default value for input *timeout* parameters. See there for exact meaning.

timeoutGranularity [seconds]

Default: 1

Available since: 8.23.0

This sets the interval in which multi-line-read timeouts are checked. The interval is specified in seconds. Note that this establishes a lower limit on the length of the timeout. For example, if a timeoutGranularity of 60 seconds is selected and a readTimeout value of 10 seconds is used, the timeout is nevertheless only checked every 60 seconds (if there is no other activity in imfile). This means that the readTimeout is also only checked every 60 seconds, which in turn means a timeout can occur only after 60 seconds.

Note that timeGranularity has some performance implication. The more frequently timeout processing is triggerred, the more processing time is needed. This effect should be neglectible, except if a very large number of files is being monitored.

PollingInterval seconds
Default: 10

This setting specifies how often files are to be polled for new data. For obvious reasons, it has effect only if imfile is running in polling mode. The time specified is in seconds. During each polling interval, all files are processed in a round-robin fashion.

A short poll interval provides more rapid message forwarding, but requires more system resources. While it is possible, we stongly recommend not to set the polling interval to 0 seconds. That will make rsyslogd become a CPU hog, taking up considerable resources. It is supported, however, for the few very unusual situations where this level may be needed. Even if you need quick response, 1 seconds should be well enough. Please note that imfile keeps reading files as long as there is any data in them. So a "polling sleep" will only happen when nothing is left to be processed.

We recommend to use inotify mode.

Input Parameters

File [/path/to/file]
(Required Parameter) The file being monitored. So far, this must be an absolute name (no macros or templates). Note that wildcards are supported at the file name level (see **WildCards** below for more details).

Tag [tag:]
(Required Parameter) The tag to be used for messages that originate from this file. If you would like to see the colon after the tag, you need to specify it here (like 'tag="myTagValue:"').

Facility [facility]
The syslog facility to be assigned to lines read. Can be specified in textual form (e.g. "local0", "local1", ...) or as numbers (e.g. 16 for "local0"). Textual form is suggested. Default is "local0".

Severity [syslogSeverity]
The syslog severity to be assigned to lines read. Can be specified in textual form (e.g. "info", "warning", ...) or as numbers (e.g. 6 for "info"). Textual form is suggested. Default is "notice".

PersistStateInterval [lines]
Specifies how often the state file shall be written when processing the input file. The **default** value is 0, which means a new state file is only written when the monitored files is being closed (end of rsyslogd execution). Any other value n means that the state file is written every time n file lines have been processed. This setting can be used to guard against message duplication due to fatal errors (like power fail). Note that this setting affects imfile performance, especially when set to a low value. Frequently writing the state file is very time consuming.

startmsg.regex [POSIX ERE regex]
This permits the processing of multi-line messages. When set, a messages is terminated when the next one begins, and `startmsg.regex` contains the regex that identifies the start of a message. As this parameter is using regular expressions, it is more flexible than `readMode` but at the cost of lower performance. Note that `readMode` and `startmsg.regex` cannot both be defined for the same input.

This parameter is available since rsyslog v8.10.0.

readTimeout [seconds]

Default: 0 (no timeout)

Available since: 8.23.0

This can be used with *startmsg.regex* (but not *readMode*). If specified, partial multi-line reads are timed out after the specified timeout interval. That means the current message fragment is being processed and the next message fragment arriving is treated as a completely new message. The typical use case for this parameter is a file that is infrequently being written. In such cases, the next message arrives relatively late, maybe hours later. Specifying a readTimeout will ensure that those "last messages" are emitted in a timely manner. In this use case, the "partial" messages being processed are actually full messages, so everything is fully correct.

To guard against accidential too-early emission of a (partial) message, the timeout should be sufficiently large (5 to 10 seconds or more recommended). Specifying a value of zero turns off timeout processing. Also note the relationship to the *timeoutGranularity* global parameter, which sets the lower bound of *readTimeout*.

Setting timeout vaues slightly increases processing time requirements; the effect should only be visible of a very large number of files is being monitored.

readMode [mode]

This provides support for processing some standard types of multiline messages. It is less flexible than `startmsg.regex` but offers higher performance than regex processing. Note that `readMode` and `startmsg.regex` cannot both be defined for the same input.

The value can range from 0-2 and determines the multiline detection method.

0 - (**default**) line based (each line is a new message)

1 - paragraph (There is a blank line between log messages)

2 - indented (new log messages start at the beginning of a line. If a line starts with a space or tab "t" it is part of the log message before it)

escapeLF [on/off] (requires v7.5.3+)

This is only meaningful if multi-line messages are to be processed. LF characters embedded into syslog messages cause a lot of trouble, as most tools and even the legacy syslog TCP protocol do not expect these. If set to "on", this option avoid this trouble by properly escaping LF characters to the 4-byte sequence "#012". This is consistent with other rsyslog control character escaping. By default, escaping is turned on. If you turn it off, make sure you test very carefully with all associated tools. Please note that if you intend to use plain TCP syslog with embedded LF characters, you need to enable octet-counted framing. For more details, see Rainer's blog posting on imfile LF escaping.

MaxLinesAtOnce [number]

This is a legacy setting that only is supported in *polling* mode. In *inotify* mode, it is fixed at 0 and all attempts to configure a different value will be ignored, but will generate an error message.

Please note that future versions of imfile may not support this parameter at all. So it is suggested to not use it.

In *polling* mode, if set to 0, each file will be fully processed and then processing switches to the next file. If it is set to any other value, a maximum of [number] lines is processed in sequence for each file, and then the file is switched. This provides a kind of mutiplexing the load of multiple files and probably leads to a more natural distribution of events when multiple busy files are monitored. For *polling* mode, the **default** is 10240.

MaxSubmitAtOnce [number]

This is an expert option. It can be used to set the maximum input batch size that imfile can generate. The **default** is 1024, which is suitable for a wide range of applications. Be sure to understand rsyslog message batch processing before you modify this option. If you do not know what this doc here talks about, this is a good indication that you should NOT modify the default.

deleteStateOnFileDelete [on/off] (requires v8.5.0+)

Default: on

This parameter controls if state files are deleted if their associated main file is deleted. Usually, this is a good idea, because otherwise problems would occur if a new file with the same name is created. In that case, imfile would pick up reading from the last position in the **deleted** file, which usually is not what you want.

However, there is one situation where not deleting associated state file makes sense: this is the case if a monitored file is modified with an editor (like vi or gedit). Most editors write out modifications by deleting the old file and creating a new now. If the state file would be deleted in that case, all of the file would be reprocessed, something that's probably not intended in most case. As a side-note, it is strongly suggested *not* to modify monitored files with editors. In any case, in such a situation, it makes sense to disable state file deletion. That also applies to similar use cases.

In general, this parameter should only by set if the users knows exactly why this is required.

Ruleset <ruleset>
Binds the listener to a specific *ruleset*.

addMetadata [on/off]
Default: see intro section on Metadata

This is used to turn on or off the addition of metadata to the message object.

addCeeTag [on/off]
Default: off

This is used to turn on or off the addition of the "@cee:" cookie to the message object.

stateFile [name-of-state-file]
Default: unset

This paramater is deprecated. It still is accepted, but should no longer be used for newly created configurations.

This is the name of this file's state file. This parameter should usually **not** be used. Check the section on "State Files" above for more details.

reopenOnTruncate [on/off] (requires v8.16.0+)
Default: off

This is an **experimental** feature that tells rsyslog to reopen input file when it was truncated (inode unchanged but file size on disk is less than current offset in memory).

trimLineOverBytes [number] (requires v8.17.0+)
Default: 0

This is used to tell rsyslog to truncate the line which length is greater than specified bytes. If it is positive number, rsyslog truncate the line at specified bytes. Default value of 'trimLineOverBytes' is 0, means never truncate line.

This option can be used when `readMode` is 0 or 2.

freshStartTail [on/off] (requires v8.18.0+)
Default: off

This is used to tell rsyslog to seek to the end/tail of input files (discard old logs) **at its first start(freshStart)** and process only new log messages.

When deploy rsyslog to a large number of servers, we may only care about new log messages generated after the deployment. set **freshstartTail** to **on** will discard old logs. Otherwise, there may be vast useless message burst on the remote central log receiver

discardTruncatedMsg <on/off>
Default: off

When messages are too long they are truncated and the following part is processed as a new message. When this parameter is turned on the truncated part is not processed but discarded.

msgDiscardingError <on/off>
Default: on

Upon truncation an error is given. When this parameter is turned off, no error will be shown upon truncation.

WildCards

Before Version: 8.25.0 Wildcards are only supported in the filename part, not in directory names.

- /var/log/*.log **works**. *
- /var/log/*/syslog.log does **not work**. *

Since Version: 8.25.0 Wildcards are supported in filename and pathes which means these samples will work:

- /var/log/*.log **works**. *
- /var/log/*/syslog.log **works**. *
- /var/log/*/*.log **works**. *

All matching files in all matching subfolders will work. Note that this may derease performance in imfile depending on how many directories and files are being watched dynamically.

Caveats/Known Bugs

- currently, wildcards are only supported in inotify mode
- read modes other than "0" currently seem to have issues in inotify mode

Configuration Example

The following sample monitors two files. If you need just one, remove the second one. If you need more, add them according to the sample ;). This code must be placed in /etc/rsyslog.conf (or wherever your distro puts rsyslog's config files). Note that only commands actually needed need to be specified. The second file uses less commands and uses defaults instead.

```
module(load="imfile" PollingInterval="10") #needs to be done just once

# File 1
input(type="imfile"
      File="/path/to/file1"
      Tag="tag1"
      Severity="error"
      Facility="local7")

# File 2
input(type="imfile"
      File="/path/to/file2"
      Tag="tag2")

# ... and so on ... #
```

Legacy Configuration

Note: in order to preserve compatibility with previous versions, the LF escaping in multi-line messages is turned off for legacy-configured file monitors (the "escapeLF" input parameter). This can cause serious problems. So it is highly suggested that new deployments use the new *input()* configuration object and keep LF escaping turned on.

Legacy Configuration Directives

$InputFileName /path/to/file
equivalent to "file"

$InputFileTag tag:
equivalent to: "tag" you would like to see the colon after the tag, you need to specify it here (as shown above).

$InputFileStateFile name-of-state-file
equivalent to: "StateFile"

$InputFileFacility facility
equivalent to: "Facility"

$InputFileSeverity severity
equivalent to: "Severity"

$InputRunFileMonitor
This **activates** the current monitor. It has no parameters. If you forget this directive, no file monitoring will take place.

$InputFilePollInterval seconds
equivalent to: "PollingInterval"

$InputFilePersistStateInterval lines
equivalent to: "PersistStateInterval"

$InputFileReadMode mode
equivalent to: "ReadMode"

$InputFileMaxLinesAtOnce number
equivalent to: "MaxLinesAtOnce"

$InputFileBindRuleset ruleset
Equivalent to: Ruleset

Legacy Example

The following sample monitors two files. If you need just one, remove the second one. If you need more, add them according to the sample ;). Note that only non-default parameters actually needed need to be specified. The second file uses less directives and uses defaults instead.

```
$ModLoad imfile # needs to be done just once
# File 1
$InputFileName /path/to/file1
$InputFileTag tag1:
$InputFileStateFile stat-file1

$InputFileSeverity error
$InputFileFacility local7
$InputRunFileMonitor
```

```
# File 2
$InputFileName /path/to/file2
$InputFileTag tag2:

$InputFileStateFile stat-file2
$InputRunFileMonitor
# ... and so on ...
# check for new lines every 10 seconds
$InputFilePollInterval 10
```

imgssapi: GSSAPI Syslog Input Module

Provides the ability to receive syslog messages from the network protected via Kerberos 5 encryption and authentication. This module also accept plain tcp syslog messages on the same port if configured to do so. If you need just plain tcp, use *imtcp* instead.

Note: This is a contributed module, which is not supported by the rsyslog team. We recommend to use RFC5425 TLS-protected syslog instead.

Author:varmojfekoj

GSSAPI module support in rsyslog v3

What is it good for.

- client-serverauthentication
- Log messages encryption

Requirements.

- Kerberos infrastructure
- rsyslog, rsyslog-gssapi

Configuration.

Let's assume there are 3 machines in kerberos Realm:

- the first is running KDC (Kerberos Authentication Service and Key Distribution Center),
- the second is a client sending its logs to the server,
- the third is receiver, gathering all logs.

1. KDC:

- Kerberos database must be properly set-up on KDC machine first. Use kadmin/kadmin.local to do that. Two principals need to be add in our case:

1. sender@REALM.ORG

- client must have ticket for pricipal sender
- REALM.ORG is kerberos Realm

1. host/receiver.mydomain.com@REALM.ORG - service principal

- Use ktadd to export service principal and transfer it to /etc/krb5.keytab on receiver

2. CLIENT:

- set-up rsyslog, in /etc/rsyslog.conf
- $ModLoad omgssapi - load output gss module
- $GSSForwardServiceName otherThanHost - set the name of service principal, "host" is the default one
- *.* :omgssapi:receiver.mydomain.com - action line, forward logs to receiver
- kinit root - get the TGT ticket
- service rsyslog start

3. SERVER:

- set-up rsyslog, in /etc/rsyslog.conf
- $ModLoad imgssapi - load input gss module
- $InputGSSServerServiceName otherThanHost - set the name of service principal, "host" is the default one
- $InputGSSServerPermitPlainTCP on - accept GSS and TCP connections (not authenticated senders), off by default
- $InputGSSServerRun 514 - run server on port
- service rsyslog start

The picture demonstrate how things work.

This documentation is part of the rsyslog project.

Configuration Directives

$InputGSSServerRun <port>
Starts a GSSAPI server on selected port - note that this runs independently from the TCP server.

$InputGSSServerServiceName <name>
The service name to use for the GSS server.

$InputGSSServerPermitPlainTCP on|off
Permits the server to receive plain tcp syslog (without GSS) on the same port

$InputGSSServerMaxSessions <number>
Sets the maximum number of sessions supported

$InputGSSServerKeepAlive on|off
Requires: 8.5.0 or above

Default: off

Enables or disable keep-alive handling.

Caveats/Known Bugs

- module always binds to all interfaces
- only a single listener can be bound

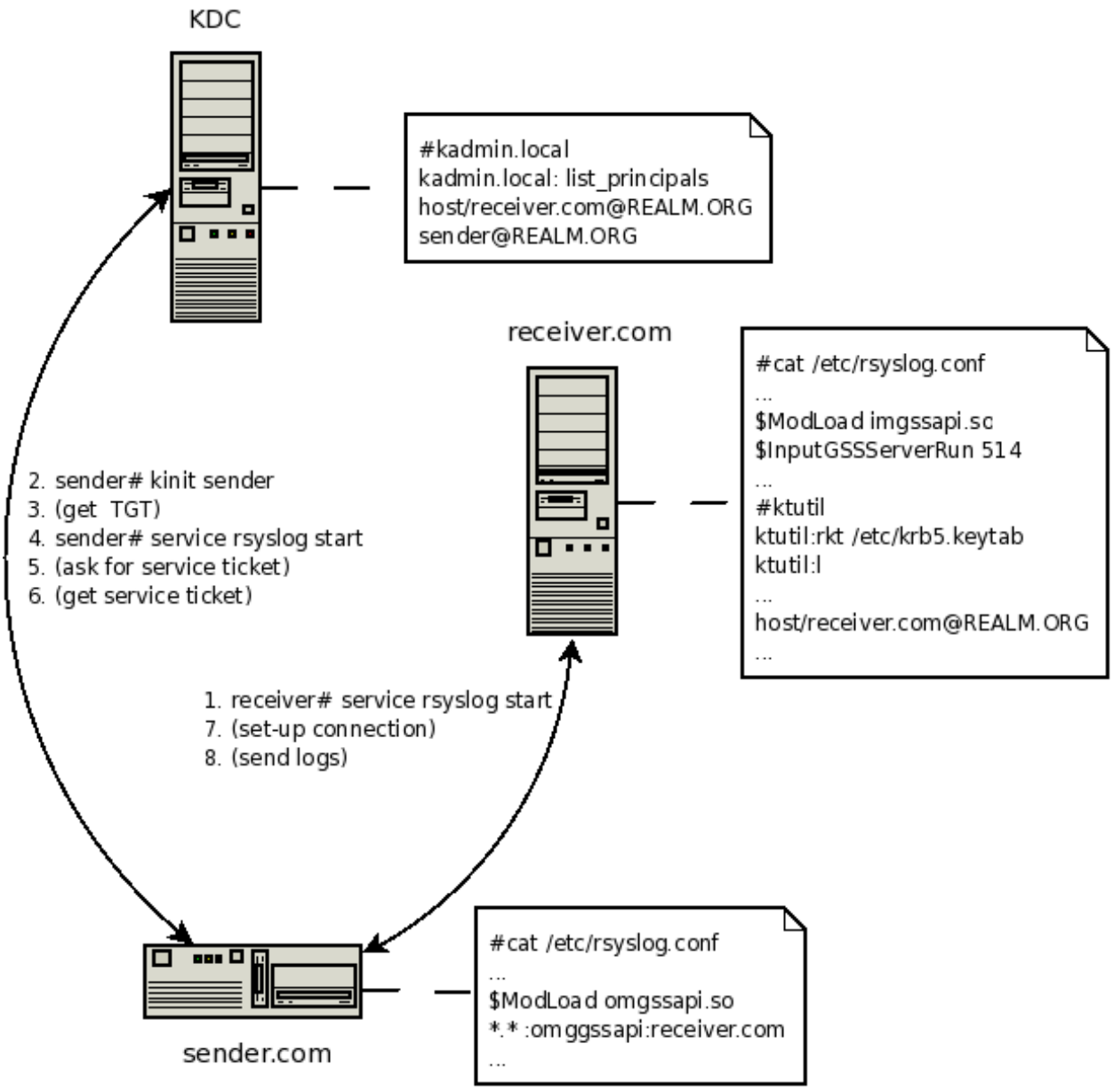


Fig. 1.2: rsyslog gssapi support

Example

This sets up a GSS server on port 1514 that also permits to receive plain tcp syslog messages (on the same port):

```
$ModLoad imgssapi # needs to be done just once
$InputGSSServerRun 1514
$InputGSSServerPermitPlainTCP on
```

imjournal: Systemd Journal Input Module

Module Name: imjournal

Author: Milan Bartos <mbartos@redhat.com> (This module is **not** project-supported)

Available since: 7.3.11

Description:

Provides the ability to import structured log messages from systemd journal to syslog.

Note that this module reads the journal database, what is considered a relativly performance-intense operation. As such, the performance of a configuration utilizing this module may be notably slower than when using imuxsock. The journal provides imuxsock with a copy of all "classical" syslog messages, however, it does not provide structured data. Only if that structured data is needed, imjournal must be used. Otherwise, imjournal may simply be replaced by imuxsock, and we highly suggest doing so.

We suggest to check out our short presentation on rsyslog journal integration to learn more details of anticipated use cases.

Warning: Some versions of systemd journal have problems with database corruption, which leads to the journal to return the same data endlessly in a tight loop. This results in massive message duplication inside rsyslog probably resulting in a denial-of-service when the system ressouces get exhausted. This can be somewhat mitigated by using proper rate-limiters, but even then there are spikes of old data which are endlessly repeated. By default, ratelimiting is activated and permits to process 20,000 messages within 10 minutes, what should be well enough for most use cases. If insufficient, use the parameters described below to adjust the permitted volume. **It is strongly recommended to use this plugin only if there is hard need to do so.**

Configuration Directives:

Module Directives

- **PersistStateInterval** number-of-messages

  This is a global setting. It specifies how often should the journal state be persisted. The persists happens after each *number-of-messages*. This option is useful for rsyslog to start reding from the last journal message it read.

- **StateFile** /path/to/file

  This is a global setting. It specifies where the state file for persisting journal state is located. If a full path name is given (starting with "/"), that path is used. Otherwise the given name is created inside the working directory.

- **ratelimit.interval** seconds (default: 600)

  Specifies the interval in seconds onto which rate-limiting is to be applied. If more than ratelimit.burst messages are read during that interval, further messages up to the end of the interval are discarded. The number of messages discarded is emitted at the end of the interval (if there were any discards). Setting this to value zero turns off ratelimiting. Note that it is **not recommended to turn of ratelimiting**, except that you know for sure journal database entries will never be corrupted. Without ratelimiting, a corrupted systemd journal database may cause a kind of denial of service (we are stressing this point as multiple users have reported us such problems with the journal database - information current as of June 2013).

- **ratelimit.burst** messages (default: 20000)

  Specifies the maximum number of messages that can be emitted within the ratelimit.interval interval. For futher information, see description there.

- **IgnorePreviousMessages** [**off**/on]

  This option specifies whether imjournal should ignore messages currently in journal and read only new messages. This option is only used when there is no StateFile to avoid message loss.

- **DefaultSeverity** <severity>

  Some messages comming from journald don't have the SYSLOG_PRIORITY field. These are typically the messages logged through journald's native API. This option specifies the default severity for these messages. Can be given either as a name or a number. Defaults to 'notice'.

- **DefaultFacility** <facility>

  Some messages comming from journald don't have the SYSLOG_FACILITY field. These are typically the messages logged through journald's native API. This option specifies the default facility for these messages. Can be given either as a name or a number. Defaults to 'user'.

- **usepidfromsystem** [**off**/on]

  Retrieves the trusted systemd parameter, _PID, instead of the user systemd parameter, SYSLOG_PID, which is the default. This option override the "usepid" option. This is now deprecated. It is better to use usepid="syslog" instead.

- **usepid** [**both**/syslog/system] Sets the PID source from journal.

  syslog *imjournal* retrieves SYSLOG_PID from journal as PID number.

  system *imjournal* retrieves _PID from journal as PID number.

  both *imjournal* trying to retrieve SYSLOG_PID first. When it is not available, it is also trying to retrive _PID. When none of them is available, message is parsed without PID number.

- **IgnoreNonValidStatefile** [**on**/off]

  When a corrupted statefile is read imjournal ignores the statefile and continues with logging from the beginning of the journal (from its end if IgnorePreviousMessages is on). After PersistStateInterval or when rsyslog is stopped invalid statefile is overwritten with a new valid cursor.

Caveats/Known Bugs:

- As stated above, a corrupted systemd journal database can cause major problems, depending on what the corruption results in. This is beyond the control of the rsyslog team.
- imjournal does not check if messages received actually originated from rsyslog itself (via omjournal or other means). Depending on configuration, this can also lead to a loop. With imuxsock, this problem does not exist.

Build Requirements:

Development headers for systemd, version >= 197.

Sample:

The following example shows pulling structured imjournal messages and saving them into /var/log/ceelog.

```
module(load="imjournal" PersistStateInterval="100"
       StateFile="/path/to/file") #load imjournal module
module(load="mmjsonparse") #load mmjsonparse module for structured logs

template(name="CEETemplate" type="string" string="%TIMESTAMP% %HOSTNAME% %syslogtag%␣
↪@cee: %$!all-json%\n" ) #template for messages
```

```
action(type="mmjsonparse")
action(type="omfile" file="/var/log/ceelog" template="CEETemplate")
```

Legacy Configuration Directives:

- **$imjournalPersistStateInterval**

  Equivalent to: PersistStateInterval

- **$imjournalStateFile**

  Equivalent to: StateFile

- **$imjournalRatelimitInterval**

  Equivalent to: ratelimit.interval

- **$imjournalRatelimitBurst**

  Equivalent to: ratelimit.burst

- **$ImjournalIgnorePreviousMessages**

  Equivalent to: ignorePreviousMessages

- **$ImjournalDefaultSeverity** Equivalent to: DefaultSeverity
- **$ImjournalDefaultFacility** Equivalent to: DefaultFacility

imkafka: read from Apache Kafka

Module Name:	imkafka
Author:	Pascal Withopf <pascalwithopf1@gmail.com>
Available since:	v8.27.0

The imkafka plug-in implements an Apache Kafka consumer, permitting rsyslog to receive data from Kafka.

Configuration Parameters

Note that imkafka supports some *Array*-type parameters. While the parameter name can only be set once, it is possible to set multiple values with that single parameter.

For example, to select a broker, you can use

```
input(type="imkafka" topic="mytopic" broker="localhost:9092" consumergroup="default")
```

which is equivalent to

```
input(type="imkafka" topic="mytopic" broker=["localhost:9092"] consumergroup="default
↪")
```

To specify multiple values, just use the bracket notation and create a comma-delimited list of values as shown here:

```
input(type="imkafka" topic="mytopic"
       broker=["localhost:9092",
               "localhost:9093",
               "localhost:9094"]
      )
```

Module Parameters

Currently none.

Action Parameters

broker <Array>
Default: "localhost:9092"

Specifies the broker(s) to use.

topic <String>
Mandatory

Default: none

Specifies the topic to produce to.

confParam <Array>
Default: none

Permits to specify Kafka options. Rather than offering a myriad of config settings to match the Kafka parameters, we provide this setting here as a vehicle to set any Kafka parameter. This has the big advantage that Kafka parameters that come up in new releases can immediately be used.

Note that we use librdkafka for the Kafka connection, so the parameters are actually those that librdkafka supports. As of our understanding, this is a superset of the native Kafka parameters.

consumergroup <String>
Default none

With this parameter the group.id for the consumer is set. All consumers sharing the same group.id belong to the same group.

ruleset <String>
Default: none

Specifies the ruleset to be used.

Caveats/Known Bugs

- currently none

Example

Sample 1:

In this sample a consumer for the topic static is created and will forward the messages to the omfile action.

```
module(load="imkafka")
input(type="imkafka" topic="static" broker="localhost:9092"
                     consumergroup="default" ruleset="pRuleset")

ruleset(name="pRuleset") {
      action(type="omfile" file="path/to/file")
}
```

imklog: Kernel Log Input Module

Reads messages from the kernel log and submits them to the syslog engine.

Author: Rainer Gerhards <rgerhards@adiscon.com>

Configuration Directives

$KLogInternalMsgFacility <facility>
The facility which messages internally generated by imklog will have. imklog generates some messages of itself (e.g. on problems, startup and shutdown) and these do not stem from the kernel. Historically, under Linux, these too have "kern" facility. Thus, on Linux platforms the default is "kern" while on others it is "syslogd". You usually do not need to specify this configuration directive - it is included primarily for few limited cases where it is needed for good reason. Bottom line: if you don't have a good idea why you should use this setting, do not touch it.

$KLogPermitNonKernelFacility off
At least under BSD the kernel log may contain entries with non-kernel facilities. This setting controls how those are handled. The default is "off", in which case these messages are ignored. Switch it to on to submit non-kernel messages to rsyslog processing.

$DebugPrintKernelSymbols on/off
Linux only, ignored on other platforms (but may be specified). Defaults to off.

$klogLocalIPIF [interface name]
If provided, the IP of the specified interface (e.g. "eth0") shall be used as fromhost-ip for imklog-originating messages. If this directive is not given OR the interface cannot be found (or has no IP address), the default of "127.0.0.1" is used.

$klogConsoleLogLevel <number>
Sets the console log level. If specified, only messages with up to the specified level are printed to the console. The default is -1, which means that the current settings are not modified. To get this behavior, do not specify $klogConsoleLogLevel in the configuration file. Note that this is a global parameter. Each time it is changed, the previous definition is re-set. The one activate will be that one that is active when imklog actually starts processing. In short words: do not specify this directive more than once!

Linux only, ignored on other platforms (but may be specified)

$klogUseSyscallInterface on/off
Linux only, ignored on other platforms (but may be specified). Defaults to off.

$klogSymbolsTwice on/off
Linux only, ignored on other platforms (but may be specified). Defaults to off.

$klogParseKernelTimestamp on/off
If enabled and the kernel creates a timestamp for its log messages, this timestamp will be parsed and converted into regular message time instead to use the receive time of the kernel message (as in 5.8.x and before). Default is 'off' to prevent parsing the kernel timestamp, because the clock used by the kernel to create the timestamps is not supposed to be as accurate as the monotonic clock required to convert it. Depending on the hardware and kernel, it can result in message time differences between kernel and system messages which occurred at same time.

$klogKeepKernelTimestamp on/off
If enabled, this option causes to keep the [timestamp] provided by the kernel at the begin of in each message rather than to remove it, when it could be parsed and converted into local time for use as regular message time. Only used, when $klogParseKernelTimestamp is on.

Caveats/Known Bugs

This is obviously platform specific and requires platform drivers. Currently, imklog functionality is available on Linux and BSD.

This module is **not supported on Solaris** and not needed there. For Solaris kernel input, use *imsolaris*.

Example

The following sample pulls messages from the kernel log. All parameters are left by default, which is usually a good idea. Please note that loading the plugin is sufficient to activate it. No directive is needed to start pulling kernel messages.

```
$ModLoad imklog
```

imkmsg: /dev/kmsg Log Input Module

Module Name: imkmsg

Authors:Rainer Gerhards <rgerhards@adiscon.com> Milan Bartos <mbartos@redhat.com>

Description:

Reads messages from the /dev/kmsg structured kernel log and submits them to the syslog engine.

The printk log buffer contains log records. These records are exported by /dev/kmsg device as structured data in the following format: "level,sequnum,timestamp;<message text>\n" There could be continuation lines starting with space that contains key/value pairs. Log messages are parsed as necessary into rsyslog msg_t structure. Continuation lines are parsed as json key/value pairs and added into rsyslog's message json representation.

Configuration Directives:

This module has no configuration directives.

Caveats/Known Bugs:

This module can't be used together with imklog module. When using one of them, make sure the other one is not enabled.

This is Linux specific module and requires /dev/kmsg device with structured kernel logs.

Sample:

The following sample pulls messages from the /dev/kmsg log device. All parameters are left by default, which is usually a good idea. Please note that loading the plugin is sufficient to activate it. No directive is needed to start pulling messages.

$ModLoad imkmsg

impstats: Generate Periodic Statistics of Internal Counters

Author: Rainer Gerhards <rgerhards@adiscon.com>

This module provides periodic output of rsyslog internal counters.

The set of available counters will be output as a set of syslog messages. This output is periodic, with the interval being configurable (default is 5 minutes). Be sure that your configuration records the counter messages (default is syslog.=info). Besides logging to the regular syslog stream, the module can also be configured to write statistics data into a (local) file.

When logging to the regular syslog stream, impstats records are emitted just like regular log messages. As such, counters increase when processing these messages. This must be taken into consideration when testing and troubleshooting.

Note that loading this module has some impact on rsyslog performance. Depending on settings, this impact may be noticable for high-load environments, but in general the overhead is pretty light.

Note that there is a rsyslog statistics online analyzer **available.** It can be given a impstats-generated file and will return problems it detects. Note that the analyzer cannot replace a human in getting things right, but it is expected to be a good aid in starting to understand and gain information from the pstats logs.

The rsyslog website has an overview of available rsyslog statistic counters. When browsing this page, please be sure to take note of which rsyslog version is required to provide a specific counter. Counters are continously being added, and older versions do not support everything.

Configuration Directives

The configuration directives for this module are designed for tailoring the method and process for outputting the rsyslog statistics to file.

Module Confguration Parameters

This module supports module parameters, only.

`interval [seconds]`
Default 300 [5minutes]

Sets the interval, in **seconds** at which messages are generated. Please note that the actual interval may be a bit longer. We do not try to be precise and so the interval is actually a sleep period which is entered after generating all messages. So the actual interval is what is configured here plus the actual time required to generate messages. In general, the difference should not really matter.

`facility [facility number]`
The numerical syslog facility code to be used for generated messages. Default is 5 (syslog). This is useful for filtering messages.

`severity [severity number]`
The numerical syslog severity code to be used for generated messages. Default is 6 (info).This is useful for filtering messages.

`resetCounters off/on`
Defaults to off

When set to "on", counters are automatically reset after they are emitted. In that case, the contain only deltas to the last value emitted. When set to "off", counters always accumulate their values. Note that in auto-reset mode not all counters can be reset. Some counters (like queue size) are directly obtained from internal object and cannot be modified. Also, auto-resetting introduces some additional slight inaccuracies due to the multi-threaded nature of rsyslog and the fact that for performance reasons it cannot serialize access to counter variables. As an alternative to auto-reset mode, you can use rsyslog's statistics manipulation scripts to create delta values from the regular statistic logs. This is the suggested method if deltas are not necessarily needed in real-time.

`format json/json-elasticsearch/cee/legacy`
Defaults to legacy

Specifies the format of emitted stats messages. The default of "legacy" is compatible with pre v6-rsyslog. The other options provide support for structured formats (note the "cee" is actually "project lumberack" logging).

The json-elasticsearch format supports the broken ElasticSearch JSON implementation. ES 2.0 no longer supports valid JSON and disallows dots inside names. The "json-elasticsearch" format option replaces those dots by the bang ("!") character. So "discarded.full" becomes "discarded!full". This option is available starting with 8.16.0.

log.syslog on/off
Defaults to on

This is a boolean setting specifying if data should be sent to the usual syslog stream. This is useful if custom formatting or more elaborate processing is desired. However, output is placed under the same restrictions as regular syslog data, especially in regard to the queue position (stats data may sit for an extended period of time in queues if they are full).

log.file [file name]
If specified, statistics data is written to the specified file. For robustness, this should be a local file. The file format cannot be customized, it consists of a date header, followed by a colon, followed by the actual statistics record, all on one line. Only very limited error handling is done, so if things go wrong stats records will probably be lost. Logging to file an be a useful alternative if for some reasons (e.g. full queues) the regular syslog stream method shall not be used solely. Note that turning on file logging does NOT turn off syslog logging. If that is desired log.syslog="off" must be explicitely set.

Ruleset [ruleset]
Binds the listener to a specific *ruleset*.

bracketing off/on
Default: off

Requires v8.4.1 or above

This is a utility setting for folks who postprocess impstats logs and would like to know the begin and end of a block of statistics. When "bracketing" is set to "on", impstats issues a "BEGIN" message before the first counter is issued, then all counter values are issued, and then an "END" message follows. As such, if and only iff messages are kept in sequence, a block of stats countes can easily be identified by those BEGIN and END messages.

Note well: in general, sequence of syslog messages is **not** strict and is not ordered in sequence of message generation. There are various occasion that can cause message reordering, some examples are:

- using multiple threads
- using UDP forwarding
- using relay systems, especially with buffering enabled
- using disk-assisted queues

This is not a problem with rsyslog, but rather the way a concurrent world works. For strict order, a specific order predicate (e.g. a sufficiently fine-grained timestamp) must be used.

As such, BEGIN and END records may actually indicate the begin and end of a block of statistics - or they may *not*. Any order is possible in theory. So the bracketing option does not in all cases work as expected. This is the reason why it is turned off by default.

However, bracketing may still be useful for many use cases. First and foremost, while there are many scenarios in which messages become reorderded, in practice it happens relatively seldom. So most of the time the statistics records will come in as expected and actually will be bracketed by the BEGIN and END messages. Consequently, if an application can handle occasional out-of-order delivery (e.g. by graceful degradation), bracketing may actually be a great solution. It is, however, very important to know and handle out of order delivery. For most real-world deployments, a good way to handle it is to ignore unexpected records and use the

previous values for ones missing in the current block. To guard against two or more blocks being mixed, it may also be a good idea to never reset a value to a lower bound, except when that lower bound is seen consistantly (which happens due to a restart). Note that such lower bound logic requires *resetCounters* to be set to off.

Statistic Counter

The impstats plugin gathers some internal *statistics*. They have different names depending on the actual statistics. Obviously, they do not relate to the plugin itself but rather to a broader object – most notably the rsyslog process itself. The "resource-usage" counter maintains process statistics. They base on the getrusage() system call. The counters are named like getrusage returned data memebers. So for details, looking them up in "man getrusage" is highly recommended, especially as value may be different depending on the platform. A getrusage() call is done immediately before the counter is emitted. The follwowing individual counters are maintained:

- **utime** - this is the user time in microseconds (thus the timeval structure combined)
- **stime** - again, time given in microseconds
- **maxrss**
- **minflt**
- **majflt**
- **inblock**
- **outblock**
- **nvcsw**
- **nivcsw**
- **openfiles** number of file handles used by rsyslog; includes actual files, sockets and others

Legacy Configuration Directives

A limited set of parameters can also be set via the legacy configuration syntax. Note that this is intended as an upward compatibilit layer, so newer features are intentionally **not** available via legacy directives.

- $PStatInterval <Seconds> - same as the "interval" parameter.
- $PStatFacility <numerical facility> - same as the "facility" parameter.
- $PStatSeverity <numerical severity> - same as the "severity" parameter.
- $PStatJSON <on/**off**> (rsyslog v6.3.8+ only) If set to on, stats messages are emitted as structured cee-enhanced syslog. If set to off, legacy format is used (which is compatible with pre v6-rsyslog).

Caveats/Known Bugs

- This module MUST be loaded right at the top of rsyslog.conf, otherwise stats may not get turned on in all places.

Example

This activates the module and records messages to /var/log/rsyslog-stats in 10 minute intervals:

```
module(load="impstats"
       interval="600"
       severity="7")

# to actually gather the data:
syslog.=debug /var/log/rsyslog-stats
```

In the next sample, the default interval of 5 minutes is used. However, this time stats data is NOT emitted to the syslog stream but to a local file instead.

```
module(load="impstats"
       interval="600"
       severity="7"
       log.syslog="off"
       /* need to turn log stream logging off! */
       log.file="/path/to/local/stats.log")
```

And finally, we log to both the regular syslog log stream as well as a file. Within the log stream, we forward the data records to another server:

```
module(load="impstats"
       interval="600"
       severity="7"
        log.file="/path/to/local/stats.log")

syslog.=debug @central.example.net
```

Legacy Sample

This activates the module and records messages to /var/log/rsyslog-stats in 10 minute intervals:

```
$ModLoad impstats
$PStatInterval 600
$PStatSeverity 7
syslog.=debug /var/log/rsyslog-stats
```

See Also

- rsyslog statistics counter
- impstats delayed or lost - cause and cure

imptcp: Plain TCP Syslog

Provides the ability to receive syslog messages via plain TCP syslog. This is a specialised input plugin tailored for high performance on Linux. It will probably not run on any other platform. Also, it does not provide TLS services. Encryption can be provided by using stunnel.

This module has no limit on the number of listeners and sessions that can be used.

Author: Rainer Gerhards <rgerhards@adiscon.com>

Error Messages

When a message is to long it will be truncated and an error will show the remaining length of the message and the beginning of it. It will be easier to comprehend the truncation.

Configuration Directives

This plugin has config directives similar named as imtcp, but they all have **P**TCP in their name instead of just TCP. Note that only a subset of the parameters are supported.

Module Parameters

These parameters can be used with the "module()" statement. They apply globaly to all inputs defined by the module.

Threads <number>
Number of helper worker threads to process incoming messages. These threads are utilized to pull data off the network. On a busy system, additional helper threads (but not more than there are CPUs/Cores) can help improving performance. The default value is two, which means there is a default thread count of three (the main input thread plus two helpers). No more than 16 threads can be set (if tried to, rsyslog always resorts to 16).

processOnPoller on/off
Defaults to on

Instructs imptcp to process messages on poller thread opportunistically. This leads to lower resource footprint(as poller thread doubles up as message-processing thread too). "On" works best when imptcp is handling low ingestion rates.

At high throughput though, it causes polling delay(as poller spends time processing messages, which keeps connections in read-ready state longer than they need to be, filling socket-buffer, hence eventually applying backpressure).

It defaults to allowing messages to be processed on poller (for backward compatibility).

Input Parameters

These parameters can be used with the "input()" statement. They apply to the input they are specified with.

port <number>
Mandatory

Select a port to listen on. It is an error to specify both *path* and *port*.

path <name>
A path on the filesystem for a unix domain socket. It is an error to specify both *path* and *port*.

discardTruncatedMsg <on/off>
Default: off

When a message is split because it is to long the second part is normally processed as the next message. This can cause Problems. When this parameter is turned on the part of the message after the truncation will be discarded.

fileOwner [userName]
Default: system default

Set the file owner for the domain socket. The parameter is a user name, for which the userid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are *not* detected.

fileOwnerNum [uid]
Default: system default

Set the file owner for the domain socket. The parameter is a numerical ID, which which is used regardless of whether the user actually exists. This can be useful if the user mapping is not available to rsyslog during startup.

fileGroup [groupName]
Default: system default

Set the group for the domain socket. The parameter is a group name, for which the groupid is obtained by rsyslogd during startup processing. Interim changes to the user mapping are not detected.

fileGroupNum [gid]
Default: system default

Set the group for the domain socket. The parameter is a numerical ID, which is used regardless of whether the group actually exists. This can be useful if the group mapping is not available to rsyslog during startup.

fileCreateMode [octalNumber]
Default: 0644

Set the access permissions for the domain socket. The value given must always be a 4-digit octal number, with the initial digit being zero. Please note that the actual permission depend on rsyslogd's process umask. If in doubt, use "$umask 0000" right at the beginning of the configuration file to remove any restrictions.

failOnChOwnFailure [switch]
Default: on

rsyslog will not start if this is on and changing the file owner, group, or access permissions fails. Disable this to ignore these errors.

unlink on/off
Default: off

If a unix domain socket is being used this controls whether or not the socket is unlinked before listening and after closing.

name <name>
Sets a name for the inputname property. If no name is set "imptcp" is used by default. Setting a name is not strictly necessary, but can be useful to apply filtering based on which input the message was received from. Note that the name also shows up in *impstats* logs.

ruleset <name>
Binds specified ruleset to this input. If not set, the default ruleset is bound.

maxFrameSize <int>
Default: 200000; Max: 200000000

When in octet counted mode, the frame size is given at the beginning of the message. With this parameter the max size this frame can have is specified and when the frame gets to large the mode is switched to octet stuffing. The max value this parameter can have was specified because otherwise the integer could become negative and this would result in a Segmentation Fault.

address <name>
Default: all interfaces

On multi-homed machines, specifies to which local address the listerner should be bound.

AddtlFrameDelimiter <Delimiter>
This directive permits to specify an additional frame delimiter for plain tcp syslog. The industry-standard specifies using the LF character as frame delimiter. Some vendors, notable Juniper in their NetScreen products, use an invalid frame delimiter, in Juniper's case the NUL character. This directive permits to specify the ASCII value of the delimiter in question. Please note that this does not guarantee that all wrong implementations can be

cured with this directive. It is not even a sure fix with all versions of NetScreen, as I suggest the NUL character is the effect of a (common) coding error and thus will probably go away at some time in the future. But for the time being, the value 0 can probably be used to make rsyslog handle NetScreen's invalid syslog/tcp framing. For additional information, see this forum thread. **If this doesn't work for you, please do not blame the rsyslog team. Instead file a bug report with Juniper!**

Note that a similar, but worse, issue exists with Cisco's IOS implementation. They do not use any framing at all. This is confirmed from Cisco's side, but there seems to be very limited interest in fixing this issue. This directive **can not** fix the Cisco bug. That would require much more code changes, which I was unable to do so far. Full details can be found at the Cisco tcp syslog anomaly page.

SupportOctetCountedFraming on/off
Defaults to "on"

The legacy octed-counted framing (similar to RFC5425 framing) is activated. This is the default and should be left unchanged until you know very well what you do. It may be useful to turn it off, if you know this framing is not used and some senders emit multi-line messages into the message stream.

NotifyOnConnectionClose on/off
Defaults to off

instructs imptcp to emit a message if a remote peer closes the connection.

NotifyOnConnectionOpen on/off
Defaults to off

instructs imptcp to emit a message if a remote peer opens a connection. Hostname of the remote peer is given in the message.

KeepAlive on/off
Defaults to off

enable of disable keep-alive packets at the tcp socket layer. The default is to disable them.

KeepAlive.Probes <number>
The number of unacknowledged probes to send before considering the connection dead and notifying the application layer. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

KeepAlive.Interval <number>
The interval between subsequential keepalive probes, regardless of what the connection has exchanged in the meantime. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

KeepAlive.Time <number>
The interval between the last data packet sent (simple ACKs are not considered data) and the first keepalive probe; after the connection is marked to need keepalive, this counter is not used any further. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

RateLimit.Interval [number]
Default is 0, which turns off rate limiting

Specifies the rate-limiting interval in seconds. Set it to a number of seconds (5 recommended) to activate rate-limiting.

RateLimit.Burst [number]
Default is 10,000

Specifies the rate-limiting burst in number of messages.

compression.mode [mode]
Default is none

This is the counterpart to the compression modes set in *omfwd*. Please see it's documentation for details.

flowControl <on/off>
Default: on

Flow control is used to throttle the sender if the receiver queue is near-full preserving some space for input that can not be throttled.

multiLine <on/off>
Default: off

Experimental parameter which caues rsyslog to recognise a new message only if the line feed is followed by a '<' or if there are no more characters.

Statistic Counter

This plugin maintains *statistics* for each listener. The statistic is named "imtcp" , followed by the bound address, listener port and IP version in parenthesis. For example, the counter for a listener on port 514, bound to all interfaces and listening on IPv6 is called "imptcp(*/514/IPv6)".

The following properties are maintained for each listener:

- **submitted** - total number of messages submitted for processing since startup

Caveats/Known Bugs

- module always binds to all interfaces

Example

This sets up a TCP server on port 514:

```
module(load="imptcp") # needs to be done just once
input(type="imptcp" port="514")
```

This creates a listener that listens on the local loopback interface, only.

```
module(load="imptcp") # needs to be done just once
input(type="imptcp" port="514" address="127.0.0.1")
```

Create a unix domain socket:

```
module(load="imptcp") # needs to be done just once
input(type="imptcp" path="/tmp/unix.sock" unlink="on")
```

Legacy Configuration Directives

$InputPTCPServerAddtlFrameDelimiter <Delimiter>
Equivalent to: AddTLFrameDelimiter

$InputPTCPSupportOctetCountedFraming on/off
Equivalent to: SupportOctetCountedFraming

$InputPTCPServerNotifyOnConnectionClose on/off
Equivalent to: NotifyOnConnectionClose.

$InputPTCPServerKeepAlive <on/off**>**
Equivalent to: KeepAlive

$InputPTCPServerKeepAlive_probes <number>
Equivalent to: KeepAlive.Probes

$InputPTCPServerKeepAlive_intvl <number>
Equivalent to: KeepAlive.Interval

$InputPTCPServerKeepAlive_time <number>
Equivalent to: KeepAlive.Time

$InputPTCPServerRun <port>
Equivalent to: Port

$InputPTCPServerInputName <name>
Equivalent to: Name

$InputPTCPServerBindRuleset <name>
Equivalent to: Ruleset

$InputPTCPServerHelperThreads <number>
Equivalent to: threads

$InputPTCPServerListenIP <name>
Equivalent to: Address

Caveats/Known Bugs

- module always binds to all interfaces

Example

This sets up a TCP server on port 514:

```
$ModLoad imptcp # needs to be done just once
$InputPTCPServerRun 514
```

imrelp: RELP Input Module

Module Name: imrelp

Author: Rainer Gerhards

Provides the ability to receive syslog messages via the reliable RELP protocol. This module requires librelp to be present on the system. From the user's point of view, imrelp works much like imtcp or imgssapi, except that no message loss can occur. Please note that with the currently supported relp protocol version, a minor message duplication may occur if a network connection between the relp client and relp server breaks after the client could successfully send some messages but the server could not acknowledge them. The window of opportunity is very slim, but in theory this is possible. Future versions of RELP will prevent this. Please also note that rsyslogd may lose a few messages if rsyslog is shutdown while a network connection to the server is broken and could not yet be recovered. Future version of RELP support in rsyslog will prevent that. Please note that both scenarios also exists with plain tcp syslog. RELP, even with the small nits outlined above, is a much more reliable solution than plain tcp syslog and so it is highly suggested to use RELP instead of plain tcp. Clients send messages to the RELP server via omrelp.

Module Parameters

Ruleset <name>
(requires v7.5.0+)
Binds the specified ruleset to **all** RELP listeners. This can be overridden at the instance level.

Input Parameters

Port <port>
Starts a RELP server on selected port

Name <name>
Sets a name for the inputname property of this listener. If no name is set, "imrelp" is used by default. Setting a name is not strictly necessary, but can be useful to apply filtering based on which input the message was received from.

Ruleset <name>
Binds specified ruleset to this listener. This overrides the module-level Ruleset parameter.

MaxDataSize <size_nbr>
Default is the global message size

Sets the max message size (in bytes) that can be received. Any messages above this size will be rejected causing the relp client to reconnect and retry.

tls on/off
Default is off

If set to "on", the RELP connection will be encrypted by TLS, so that the data is protected against observers. Please note that both the client and the server must have set TLS to either "on" or "off". Other combinations lead to unpredictable results.

Attention when using GnuTLS 2.10.x or older

Versions older than GnuTLS 2.10.x may cause a crash (Segfault) under certain circumstances. Most likely when an imrelp inputs and an omrelp output is configured. The crash may happen when you are receiving/sending messages at the same time. Upgrade to a newer version like GnuTLS 2.12.21 to solve the problem.

tls.compression on/off
Default is off

The controls if the TLS stream should be compressed (zipped). While this increases CPU use, the network bandwidth should be reduced. Note that typical text-based log records usually compress rather well.

tls.dhbits <integer>
This setting controls how many bits are used for Diffie-Hellman key generation. If not set, the librelp default is used. For secrity reasons, at least 1024 bits should be used. Please note that the number of bits must be supported by GnuTLS. If an invalid number is given, rsyslog will report an error when the listener is started. We do this to be transparent to changes/upgrades in GnuTLS (to check at config processing time, we would need to hardcode the supported bits and keep them in sync with GnuTLS - this is even impossible when custom GnuTLS changes are made...).

tls.**permittedPeer**()
Peer Places access restrictions on this listener. Only peers which have been listed in this parameter may connect. The validation bases on the certificate the remote peer presents.

The *peer* parameter lists permitted certificate fingerprints. Note that it is an array parameter, so either a single or multiple fingerprints can be listed. When a non-permitted peer connects, the refusal is logged together with

it's fingerprint. So if the administrator knows this was a valid request, he can simple add the fingerprint by copy and paste from the logfile to rsyslog.conf.

To specify multiple fingerprints, just enclose them in braces like this:

```
tls.permittedPeer=["SHA1:...1", "SHA1:....2"]
```

To specify just a single peer, you can either specify the string directly or enclose it in braces.

tls.authMode <mode>
Sets the mode used for mutual authentication.

Supported values are either "*fingerprint*" or "*name*".

Fingerprint mode basically is what SSH does. It does not require a full PKI to be present, instead self-signed certs can be used on all peers. Even if a CA certificate is given, the validity of the peer cert is NOT verified against it. Only the certificate fingerprint counts.

In "name" mode, certificate validation happens. Here, the matching is done against the certificate's subjectAltName and, as a fallback, the subject common name. If the certificate contains multiple names, a match on any one of these names is considered good and permits the peer to talk to rsyslog.

tls.prioritystring <string>
This parameter permits to specify the so-called "priority string" to GnuTLS. This string gives complete control over all crypto parameters, including compression setting. For this reason, when the prioritystring is specified, the "tls.compression" parameter has no effect and is ignored.

Full information about how to construct a priority string can be found in the GnuTLS manual. At the time of this writing, this information was contained in section 6.10 of the GnuTLS manual.

Note: this is an expert parameter. Do not use if you do not exactly know what you are doing.

KeepAlive on/off
enable of disable keep-alive packets at the tcp socket layer. The default is to disable them.

KeepAlive.Probes <number>
Default is 0

The number of unacknowledged probes to send before considering the connection dead and notifying the application layer. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

KeepAlive.Interval <number>
Default is 0

The interval between subsequent keepalive probes, regardless of what the connection has exchanged in the meantime. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

KeepAlive.Time <number>
Default is 0

The interval between the last data packet sent (simple ACKs are not considered data) and the first keepalive probe; after the connection is marked to need keepalive, this counter is not used any further. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

Statistic Counter

This plugin maintains *statistics* for each listener. The statistic by default is named "imrelp" , followed by the listener port in parenthesis. For example, the counter for a listener on port 514 is called "imprelp(514)". If the input is given a

name, that input name is used instead of "imrelp". This counter is available starting rsyslog 7.5.1

The following properties are maintained for each listener: - **submitted** - total number of messages submitted for processing since startup

Caveats/Known Bugs

- see description
- To obtain the remote system's IP address, you need to have at least librelp 1.0.0 installed. Versions below it return the hostname instead of the IP address.

Sample

This sets up a RELP server on port 20514 with a max message size of 10,000 bytes.

```
module(load="imrelp") # needs to be done just once
input(type="imrelp" port="20514" maxDataSize="10k")
```

Legacy Configuration Directives

- InputRELPServerBindRuleset <name> (available in 6.3.6+) equivalent to: RuleSet
- InputRELPServerRun <port> equivalent to: Port

Caveats/Known Bugs

- To obtain the remote system's IP address, you need to have at least librelp 1.0.0 installed. Versions below it return the hostname instead of the IP address.
- Contrary to other inputs, the ruleset can only be bound to all listeners, not specific ones. This issue is resolved in the non-Legacy configuration format.

Sample:

Legacy Sample

This sets up a RELP server on port 20514.

```
$ModLoad imrelp # needs to be done just once
$InputRELPServerRun 20514
```


imsolaris: Solaris Input Module

Reads local Solaris log messages including the kernel log.

This module is specifically tailored for Solaris. Under Solaris, there is no special kernel input device. Instead, both kernel messages as well as messages emitted via syslog() are received from a single source.

This module obeys the Solaris door() mechanism to detect a running syslogd instance. As such, only one can be active at one time. If it detects another active instance at startup, the module disables itself, but rsyslog will continue to run.

Author: Rainer Gerhards <rgerhards@adiscon.com>

Configuration Directives

functions:: $IMSolarisLogSocketName <name>

This is the name of the log socket (stream) to read. If not given, /dev/log is read.

Caveats/Known Bugs

None currently known. For obvious reasons, works on Solaris, only (and compilation will most probably fail on any other platform).

Examples

The following sample pulls messages from the default log source

```
$ModLoad imsolaris
```

imtcp: TCP Syslog Input Module

Provides the ability to receive syslog messages via TCP. Encryption is natively provided by selecting the approprioate network stream driver and can also be provided by using stunnel (an alternative is the use the imgssapi module).

Author: Rainer Gerhards <rgerhards@adiscon.com>

Configuration Parameters

Module Parameters

AddtlFrameDelimiter <Delimiter>
This directive permits to specify an additional frame delimiter for Multiple receivers may be configured by specifying $InputTCPServerRun multiple times. This is available since version 4.3.1, earlier versions do NOT support it.

DisableLFDelimiter on/off
Default is off

Industry-strandard plain text tcp syslog uses the LF to delimit syslog frames. However, some users brought up the case that it may be useful to define a different delimiter and totally disable LF as a delimiter (the use case named were multi-line messages). This mode is non-standard and will probably come with a lot of problems. However, as there is need for it and it is relatively easy to support, we do so. Be sure to turn this setting to "on" only if you exactly know what you are doing. You may run into all sorts of troubles, so be prepared to wrangle with that!

maxFrameSize <int>
Default: 200000; Max: 200000000

When in octet counted mode, the frame size is given at the beginning of the message. With this parameter the max size this frame can have is specified and when the frame gets to large the mode is switched to octet stuffing. The max value this parameter can have was specified because otherwise the integer could become negative and this would result in a Segmentation Fault.

NotifyOnConnectionClose on/off
Default is off

Instructs imtcp to emit a message if the remote peer closes a connection.

Important: This directive is global to all listeners and must be given right after loading imtcp, otherwise it may have no effect.

KeepAlive on/off
Default is off

enable of disable keep-alive packets at the tcp socket layer. The default is to disable them.

KeepAlive.Probes <number>
The number of unacknowledged probes to send before considering the connection dead and notifying the application layer. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

KeepAlive.Interval <number>
The interval between subsequential keepalive probes, regardless of what the connection has exchanged in the meantime. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

KeepAlive.Time <number>
The interval between the last data packet sent (simple ACKs are not considered data) and the first keepalive probe; after the connection is marked to need keepalive, this counter is not used any further. The default, 0, means that the operating system defaults are used. This has only effect if keep-alive is enabled. The functionality may not be available on all platforms.

FlowControl on/off
Default is on

This setting specifies whether some message flow control shall be exercised on the related TCP input. If set to on, messages are handled as "light delayable", which means the sender is throttled a bit when the queue becomes near-full. This is done in order to preserve some queue space for inputs that can not throttle (like UDP), but it may have some undesired effect in some configurations. Still, we consider this as a useful setting and thus it is the default. To turn the handling off, simply configure that explicitely.

MaxListeners <number>
Default is 20

Sets the maximum number of listeners (server ports) supported. This must be set before the first $InputTCPServerRun directive.

MaxSessions <number>
Default is 200

Sets the maximum number of sessions supported. This must be set before the first $InputTCPServerRun directive

StreamDriver.Name <string>
Selects *network stream driver* for all inputs using this module.

StreamDriver.Mode <number>
Sets the driver mode for the currently selected *network stream driver*. <number> is driver specific.

StreamDriver.AuthMode <mode-string>
Sets permitted peer IDs. Only these peers are able to connect to the listener. <id-string> semantics depend on the currently selected AuthMode and *network stream driver*. PermittedPeers may not be set in anonymous modes.

PermittedPeer <id-string>
Sets permitted peer IDs. Only these peers are able to connect to the listener. <id-string> semantics depend on the currently selected AuthMode and *network stream driver*. PermittedPeer may not be set in anonymous modes. PermittedPeer may be set either to a single peer or an array of peers either of type IP or name, depending on the tls certificate.

Single peer: PermittedPeer="127.0.0.1"

Array of peers: PermittedPeer=["test1.example.net","10.1.2.3","test2.example.net","..."]

discardTruncatedMsg on/off
Default is off

Normally when a message is truncated in octet stuffing mode the part that is cut off is processed as the next message. When this parameter is activated, the part that is cut off after a truncation is discarded and not processed.

gnutlsPriorityString <string>
Default is NULL

Available since 8.29.0

The GnuTLS priority strings specify the TLS session's handshake algorithms and options. These strings are intended as a user-specified override of the library defaults. If this parameter is NULL, the default settings are used. More information about priority Strings here.

Input Parameters

Port <port>
Starts a TCP server on selected port

address <name>
Default: all interfaces

On multi-homed machines, specifies to which local address the listerner should be bound.

Name <name>
Sets a name for the inputname property. If no name is set "imtcp" is used by default. Setting a name is not strictly necessary, but can be useful to apply filtering based on which input the message was received from.

Ruleset <ruleset>
Binds the listener to a specific *ruleset*.

SupportOctetCountedFraming on/off
Default is on

If set to "on", the legacy octed-counted framing (similar to RFC5425 framing) is activated. This should be left unchanged until you know very well what you do. It may be useful to turn it off, if you know this framing is not used and some senders emit multi-line messages into the message stream.

RateLimit.Interval [number]
Specifies the rate-limiting interval in seconds. Default value is 0, which turns off rate limiting. Set it to a number of seconds (5 recommended) to activate rate-limiting.

RateLimit.Burst [number]
Specifies the rate-limiting burst in number of messages. Default is 10,000.

Statistic Counter

This plugin maintains *statistics* for each listener. The statistic is named after the given input name (or "imtcp" if none is configured), followed by the listener port in parenthesis. For example, the counter for a listener on port 514 with no set name is called "imtcp(514)".

The following properties are maintained for each listener:

- **submitted** - total number of messages submitted for processing since startup

Caveats/Known Bugs

- module always binds to all interfaces
- can not be loaded together with imgssapi (which includes the functionality of imtcp)

See also

- rsyslog video tutorial on how to store remote messages in a separate file (for legacy syntax, but you get the idea).

Example

This sets up a TCP server on port 514 and permits it to accept up to 500 connections:

```
module(load="imtcp" MaxSessions="500")
input(type="imtcp" port="514")
```

Note that the global parameters (here: max sessions) need to be set when the module is loaded. Otherwise, the parameters will not apply.

Legacy Configuration Directives

$InputTCPServerAddtlFrameDelimiter <Delimiter>
equivalent to: AddtlFrameDelimiter

$InputTCPServerDisableLFDelimiter on/off
equivalent to: DisableLFDelimiter

$InputTCPServerNotifyOnConnectionClose on/off
equivalent to: NotifyOnConnectionClose

$InputTCPServerKeepAlive <on/**off**>**
equivalent to: KeepAlive

$InputTCPServerKeepAlive_probes <number>
Equivalent to: KeepAlive.Probes

$InputTCPServerKeepAlive_intvl <number>
Equivalent to: KeepAlive.Interval

$InputTCPServerKeepAlive_time <number>
Equivalent to: KeepAlive.Time

$InputTCPServerRun <port>
equivalent to: Port

$InputTCPFlowControl on/off
equivalent to: FlowControl

$InputTCPMaxListeners <number>
equivalent to: MaxListeners

$InputTCPMaxSessions <number>
equivalent to: MaxSessions

$InputTCPServerStreamDriverMode <number>
equivalent to: StreamDriver.Mode

$InputTCPServerInputName <name>
equivalent to: Name

$InputTCPServerStreamDriverAuthMode <mode-string>
equivalent to: StreamDriver.AuthMode

$InputTCPServerStreamDriverPermittedPeer <id-string>
equivalent to: PermittedPeer.

$InputTCPServerBindRuleset <ruleset>
equivalent to: Ruleset.

$InputTCPSupportOctetCountedFraming on/off
equivalent to: SupportOctetCountedFraming

Caveats/Known Bugs

- module always binds to all interfaces
- can not be loaded together with imgssapi (which includes the functionality of imtcp)
- increasing MaxSessions and MaxListeners doesn't change MaxOpenFiles, consider increasing this global configuration parameter too (on Linux check the actual value for running process by in /proc/PID/limits; default limit on Linux is 1024)

Example

This sets up a TCP server on port 514 and permits it to accept up to 500 connections:

```
$ModLoad imtcp # needs to be done just once
$InputTCPMaxSessions 500
$InputTCPServerRun 514
```

Note that the parameters (here: max sessions) need to be set **before** the listener is activated. Otherwise, the parameters will not apply.

imudp: UDP Syslog Input Module

Module Name:	**imudp**
Author:	Rainer Gerhards <rgerhards@adiscon.com>

Provides the ability to receive syslog messages via UDP.

Multiple receivers may be configured by specifying multiple input statements.

Note that in order to enable UDP reception, Firewall rules probably need to be modified as well. Also, SELinux may need additional rules.

Configuration Parameters

Module Parameters

TimeRequery <nbr-of-times>
Default: 2

This is a performance optimization. Getting the system time is very costly. With this setting, imudp can be instructed to obtain the precise time only once every n-times. This logic is only activated if messages come in at a very fast rate, so doing less frequent time calls should usually be acceptable. The default value is two, because we have seen that even without optimization the kernel often returns twice the identical time. You can set this value as high as you like, but do so at your own risk. The higher the value, the less precise the timestamp.

Note: the timeRequery is done based on executed system calls (**not** messages received). So when batch sizes are used, multiple messages are received with one system call. All of these messages always receive the same timestamp, as they are effectively received at the same time. When there is very high traffic and successive system calls immediately return the next batch of messages, the time requery logic kicks in, which means that by default time is only queried for every second batch. Again, this should not cause a too-much deviation as it requires messages to come in very rapidly. However, we advise not to set the "timeRequery" parameter to a large value (larger than 10) if input batches are used.

SchedulingPolicy <rr/fifo/other>
Default: unset

Can be used the set the scheduler priority, if the necessary functionality is provided by the platform. Most useful to select "fifo" for real-time processing under Linux (and thus reduce chance of packet loss).

SchedulingPriority <number>
Default: unset

Scheduling priority to use.

batchSize <number>
Default: 32

This parameter is only meaningful if the system support recvmmsg() (newer Linux OSs do this). The parameter is silently ignored if the system does not support it. If supported, it sets the maximum number of UDP messages that can be obtained with a single OS call. For systems with high UDP traffic, a relatively high batch size can reduce system overhead and improve performance. However, this parameter should not be overdone. For each buffer, max message size bytes are statically required. Also, a too-high number leads to reduced efficiency, as some structures need to be completely initialized before the OS call is done. We would suggest to not set it above a value of 128, except if experimental results show that this is useful.

threads <number>
Available since: 7.5.5

Default: 1

Number of worker threads to process incoming messages. These threads are utilized to pull data off the network. On a busy system, additional threads (but not more than there are CPUs/Cores) can help improving performance and avoiding message loss. Note that with too many threads, performance can suffer. There is a hard upper limit

on the number of threads that can be defined. Currently, this limit is set to 32. It may increase in the future when massive multicore processors become available.

Input Parameters

..index:: imudp; address (input parameter)

Address <IP>
*Default: **

Local IP address (or name) the UDP server should bind to. Use "*" to bind to all of the machine's addresses.

Port <port>
Default: 514

Specifies the port the server shall listen to.. Either a single port can be specified or an array of ports. If multiple ports are specified, a listener will be automatically started for each port. Thus, no additional inputs need to be configured.

Single port: Port="514"

Array of ports: Port=["514","515","10514","..."]

Device <device>
Default: none

Bind socket to given device (e.g., eth0)

For Linux with VRF support, the Device option can be used to specify the VRF for the Address.

Ruleset <ruleset>
Default: RSYSLOG_DefaultRuleset

Binds the listener to a specific *ruleset*.

RateLimit.Interval [number]
Available since: 7.3.1

Default: 0

The rate-limiting interval in seconds. Value 0 turns off rate limiting. Set it to a number of seconds (5 recommended) to activate rate-limiting.

RateLimit.Burst [number]
Available since: 7.3.1

Default: 10000

Specifies the rate-limiting burst in number of messages.

name [name]
Available since: 8.3.3

Default: imudp

specifies the value of the inputname property. In older versions, this was always "imudp" for all listeners, which still is the default. Starting with 7.3.9 it can be set to different values for each listener. Note that when a single input statement defines multipe listner ports, the inputname will be the same for all of them. If you want to differentiate in that case, use "name.appendPort" to make them unique. Note that the "name" parameter can be an empty string. In that case, the corresponding inputname property will obviously also be the empty string. This is primarily meant to be used together with "name.appendPort" to set the inputname equal to the port.

InputName [name]
Available since: 7.3.9

Deprecated

This provides the same functionality as "name". It is the historical parameter name and should not be used in new configurations. It is scheduled to be removed some time in the future.

name.appendPort [on/off]
Available since: 7.3.9

Default: off

Appends the port the the inputname property. Note that when no "name" is specified, the default of "imudp" is used and the port is appended to that default. So, for example, a listner port of 514 in that case will lead to an inputname of "imudp514". The ability to append a port is most useful when multiple ports are defined for a single input and each of the inputnames shall be unique. Note that there currently is no differentiation between IPv4/v6 listeners on the same port.

InputName.AppendPort [on/off]
Available since: 7.3.9

Deprecated

This provides the same functionality as "name.appendPort". It is the historical parameter name and should not be used in new configurations. It is scheduled to be removed some time in the future.

defaultTZ <timezone-info>
Default: unset

This is an **experimental** parameter; details may change at any time and it may also be discoutinued without any early warning. Permits to set a default timezone for this listener. This is useful when working with legacy syslog (RFC3164 et al) residing in different timezones. If set it will be used as timezone for all messages **that do not contain timezone info**. Currently, the format **must** be "+/-hh:mm", e.g. "-05:00", "+01:30". Other formats, including TZ names (like EST) are NOT yet supported. Note that consequently no daylight saving settings are evaluated when working with timezones. If an invalid format is used, "interesting" things can happen, among them malformed timestamps and rsyslogd segfaults. This will obviously be changed at the time this feature becomes non-experimental.

rcvbufSize [size]
Available since: 7.5.3

Default: unset

Maximum Value: 1G

This request a socket receive buffer of specific size from the operating system. It is an expert parameter, which should only be changed for a good reason. Note that setting this parameter disables Linux auto-tuning, which usually works pretty well. The default value is 0, which means "keep the OS buffer size unchanged". This is a size value. So in addition to pure integer values, sizes like "256k", "1m" and the like can be specified. Note that setting very large sizes may require root or other special privileges. Also note that the OS may slightly adjust the value or shrink it to a system-set max value if the user is not sufficiently privileged. Technically, this parameter will result in a setsockopt() call with SO_RCVBUF (and SO_RCVBUFFORCE if it is available).

Statistic Counter

This plugin maintains *statistics* for each listener and for each worker thread.

The listener statistic is named starting with "imudp", followed followed by the listener IP, a colon and port in parenthesis. For example, the counter for a listener on port 514 (on all IPs) with no set name is called "imudp(*:514)".

If an "inputname" is defined for a listener, that inputname is used instead of "imudp" as statistic name. For example, if the inputname is set to "myudpinut", that corresponding statistic name in above case would be "myudpinput(*:514)". This has been introduced in 7.5.3.

The following properties are maintained for each listener:

- **submitted** - total number of messages submitted for processing since startup

The worker thread (in short: worker) statistic is named "imudp(wX)" where "X" is the worker thread ID, which is an monotonically increasing integer starting at 0. This means the first worker will have the name "imudp(w0), the second "imudp(w1)" and so on. Note that workers are all equal. It doesn't really matter which worker processes which messages, so the actual worker ID is not of much concern. More interesting is to check how the load is spread between the worker. Also note that there is no fixed worker-to-listener relationship: all workers process messages from all listeners.

Note: worker thread statistics are available starting with rsyslog 7.5.5.

The following properties are maintained for each worker thread:

- **called.recvmmsg** - number of recvmmsg() OS calls done
- **called.recvmsg** - number of recvmsg() OS calls done
- **msgs.received** - number of actual messages received

See Also

- rsyslog video tutorial on how to store remote messages in a separate file.
- Description of rsyslog statistic counters. This also describes all imudp counters.

Caveats/Known Bugs

- Scheduling parameters are set **after** privileges have been dropped. In most cases, this means that setting them will not be possible after privilege drop. This may be worked around by using a sufficiently-privileged user account.

Samples

This sets up an UPD server on port 514:

```
module(load="imudp") # needs to be done just once
input(type="imudp" port="514")
```

This sets up a UDP server on port 514 bound to device eth0:

```
module(load="imudp") # needs to be done just once
input(type="imudp" port="514" device="eth0")
```

The following sample is mostly equivalent to the first one, but request a larger rcvuf size. Note that 1m most probably will not be honored by the OS until the user is sufficiently privileged.

```
module(load="imudp") # needs to be done just once
input(type="imudp" port="514" rcvbufSize="1m")
```

In the next example, we set up three listeners at ports 10514, 10515 and 10516 and assign a listner name of "udp" to it, followed by the port number:

```
module(load="imudp")
input(type="imudp" port=["10514","10515","10516"]
      inputname="udp" inputname.appendPort="on")
```

The next example is almost equal to the previous one, but now the inputname property will just be set to the port number. So if a message was received on port 10515, the input name will be "10515" in this example whereas it was "udp10515" in the previous one. Note that to do that we set the inputname to the empty string.

```
module(load="imudp")
input(type="imudp" port=["10514","10515","10516"]
      inputname="" inputname.appendPort="on")
```

Legacy Configuration Directives

Legacy configuration parameters should **not** be used when crafting new configuration files. It is easy to get things wrong with them.

Directive	Equivalent Parameter	Requires
$UDPServerTimeRequery <nbr-of-times>	*TimeRequery*	
$IMUDPSchedulingPolicy <rr/fifo/other>	*SchedulingPolicy*	4.7.4+, 5.7.3+, 6.1.3+
$IMUDPSchedulingPriority <number>	*SchedulingPriority*	4.7.4+, 5.7.3+, 6.1.3+
$UDPServerAddress <IP>	Address	
$UDPServerRun <port>	Port	
$InputUDPServerBindRuleset <ruleset>	Ruleset	5.3.2+

Note: module parameters are given in *italics*. All others are input parameters.

Multiple receivers may be configured by specifying $UDPServerRun multiple times.

Legacy Sample

This sets up an UDP server on port 514:

```
$ModLoad imudp # needs to be done just once
$UDPServerRun 514
```


imuxsock: Unix Socket Input

Module Name:	**imuxsock**
Author:	Rainer Gerhards <rgerhards@adiscon.com>

Provides the ability to accept syslog messages via local Unix sockets. Most importantly, this is the mechanism by which the syslog(3) call delivers syslog messages to rsyslogd. So you need to have this module loaded to read the system log socket and be able to process log messages from applications running on the local system.

Application-provided timestamps are ignored by default. This is needed, as some programs (e.g. sshd) log with inconsistent timezone information, what messes up the local logs (which by default don't even contain time zone information). This seems to be consistent with what sysklogd did for the past four years. Alternate behaviour may

be desirable if gateway-like processes send messages via the local log slot - in this case, it can be enabled via the IgnoreTimestamp and SysSock.IgnoreTimestamp config directives

There is input rate limiting available, (since 5.7.1) to guard you against the problems of a wild running logging process. If more than SysSock.RateLimit.Interval * SysSock.RateLimit.Burst log messages are emitted from the same process, those messages with SysSock.RateLimit.Severity or lower will be dropped. It is not possible to recover anything about these messages, but imuxsock will tell you how many it has dropped one the interval has expired AND the next message is logged. Rate-limiting depends on SCM_CREDENTIALS. If the platform does not support this socket option, rate limiting is turned off. If multiple sockets are configured, rate limiting works independently on each of them (that should be what you usually expect). The same functionality is available for additional log sockets, in which case the config statements just use the prefix RateLimit... but otherwise works exactly the same. When working with severities, please keep in mind that higher severity numbers mean lower severity and configure things accordingly. To turn off rate limiting, set the interval to zero.

Unix log sockets can be flow-controlled. That is, if processing queues fill up, the unix socket reader is blocked for a short while. This may be useful to prevent overruning the queues (which may cause excessive disk-io where it actually would not be needed). However, flow-controlling a log socket (and especially the system log socket) can lead to a very unresponsive system. As such, flow control is disabled by default. That means any log records are places as quickly as possible into the processing queues. If you would like to have flow control, you need to enable it via the SysSock.FlowControl and FlowControl config directives. Just make sure you thought about the implications. Note that for many systems, turning on flow control does not hurt.

Starting with rsyslog 5.9.4, trusted syslog properties are available. These require a recent enough Linux Kernel and access to the /proc file system. In other words, this may not work on all platforms and may not work fully when privileges are dropped (depending on how they are dropped). Note that trusted properties can be very useful, but also typically cause the message to grow rather large. Also, the format of log messages is obviously changed by adding the trusted properties at the end. For these reasons, the feature is **not enabled by default**. If you want to use it, you must turn it on (via SysSock.Annotate and Annotate).

Configuration Parameters

Global Parameters

- **SysSock.IgnoreTimestamp** [**on**/off] Ignore timestamps included in the messages, applies to messages received via the system log socket.
- **SysSock.IgnoreOwnMessages** [**on**/off] (available since 7.3.7) Ignores messages that originated from the same instance of rsyslogd. There usually is no reason to receive messages from ourselves. This setting is vital when writing messages to the Linux journal. See omjournal module documentation for a more in-depth description.
- **SysSock.Use** (imuxsock) [**on**/off] - Listen on the local log socket. This is most useful if you run multiple instances of rsyslogd where only one shall handle the system log socket.
- **SysSock.Name** <name-of-socket>
- **SysSock.FlowControl** [on/**off**] - specifies if flow control should be applied to the system log socket.
- **SysSock.UsePIDFromSystem** [on/**off**] - specifies if the pid being logged shall be obtained from the log socket itself. If so, the TAG part of the message is rewritten. It is recommended to turn this option on, but the default is "off" to keep compatible with earlier versions of rsyslog.
- **SysSock.RateLimit.Interval** [number] - specifies the rate-limiting interval in seconds. Default value is 0, which turns off rate limiting. Set it to a number of seconds (5 recommended) to activate rate-limiting. The default of 0 has been chosen as people experienced problems with this feature activated by default. Now it needs an explicit opt-in by setting this parameter.
- **SysSock.RateLimit.Burst** [number] - specifies the rate-limiting burst in number of messages. Default is 200.

- **SysSock.RateLimit.Severity** [numerical severity] - specifies the severity of messages that shall be rate-limited.
- **SysSock.UseSysTimeStamp** [**on**/off] the same as the input parameter UseSysTimeStamp, but for the system log socket. See description there.
- **SysSock.Annotate** <on/**off**> turn on annotation/trusted properties for the system log socket.
- **SysSock.ParseTrusted** <on/**off**> if Annotation is turned on, create JSON/lumberjack properties out of the trusted properties (which can be accessed via RainerScript JSON Variables, e.g. "$!pid") instead of adding them to the message.
- **SysSock.Unlink** <**on**/off> (available since 7.3.9) if turned on (default), the system socket is unlinked and re-created when opened and also unlinked when finally closed. Note that this setting has no effect when running under systemd control (because systemd handles the socket).
- **sysSock.useSpecialParser** (available since 8.9.0) The equivalent of the "useSpecialParser" input parameter for the system socket.
- **sysSock.parseHostname** (available since 8.9.0) The equivalent of the "parseHostname" input parameter for the system socket.

Input Parameters

- **ruleset** [name] Binds specified ruleset to this input. If not set, the default ruleset is bound. (available since 8.17.0)
- **IgnoreTimestamp** [**on**/off] Ignore timestamps included in the message. Applies to the next socket being added.
- **IgnoreOwnMessages** [**on**/off] (available since 7.3.7) Ignore messages that originated from the same instance of rsyslogd. There usually is no reason to receive messages from ourselfs. This setting is vital when writing messages to the Linux journal. See omjournal module documentation for a more in-depth description.
- **FlowControl** [on/**off**] - specifies if flow control should be applied to the next socket.
- **RateLimit.Interval** [number] - specifies the rate-limiting interval in seconds. Default value is 0, which turns off rate limiting. Set it to a number of seconds (5 recommended) to activate rate-limiting. The default of 0 has been chosen as people experienced problems with this feature activated by default. Now it needs an explicit opt-in by setting this parameter.
- **RateLimit.Burst** [number] - specifies the rate-limiting burst in number of messages. Default is 200.
- **RateLimit.Severity** [numerical severity] - specifies the severity of messages that shall be rate-limited.
- **UsePIDFromSystem** [on/**off**] - specifies if the pid being logged shall be obtained from the log socket itself. If so, the TAG part of the message is rewritten. It is recommended to turn this option on, but the default is "off" to keep compatible with earlier versions of rsyslog.
- **UseSysTimeStamp** [**on**/off] instructs imuxsock to obtain message time from the system (via control messages) instead of using time recorded inside the message. This may be most useful in combination with systemd. Note: this option was introduced with version 5.9.1. Due to the usefulness of it, we decided to enable it by default. As such, 5.9.1 and above behave slightly different than previous versions. However, we do not see how this could negatively affect existing environments.
- **CreatePath** [on/**off**] - create directories in the socket path if they do not already exist. They are created with 0755 permissions with the owner being the process under which rsyslogd runs. The default is not to create directories. Keep in mind, though, that rsyslogd always creates the socket itself if it does not exist (just not the directories by default). Note that this statement affects the next Socket directive that follows in sequence in the configuration file. It never works on the system log socket (where it is deemed unnecessary). Also note that it is automatically being reset to "off" after the Socket directive, so if you would have it active for two additional listen sockets, you need to specify it in front of each one. This option is primarily considered useful for defining

additional sockets that reside on non-permanent file systems. As rsyslogd probably starts up before the daemons that create these sockets, it is a vehicle to enable rsyslogd to listen to those sockets even though their directories do not yet exist.

- **Socket** <name-of-socket> adds additional unix socket, default none – former -a option
- **HostName** <hostname> permits to override the hostname that shall be used inside messages taken from the **next** Socket socket. Note that the hostname must be specified before the $AddUnixListenSocket configuration directive, and it will only affect the next one and then automatically be reset. This functionality is provided so that the local hostname can be overridden in cases where that is desired.
- **Annotate** <on/**off**> turn on annotation/trusted properties for the non-system log socket in question.
- **ParseTrusted** <on/**off**> equivalent to the SysSock.ParseTrusted module parameter, but applies to the input that is being defined.
- **Unlink <on**/off> (available since 7.3.9) if turned on (default), the socket is unlinked and re-created when opened and also unlinked when finally closed. Set it to off if you handle socket creation yourself. Note that handling socket creation oneself has the advantage that a limited amount of messages may be queued by the OS if rsyslog is not running.
- **useSpecialParser <on**/off> (available since 8.9.0) If turned on (the default and the way it was up until 8.8.0) a special parser is used that parses the format that is usually used on the system log socket (the one syslog(3) creates). If set to "off", the regular parser chain is used, in which case the format on the log socket can be arbitrary. Note that when the special parser is used, rsyslog is able to inject a more precise timestamp into the message (it is obtained from the log socket). If the regular parser chain is used, this is not possible.
- **parseHostname** <on/**off**> (available since 8.9.0) Normally, the local log sockets do *not* contain hostnames. With this directive, the parser chain can be instructed to not expect them (setting "off", the default). If set to on, parsers will expect hostnames just like in regular formats. Note: this option only has an effect if *useSpecialParsers* is set to "off".

Statistic Counter

This plugin maintains a global *statistics* with the following properties:

- **submitted** - total number of messages submitted for processing since startup
- **ratelimit.discarded** - number of messages discarded due to rate limiting
- **ratelimit.numratelimiters** - number of currently active rate limiters (smal data structures used for the rate limiting logic)

See Also

- What are "trusted properties"?
- Why does imuxsock not work on Solaris?

Caveats/Known Bugs

- There is a compile-time limit of 50 concurrent sockets. If you need more, you need to change the array size in imuxsock.c.
- This documentation is sparse and incomplete.

Samples

The following sample is the minimum setup required to accept syslog messages from applications running on the local system.

```
module(load="imuxsock" # needs to be done just once
       SysSock.FlowControl="on") # enable flow control (use if needed)
```

The following sample is similiar to the first one, but enables trusted properties, which are put into JSON/lumberjack variables.

```
module(load="imuxsock" SysSock.Annotate="on" SysSock.ParseTrusted="on")
```

The following sample is a configuration where rsyslogd pulls logs from two jails, and assigns different hostnames to each of the jails:

```
module(load="imuxsock") # needs to be done just once
input(type="imuxsock" HostName="jail1.example.net"
      Socket="/jail/1/dev/log") input(type="imuxsock"
      HostName="jail2.example.net" Socket="/jail/2/dev/log")
```

The following sample is a configuration where rsyslogd reads the openssh log messages via a separate socket, but this socket is created on a temporary file system. As rsyslogd starts up before the sshd, it needs to create the socket directories, because it otherwise can not open the socket and thus not listen to openssh messages. Note that it is vital not to place any other socket between the CreatePath and the Socket.

```
module(load="imuxsock") # needs to be done just once
input(type="imuxsock" Socket="/var/run/sshd/dev/log" CreatePath="on")
```

The following sample is used to turn off input rate limiting on the system log socket.

```
module(load="imuxsock" # needs to be done just once
       SysSock.RateLimit.Interval="0") # turn off rate limiting
```

The following sample is used activate message annotation and thus trusted properties on the system log socket. module(load=”imuxsock” # needs to be done just once SysSock.Annotate=”on”)

Legacy Configuration Directives

Legacy directives should NOT be used when writing new configuration files.

Note that the legacy configuration parameters do **not** affect new-style definitions via the input() object. This is by design. To set defaults for input() objects, use module parameters in the

```
module(load="imuxsock" ...)
```

object.

Read about *the importance of order in legacy configuration* to understand how to use these configuration directives.

- **$InputUnixListenSocketIgnoreMsgTimestamp** [**on**/off] equivalent to: IgnoreTimestamp.
- **$InputUnixListenSocketFlowControl** [on/**off**] - equivalent to: FlowControl .
- **$IMUXSockRateLimitInterval** [number] - equivalent to: RateLimit.Interval
- **$IMUXSockRateLimitBurst** [number] - equivalent to: RateLimit.Burst

- **$IMUXSockRateLimitSeverity** [numerical severity] - equivalent to: RateLimit.Severity
- **$IMUXSockLocalIPIF** [interface name] - (available since 5.9.6) - if provided, the IP of the specified interface (e.g. "eth0") shall be used as fromhost-ip for imuxsock-originating messages. If this directive is not given OR the interface cannot be found (or has no IP address), the default of "127.0.0.1" is used.
- **$InputUnixListenSocketUsePIDFromSystem** [on/**off**] - equivalent to: UsePIDFromSystem. This option was introduced in 5.7.0.
- **$InputUnixListenSocketUseSysTimeStamp** [**on**/off] equivalent to: UseSysTimeStamp .
- **$SystemLogSocketIgnoreMsgTimestamp [on/off]** equivalent to: SysSock.IgnoreTimestamp.
- **$OmitLocalLogging** (imuxsock) [on/**off**] - The **inverse** of SysSock.Use.
- **$SystemLogSocketName** <name-of-socket> equivalent to: SysSock.Name
- **$SystemLogFlowControl** [on/**off**] - equivalent to: SysSock.FlowControl.
- **$SystemLogUsePIDFromSystem** [on/**off**] - equivalent to: SysSock.UsePIDFromSystem. This option was introduced in 5.7.0.
- **$SystemLogRateLimitInterval** [number] - equivalent to: SysSock.RateLimit.Interval.
- **$SystemLogRateLimitBurst** [number] - equivalent to: SysSock.RateLimit.Burst
- **$SystemLogRateLimitSeverity** [numerical severity] - equivalent to: SysSock.RateLimit.Severity
- **$SystemLogUseSysTimeStamp** [**on**/off] equivalent to: SysSock.UseSysTimeStamp.
- **$InputUnixListenSocketCreatePath** [on/**off**] - equivalent to: CreatePath [available since 4.7.0 and 5.3.0]
- **$AddUnixListenSocket** <name-of-socket> equivalent to: Socket
- **$InputUnixListenSocketHostName** <hostname> equivalent to: HostName.
- **$InputUnixListenSocketAnnotate** <on/**off**> equivalent to: Annotate.
- **$SystemLogSocketAnnotate** <on/**off**> equivalent to: SysSock.Annotate.
- **$SystemLogSocketParseTrusted** <on/**off**> equivalent to: SysSock.ParseTrusted.

Caveats/Known Bugs:

- There is a compile-time limit of 50 concurrent sockets. If you need more, you need to change the array size in imuxsock.c.
- This documentation is sparse and incomplete.

Sample:

The following sample is the minimum setup required to accept syslog messages from applications running on the local system.

```
$ModLoad imuxsock # needs to be done just once
$SystemLogSocketFlowControl on # enable flow control (use if needed)
```

The following sample is a configuration where rsyslogd pulls logs from two jails, and assigns different hostnames to each of the jails:

```
$ModLoad imuxsock # needs to be done just once
$InputUnixListenSocketHostName jail1.example.net
$AddUnixListenSocket /jail/1/dev/log
$InputUnixListenSocketHostName jail2.example.net
$AddUnixListenSocket /jail/2/dev/log
```

The following sample is a configuration where rsyslogd reads the openssh log messages via a separate socket, but this socket is created on a temporary file system. As rsyslogd starts up before the sshd, it needs to create the socket directories, because it otherwise can not open the socket and thus not listen to openssh messages. Note that it is vital not to place any other socket between the $InputUnixListenSocketCreatePath and the $InputUnixListenSocketHostName.

```
$ModLoad imuxsock # needs to be done just once
$InputUnixListenSocketCreatePath on # turn on for *next* socket
$InputUnixListenSocket /var/run/sshd/dev/log
```

The following sample is used to turn off input rate limiting on the system log socket.

```
$ModLoad imuxsock # needs to be done just once
$SystemLogRateLimitInterval 0 # turn off rate limiting
```

The following sample is used to activate message annotation and thus trusted properties on the system log socket.

```
$ModLoad imuxsock # needs to be done just once
$SystemLogSocketAnnotate on
```


Parser Modules

Parser modules are used to parse message content, once the message has been received. They can be used to process custom message formats or invalidly formatted messages. For details, please see the *rsyslog message parser documentation*.

The current modules are currently provided as part of rsyslog:

pmciscoios

Author: Rainer Gerhards

Available since: 8.3.4+

This is a parser that understands Cisco IOS "syslog" format. Note that this format is quite different from RFC syslog format, and so the default parser chain cannot deal with it.

Note that due to large differences in IOS logging format, pmciscoios may currently not be able to handle all possible format variations. Nevertheless, it should be fairly easy to adapt it to additional requirements. So be sure to ask if you run into problems with format issues.

Note that if your Cisco system emits timezone information in a supported format, rsyslog will pick it up. In order to apply proper timezone offsets, the timezone ids (e.g. "EST") must be configured via the *timezone object*.

Note if the clock on the Cisco device has not been set and cannot be verified the Cisco will prepend the timestamp field with an asterisk (*). If the clock has gone out of sync with its configured NTP server the timestamp field will be prepended with a dot (.). In both of these cases parsing the timestamp would fail, therefore any preceding asterisks (*) or dots (.) are ignored. This may lead to "incorrect" timestamps being logged.

Parser Parameters

present.origin <boolean>
Default: off

This setting tell the parser if the origin field is present inside the message. Due to the nature of Cisco's logging format, the parser cannot sufficiently correctly deduce if the origin field is present or not (at least not with reasonable performance). As such, the parser must be provided with that information. If the origin is present, its value is stored inside the HOSTNAME message property.

present.xr <boolean>

Default: off

If syslog is recviced from an IOSXR device the syslog format will usually start with the RSP/LC/etc that produced the log, then the timestamp. It will also contain an additional syslog tag before the standard Cisco %TAG, this tag references the process that produced the log. In order to use this Cisco IOS parser module with XR format messages both of these additional fields must be ignored.

Example

We assume a scenario where we have some devices configured to emit origin information whereas some others do not. In order to differentiate between the two classes, rsyslog accepts input on different ports, one per class. For each port, an input() object is defined, which binds the port to a ruleset that uses the appropriately-configured parser. Except for the different parsers, processing shall be identical for both classes. In our first example we do this via a common ruleset which carries out the actual processing:

```
module(load="imtcp")
module(load="pmciscoios")

input(type="imtcp" port="10514" ruleset="withoutOrigin")
input(type="imtcp" port="10515" ruleset="withOrigin")

ruleset(name="common") {
    ... do processing here ...
}

ruleset(name="withoutOrigin" parser="rsyslog.ciscoios") {
    /* this ruleset uses the default parser which was
     * created during module load
     */
    call common
}

parser(name="custom.ciscoios.withOrigin" type="pmciscoios"
       present.origin="on")
ruleset(name="withOrigin" parser="custom.ciscoios.withOrigin") {
    /* this ruleset uses the parser defined immediately above */
    call common
}
```

The example configuration above is a good solution. However, it is possible to do the same thing in a somewhat condensed way, but if and only if the date stamp immediately follows the origin. In that case, the parser has a chance to detect if the origin is present or not. The key point here is to make sure the parser checking for the origin is given before the default one, in which case the first on will detect it does not match an pass on to the next one inside the parser chain. However, this comes at the expense of additional runtime overhead. The example below is **not** good practice – it is given as a purely educational sample to show some fine details of how parser definitions interact. In this case, we can use a single listener.

```
module(load="imtcp")
module(load="pmciscoios")
```

```
input(type="imtcp" port="10514" ruleset="ciscoBoth")

parser(name="custom.ciscoios.withOrigin" type="pmciscoios"
       present.origin="on")
ruleset(name="ciscoBoth"
        parser=["custom.ciscoios.withOrigin", "rsyslog.ciscoios"]) {
    ... do processing here ...
}
```

The following sample demonstrates how to handle Cisco IOS and IOSXR formats

```
module(load="imudp")
module(load="pmciscoios")

input(type="imudp" port="10514" ruleset="ios")
input(type="imudp" port="10515" ruleset="iosxr")

ruleset(name="common") {
    ... do processing here ...
}

ruleset(name="ios" parser="rsyslog.ciscoios") {
    call common
}

parser(name="custom.ciscoios.withXr" type="pmciscoios"
       present.xr="on")
ruleset(name="iosxr" parser="custom.ciscoios.withXr"] {
    call common
}
```

pmlastmsg: last message repeated n times

Module Name: **pmlastmsg**

Module Type: **parser module**

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available Since: 5.5.6

Description:

Some syslogds are known to emit severily malformed messages with content "last message repeated n times". These messages can mess up message reception, as they lead to wrong interpretation with the standard RFC3164 parser. Rather than trying to fix this issue in pmrfc3164, we have created a new parser module specifically for these messages. The reason is that some processing overhead is involved in processing these messages (they must be recognized) and we would not like to place this toll on every user but only on those actually in need of the feature. Note that the performance toll is not large – but if you expect a very high message rate with tenthousands of messages per second, you will notice a difference.

This module should be loaded first inside rsyslog's parser chain. It processes all those messages that contain a PRI, then none or some spaces and then the exact text (case-insensitive) "last message repeated n times" where n must be an integer. All other messages are left untouched.

Configuration Directives:

There do not currently exist any configuration directives for this module.

Examples:

This example is the typical use case, where some systems emit malformed "repeated msg" messages. Other than that, the default RFC5424 and RFC3164 parsers should be used. Note that when a parser is specified, the default parser chain is removed, so we need to specify all three parsers. We use this together with the default ruleset.

```
$ModLoad pmlastmsg   # This parser is not a built-in module.

# Note that parsers are tried in the order they appear in
# rsyslog.conf, so put pmlastmsg first.

$RulesetParser rsyslog.lastline

# As we have removed the default parser chain, we need to add the
# default parsers as well.
$RulesetParser rsyslog.rfc5424
$RulesetParser rsyslog.rfc3164

# Here come the typical rules, like... \*.\* /path/to/file.log
```

Caveats/Known Bugs:

currently none

This documentation is part of the rsyslog project.

Log Message Normalization Parser Module (pmnormalize)

Module Name: **pmnormalize**

Available since: 8.27.0

Author: Pascal Withopf <pascalwithopf1@gmail.com>

Description:

This parser normalizes messages with the specified rules and populates the properties for further use.

Action Parameters

`rulebase <word>`
Specifies which rulebase file is to use. If there are multiple pmnormalize instances, each one can use a different file. However, a single instance can use only a single file. This parameter or **rule** MUST be given, because normalization can only happen based on a rulebase. It is recommended that an absolute path name is given. Information on how to create the rulebase can be found in the liblognorm manual.

`rule <array>`
Contains an array of strings which will be put together as the rulebase. This parameter or **rulebase** MUST be given, because normalization can only happen based on a rulebase.

`undefinedpropertyerror <on/off>`
Default: off

With this parameter an error message is controlled, which will be put out every time pmnormalize can't normalize a message.

See Also

Caveats/Known Bugs

None known at this time.

Example

Sample 1:

In this sample messages are received via imtcp. Then they are normalized with the given rulebase and written to a file.

```
module(load="imtcp")
module(load="pmnormalize")

input(type="imtcp" port="13514" ruleset="ruleset")

parser(name="custom.pmnormalize" type="pmnormalize" rulebase="/tmp/rules.rulebase")

ruleset(name="ruleset" parser="custom.pmnormalize") {
      action(type="omfile" file="/tmp/output")
}
```

Sample 2:

In this sample messages are received via imtcp. Then they are normalized with the given rule array. After that they are written in a file.

```
module(load="imtcp")
module(load="pmnormalize")

input(type="imtcp" port="10514" ruleset="outp")

parser(name="custom.pmnormalize" type="pmnormalize" rule=[
              "rule=:<%pri:number%> %fromhost-ip:ipv4% %hostname:word%
↪%syslogtag:char-to:\\x3a%: %msg:rest%",
              "rule=:<%pri:number%> %hostname:word% %fromhost-ip:ipv4%
↪%syslogtag:char-to:\\x3a%: %msg:rest%"])

ruleset(name="outp" parser="custom.pmnormalize") {
      action(type="omfile" File="/tmp/output")
}
```

pmnull: Syslog Null Parser Module

Author: Pascal Withopf <pascalwithopf1@gmail.com>

When a message is received it is tried to match a set of parsers to get properties populated. This parser module sets all attributes to “” but rawmsg. There usually should be no need to use this module. It may be useful to process certain known-not-syslog messages.

Parser Parameters

tag <string>
Default: Empty (“”)

This setting sets the tag value to the message.

syslogfacility <int>
Default: 1

This setting sets the syslog facility value. The default comes from the rfc3164 standard.

syslogseverity <int>
Default: 5

This setting sets the syslog severity value. The default comes from the rfc3164 standard.

Example

In this example messages are received through imtcp on port 13514. The ruleset uses the parser pmnull which has the parameters tag, syslogfacility and syslogseverity given.

```
module(load="imtcp")
module(load="pmnull")

input(type="imtcp" port="13514" ruleset="ruleset")
parser(name="custom.pmnull" type="pmnull" tag="mytag" syslogfacility="3"
       syslogseverity="1")

ruleset(name="ruleset" parser=["custom.pmnull", "rsyslog.pmnull"]) {
      action(type="omfile" file="rsyslog.out.log")
}
```

In this example the ruleset uses the parser pmnull with the default parameters because no specifics were given.

```
module(load="imtcp")
module(load="pmnull")

input(type="imtcp" port="13514" ruleset="ruleset")
parser(name="custom.pmnull" type="pmnull")

ruleset(name="ruleset" parser="custom.pmnull") {
      action(type="omfile" file="rsyslog.out.log")
}
```

pmrfc3164: Parse RFC3164-formatted messages

Author: Rainer Gerhards

This parser module is for parsing messages according to the traditional/legacy syslog standard **RFC 3164**

It is part of the default parser chain.

The parser can also be customized to allow the parsing of specific formats, if they occur.

Parser Parameters

permit.squareBracketsInHostname <boolean>
Default: off

This setting tells the parser that hostnames that are enclosed by brackets should omit the brackets.

permit.slashesInHostname <boolean>
Default: off

Available since: 8.20.0

This setting tells the parser that hostnames may contain slashes. This is useful when messages e.g. from a syslog-ng releay chain are received. Syslog-ng puts the various relay hosts via slashes into the hostname field.

permit.AtSignsInHostname <boolean>
Default: off

Available since: 8.25.0

This setting tells the parser that hostnames may contain at-signs. This is useful when messages are relayed from a syslog-ng server in rfc3164 format. The hostname field sent by syslog-ng may be prefixed by the source name followed by an at-sign character.

force.tagEndingByColon <boolean>
Default: off

Available since: 8.25.0

This setting tells the parser that tag need to be ending by colon to be valid. In others case, the tag is set to dash ("-") without changing message.

remove.msgFirstSpace <boolean>
Default: off

Available since: 8.25.0

rfc3164 tell message is directly after tag including first white space. This option tell to remove the first white space in message just after reading. It make rfc3164 & rfc5424 syslog messages working in a better way.

detect.YearAfterTimestamp <boolean>
Default: off

Some devices send syslog messages in a format that is similar to RFC3164, but they also attach the year to the timestamp (which is not compliant to the RFC). With regular parsing, the year would be recognized to be the hostname and the hostname would become the syslogtag. This setting should prevent this. It is also limited to years between 2000 and 2099, so hostnames with numbers as their name can still be recognized correctly. But everything in this range will be detected as a year.

Example

We assume a scenario where some of the devices send malformed RFC3164 messages. The parser module will automatically detect the malformed sections and parse them accordingly.

```
module(load="imtcp")

input(type="imtcp" port="514" ruleset="customparser")

parser(name="custom.rfc3164"
       type="pmrfc3164"
       permit.squareBracketsInHostname="on"
```

```
      detect.YearAfterTimestamp="on")

ruleset(name="customparser" parser="custom.rfc3164") {
      ... do processing here ...
}
```

pmrfc3164sd: Parse RFC5424 structured data inside RFC3164 messages

A contributed module for supporting RFC5424 structured data inside RFC3164 messages (not supported by the rsyslog team)

pmrfc5424: Parse RFC5424-formatted messages

This is the new Syslog Standard.

RFC 5424

Message Modification Modules

Message modification modules are used to change the content of messages being processed. They can be implemented using either the output module or the parser module interface. From the rsyslog core's point of view, they actually are output or parser modules, it is their implementation that makes them special.

IP Address Anonymization Module (mmanon)

Module Name: mmanon

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available since: 7.3.7

Description:

The mmanon module permits to anonymize IP addresses. It is a message modification module that actually changes the IP address inside the message, so after calling mmanon, the original message can no longer be obtained. Note that anonymization will break digital signatures on the message, if they exist.

How are IP-Addresses defined?

We assume that an IP address consists of four octets in dotted notation, where each of the octets has a value between 0 and 255, inclusively.

Module Configuration Parameters:

Currently none.

Action Confguration Parameters:

- **mode** - default "rewrite"

  There exists the "simple" and "rewrite" mode. In simple mode, only octets as whole can be anonymized and the length of the message is never changed. This means that when the last three octets of the address 10.1.12.123 are anonymized, the result will be 10.0.00.000. This means that the length of the original octets is still visible and may be used to draw some privacy-evasive conclusions. This mode is slightly faster than "overwrite" mode, and this may matter in high throughput environments. The default "rewrite" mode will do full anonymization of

any number of bits and it will also normlize the address, so that no information about the original IP address is available. So in the above example, 10.1.12.123 would be anonymized to 10.0.0.0.

- **ipv4.bits** - default "16"

  This set the number of bits that should be anonymized (bits are from the right, so lower bits are anonymized first). This setting permits to save network information while still anonymizing user-specific data. The more bits you discard, the better the anonymization obviously is. The default of 16 bits reflects what German data privacy rules consider as being sufficinetly anonymized. We assume, this can also be used as a rough but conservative guideline for other countries. Note: when in simple mode, only bits on a byte boundary can be specified. As such, any value other than 8, 16, 24 or 32 is invalid. If an invalid value is given, it is rounded to the next byte boundary (so we favor stronger anonymization in that case). For example, a bit value of 12 will become 16 in simple mode (an error message is also emitted).

- **replacementChar** - default "x"

  In simple mode, this sets the character that the to-be-anonymized part of the IP address is to be overwritten with. In rewrite mode, this parameter is **not permitted**, as in this case we need not necessarily rewrite full octets. As such, the anonymized part is always zero-filled and replacementChar is of no use. If it is specified, an error message is emitted and the parameter ignored.

See Also

- Howto anonymize messages that go to specific files

Caveats/Known Bugs:

- **only IPv4** is supported

Samples:

In this snippet, we write one file without anonymization and another one with the message anonymized. Note that once mmanon has run, access to the original message is no longer possible (execept if stored in user variables before anonymization).

```
module(load="mmanon")
action(type="omfile" file="/path/to/non-anon.log")
action(type="mmanon")
action(type="omfile" file="/path/to/anon.log")
```

This next snippet is almost identical to the first one, but here we anonymize the full IPv4 address. Note that by modifying the number of bits, you can anonymize different parts of the address. Keep in mind that in simple mode (used here), the bit values must match IP address bytes, so for IPv4 only the values 8, 16, 24 and 32 are valid. Also, in this example the replacement is done via asterisks instead of lower-case "x"-letters. Also keep in mind that "replacementChar" can only be set in simple mode.

```
module(load="mmanon") action(type="omfile" file="/path/to/non-anon.log")
action(type="mmanon" ipv4.bits="32" mode="simple" replacementChar="\*")
action(type="omfile" file="/path/to/anon.log")
```

The next snippet is also based on the first one, but anonimzes an "odd" number of bits, 12. The value of 12 is used by some folks as a compromise between keeping privacy and still permiting to gain some more in-depth insight from log files. Note that anonymizing 12 bits may be insufficient to fulfill legal requirements (if such exist).

```
module(load="mmanon") action(type="omfile" file="/path/to/non-anon.log")
action(type="mmanon" ipv4.bits="12") action(type="omfile"
file="/path/to/anon.log")
```

mmcount

Module Name: mmcount

Author: Bala.FA <barumuga@redhat.com>

Status: Non project-supported module - contact author or rsyslog mailing list for questions

Available since: 7.5.0

Description:

```
mmcount: message modification plugin which counts messages

This module provides the capability to count log messages by severity
or json property of given app-name.  The count value is added into the
log message as json property named 'mmcount'

Example usage of the module in the configuration file

 module(load="mmcount")

 # count each severity of appname gluster
 action(type="mmcount" appname="gluster")

 # count each value of gf_code of appname gluster
 action(type="mmcount" appname="glusterd" key="!gf_code")

 # count value 9999 of gf_code of appname gluster
 action(type="mmcount" appname="glusterfsd" key="!gf_code" value="9999")

 # send email for every 50th mmcount
 if $app-name == 'glusterfsd' and $!mmcount <> 0 and $!mmcount % 50 == 0 then {
    $ActionMailSMTPServer smtp.example.com
    $ActionMailFrom rsyslog@example.com
    $ActionMailTo glusteradmin@example.com
    $template mailSubject,"50th message of gf_code=9999 on %hostname%"
    $template mailBody,"RSYSLOG Alert\r\nmsg='%msg%'"
    $ActionMailSubject mailSubject
    $ActionExecOnlyOnceEveryInterval 30
    :ommail:;RSYSLOG_SyslogProtocol23Format
 }
```


MaxMind/GeoIP DB lookup (mmdblookup)

Module Name:	mmdblookup
Author:	chenryn
Available:	8.24+

Description:

MaxMindDB is the new file format for storing information about IP addresses in a highly optimized, flexible database format. GeoIP2 Databases are available in the MaxMind DB format.

Plugin author claimed a MaxMindDB vs GeoIP speed around 4 to 6 times.

Module parameters

- **container** - default "!iplocation" v8.28.0+

  Specifies the container to be used to store the fields ammended by mmdblookup.

Input parameters

- **key** - default none

  Name of field containing IP address.

- **mmdbfile** - default none

  Location of Maxmind DB file.

- **fields** - default none

  Fields that will be appended to processed message. The fields will always be appended in the container used by mmdblookup (which may be overriden by the "container" parameter on module load).

  By default, the maxmindb field name is used for variables. This can be overriden by specifying a custom name between colons at the beginnig of the field name. As usual, bang signs denote path levels. So for example, if you want to extract "!city!names!en" but rename it to "cityname", you can use ":cityname:!city!names!en" as field name.

Examples

```
# load module
module( load="mmdblookup" )

action( type="mmdblookup" mmdbfile="/etc/rsyslog.d/GeoLite2-City.mmdb" fields=["!
↪continent!code","!location"] key="!clientip" )
```

The following example uses a custom container and custom field name:

```
# load module
module(load="mmdblookup" container="!geo_ip")

action(type="mmdblookup" mmdbfile="/etc/rsyslog.d/GeoLite2-City.mmdb"
       fields=[":continent:!continent!code", ":loc:!location"]
       key="!clientip"
      )
```

Building

To compile Rsyslog with mmdblookup you'll need to:

- install *libmaxminddb-devel* package
- set *–enable-mmdblookup* on configure

Support module for external message modification modules

Module Name: mmexternal

Available since: 8.3.0

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module permits to integrate external message modification plugins into rsyslog.

For details on the interface specification, see rsyslog's source in the ./plugins/external/INTERFACE.md.

Action Parameters:

- **binary**

  The name of the external message modification plugin to be called. This can be a full path name.

- **interface.input**

  This can either be "msg", "rawmsg" or "fulljson". In case of "fulljson", the message object is provided as a json object. Otherwise, the respective property is provided. This setting **must** match the external plugin's expectations. Check the external plugin documentation for what needs to be used.

- **output**

  This is a debug aid. If set, this is a filename where the plugins output is logged. Note that the output is also being processed as usual by rsyslog. Setting this parameter thus gives insight into the internal processing that happens between plugin and rsyslog core.

- **forcesingleinstance**

  This is an expert parameter, just like the equivalent *omprog* parameter. See the message modification plugin's documentation if it is needed.

Caveats/Known Bugs:

- None.

Sample:

The following config file snippet is used to write execute an external message modification module "mmexternal.py". Note that the path to the module is specified here. This is necessary if the module is not in the default search path.

```
module (load="mmexternal") # needs to be done only once inside the config

action(type="mmexternal" binary="/path/to/mmexternal.py")
```

Fields Extraction Module (mmfields)

Module Name: mmfields

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available since: 7.5.1

Description:

The mmfield module permits to extract fields. It is an alternate to using the property replacer field extraction capabilities. In contrast to the property replacer, all fields are extracted as once and stored inside the structured data part (more precisely: they become Lumberjack [JSON] properties).

Using this module is of special advantage if a field-based log format is to be processed, like for example CEF **and** and either a large number of fields is needed or a specific field is used multiple times inside filters. In these scenarios, mmfields potentially offers better performance than the property replacer of the RainerScript field extraction method. The reason is that mmfields extracts all fields as one big sweep, whereas the other methods extract fields individually, which requires multiple passes through the same data. On the other hand, adding field content to the rsyslog property dictionary also has some overhead, so for high-performance use cases it is suggested to do some performance testing before finally deciding which method to use. This is most important if only a smaller subset of the fields is actually needed.

In any case, mmfields provides a very handy and easy to use way to parse structured data into a it's individual data items. Again, a primiary use case was support for CEF (Common Event Format), which is made extremely easy to do with this module.

This module is implemented via the action interface. Thus it can be conditionally used depending on some prequisites.

Module Configuration Parameters:

Currently none.

Action Confguration Parameters:

- **separator** - separatorChar (default ',') This is the character used to separate fields. Currently, only a single character is permitted, while the RainerScript method permits to specify multi-character separator strings. For CEF, this is not required. If there is actual need to support multi-character separator strings, support can relatively easy be added. It is suggested to request it on the rsyslog mailing list, together with the use case - we intend to add functionality only if there is a real use case behind the request (in the past we too-often implemented things that actually never got used). The fields are named f*nbr*, where *nbr* is the field number starting with one and being incremented for each field.
- **jsonRoot** - path (default "!") This parameters specifies into which json path the extracted fields shall be written. The default is to use the json root object itself.

Caveats/Known Bugs:

- Currently none.

Samples:

This is a very simple use case where each message is parsed. The default separator character of comma is being used.

```
module(load="mmfields")
template(name="ftpl"
         type=string
         string="%$!%\\n")
action(type="mmfields")
action(type="mmfile"
       file="/path/to/logfile"
       template="ftpl")
```

The following sample is similar to the previous one, but this time the colon is used as separator and data is written into the "$!mmfields" json path.

```
module(load="mmfields")
template(name="ftpl"
         type=string
         string="%$!%\\n")
action(type="mmfields"
       separator=":"
       jsonRoot="!mmfields")
       action(type="mmfile"
```

```
        file="/path/to/logfile"
        template="ftpl")
```


JSON/CEE Structured Content Extraction Module (mmjsonparse)

Module Name: **mmjsonparse**

Available since:6.6.0+

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module provides support for parsing structured log messages that follow the CEE/lumberjack spec. The so-called "CEE cookie" is checked and, if present, the JSON-encoded structured message content is parsed. The properties are then available as original message properties.

The "CEE cookie" is the character squence "@cee:" which must prepend the actual JSON. Note that the JSON must be valid and MUST NOT be followed by any non-JSON message. If either of these conditions is not true, mmjsonparse will **not** parse the associated JSON. This is based on the cookie definition used in CEE/project lumberjack and is meant to aid against an erroneous detection of a message as being CEE where it is not.

This also means that mmjsonparse currently is NOT a generic JSON parser that picks up JSON from whereever it may occur in the message. This is intentional, but future versions may support config parameters to relax the format requirements.

Action Parameters

- **cookie** [string] defaults to "@cee:"

  Permits to set the cookie that must be present in front of the JSON part of the message.

  Most importantly, this can be set to the empty string ("") in order to not require any cookie. In this case, leading spaces are permitted in front of the JSON. No non-whitespace characters are permitted after the JSON. If such is required, mmnormalize must be used.

Check parsing result

You can check whether rsyslogd was able to successfully parse the message by reading the $parsesuccess variable :

```
action(type="mmjsonparse")
if $parsesuccess == "OK" then {
   action(type="omfile" File="/tmp/output")
}
else if $parsesuccess == "FAIL" then {
   action(type="omfile" File="/tmp/parsing_failure")
}
```

Example

This activates the module and applies normalization to all messages:

```
module(load="mmjsonparse")
action(type="mmjsonparse")
```

To permit parsing messages without cookie, use this action statement:

```
action(type="mmjsonparse" cookie="")
```

The same in legacy format:

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

Log Message Normalization Module (mmnormalize)

Module Name: **mmnormalize**

Available since: 6.1.2+

Author: Rainer Gerhards <rgerhards@adiscon.com>

Description:

This module provides the capability to normalize log messages via liblognorm. Thanks to liblognorm, unstructured text, like usually found in log messages, can very quickly be parsed and put into a normal form. This is done so quickly, that it should be possible to normalize events in realtime.

This module is implemented via the output module interface. This means that mmnormalize should be called just like an action. After it has been called, the normalized message properties are available and can be accessed. These properties are called the "CEE/lumberjack" properties, because liblognorm creates a format that is inspired by the CEE/lumberjack approach.

Please note: CEE/lumberjack properties are different from regular properties. They have always "$!" prepended to the property name given in the rulebase. Such a property needs to be called with **%$!propertyname%**.

Note that mmnormalize should only be called once on each message. Behaviour is undefined if multiple calls to mmnormalize happen for the same message.

Module Parameters

allow_regex <boolean>
Default: off

Specifies if regex field-type should be allowed. Regex field-type has significantly higher computational overhead compared to other fields, so it should be avoided when another field-type can achieve the desired effect. Needs to be "on" for regex field-type to work.

Action Parameters

ruleBase <word>
Specifies which rulebase file is to use. If there are multiple mmnormalize instances, each one can use a different file. However, a single instance can use only a single file. This parameter or **rule** MUST be given, because normalization can only happen based on a rulebase. It is recommended that an absolute path name is given. Information on how to create the rulebase can be found in the liblognorm manual.

rule <array>
(Available since: 8.26.0)

Contains an array of strings which will be put together as the rulebase. This parameter or **rulebase** MUST be given, because normalization can only happen based on a rulebase.

useRawMsg <boolean>
Default: off

Specifies if the raw message should be used for normalization (on) or just the MSG part of the message (off).

path <word>
Default: $!

Specifies the JSON path under which parsed elements should be placed. By default, all parsed properties are merged into root of message properties. You can place them under a subtree, instead. You can place them in local variables, also, by setting path="$.".

variable <word>
(Available since: 8.5.1)

Specifies if a variable insteed of property 'msg' should be used for normalization. A varible can be property, local variable, json-path etc. Please note that **useRawMsg** overrides this parameter, so if **useRawMsg** is set, **variable** will be ignored and raw message will be used.

Legacy Configuration Directives

- $mmnormalizeRuleBase <rulebase-file> - equivalent to the "ruleBase" parameter.
- $mmnormalizeUseRawMsg <on/off> - equivalent to the "useRawMsg" parameter.

See Also

- First steps for mmnormalize
- Log normalization and special characters
- Log normalization and the leading space
- Using mmnormalize effectively with Adiscon LogAnalyzer

Caveats/Known Bugs

None known at this time.

Example

Sample 1:

In this sample messages are received via imtcp. Then they are normalized with the given rulebase. After that they are written in a file.

```
module(load="mmnormalize")
module(load="imtcp")

input(type="imtcp" port="10514" ruleset="outp")
```

```
ruleset(name="outp") {
      action(type="mmnormalize" rulebase="/tmp/rules.rulebase")
      action(type="omfile" File="/tmp/output")
}
```

Sample 2:

In this sample messages are received via imtcp. Then they are normalized based on the given rules. The strings from **rule** are put together and are equal to a rulebase with the same content.

```
module(load="mmnormalize")
module(load="imtcp")

input(type="imtcp" port="10514" ruleset="outp")

ruleset(name="outp") {
      action(type="mmnormalize" rule=["rule=:%host:word% %tag:char-to:\\x3a%: no␣
↪longer listening on %ip:ipv4%#%port:number%", "rule=:%host:word% %ip:ipv4% user was␣
↪logged out"])
      action(type="omfile" File="/tmp/output")
}
```

Sample 3:

This activates the module and applies normalization to all messages:

```
module(load="mmnormalize")
action(type="mmnormalize" ruleBase="/path/to/rulebase.rb")
```

The same in legacy format:

```
$ModLoad mmnormalize
$mmnormalizeRuleBase /path/to/rulebase.rb
*.* :mmnormalize:
```

RFC5424 structured data parsing module (mmpstrucdata)

Module Name: mmpstrucdata

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available since: 7.5.4

Description:

The mmpstrucdata parses the structured data of RFC5424 into the message json variable tree. The data parsed, if available, is stored under "jsonRoot!rfc5424-sd!...". Please note that only RFC5424 messages will be processed.

The difference of RFC5424 is in the message layout: the SYSLOG-MSG part only contains the structured-data part instead of the normal message part. Further down you can find a example of a structured-data part.

Module Configuration Parameters:

Currently none.

Action Confguration Parameters:

- **jsonRoot - default "!"** Specifies into which json container the data shall be parsed to.

See Also

- Howto anonymize messages that go to specific files

Caveats/Known Bugs:

- this module is currently experimental; feedback is appreciated
- property names are treated case-insensitive in rsyslog. As such, RFC5424 names are treated case-insensitive as well. If such names only differ in case (what is not recommended anyways), problems will occur.
- structured data with duplicate SD-IDs and SD-PARAMS is not properly processed

Samples:

Below you can find a structured data part of a random message which has three parameters.

```
[exampleSDID@32473 iut="3" eventSource="Application"eventID="1011"]
```

In this snippet, we parse the message and emit all json variable to a file with the message anonymized. Note that once mmpstrucdata has run, access to the original message is no longer possible (execept if stored in user variables before anonymization).

```
module(load="mmpstrucdata") action(type="mmpstrucdata")
template(name="jsondump" type="string" string="%msg%: %$!%\\n")
action(type="omfile" file="/path/to/log" template="jsondump")
```

A more practical one:

Take this example message (inspired by RFC5424 sample;)):

```
<34>1 2003-10-11T22:14:15.003Z mymachine.example.com su - ID47
[exampleSDID@32473 iut="3" eventSource="Application" eventID="1011"][id@2
test="tast"] BOM'su root' failed for lonvick on /dev/pts/8
```

We apply this configuration:

```
module(load="mmpstrucdata") action(type="mmpstrucdata")
template(name="sample2" type="string" string="ALL: %$!%\\nSD:
%$!RFC5424-SD%\\nIUT:%$!rfc5424-sd!exampleSDID@32473!iut%\\nRAWMSG:
%rawmsg%\\n\\n") action(type="omfile" file="/path/to/log"
template="sample2")
```

This will output:

```
ALL: { "rfc5424-sd": { "examplesdid@32473": { "iut": "3", "eventsource":
"Application", "eventid": "1011" }, "id@2": { "test": "tast" } } } SD:
{ "examplesdid@32473": { "iut": "3", "eventsource": "Application",
"eventid": "1011" }, "id@2": { "test": "tast" } } IUT:3 RAWMSG: <34>1
2003-10-11T22:14:15.003Z mymachine.example.com su - ID47 [exampleSDID@32473
iut="3" eventSource="Application" eventID="1011"][id@2 test="tast"] BOM'su
root' failed for lonvick on /dev/pts/8
```

As you can seem, you can address each of the individual items. Note that the case of the RFC5424 parameter names has been converted to lower case.

mmrfc5424addhmac

Module Name: mmrfc5424addhmac

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available since: 7.5.6

Description:

This module adds a hmac to RFC5424 structured data if not already present. This is a custom module and uses openssl as requested by the sponsor. This works exclusively for RFC5424 formatted messages; all others are ignored.

If both mmpstrucdata and mmrfc5424addhmac are to be used, the recommended calling sequence is

1. mmrfc5424addhmac
2. mmpstrucdata

with that sequence, the generated hash will become available for mmpstrucdata.

Module Configuration Parameters:

Currently none.

Action Confguration Parameters:

- **key** The "key" (string) to be used to generate the hmac.
- **hashfunction** An openssl hash function name for the function to be used. This is passed on to openssl, so see the openssl list of supported function names.
- **sd_id** The RFC5424 structured data ID to be used by this module. This is the SD-ID that will be added. Note that nothing is added if this SD-ID is already present.

Verification method

rsyslog does not contain any tools to verify a log file (this was not part of the custom project). So you need to write your own verifier.

When writing the verifier, keep in mind that the log file contains messages with the hash SD-ID included. For obvious reasons, this SD-ID was not present when the hash was created. So before the actual verification is done, this SD-ID must be removed, and the remaining (original) message be verified. Also, it is important to note that the output template must write the exact same message format that was received. Otherwise, a verification failure will obviously occur - and must so, because the message content actually was altered.

So in a more formal description, verification of a message m can be done as follows:

1. let m' be m with the configured SD-ID removed (everything between []). Otherwise, m' must be an exact duplicate of m.
2. call openssl's HMAC function as follows: `HMAC(hashfunction, key, len(key), m', len(m'), hash, &hashlen);` Where hashfunction and key are the configured values and hash is an output buffer for the hash.
3. let h be the extracted hash value obtained from m within the relevant SD-ID. Be sure to convert the hex string back to the actual byte values.
4. now compare hash and h under consideration of the sizes. If these values match the verification succeeds, otherwise the message was modified.

If you neeed help implementing a verifier function or want to sponsor development of a verification tool, please simply email sales@adiscon.com for a quote.

See Also

- How to add a HMAC to RFC5424 messages

Caveats/Known Bugs:

- none

mmrm1stspace: First Space Modification Module

Author: Pascal Withopf <pascalwithopf1@gmail.com>

In rfc3164 the msg begins at the first letter after the tag. It is often the case that this is a unnecessary space. This module removes this first character if it is a space.

Configuration Parameters

Currently none.

Examples

This example receives messages over imtcp and modifies them, before sending them to a file.

:: module(load="imtcp") module(load="mmrm1stspace") input(type="imtcp" port="13514") action(type="mmrm1stspace") action(type="omfile" file="output.log")

Number generator and counter module (mmsequence)

Module Name: mmsequence

Author: Pavel Levshin <pavel@levshin.spb.ru>

Status: Non project-supported module - contact author or rsyslog mailing list for questions

This module is deprecated in v8 and solely provided for backward compatibility reasons. It was written as a workaround for missing global variable support in v7. Global variables are available in v8, and at some point in time this module will entirely be removed.

Do not use this module for newly crafted config files. Use global variables instead.

Available since: 7.5.6

Description:

This module generates numeric sequences of different kinds. It can be used to count messages up to a limit and to number them. It can generate random numbers in a given range.

This module is implemented via the output module interface, so it is called just as an action. The number generated is stored in a variable.

Action Parameters:

- **mode** "random" or "instance" or "key"

  Specifies mode of the action. In "random" mode, the module generates uniformly distributed integer numbers in a range defined by "from" and "to".

In "instance" mode, which is default, the action produces a counter in range [from, to). This counter is specific to this action instance.

In "key" mode, the counter can be shared between multiple instances. This counter is identified by a name, which is defined with "key" parameter.

- **from** [non-negative integer], default "0"

  Starting value for counters and lower margin for random generator.

- **to** [positive integer], default "INT_MAX"

  Upper margin for all sequences. Note that this margin is not inclusive. When next value for a counter is equal or greater than this parameter, the counter resets to the starting value.

- **step** [non-negative integer], default "1"

  Increment for counters. If step is "0", it can be used to fetch current value without modification. The latter not applies to "random" mode. This is useful in "key" mode or to get constant values in "instance" mode.

- **key** [word], default ""

  Name of the global counter which is used in this action.

- **var** [word], default "$!mmsequence"

  Name of the variable where the number will be stored. Should start with "$".

Sample:

```
# load balance
Ruleset(
    name="logd"
    queue.workerthreads="5"
    ){

    Action(
        type="mmsequence"
        mode="instance"
        from="0"
        to="2"
        var="$.seq"
    )

    if $.seq == "0" then {
        Action(
            type="mmnormalize"
            userawmsg="on"
            rulebase="/etc/rsyslog.d/rules.rb"
        )
    } else {
        Action(
            type="mmnormalize"
            userawmsg="on"
            rulebase="/etc/rsyslog.d/rules.rb"
        )
    }

    # output logic here
}
    # generate random numbers
    action(
        type="mmsequence"
```

```
        mode="random"
        to="100"
        var="$!rndz"
    )
    # count from 0 to 99
    action(
        type="mmsequence"
        mode="instance"
        to="100"
        var="$!cnt1"
    )
    # the same as before but the counter is global
    action(
        type="mmsequence"
        mode="key"
        key="key1"
        to="100"
        var="$!cnt2"
    )
    # count specific messages but place the counter in every message
    if $msg contains "txt" then
        action(
            type="mmsequence"
            mode="key"
            to="100"
            var="$!cnt3"
        )
    else
        action(
            type="mmsequence"
            mode="key"
            to="100"
            step="0"
            var="$!cnt3"
            key=""
        )
```

Legacy Configuration Directives:

Not supported.

mmsnmptrapd message modification module

Module Name: mmsnmptrapd

Author: Rainer Gerhards <rgerhards@adiscon.com> (custom-created)

Multi-Ruleset Support: since 5.8.1

Description:

This module uses a specific configuration of snmptrapd's tag values to obtain information of the original source system and the severity present inside the original SNMP trap. It then replaces these fields inside the syslog message.

Let's look at an example. Essentially, SNMPTT will invoke something like this:

```
logger -t snmptrapd/warning/realhost Host 003c.abcd.ffff in vlan 17 is flapping␣
↪between port Gi4/1 and port Gi3/2
```

This message modification module will change the tag (removing the additional information), hostname and severity (not shown in example), so the log entry will look as follows:

```
2011-04-21T16:43:09.101633+02:00 realhost snmptrapd: Host 003c.abcd.ffff in vlan 122␣
↪is flapping between port Gi4/1 and port Gi3/2
```

The following logic is applied to all message being processed:

1. The module checks incoming syslog entries. If their TAG field starts with "snmptrapd/" (configurable), they are modified, otherwise not. If the are modified, this happens as follows:
2. It will derive the hostname from the tag field which has format snmptrapd/severity/hostname
3. It should derive the severity from the tag field which has format snmptrapd/severity/hostname. A configurable mapping table will be used to drive a new severity value from that severity string. If no mapping has been defined, the original severity is not changed.
4. It replaces the "FromHost" value with the derived value from step 2
5. It replaces the "Severity" value with the derived value from step 3

Note that the placement of this module inside the configuration is important. All actions before this modules is called will work on the unmodified message. All messages after it's call will work on the modified message. Please also note that there is some extra power in case it is required: as this module is implemented via the output module interface, a filter can be used (actually must be used) in order to tell when it is called. Usually, the catch-all filter (*.*) is used, but more specific filters are fully supported. So it is possible to define different parameters for this module depending on different filters. It is also possible to just run messages from one remote system through this module, with the help of filters or multiple rulesets and ruleset bindings. In short words, all capabilities rsyslog offers to control output modules are also available to mmsnmptrapd.

Configuration Directives:

- **$mmsnmptrapdTag** [tagname]

  tells the module which start string inside the tag to look for. The default is "snmptrapd". Note that a slash is automatically added to this tag when it comes to matching incoming messages. It MUST not be given, except if two slashes are required for whatever reasons (so "tag/" results in a check for "tag//" at the start of the tag field).

- **$mmsnmptrapdSeverityMapping** [severitymap] This specifies the severity mapping table. It needs to be specified as a list. Note that due to the current config system **no whitespace** is supported inside the list, so be sure not to use any whitespace inside it. The list is constructed of Severtiy-Name/Severity-Value pairs, delimited by comma. Severity-Name is a case-sensitive string, e.g. "warning" and an associated numerical value (e.g. 4). Possible values are in the rage 0..7 and are defined in RFC5424, table 2. The given sample would be specified as "warning/4". If multiple instances of mmsnmptrapd are used, each instance uses the most recently defined $mmsnmptrapdSeverityMapping before itself.

Caveats/Known Bugs:

- currently none known

Example:

This enables to rewrite messages from snmptrapd and configures error and warning severities. The default tag is used.

```
$ModLoad mmsnmptrapd # needs to be done just once
# ... other module loads and listener setup ...
*.* /path/to/file/with/orignalMessage # this file receives unmodified messages
$mmsnmptrapdSeverityMapping warning/4,error/3
```

```
*.* :mmsnmptrapd: # now message is modified
*.* /path/to/file/with/modifiedMessage # this file receives modified messages
# ... rest of config ...
```

This documentation is part of the rsyslog project.

Fix invalid UTF-8 Sequences (mmutf8fix)

Module Name: mmutf8fix

Author: Rainer Gerhards <rgerhards@adiscon.com>

Available since: 7.5.4

Description:

The mmutf8fix module permits to fix invalid UTF-8 sequences. Most often, such invalid sequences result from syslog sources sending in non-UTF character sets, e.g. ISO 8859. As syslog does not have a way to convey the character set information, these sequences are not properly handled. While they are typically uncritical with plain text files, they can cause big headache with database sources as well as systems like ElasticSearch.

The module supports different "fixing" modes and fixes. The current implementation will always replace invalid bytes with a single US ASCII character. Additional replacement modes will probably be added in the future, depending on user demand. In the longer term it could also be evolved into an any-charset-to-UTF8 converter. But first let's see if it really gets into widespread enough use.

Proper Usage:

Some notes are due for proper use of this module. This is a message modification module utilizing the action interface, which means you call it like an action. This gives great flexibility on the question on when and how to call this module. Note that once it has been called, it actually modifies the message. The original messsage is then no longer available. However, this does **not** change any properties set, used or extracted before the modification is done.

One potential use case is to normalize all messages. This is done by simply calling mmutf8fix right in front of all other actions.

If only a specific source (or set of sources) is known to cause problems, mmutf8fix can be conditionally called only on messages from them. This also offers performance benefits. If such multiple sources exists, it probably is a good idea to define different listeners for their incoming traffic, bind them to specific ruleset and call mmutf8fix as first action in this ruleset.

Module Configuration Parameters:

Currently none.

Action Confguration Parameters:

- **mode** - **utf-8**/controlcharacters

  This sets the basic detection mode. In **utf-8** mode (the default), proper UTF-8 encoding is checked and bytes which are not proper UTF-8 sequences are acted on. If a proper multi-byte start sequence byte is detected but any of the following bytes is invalid, the whole sequence is replaced by the replacement method. This mode is most useful with non-US-ASCII character sets, which validly includes multibyte sequences. Note that in this mode control characters are NOT being replaced, because they are valid UTF-8. In **controlcharacters** mode, all bytes which do not represent a printable US-ASCII character (codes 32 to 126) are replaced. Note that this also mangles valid UTF-8 multi-byte sequences, as these are (deliberately) outside of that character range. This mode is most useful if it is known that no characters outside of the US-ASCII alphabet need to be processed.

- **replacementChar** - default " " (space), a single character

  This is the character that invalid sequences are replaced by. Currently, it MUST be a **printable** US-ASCII character.

Caveats/Known Bugs:

- overlong UTF-8 encodings are currently not detected in utf-8 mode.

Samples:

In this snippet, we write one file without fixing UTF-8 and another one with the message fixed. Note that once mmutf8fix has run, access to the original message is no longer possible.

```
module(load="mmutf8fix") action(type="omfile"
file="/path/to/non-fixed.log") action(type="mmutf8fix")
action(type="omfile" file="/path/to/fixed.log")
```

In this sample, we fix only message originating from host 10.0.0.1.

```
module(load="mmutf8fix") if $fromhost-ip == "10.0.0.1" then
action(type="mmutf8fix") # all other actions here...
```

This is mostly the same as the previous sample, but uses "controlcharacters" processing mode.

```
module(load="mmutf8fix") if $fromhost-ip == "10.0.0.1" then
action(type="mmutf8fix" mode="controlcharacters") # all other actions here...
```


String Generator Modules

String generator modules are used, as the name implies, to generate strings based on the message content. They are currently tightly coupled with the template system. Their primary use is to speed up template processing by providing a native C interface to template generation. These modules exist since 5.5.6. To get an idea of the potential speedup, the default file format, when generated by a string generator, provides a roughly 5% speedup. For more complex strings, especially those that include multiple regular expressions, the speedup may be considerably higher.

String generator modules are written to a quite simple interface. However, a word of caution is due: they access the rsyslog message object via a low-level interface. That interface is not guaranteed yet to stay stable. So it may be necessary to modify string generator modules if the interface changes. Obviously, we will not do that without good reason, but it may happen.

Rsyslog comes with a set of core, build-in string generators, which are used to provide those default templates that we consider to be time-critical:

- smfile - the default rsyslog file format
- smfwd - the default rsyslog (network) forwarding format
- smtradfile - the traditional syslog file format
- smfwd - the traditional syslog (network) forwarding format

Note that when you replace these defaults be some custom strings, you will loose some performance (around 5%). For typical systems, this is not really relevant. But for a high-performance systems, it may be very relevant. To solve that issue, create a new string generator module for your custom format, starting out from one of the default generators provided. If you can not do this yourself, you may want to contact Adiscon as we offer custom development of string generators at a very low price.

Note that string generator modules can be dynamically loaded. However, the default ones provided are so important that they are build right into the executable. But this does not need to be done that way (and it is straightforward to do it dynamic).

Library Modules

Library modules provide dynamically loadable functionality for parts of rsyslog, most often for other loadable modules. They can not be user-configured and are loaded automatically by some components. They are just mentioned so that error messages that point to library moduls can be understood. No module list is provided.

Where are the modules integrated into the Message Flow?

Depending on their module type, modules may access and/or modify messages at various stages during rsyslog's processing. Note that only the "core type" (e.g. input, output) but not any type derived from it (message modification module) specifies when a module is called.

The simplified workflow is as follows:

As can be seen, messages are received by input modules, then passed to one or many parser modules, which generate the in-memory representation of the message and may also modify the message itself. The internal representation is passed to output modules, which may output a message and (with the interfaces introduced in v5) may also modify message object content.

String generator modules are not included inside this picture, because they are not a required part of the workflow. If used, they operate "in front of" the output modules, because they are called during template generation.

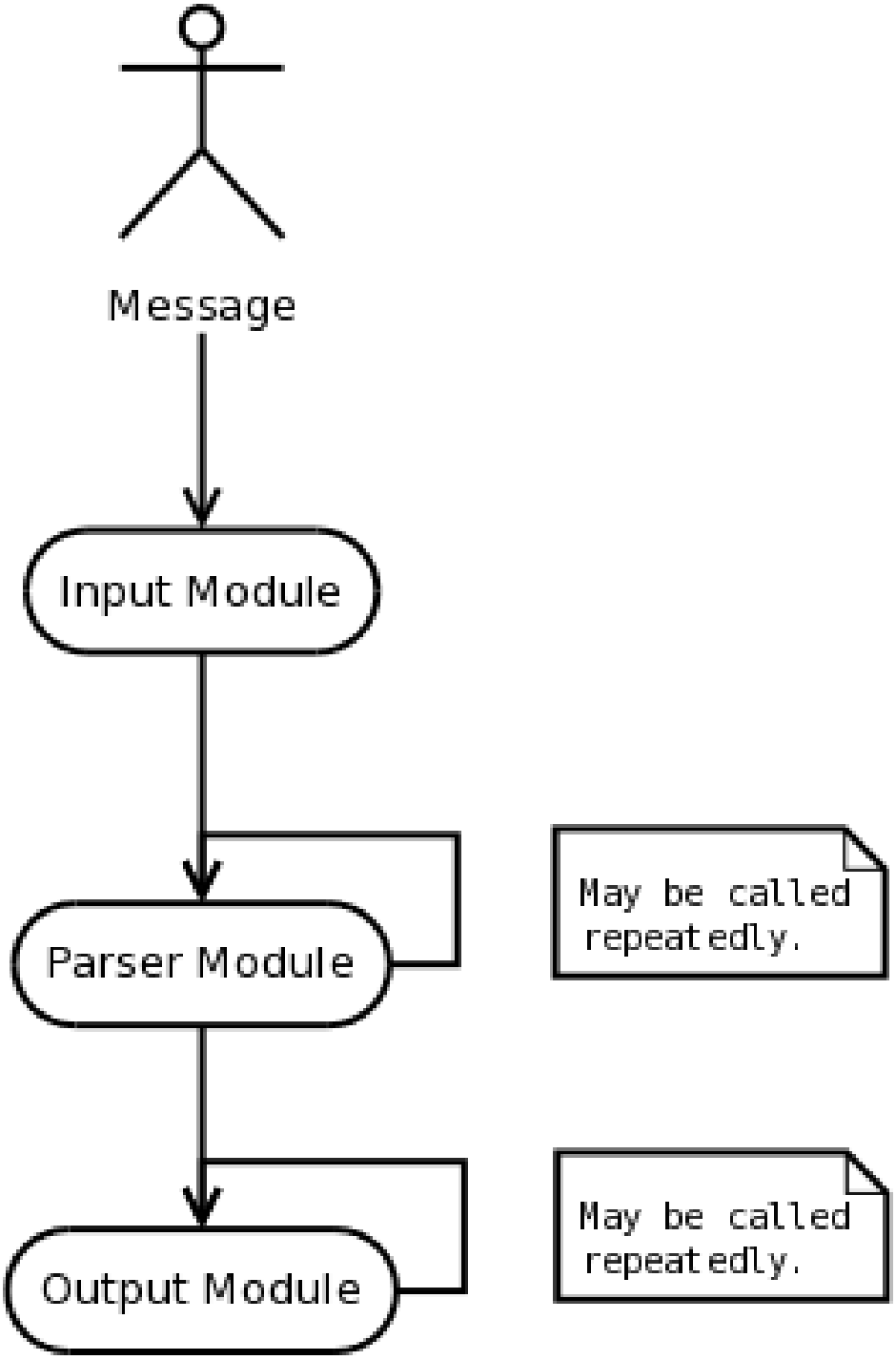
Note that the actual flow is much more complex and depends a lot on queue and filter settings. This graphic above is a high-level message flow diagram.

Output Channels

Output Channels are a new concept first introduced in rsyslog 0.9.0. **As of this writing, it is most likely that they will be replaced by something different in the future.** So if you use them, be prepared to change you configuration file syntax when you upgrade to a later release. The idea behind output channel definitions is that it shall provide an umbrella for any type of output that the user might want. In essence, this is the "file" part of selector lines (and this is why we are not sure output channel syntax will stay after the next review). There is a difference, though: selector channels both have filter conditions (currently facility and severity) as well as the output destination. they can only be used to write to files - not pipes, ttys or whatever Output channels define the output definition, only. As of this build, else. If we stick with output channels, this will change over time.

In concept, an output channel includes everything needed to know about an output actions. In practice, the current implementation only carries a filename, a maximum file size and a command to be issued when this file size is reached. More things might be present in future version, which might also change the syntax of the directive.

Output channels are defined via an $outchannel directive. It's syntax is as follows: $outchannel name,file-name,max-size,action-on-max-size name is the name of the output channel (not the file), file-name is the file name to be written to, max-size the maximum allowed size and action-on-max-size a command to be issued when the max size is reached. This command always has exactly one parameter. The binary is that part of action-on-max-size before the first space, its parameter is everything behind that space. Please note that max-size is queried BEFORE writing the log message to the file. So be sure to set this limit reasonably low so that any message might fit. For the current release, setting it 1k lower than you expected is helpful. The max-size must always be specified in bytes - there are no special symbols (like 1k, 1m,...) at this point of development. Keep in mind that $outchannel just defines a channel with "name". It does not activate it. To do so, you must use a selector line (see below). That selector line includes the channel name plus an $ sign in front of it. A sample might be: *.* :omfile:$mychannel In its current form, output channels primarily provide the ability to size-limit an output file. To do so, specify a maximum size. When this size is reached, rsyslogd


Message
Input Module
Parser Module
May be called repeatedly.
Output Module
May be called repeatedly.

will execute the action-on-max-size command and then reopen the file and retry. The command should be something like a log rotation script or a similar thing.

If there is no action-on-max-size command or the command did not resolve the situation, the file is closed and never reopened by rsyslogd (except, of course, by huping it). This logic was integrated when we first experienced severe issues with files larger 2gb, which could lead to rsyslogd dumping core. In such cases, it is more appropriate to stop writing to a single file. Meanwhile, rsyslogd has been fixed to support files larger 2gb, but obviously only on file systems and operating system versions that do so. So it can still make sense to enforce a 2gb file size limit.

This documentation is part of the rsyslog project.

Dropping privileges in rsyslog

Available since: 4.1.1

Description:

Rsyslogd provides the ability to drop privileges by impersonating as another user and/or group after startup.

Please note that due to POSIX standards, rsyslogd always needs to start up as root if there is a listener who must bind to a network port below 1024. For example, the UDP listener usually needs to listen to 514 and therefore rsyslogd needs to start up as root.

If you do not need this functionality, you can start rsyslog directly as an ordinary user. That is probably the safest way of operations. However, if a startup as root is required, you can use the $PrivDropToGroup and $PrivDropToUser config directives to specify a group and/or user that rsyslogd should drop to after initialization. Once this happens, the daemon runs without high privileges (depending, of course, on the permissions of the user account you specified).

There is some additional information available in the rsyslog wiki.

Configuration Directives:

- **$PrivDropToUser** Name of the user rsyslog should run under after startup. Please note that this user is looked up in the system tables. If the lookup fails, privileges are NOT dropped. Thus it is advisable to use the less convenient $PrivDropToUserID directive. If the user id can be looked up, but can not be set, rsyslog aborts.
- **$PrivDropToUserID** Much the same as $PrivDropToUser, except that a numerical user id instead of a name is specified.Thus, privilege drop will always happen. rsyslogd aborts.
- **$PrivDropToGroup** Name of the group rsyslog should run under after startup. Please note that this user is looked up in the system tables. If the lookup fails, privileges are NOT dropped. Thus it is advisable to use the less convenient $PrivDropToGroupID directive. Note that all supplementary groups are removed from the process if $PrivDropToGroup is specified. If the group id can be looked up, but can not be set, rsyslog aborts.
- **$PrivDropToGroupID** Much the same as $PrivDropToGroup, except that a numerical group id instead of a name is specified. Thus, privilege drop will always happen.

Notes on IPv6 Handling in Rsyslog

Rsyslog fully* supports sending and receiving syslog messages via both IPv4 and IPv6. IPv6 is natively supported for both UDP and TCP. However, there are some options that control handling of IPv6 operations. I thought it is is a good idea to elaborate a little about them, so that you can probably find your way somewhat easier.

First of all, you can restrict rsyslog to using IPv4 or IPv6 addresses only by specifying the -4 or -6 command line option (now guess which one does what...). If you do not provide any command line option, rsyslog uses IPv4 and

IPv6 addresses concurrently. In practice, that means the listener binds to both addresses (provided they are configured). When sending syslog messages, rsyslog uses IPv4 addresses when the receiver can be reached via IPv4 and IPv6 addresses if it can be reached via IPv6. If it can be reached on either IPv4 and v6, rsyslog leaves the choice to the socket layer. The important point to know is that it uses whatever connectivity is available to reach the destination.

There is one subtle difference between UDP and TCP. With the new IPv4/v6 ignorant code, rsyslog has potentially different ways to reach destinations. The socket layer returns all of these paths in a sorted array. For TCP, rsyslog loops through this array until a successful TCP connect can be made. If that happens, the other addresses are ignored and messages are sent via the successfully-connected socket.

For UDP, there is no such definite success indicator. Sure, the socket layer may detect some errors, but it may not notice other errors (due to the unreliable nature of UDP). By default, the UDP sender also tries one entry after the other in the sorted array of destination addresses. When a send fails, the next address is tried. When the send function finally succeeds, rsyslogd assumes the UDP packet has reached its final destination. However, if rsyslogd is started with the "-A" (capital A!) was given on the command line, rsyslogd will continue to send messages until the end of the destination address array is reached. This may result in duplicate messages, but it also provides some additional reliability in case a message could not be received. You need to be sure about the implications before applying this option. In general, it is NOT recommended to use the -A option.

*rsyslog does not support RFC 3195 over IPv6. The reason is that the RFC 3195 library, liblogging, supports IPv4, only. Currently, there are no plans to update either rsyslog to another RFC 3195 stack or update liblogging. There is simply no demand for 3195 solutions.

libgcrypt Log Crypto Provider (gcry)

Crypto Provider Name: gcry

Author: Rainer Gerhards <rgerhards@adiscon.com>

Supported Since: since 7.3.10

Description:

Provides encryption support to rsyslog.

Configuration Parameters:

Crypto providers are loaded by omfile, when the provider is selected in its "cry.providerName" parameter. Parameters for the provider are given in the omfile action instance line.

This provider creates an encryption information file with the same base name but the extension ".encinfo" for each log file (both for fixed-name files as well as dynafiles). Both files together form a set. So you need to archive both in order to prove integrity.

- **cry.algo** <Encryption Algorithm> The algorithm (cipher) to be used for encryption. The default algorithm is "AES128". Currently, the following Algorithms are supported:
  - 3DES
  - CAST5
  - BLOWFISH
  - AES128
  - AES192
  - AES256
  - TWOFISH

  - TWOFISH128
  - ARCFOUR
  - DES
  - SERPENT128
  - SERPENT192
  - SERPENT256
  - RFC2268_40
  - SEED
  - CAMELLIA128
  - CAMELLIA192
  - CAMELLIA256

  The actual availability of an algorithms depends on which ones are compiled into libgcrypt. Note that some versions of libgcrypt simply abort the process (rsyslogd in this case!) if a supported algorithm is select but not available due to libgcrypt build settings. There is nothing rsyslog can do against this. So in order to avoid production downtime, always check carefully when you change the algorithm.

- **cry.mode** <Algorithm Mode> The encryption mode to be used. Default ist Cipher Block Chaining (CBC). Note that not all encryption modes can be used together with all algorithms. Currently, the following modes are supported:
  - ECB
  - CFB
  - CBC
  - STREAM
  - OFB
  - CTR
  - AESWRAP
- **cry.key** <encryption key> TESTING AID, NOT FOR PRODUCTION USE. This uses the KEY specified inside rsyslog.conf. This is the actual key, and as such this mode is highly insecure. However, it can be useful for intial testing steps. This option may be removed in the future.
- **cry.keyfile** <filename> Reads the key from the specified file. The file must contain the key, only, no headers or other meta information. Keyfiles can be generated via the rscrytool utility.
- **cry.keyprogram** <path to program> If given, the key is provided by a so-called "key program". This program is executed and must return the key (as well as some meta information) via stdout. The core idea of key programs is that using this interface the user can implement as complex (and secure) method to obtain keys as desired, all without the need to make modifications to rsyslog.

Caveats/Known Bugs:

- currently none known

Samples:

This encrypts a log file. Default parameters are used, they key is provided via a keyfile.

```
action(type="omfile" file="/var/log/somelog" cry.provider="gcry"
       cry.keyfile="/secured/path/to/keyfile")
```

Note that the keyfile can be generated via the rscrytool utility (see its documentation for how to actually do that).

Dynamic Stats

Rsyslog produces runtime-stats to allow user to study service health, performance, bottlenecks etc. Runtime-stats counters that Rsyslog components publish are statically defined.

Dynamic Stats (called dyn-stats henceforth) component allows user to configure stats-namespaces (called stats-buckets) and increment counters within these buckets using Rainerscript function call.

The metric-name in this case can be a message-property or a sub-string extracted from message etc.

Dyn-stats configuration

Dyn-stats configuration involves a **two part setup**.

dyn_stats(name="<bucket>"...) (object)

Defines the bucket(identified by the bucket-name) and allows user to set some properties that control behavior of the bucket.

```
dyn_stats(name="msg_per_host")
```

Parameters: **name** <string litteral, mandatory> : Name of the bucket.

resettable <onloff, default: on> : Whether or not counters should be reset every time they are reported. This works independent of `resetCounters` config parameter in *impstats: Generate Periodic Statistics of Internal Counters*.

maxCardinality <number, default: 2000> : Maximum number of unique counter-names to track.

unusedMetricLife <number, default: 3600> : Interval between full purges (in seconds). This prevents unused counters from occupying resources forever.

A definition setting all the parameters looks like:

```
dyn_stats(name="msg_per_host" resettable="on" maxCardinality="3000" unusedMetricLife=
↪"600")
```

dyn_inc("<bucket>", <expr>) (function)

Increments counter identified by value of variable in bucket identified by name.

Parameters: **name** <string litteral, mandatory> : Name of the bucket

expr <expression resulting in a string> : Name of counter (this name will be reported by impstats to identify the counter)

A `dyn_inc` call looks like:

```
set $.inc = dyn_inc("msg_per_host", $hostname);

if ($.inc != 0) then {
    ....
}
```

`$.inc` captures the error-code. It has value `0` when increment operation is successful and non-zero when it fails. It uses Rsyslog error-codes.

Reporting

Legacy format:

```
...
global: origin=dynstats msg_per_host.ops_overflow=1 msg_per_host.new_metric_add=3 msg_
↪per_host.no_metric=0 msg_per_host.metrics_purged=0 msg_per_host.ops_ignored=0
...
msg_per_host: origin=dynstats.bucket foo=2 bar=1 baz=1
...
```

Json(variants with the same structure are used in other Json based formats such as `cee` and `json-elasticsearch`) format:

```
...
{ "name": "global", "origin": "dynstats", "values": { "msg_per_host.ops_overflow": 1,
↪"msg_per_host.new_metric_add": 3, "msg_per_host.no_metric": 0, "msg_per_host.
↪metrics_purged": 0, "msg_per_host.ops_ignored": 0 } }
...
{ "name": "msg_per_host", "origin": "dynstats.bucket", "values": { "foo": 2, "bar": 1,
↪ "baz": 1 } }
...
```

In this case counters are encapsulated inside an object hanging off top-level-key `values`.

Fields

global: origin=dynstats: ops_overflow: Number of operations ignored because number-of-counters-tracked has hit configured max-cardinality.

new_metric_add: Number of "new" metrics added (new counters created).

no_metric: Counter-name given was invalid (length = 0).

metrics_purged: Number of counters discarded at discard-cycle (controlled by **unusedMetricLife**).

ops_ignored: Number of operations ignored due to potential performance overhead. Dyn-stats subsystem ignores operations to avoid performance-penalty if it can't get access to counter without delay(lock acquiring latency).

purge_triggered: Indicates that a discard was performed (1 implies a discard-cycle run).

msg_per_host: origin=dynstats.bucket: <metric_name>: Value of counter identified by <metric-name>.

Lookup Tables

Lookup tables are a powerful construct to obtain *class* information based on message content. It works on top of a data-file which maps key (to be looked up) to value (the result of lookup).

The idea is to use a message properties (or derivatives of it) as an index into a table which then returns another value. For example, $fromhost-ip could be used as an index, with the table value representing the type of server or the department or remote office it is located in.

This can be emulated using if and else-if stack, but implementing it as a dedicated component allows `lookup` to be made fast.

The lookup tables itself exists in a separate data file (one per table). This file is loaded on Rsyslog startup and when a reload is requested.

There are different types of lookup tables (identified by "type" field in json data-file).

Types

string

The key to be looked up is an arbitrary string.

Match criterion: The key must be exactly equal to index from one of the entries.

array

The value to be looked up is an integer number from a consequtive set. The set does not need to start at zero or one, but there must be no number missing. So, for example `5,6,7,8,9` would be a valid set of index values, while `1,2,4,5` would not be (due to missing `3`).

Match criterion: Looked-up number(key) must be exactly equal to index from one of the entries.

sparseArray

The value to be looked up is an integer value, but there may be gaps inside the set of values (usually there are large gaps). A typical use case would be the matching of IPv4 address information.

Match criterion: A match happens on the first index that is less than or equal to the looked-up key.

Note that index integer numbers are represented by unsigned 32 bits.

Lookup Table File Format

Lookup table files contain a single JSON object. This object contains of a header and a table part.

Header

The header is the top-level json. It has paramters "version", "nomatch", and "type".

Parameters: **version** <number, default: 1> : Version of table-definition format (so improvements in the future can be managed in a backward compatible way).

nomatch <string litteral, default: ""> : Value to be returned for a lookup when match fails.

type <*string*, *array* or *sparseArray*, default: *string*> : Type of lookup-table (controls how matches are performed).

Table

This must be an array of elements, even if only a single value exists (for obvious reasons, we do not expect this to occur often). Each array element must contain two fields "index" and "value".

This is a sample of how an ip-to-office mapping may look like:

```
{ "version" : 1,
  "nomatch" : "unk",
  "type" : "string",
  "table" : [
    {"index" : "10.0.1.1", "value" : "A" },
    {"index" : "10.0.1.2", "value" : "A" },
    {"index" : "10.0.1.3", "value" : "A" },
    {"index" : "10.0.2.1", "value" : "B" },
    {"index" : "10.0.2.2", "value" : "B" },
    {"index" : "10.0.2.3", "value" : "B" }]}
```

Note: In the example above, if a different IP comes in, the value "unk" is returend thanks to the nomatch parameter in the first line.

Lookup tables can be access via the `lookup()` built-in function. Common usage pattern is to set a local variable to the lookup result and later use that variable in templates.

Lookup-table configuration

Lookup-table configuration involves a **two part setup** (definition and usage(lookup)), with an optional third part, which allows reloading table using internal trigger.

lookup_table(name="<table>" file="</path/to/file>"...) (object)

Defines the table(identified by the table-name) and allows user to set some properties that control behavior of the table.

```
dyn_stats(name="msg_per_host")
```

Parameters: **name** <string litteral, mandatory> : Name of the table.

file <string litteral, file path, mandatory> : Path to external json database file.

reloadOnHUP <on|off, default: on> : Whether or not table should be reloaded when process receives HUP signal.

A definition setting all the parameters looks like:

```
lookup_table(name="host_bu" file="/var/lib/host_billing_unit_mapping.json"␣
↪reloadOnHUP="on")
```

lookup("<table>", <expr>) (function)

Looks up and returns the value that is associated with the given key (passed as <variable>) in lookup table identified by table-name. If no match is found (according to table-type matching-criteria specified above), the "nomatch" string is returned (or an empty string if it is not defined).

Parameters: **name** <string litteral, mandatory> : Name of the table.

expr <expression resulting in string or number according to lookup-table type, mandatory> : Key to be looked up.

A `lookup` call looks like:

```
set $.bu = lookup("host_bu", $hostname);

if ($.bu == "unknown") then {
    ....
}
```

Some examples of different match/no-match senarios:

string table:

```
{ "nomatch" : "none",
  "type" : "string",
  "table":[
    {"index" : "foo", "value" : "bar" },
    {"index" : "baz", "value" : "quux" }]}
```

Match/no-Match behaviour:

key	return
foo	bar
baz	quux
corge	none

array table:

```
{ "nomatch" : "nothing",
  "type" : "array",
  "table":[
    {"index" : 9, "value" : "foo" },
    {"index" : 10, "value" : "bar" },
    {"index" : 11, "value" : "baz" }]}
```

Match/no-Match behaviour:

key	return
9	foo
11	baz
15	nothing
0	nothing

sparseArray table:

```
{ "nomatch" : "no_num",
  "type" : "sparseArray",
  "table":[
    {"index" : "9", "value" : "foo" },
    {"index" : "11", "value" : "baz" }]}
```

Match/no-Match behaviour:

key	return
8	no_num
9	foo
10	foo
11	baz
12	baz
100	baz

reload_lookup_table("<table>", "<stub value>") (statement)

Reloads lookup table identified by given table name **asynchronously** (by internal trigger, as opposed to HUP).

This statement isn't always useful. It needs to be used only when lookup-table-reload needs to be triggered in response to a message.

Messages will continue to be processed while table is asynchronously reloaded.

Note: For performance reasons, message that triggers reload should be accepted only from a trusted source.

Parameters: **name** <string litteral, mandatory> : Name of the table.

stub value <string litteral, optional> : Value to stub the table in-case reload-attempt fails.

A `reload_lookup_table` invocation looks like:

```
if ($.do_reload == "y") then {
    reload_lookup_table("host_bu", "unknown")
}
```

Implementation Details

The lookup table functionality is implemented via efficient algorithms.

The string and sparseArray lookup have O(log(n)) time complexity, while array lookup is O(1).

To preserve space and, more important, increase cache hit performance, equal data values are only stored once, no matter how often a lookup index points to them.

Configuration file examples can be found in the rsyslog wiki. Also keep the rsyslog config snippets on your mind. These are ready-to-use real building blocks for rsyslog configuration.

There is also one sample file provided together with the documentation set. If you do not like to read, be sure to have at least a quick look at `rsyslog-example.conf`.

While rsyslogd contains enhancements over standard syslogd, efforts have been made to keep the configuration file as compatible as possible. While, for obvious reasons, *enhanced features* require a different config file syntax, rsyslogd should be able to work with a standard syslog.conf file. This is especially useful while you are migrating from syslogd to rsyslogd.

Installation

Installation is usually as simple as saying

```
$ sudo yum install rsyslog
```

or

```
$ sudo apt-get install rsyslog
```

However, distributions usually provide rather old versions of rsyslog, and so there are chances you want to have something newer. In that case, you need to either use the rsyslog project's own packages, a distribution tarball or build directly from repo. All of this is detailled in this chapter.

Installing rsyslog from Package

Installing from package is usually the most convenient way to install rsyslog. Usually, the regular package manager can be used.

Package Availability

Rsyslog is included in all major distributions. So you do not necessarily need to take care of where packages can be found - they are "just there". Unfortunately, the distros provide often rather old versions. This is especially the case for so-called enterprise distributions.

As long as you do not run into trouble with one of these old versions, using the distribution-provided packages is easy and a good idea. If you need new features, better performance and sometimes even a fix for a bug that the distro did not backport, you can use alternative packages. Please also note that the project team does not support outdated versions. While we probably can help with simple config questions, for anything else we concentracte on current versions.

The rsyslog project offers current packages for a number of "big" distributions. They can be found at http://www.rsyslog.com in the download section.

Note that some distributions (like Fedora) usually keep up with development rather quickly and so we do not provide special packages for them.

If you do not find a suitable package for your distribution, there is no reason to panic. It is quite simple to *install rsyslog from the source tarball*, so you should consider that.

Package Structure

Almost all distributions package rsyslog in multiple packages. This is also the way Adiscon packages are created. The reason is that rsyslog has so many input and output plugins that enable it to connect to different systems like MySQL, HDFS, ElasticSearch and so on. If everything were provided in a single giantic package, you would need to install all of these dependencies, even though they are mostly not needed.

For that reason, rsyslog comes with multiple packages:

- *core package* (usually just called "rsyslog") - this contains core technology that is required as a base for all other packages. It also contains modules like the file writer or syslog forwareder that is extremely often used and has little dependencies.
- *feature package* (usually called "rsyslog-feature") - there are multiple of these packages. What exactly is available and how it is named depends on the distro. This unfortunately is a bit consistent. Usually, it is a good guess that the package is intuitively named, e.g. "rsyslog-mysql" for the MySQL component and "rsyslog-elasticsearch" for ElasticSearch support. If in doubt, it is suggested to use the distro's package manager and search for "rsyslog*".

Contributing

Packaging is a community effort. If you would like to see support for an additional distribution and know how to build packages, please consider contributing to the project and joining the packaging team. Also, rsyslog's presence on github also contains the sources for the currently maintained packages. They can be found at https://github.com/rsyslog.

Installing rsyslog from Source

Written by Rainer Gerhards

In this paper, I describe how to install rsyslog. It is intentionally a brief step-by-step guide, targeted to those who want to quickly get it up and running. For more elaborate information, please consult the rest of the *manual set*.

How to make your life easier...

There are *RPMs/packages for rsyslog* available. If you use them, you can spare yourself many of the steps below. This is highly recommended if there is a package for your distribution available.

Steps To Do

Step 1 - Download Software

For obvious reasons, you need to download rsyslog. Here, I assume that you use a distribution tarball. If you would like to use a version directly from the repository, see build rsyslog from repository instead.

Load the most recent build from http://www.rsyslog.com/downloads. Extract the software with "tar xzf -nameOfDownloadSet-". This will create a new subdirectory rsyslog-version in the current working directory. cd into that.

Depending on your system configuration, you also need to install some build tools, most importantly make, the gcc compiler and the MySQL development system (if you intend to use MySQL - the package is often named "mysql-dev"). On many systems, these things should already be present. If you don't know exactly, simply skip this step for now and see if nice error messages pop up during the compile process. If they do, you can still install the missing build environment tools. So this is nothing that you need to look at very carefully.

Build Requirements

At a minimum, the following development tools must be present on the system:

- C compiler (usually gcc)
- make
- libtool
- rst2man (part of Python docutils) if you want to generate the man files
- Bison and Flex (preferrably, otherwise yacc and lex)
- zlib development package (usually *libz-dev*)
- json-c (usually named *libjson0-dev* or similar)
- libuuid (usually *uuid-dev*, if not present use –disable-uuid)
- libgcrypt (usually *libgcrypt-dev*)

Also, development versions of the following supporting libraries that the rsyslog project provides are necessary:

- liblogging (only stdlog component is hard requirement)
- libestr

In contrast to the other dependencies, recent versions of rsyslog may require recent versions of these libraries as well, so there is a chance that they must be built from source, too.

Depending on which plugins are enabled, additional dependencies exist. These are reported during the ./configure run.

Important: you need the **development** version of the packages in question. That is the version which is used by developers to build software that uses the respecitive package. Usually, they are separate from the regular user package. So if you just install the regular package but not the development one, ./configure will fail.

As a concrete example, you may want to build ommysql. It obviously requires a package like *mysql-client*, but that is just the regular package and not sufficient to buid rsyslog succesfully. To do so, you need to also install something named like *mysql-client-dev*.

Usually, the regular package is automatically installed, when you select the development package, but not vice versa. The exact behaviour and names depend on the distribution you use. It is quite common to name development packages something along the line of *pkgname-dev* or *pkgname-devel* where *pkgname* is the regular package name (like *mysql-client* in the above example).

Step 2 - Run ./configure

Run ./configure to adopt rsyslog to your environment. While doing so, you can also enable options. Configure will display selected options when it is finished. For example, to enable MySQL support, run

```
./configure --enable-mysql
```

Please note that MySQL support by default is NOT disabled.

To learn which ./configure options are available and what their default values are, use

```
./configure --help
```

Step 3 - Compile

That is easy. Just type "make" and let the compiler work. On any recent system, that should be a very quick task, on many systems just a matter of a few seconds. If an error message comes up, most probably a part of your build environment is not installed. Check with step 1 in those cases.

Step 4 - Install

Again, that is quite easy. All it takes is a "sudo make install". That will copy the rsyslogd and the man pages to the relevant directories.

Step 5 - Configure rsyslogd

In this step, you tell rsyslogd what to do with received messages. If you are upgrading from stock syslogd, /etc/syslog.conf is probably a good starting point. Rsyslogd understands stock syslogd syntax, so you can simply copy over /etc/syslog.conf to /etc/rsyslog.conf. Note since version 3 rsyslog requires to load plug-in modules to perform useful work (more about compatibilty notes v3). To load the most common plug-ins, add the following to the top of rsyslog.conf:

```
$ModLoad immark # provides --MARK-- message capability
$ModLoad imudp # provides UDP syslog reception
$ModLoad imtcp # provides TCP syslog reception
$ModLoad imuxsock # provides support for local system logging
$ModLoad imklog # provides kernel logging support
```

Change rsyslog.conf for any further enhancements you would like to see. For example, you can add database writing as outlined in the paper "Writing syslog Data to MySQL" (remember you need to enable MySQL support during step 2 if you want to do that!).

Step 6 - Disable stock syslogd

You can skip this and the following steps if rsyslog was already installed as the stock syslogd on your system (e.g. via a distribution default or package). In this case, you are finished.

If another syslogd is installed, it must be disabled and rsyslog set to become the default. This is because both it and rsyslogd listen to the same sockets, they can NOT be run concurrently. So you need to disable the stock syslogd. To do this, you typically must change your rc.d startup scripts.

For example, under Debian this must be done as follows: The default runlevel is 2. We modify the init scripts for runlevel 2 - in practice, you need to do this for all run levels you will ever use (which probably means all). Under /etc/rc2.d there is a S10sysklogd script (actually a symlink). Change the name to _S10sysklogd (this keeps the symlink in place, but will prevent further execution - effectively disabling it).

Step 7 - Enable rsyslogd Autostart

This step is very close to step 3. Now, we want to enable rsyslogd to start automatically. The rsyslog package contains a (currently small) number of startup scripts. They are inside the distro-specific directory (e.g. debian). If there is nothing for your operating system, you can simply copy the stock syslogd startup script and make the minor modifications to run rsyslogd (the samples should be of help if you intend to do this).

In our Debian example, the actual scripts are stored in /etc/init.d. Copy the standard script to that location. Then, you need to add a symlink to it in the respective rc.d directory. In our sample, we modify rc2.d, and can do this via the command "ln -s ../init.d/rsyslogd S10rsyslogd". Please note that the S10 prefix tells the system to start rsyslogd at the same time stock sysklogd was started.

Important: if you use the database functionality, you should make sure that MySQL starts before rsyslogd. If it starts later, you will receive an error message during each restart (this might be acceptable to you). To do so, either move MySQL's start order before rsyslogd or rsyslogd's after MySQL.

Step 8 - Check daily cron scripts

Most distributions come pre-configured with some daily scripts for log rotation. As long as you use the same log file names, the log rotation scripts will probably work quite well. There is one caveat, though. The scripts need to tell syslogd that the files have been rotated. To do this, they typically have a part using syslogd's init script to do that. Obviously, scripts for other default daemons do not know about rsyslogd, so they manipulate the other one. If that happens, in most cases an additional instance of that daemon is started. It also means that rsyslogd is not properly told about the log rotation, which will lead it to continue to write to the now-rotated files.

So you need to fix these scripts. See your distro-specific documentation how they are located.

Done

This concludes the steps necessary to install rsyslog. Of course, it is always a good idea to test everything thoroughly. At a minimalist level, you should do a reboot and after that check if everything has come up correctly. Pay attention not only to running processes, but also check if the log files (or the database) are correctly being populated.

If rsyslogd encounters any serious errors during startup, you should be able to see them at least on the system console. They might not be in log file, as errors might occur before the log file rules are in place. So it is always a good idea

to check system console output when things don't go smooth. In some rare cases, enabling debug logging (-d option) in rsyslogd can be helpful. If all fails, go to www.rsyslog.com and check the forum or mailing list for help with your issue.

Housekeeping stuff

This section and its subsections contain all these nice things that you usually need to read only if you are really curios ;)

Feedback requested

I would appreciate feedback on this tutorial. Additional ideas, comments or bug sighting reports are very welcome. Please let me know about them.

Revision History

- 2005-08-08 * Rainer Gerhards * Initial version created
- 2005-08-09 * Rainer Gerhards * updated to include distro-specific directories, which are now mandatory
- 2005-09-06 * Rainer Gerhards * added information on log rotation scripts
- 2007-07-13 * Rainer Gerhards * updated to new autotools-based build system
- 2008-10-01 * Rainer Gerhards * added info on building from source repository
- 2014-03181 * Rainer Gerhards * revamped doc to match current state.

Copyright

Installing rsyslog from the source repository

In most cases, people install rsyslog either via a package or use an "official" distribution tarball to generate it. But there may be situations where it is desirable to build directly from the source repository. This is useful for people who would like to participate in development or who would like to use the latest, not-yet-released code. The later may especially be the case if you are asked to try out an experimental version.

Building from the repsitory is not much different than building from the source tarball, but some files are missing because they are output files and thus do not belong into the repository.

Obtaining the Source

First of all, you need to download the sources. Rsyslog is kept in git. The "Where to find the rsyslog source code" page on the project site will point you to the current repository location.

After you have cloned the repository, you are in the master branch by default. This is where we keep the devel branch. If you need any other branch, you need to do a "git checkout –track -b branch origin/branch". For example, the command to check out the beta branch is "git checkout –track -b beta origin/beta".

Prequisites

To build the compilation system, you need

- GNU autotools (autoconf, automake, ...)
- libtool
- pkg-config

Unfortunately, the actual package names vary between distributions. Doing a search for the names above inside the packaging system should lead to the right path, though.

If some of these tools are missing, you will see errors like this one:

```
checking for SYSLOG_UNIXAF support... yes
checking for FSSTND support... yes
./configure: line 25895: syntax error near unexpected token `RELP,'
./configure: line 25895: ` PKG_CHECK_MODULES(RELP, relp >= 0.1.1)'
```

The actual error message will vary. In the case shown here, pkg-config was missing.

Important: the build dependencies must be present **before** creating the build environment is begun. Otherwise, some hard to interpret errors may occur. For example, the error above will also occur if you install pkg-config, but *after* you have run *autoreconf*. So be sure everything is in place *before* you create the build environment.

Creating the Build Environment

This is fairly easy: just issue "**autoreconf -fvi**", which should do everything you need. Once this is done, you can follow the usual ./configure steps just like when you downloaded an official distribution tarball (see the rsyslog install guide, starting at step 2, for further details about that).

Troubleshooting

Typical Problems

Output File is not Being Written

Note: current rsyslog versions have somewhat limited error reporting inside omfile. If a problem persists, you may want to generate a rsyslog debug log, which often can help you pinpoint the actual root cause of the problem more quickly.

To learn more about the current state of error reporting, follow our bug tracker for this issue.

The following subsections list frequent causes for file writing problems. You can quickly check this without the need to create a debug log.

SELinux

This often stems back to **selinux** permission errors, especially if files outside of the `/var/log` directory shall be written to.

Follow the *SELinux troubleshooting guide* to check for this condition.

Max Number of Open Files

This can also be caused by a too low limit on number of open file handles, especially when dynafiles are being written.

Note that some versions of systemd limit the process to 1024 files by default. The current set limit can be validated by doing:

```
cat /proc/<pid>/limits
```

and the currently open number of files can be obtained by doing:

```
ls /proc/<pid>/fd | wc -l
```

Also make sure the system-wide max open files is appropriate using:

```
sysctl fs.file-max
```

Some versions of systemd completely ignore `/etc/security/limits*`. To change limits for a service in systemd, edit `/usr/lib/systemd/system/rsyslog.service` and under `[Service]` add: `LimitNOFILE=<new value>`.

Then run:

```
systemctl daemon-reload
systemctl restart rsyslog
```

Troubleshooting SELinux-Related Issues

SELinux by its very nature can block many features of rsyslog (or any other process, of course), even when run under root. Actually, this is what it is supposed to do, so there is nothing bad about it.

If you suspect that some issues stems back to SELinux configuration, do the following:

- *temporarily* disabling SELinux
- restart rsyslog
- retry the operation

If it now succeeds, you know that you have a SELinux policy issue. The solution here is **not** to keep SELinux disabled. Instead do:

- reenable SELinux (set back to previous state, whatever that was)
- add proper SELinux policies for what you want to do with rsyslog

With SELinux running, restart rsyslog $ sudo audit2allow -a audit2allow will read the audit.log and list any SELinux infractions, namely the rsyslog infractions $ sudo audit2allow -a -M <FRIENDLY_NAME_OF_MODULE>.pp audit2allow will create a module allowing all previous infractions to have access $ sudo semodule -i <FRIENDLY_NAME_OF_MODULE>.pp Your module is loaded! Restart rsyslog and continue to audit until no

more infractions are detected and rsyslog has proper access. Additionally, you can save these modules and install them on future machines where rsyslog will need the same access.

General Procedure

Rsyslog Debug Support

For harder to find issues, rsyslog has integrated debug support. Usually, this is not required for finding configuration issues but rather to hunt for program or plugin bugs. However, there are several occasions where debug log has proven to be quite helpful in finding out configuration issues.

Part of debug supports is activated by adding the `--enable-rtinst` ./configure option ("rtinst" means runtime instrumentation). Turning debugging on obviously costs some performance (in some cases considerable). For typical cases, runtime instrumentation is *not* required.

Signals supported

SIGUSR1 - turns debug messages on and off. Note that for this signal to work, rsyslogd must be running with debugging enabled, either via the -d command line switch or the environment options specified below. It is **not** required that rsyslog was compiled with debugging enabled (but depending on the settings this may lead to better debug info).

SIGUSR2 - outputs debug information (including active threads and a call stack) for the state when SIGUSR2 was received. This is a one-time output. Can be sent as often as the user likes.

Note: this signal **may go away** in later releases and may be replaced by something else.

Environment Variables

There are two environment variables that set several debug settings:

- The "RSYSLOG_DEBUGLOG" (sample: RSYSLOG_DEBUGLOG="/path/to/debuglog/debug.log") writes (allmost) all debug message to the specified log file in addition to stdout. Some system messages (e.g. segfault or abort message) are not written to the file as we can not capture them.
- Runtime debug support is controlled by "RSYSLOG_DEBUG".

  The "RSYSLOG_DEBUG" environment variable contains an option string with the following options possible (all are case insensitive):

  - **LogFuncFlow** - print out the logical flow of functions (entering and exiting them)
  - **FileTrace** - specifies which files to trace LogFuncFlow. If **not** set (the default), a LogFuncFlow trace is provided for all files. Set to limit it to the files specified. FileTrace may be specified multiple times, one file each (e.g. export RSYSLOG_DEBUG="LogFuncFlow FileTrace=vm.c FileTrace=expr.c"
  - **PrintFuncDB** - print the content of the debug function database whenever debug information is printed (e.g. abort case)!
  - **PrintAllDebugInfoOnExit** - print all debug information immediately before rsyslogd exits (currently not implemented!)
  - **PrintMutexAction** - print mutex action as it happens. Useful for finding deadlocks and such.
  - **NoLogTimeStamp** - do not prefix log lines with a timestamp (default is to do that).

- **NoStdOut** - do not emit debug messages to stdout. If RSYSLOG_DEBUGLOG is not set, this means no messages will be displayed at all.
- **Debug** - if present, turns on the debug system and enables debug output
- **DebugOnDemand** - if present, turns on the debug system but does not enable debug output itself. You need to send SIGUSR1 to turn it on when desired.
- **OutputTidToStderr** - if present, makes rsyslog output information about the thread id (tid) of newly create processesto stderr. Note that not necessarily all new threads are reported (depends on the code, e.g. of plugins). This is only available under Linux. This usually does NOT work when privileges have been dropped (that's not a bug, but the way it is).
- **help** - display a very short list of commands - hopefully a life saver if you can't access the documentation...

Individual options are separated by spaces.

Why Environment Variables?

You may ask why we use environment variables for debug-system parameters and not the usual rsyslog.conf configuration commands. After all, environment variables force one to change distro-specific configuration files, whereas regular configuration directives would fit nicely into the one central rsyslog.conf.

The problem here is that many settings of the debug system must be initialized before the full rsyslog engine starts up. At that point, there is no such thing like rsyslog.conf or the objects needed to process it present in an running instance. And even if we would enable to change settings some time later, that would mean that we can not correctly monitor (and debug) the initial startup phase of rsyslogd. What makes matters worse is that during this startup phase (and never again later!) some of the base debug structure needs to be created, at least if the build is configured for that (many of these things only happen in –enable-rtinst mode). So if we do not initialize the debug system **before** actually startig up the rsyslog core, we get a number of data structures wrong.

For these reasons, we utilize environment variables to initialize and configure the debugging system. We understand this may be somewhat painful, but now you know there are at least some good reasons for doing so.

HOWEVER, if you have a too hard time to set debug instructions using the environment variables, there is a cure, described in the next paragraph.

Enabling Debug via rsyslog.conf

As described in the previous paragraph, enabling debug via rsyslog.conf may not be perfect for some debugging needs, but basic debug output will work - and that is what most often is requried. There are limited options available, but these cover the most important use cases.

Debug processing is done via legacy config statements. There currently is no plan to move these over to the v6+ config system. Availabe settings are

- $DebugFile <filename> - sets the debug file name
- $DebugLevel <0|1|2> - sets the respective debug level, where 0 means debug off, 1 is debug on demand activated (but debug mode off) and 2 is full debug mode.

Note that in theory it is forbidden to specify these parameters more than once. However, we do not enforce that and if it happens results are undefined.

Getting debug information from a running Instance

It is possible to obtain debugging information from a running instance, but this requires some setup. We assume that the instance runs in the background, so debug output to stdout is not desired. As such, all debug information needs to go into a log file.

To create this setup, you need to

- point the RSYSLOG_DEBUGLOG environment variable to a file that is accessible during the while runtime (we strongly suggest a file in the local file system!)
- set RSYSLOG_DEBUG at least to "DebugOnDeman NoStdOut"
- make sure these environment variables are set in the correct (distro-specifc) startup script if you do not run rsyslogd interactively

These settings enable the capability to react to SIGUSR1. The signal will toggle debug status when received. So send it one to turn debug loggin on, and send it again to turn debug logging off again. The third time it will be turned on again ... and so on.

On a typical system, you can signal rsyslogd as follows:

```
kill -USR1 `cat /var/run/rsyslogd.pid`
```

Important: there are backticks around the "cat"-command. If you use the regular quote it won't work. The debug log will show whether debug logging has been turned on or off. There is no other indication of the status.

Note: running with DebugOnDemand by itself does in practice not have any performance toll. However, switching debug logging on has a severe performance toll. Also, debug logging synchronizes much of the code, removing a lot of concurrency and thus potential race conditions. As such, the very same running instance may behave very differently with debug logging turned on vs. off. The on-demand debug log functionality is considered to be very valuable to analyze hard-to-find bugs that only manifest after a long runtime. Turning debug logging on a failing instance may reveal the cause of the failure. However, depending on the failure, debug logging may not even be successfully be turned on. Also note that with this rsyslog version we cannot obtain any debug information on events that happened *before* debug logging was turned on.

If an instance hangs, it is possible to obtain some useful information about the current threads and their calling stack by sending SIGUSR2. However, the usefulness of that information is very much depending on rsyslog compile-time settings, must importantly the –enable-rtinst configure flag. Note that activating this option causes additional overhead and slows down rsyslgod considerable. So if you do that, you need to check if it is capable to handle the workload. Also, threading behavior is modified by the runtime instrumentation.

Sending SIGUSR2 writes new process state information to the log file each time it is sent. So it may be useful to do that from time to time. It probably is most useful if the process seems to hang, in which case it may (may!) be able to output some diagnostic information on the current processing state. In that case, turning on the mutex debugging options (see above) is probably useful.

Interpreting the Logs

Debug logs are primarily meant for rsyslog developers. But they may still provide valuable information to users. Just be warned that logs sometimes contains informaton the looks like an error, but actually is none. We put a lot of extra information into the logs, and there are some cases where it is OK for an error to happen, we just wanted to record it inside the log. The code handles many cases automatically. So, in short, the log may not make sense to you, but it (hopefully) makes sense to a developer. Note that we developers often need many lines of the log file, it is relatively rare that a problem can be diagnosed by looking at just a couple of (hundered) log records.

Security Risks

The debug log will reveal potentially sensible information, including user accounts and passwords, to anyone able to read the log file. As such, it is recommended to properly guard access to the log file. Also, an instance running with debug log enabled runs much slower than one without. An attacker may use this to place carry out a denial-of-service attack or try to hide some information from the log file. As such, it is suggested to enable DebugOnDemand mode only for a reason. Note that when no debug mode is enabled, SIGUSR1 and SIGUSR2 are completely ignored.

When running in any of the debug modes (including on demand mode), an interactive instance of rsyslogd can be aborted by pressing ctl-c.

See Also

- How to use debug on demand

troubleshooting problems

Having trouble with rsyslog? This page provides some tips on where to look for help and what to do if you need to ask for assistance. This page is continously being expanded.

Useful troubleshooting ressources are:

- The rsyslog documentation - note that the online version always covers the most recent development version. However, there is a version-specific doc set in each tarball. If you installed rsyslog from a package, there usually is a rsyslog-doc package, that often needs to be installed separately.
- The rsyslog wiki provides user tips and experiences.
- Check the rsyslog github issue tracker and the bugzilla to see if your problem is a known (and even fixed ;)) bug. **Note:** the preferred way to create new bugs is via github. The bugzilla does no longer accept new bugs. It is just kept to work on old ones and as a reference source for ChangeLog entries.

Malformed Messages and Message Properties

A common trouble source are ill-formed syslog messages, which lead to to all sorts of interesting problems, including malformed hostnames and dates. Read the quoted guide to find relief. A common symptom is that the %HOSTNAME% property is used for generating dynafile names, but some glibberish shows up. This is caused by the malformed syslog messages, so be sure to read the *guide on syslog parsing* if you face that problem. Just let me add that the common work-around is to use %FROMHOST% or %FROMHOST-IP% instead. These do not take the hostname from the message, but rather use the host that sent the message (taken from the socket layer). Of course, this does not work over NAT or relay chains, where the only cure is to make sure senders emit well-formed messages.

Configuration Problems

Rsyslog has support for configuration checking. It offers a special command line switch (-N<*value*>) that puts it into "config verfication mode". In that mode, it interprets and checks the configuration file, but does not startup. This mode can be used in parallel to a running instance of rsyslogd.

The *value* is a set of binary values. Currently, there only is

value	meaning
1	turn on config checking
2	permit checking of include files

Where 2 automatically turns on config checking mode, if not given. In that sense `-N2` and `-N3` are equivalent.

When set to check include files, some conditions are relaxed. For example, rsyslog usually requires that at least one action is defined somewhere in the configuration. For obvious reasons, it would not make much sense to run an instance without any action. However, when an include file is checked, it may happen that it contains no actions as all. As such, the requirement to include one action has been lifted in include file checking.

To check a full rsyslog configuration, run rsyslog interactively as follows:

```
$ /path/to/rsyslogd -f/path/to/config-file -N1
```

You should also specify other options you usually give. Any problems experienced are reported to stderr [aka "your screen" (if not redirected)].

If you would like to check just an include file, instead use:

```
$ /path/to/rsyslogd -f/path/to/config-file -N3
```

Checking Connection Problems

If a client cannot connect via the network to the rsyslog server, you can do a connection check via netcat. That will verify if the sender is able to deliver to an application running on the receiver. Netcat is a very simple receiver, so we can be sure that no netcat problem will interfere with this test.

With netcat, you can test UDP and TCP syslog connections, but not TLS.

To do this test, you need to

- on the client
  - stop the syslog sender process, if possible. If the sender is rsyslog, you can use the same procedure described below for the server.
- on the rsyslog server
  - stop and/or disable rsyslog On systemd systems (newer distro versions), systemd might automatically restart rsyslog when data is written to the system log socket. To be sure, we recommend to disable the service on those systems. This sequence should work: $ systemctl disable rsyslog.service $ systemctl stop rsyslog.service
  - open a terminal session, and start a netcat listener **on the same listening port** that you have configured inside rsyslog. Note that if you use a privileged port, you need to execute nc as root. We assume port 13515 is used for rsyslog, so do this: $ nc -k -l <ip-of-server> 13515 # [FOR TCP] OR sudo nc ... $ nc -u -l <ip-of-server> 13515 # [FOR UDP] OR sudo nc ...
- on the syslog client
  - send a test message via netcat: $ echo "test message 1" | nc <ip-of-server> 13515 # [FOR TCP] $ echo "test message 1" | nc <ip-of-server> 13515 # [FOR UDP]
- on the server
  - check if you received the test message. Note that you might also have received additional messages if the original sender process was not stopped. If you see garbagge, most probably some sender tries to send via TLS.
  - you can stop nc by <ctl>-c

If you did not see the test message arrive at the centeral server, the problem is most probably rooted in the network configuration or other parts of the system configuration. Things to check are - firewall settings

- for UDP: does the sender have a route back to the original sender? This is often required by modern systems to prevent spoofing; if the sender cannot be reached, UDP messages are discarded AFTER they have been received by the OS (an app like netcat or rsyslog will never see them)
- if that doesn't help, you a network monitor (or tcpdump, Wireshark, ...) to verify that the network packet at least reaches the system.

If you saw the test message arrive at the central server, the problem most probably is related to the rsyslog configuration or the system configuration that affects rsyslog (SELinux, AppArmor, ...).

A good next test is to run rsyslog interactively, just like you did with netcat:

- on the server - make sure the rsyslog service is still stopped
  - run $ sudo /usr/sbin/rsyslogd -n
- on the client
  - send a test message
- on the server - check if the message arrived
  - terminate rsyslog by pressing <ctl>-c

If the test message arrived, you definitely have a problem with the system configuration, most probably in SELinux, AppArmor or a similar subsystem. Note that your interactive security context is quite different from the rsyslog system service context.

If the test message did not arrive, it is time to generate a debug log to see exactly what rsyslog does. A full description is in this file a bit down below, but in essence you need to do

- on the server - make sure the rsyslog service is still stopped - run

  $ sudo /usr/sbin/rsyslogd -nd 2> rsyslog-debug.log

- on the client - send a test message
- on the server - stop rsyslog by pressing <ctl>- - review debug log

Asking for Help

If you can't find the answer yourself, you should look at these places for community help.

- The rsyslog mailing list. This is a low-volume list which occasional gets traffic spikes. The mailing list is probably a good place for complex questions. This is the preferred method of obtaining support.
- The rsyslog forum.

Debug Log

If you ask for help, there are chances that we need to ask for an rsyslog debug log. The debug log is a detailled report of what rsyslog does during processing. As such, it may even be useful for your very own troubleshooting. People have seen things inside their debug log that enabled them to find problems they did not see before. So having a look at the debug log, even before asking for help, may be useful.

Note that the debug log contains most of those things we consider useful. This is a lot of information, but may still be too few. So it sometimes may happen that you will be asked to run a specific version which has additional debug output. Also, we revise from time to time what is worth putting into the standard debug log. As such, log content

may change from version to version. We do not guarantee any specific debug log contents, so do not rely on that. The amount of debug logging can also be controlled via some environment options. Please see debugging support for further details.

In general, it is advisable to run rsyslogd in the foreground to obtain the log. To do so, make sure you know which options are usually used when you start rsyslogd as a background daemon. Let's assume "-c5" is the only option used. Then, do the following:

- make sure rsyslogd as a daemon is stopped (verify with ps -eflgrep rsyslogd)
- make sure you have a console session with root permissions
- run rsyslogd interactively: `` `/sbin/rsyslogd ..your options.. -dn > logfile` `` where "your options" is what you usually use. /sbin/rsyslogd is the full path to the rsyslogd binary (location different depending on distro). In our case, the command would be `` `/sbin/rsyslogd -c5 -dn > logfile` ``
- press ctrl-C when you have sufficient data (e.g. a device logged a record) **NOTE: rsyslogd will NOT stop automatically - you need to ctrl-c out of it!**
- Once you have done all that, you can review logfile. It contains the debug output.
- When you are done, make sure you re-enable (and start) the background daemon!

If you need to submit the logfile, you may want to check if it contains any passwords or other sensitive data. If it does, you can change it to some **consistent** meaningless value. **Do not delete the lines**, as this renders the debug log unusable (and makes Rainer quite angry for wasted time, aka significantly reduces the chance he will remain motivated to look at your problem ;)). For the same reason, make sure whatever you change is change consistently. Really!

Debug log file can get quite large. Before submitting them, it is a good idea to zip them. Rainer has handled files of around 1 to 2 GB. If your's is larger ask before submitting. Often, it is sufficient to submit the first 2,000 lines of the log file and around another 1,000 around the area where you see a problem. Also, ask you can submit a file via private mail. Private mail is usually a good way to go for large files or files with sensitive content. However, do NOT send anything sensitive that you do not want the outside to be known. While Rainer so far made effort no to leak any sensitive information, there is no guarantee that doesn't happen. If you need a guarantee, you are probably a candidate for a commercial support contract. Free support comes without any guarantees, include no guarantee on confidentiality [aka "we don't want to be sued for work were are not even paid for ;)]. **So if you submit debug logs, do so at your sole risk**. By submitting them, you accept this policy.

Segmentation Faults

Rsyslog has a very rapid development process, complex capabilities and now gradually gets more and more exposure. While we are happy about this, it also has some bad effects: some deployment scenarios have probably never been tested and it may be impossible to test them for the development team because of resources needed. So while we try to avoid this, you may see a serious problem during deployments in demanding, non-standard, environments (hopefully not with a stable version, but chances are good you'll run into troubles with the development versions).

In order to aid the debugging process, it is useful to have debug symbols on the system. If you build rsyslog yourself, make sure that the `-g` option is included in CFLAGS. If you use packages, the debug symbols come in their own package. **It is highly recommended to install that package as it provides tremendous extra benefit.** To do so, do:

```
yum install rsyslog-debuginfo
```

Obviously, this is for RPM-based systems, but it's essentially the same with other packaging systems, just use the native commands. Note that the package may be named slightly different, but it should always be fairly easy to locate.

Active support from the user base is very important to help us track down those things. Most often, serious problems are the result of some memory misadressing. During development, we routinely use valgrind, a very well and capable memory debugger. This helps us to create pretty clean code. But valgrind can not detect everything, most importantly not code pathes that are never executed. So of most use for us is information about aborts and abort locations.

Unfortunately, faults rooted in adressing errors typically show up only later, so the actual abort location is in an unrelated spot. To help track down the original spot, libc later than 5.4.23 offers support for finding, and possible temporary relief from it, by means of the MALLOC_CHECK_ environment variable. Setting it to 2 is a useful troubleshooting aid for us. It will make the program abort as soon as the check routines detect anything suspicious (unfortunately, this may still not be the root cause, but hopefully closer to it). Setting it to 0 may even make some problems disappear (but it will NOT fix them!). With functionality comes cost, and so exporting MALLOC_CHECK_ without need comes at a performance penalty. However, we strongly recommend adding this instrumentation to your test environment should you see any serious problems. Chances are good it will help us interpret a dump better, and thus be able to quicker craft a fix.

In order to get useful information, we need some backtrace of the abort. First, you need to make sure that a core file is created. Under Fedora, for example, that means you need to have an "ulimit -c unlimited" in place.

Now let's assume you got a core file (e.g. in /core.1234). So what to do next? Sending a core file to us is most often pointless - we need to have the exact same system configuration in order to interpret it correctly. Obviously, chances are extremely slim for this to be. So we would appreciate if you could extract the most important information. This is done as follows:

```
$ gdb /path/to/rsyslogd
$ core /core.1234
$ info thread
$ thread apply all bt full
$ q # quits gdb
```

Then please send all information that gdb spit out to the development team. It is best to first ask on the forum or mailing list on how to do that. The developers will keep in contact with you and, I fear, will probably ask for other things as well ;)

Note that we strive for highest reliability of the engine even in unusual deployment scenarios. Unfortunately, this is hard to achieve, especially with limited resources. So we are depending on cooperation from users. This is your chance to make a big contribution to the project without the need to program or do anything else except get a problem solved.

FAQ

What is the difference between the main_queue and a queue with a ruleset tied to an input?

A queue on a ruleset tied to an input replaces the main queue for that input. The only difference is the higher default size of the main queue.

If an input bounded ruleset does not have a queue defined, what default does it have?

Rulesets without a queue on them use the main queue as a default.

What is the recommended way to use several inputs? Should there be a need to define a queue for the rulesets?

If you are going to do the same thing with all logs, then they should share a ruleset.

If you are doing different things with different logs, then they should have different rulesets.

For example take a system where the default ruleset processes things and sends them over the network to the nearest relay system. All systems have this same default ruleset. Then in the relay systems there is a ruleset which is tied

to both TCP and UDP listeners, and it receives the messages from the network, cleans them up, and sends them on. There is no mixing of these two processing paths, so having them as completly separate paths with rulesets tied to the inputs and queues on the rulesets makes sense.

A queue on a ruleset tied to one or more inputs can be thought of as a seperate instance of rsyslog, which processes those logs.

Credits: davidelang

Concepts

This chapter describes important rsyslog concepts and objects. Where appropriate, it also refers to configurations settings to affect the respective objects.

Understanding rsyslog Queues

Rsyslog uses queues whenever two activities need to be loosely coupled. With a queue, one part of the system "produces" something while another part "consumes" this something. The "something" is most often syslog messages, but queues may also be used for other purposes.

This document provides a good insight into technical details, operation modes and implications. In addition to it, an *rsyslog queue concepts overview* document exists which tries to explain queues with the help of some analogies. This may probably be a better place to start reading about queues. I assume that once you have understood that document, the material here will be much easier to grasp and look much more natural.

The most prominent example is the main message queue. Whenever rsyslog receives a message (e.g. locally, via UDP, TCP or in whatever else way), it places these messages into the main message queue. Later, it is dequeued by the rule processor, which then evaluates which actions are to be carried out. In front of each action, there is also a queue, which potentially de-couples the filter processing from the actual action (e.g. writing to file, database or forwarding to another host).

Where are Queues Used?

Currently, queues are used for the main message queue and for the actions.

There is a single main message queue inside rsyslog. Each input module delivers messages to it. The main message queue worker filters messages based on rules specified in rsyslog.conf and dispatches them to the individual action queues. Once a message is in an action queue, it is deleted from the main message queue.

There are multiple action queues, one for each configured action. By default, these queues operate in direct (non-queueing) mode. Action queues are fully configurable and thus can be changed to whatever is best for the given use case.

Future versions of rsyslog will most probably utilize queues at other places, too.

Wherever "<*object*>" is used in the config file statements, substitute "<*object*>" with either "MainMsg" or "Action". The former will set main message queue parameters, the later parameters for the next action that will be created. Action queue parameters can not be modified once the action has been specified. For example, to tell the main message queue to save its content on shutdown, use *$MainMsgQueueSaveOnShutdown on*".

If the same parameter is specified multiple times before a queue is created, the last one specified takes precedence. The main message queue is created after parsing the config file and all of its potential includes. An action queue is created each time an action selector is specified. Action queue parameters are reset to default after an action queue has been created (to provide a clean environment for the next action).

Not all queues necessarily support the full set of queue configuration parameters, because not all are applicable. For example, in current output module design, actions do not support multi-threading. Consequently, the number of worker threads is fixed to one for action queues and can not be changed.

Queue Modes

Rsyslog supports different queue modes, some with submodes. Each of them has specific advantages and disadvantages. Selecting the right queue mode is quite important when tuning rsyslogd. The queue mode (aka "type") is set via the "*$<object>QueueType*" config directive.

Direct Queues

Direct queues are **non**-queuing queues. A queue in direct mode does neither queue nor buffer any of the queue elements but rather passes the element directly (and immediately) from the producer to the consumer. This sounds strange, but there is a good reason for this queue type.

Direct mode queues allow to use queues generically, even in places where queuing is not always desired. A good example is the queue in front of output actions. While it makes perfect sense to buffer forwarding actions or database writes, it makes only limited sense to build up a queue in front of simple local file writes. Yet, rsyslog still has a queue in front of every action. So for file writes, the queue mode can simply be set to "direct", in which case no queuing happens.

Please note that a direct queue also is the only queue type that passes back the execution return code (success/failure) from the consumer to the producer. This, for example, is needed for the backup action logic. Consequently, backup actions require the to-be-checked action to use a "direct" mode queue.

To create a direct queue, use the "*$<object>QueueType Direct*" config directive.

Disk Queues

Disk queues use disk drives for buffering. The important fact is that they always use the disk and do not buffer anything in memory. Thus, the queue is ultra-reliable, but by far the slowest mode. For regular use cases, this queue mode is not recommended. It is useful if log data is so important that it must not be lost, even in extreme cases.

When a disk queue is written, it is done in chunks. Each chunk receives its individual file. Files are named with a prefix (set via the "*$<object>QueueFilename*" config directive) and followed by a 7-digit number (starting at one and incremented for each file). Chunks are 10mb by default, a different size can be set via the"*$<object>QueueMaxFileSize*" config directive. Note that the size limit is not a sharp one: rsyslog always writes one complete queue entry, even if it violates the size limit. So chunks are actually a little but (usually less than 1k) larger then the configured size. Each chunk also has a different size for the same reason. If you observe different chunk sizes, you can relax: this is not a problem.

Writing in chunks is used so that processed data can quickly be deleted and is free for other uses - while at the same time keeping no artificial upper limit on disk space used. If a disk quota is set (instructions further below), be sure that the quota/chunk size allows at least two chunks to be written. Rsyslog currently does not check that and will fail miserably if a single chunk is over the quota.

Creating new chunks costs performance but provides quicker ability to free disk space. The 10mb default is considered a good compromise between these two. However, it may make sense to adapt these settings to local policies. For example, if a disk queue is written on a dedicated 200gb disk, it may make sense to use a 2gb (or even larger) chunk size.

Please note, however, that the disk queue by default does not update its housekeeping structures every time it writes to disk. This is for performance reasons. In the event of failure, data will still be lost (except when manually is mangled with the file structures). However, disk queues can be set to write bookkeeping information on checkpoints (every n

records), so that this can be made ultra-reliable, too. If the checkpoint interval is set to one, no data can be lost, but the queue is exceptionally slow.

Each queue can be placed on a different disk for best performance and/or isolation. This is currently selected by specifying different *$WorkDirectory* config directives before the queue creation statement.

To create a disk queue, use the "*$<object>QueueType Disk*" config directive. Checkpoint intervals can be specified via "*$<object>QueueCheckpointInterval*", with 0 meaning no checkpoints. Note that disk-based queues can be made very reliable by issuing a (f)sync after each write operation. Starting with version 4.3.2, this can be requested via "*<object>QueueSyncQueueFiles on/off* with the default being off. Activating this option has a performance penalty, so it should not be turned on without reason.

If you happen to lose or otherwise need the housekeeping structures and have all yours queue chunks you can use perl script included in rsyslog package to generate it. Usage: recover_qi.pl -w *$WorkDirectory* -f QueueFileName -d 8 > QueueFileName.qi

In-Memory Queues

In-memory queue mode is what most people have on their mind when they think about computing queues. Here, the enqueued data elements are held in memory. Consequently, in-memory queues are very fast. But of course, they do not survive any program or operating system abort (what usually is tolerable and unlikely). Be sure to use an UPS if you use in-memory mode and your log data is important to you. Note that even in-memory queues may hold data for an infinite amount of time when e.g. an output destination system is down and there is no reason to move the data out of memory (lying around in memory for an extended period of time is NOT a reason). Pure in-memory queues can't even store queue elements anywhere else than in core memory.

There exist two different in-memory queue modes: LinkedList and FixedArray. Both are quite similar from the user's point of view, but utilize different algorithms.

A FixedArray queue uses a fixed, pre-allocated array that holds pointers to queue elements. The majority of space is taken up by the actual user data elements, to which the pointers in the array point. The pointer array itself is comparatively small. However, it has a certain memory footprint even if the queue is empty. As there is no need to dynamically allocate any housekeeping structures, FixedArray offers the best run time performance (uses the least CPU cycle). FixedArray is best if there is a relatively low number of queue elements expected and performance is desired. It is the default mode for the main message queue (with a limit of 10,000 elements).

A LinkedList queue is quite the opposite. All housekeeping structures are dynamically allocated (in a linked list, as its name implies). This requires somewhat more runtime processing overhead, but ensures that memory is only allocated in cases where it is needed. LinkedList queues are especially well-suited for queues where only occasionally a than-high number of elements need to be queued. A use case may be occasional message burst. Memory permitting, it could be limited to e.g. 200,000 elements which would take up only memory if in use. A FixedArray queue may have a too large static memory footprint in such cases.

In general, it is advised to use LinkedList mode if in doubt. The processing overhead compared to FixedArray is low and may be outweigh by the reduction in memory use. Paging in most-often-unused pointer array pages can be much slower than dynamically allocating them.

To create an in-memory queue, use the "*$<object>QueueType LinkedList*" or "*$<object>QueueType FixedArray*" config directive.

Disk-Assisted Memory Queues

If a disk queue name is defined for in-memory queues (via *$<object>QueueFileName*), they automatically become "disk-assisted" (DA). In that mode, data is written to disk (and read back) on an as-needed basis.

Actually, the regular memory queue (called the "primary queue") and a disk queue (called the "DA queue") work in tandem in this mode. Most importantly, the disk queue is activated if the primary queue is full or needs to be persisted on shutdown. Disk-assisted queues combine the advantages of pure memory queues with those of pure disk queues. Under normal operations, they are very fast and messages will never touch the disk. But if there is need to, an unlimited amount of messages can be buffered (actually limited by free disk space only) and data can be persisted between rsyslogd runs.

With a DA-queue, both disk-specific and in-memory specific configuration parameters can be set. From the user's point of view, think of a DA queue like a "super-queue" which does all within a single queue [from the code perspective, there is some specific handling for this case, so it is actually much like a single object].

DA queues are typically used to de-couple potentially long-running and unreliable actions (to make them reliable). For example, it is recommended to use a disk-assisted linked list in-memory queue in front of each database and "send via tcp" action. Doing so makes these actions reliable and de-couples their potential low execution speed from the rest of your rules (e.g. the local file writes). There is a howto on massive database inserts which nicely describes this use case. It may even be a good read if you do not intend to use databases.

With DA queues, we do not simply write out everything to disk and then run as a disk queue once the in-memory queue is full. A much smarter algorithm is used, which involves a "high watermark" and a "low watermark". Both specify numbers of queued items. If the queue size reaches high watermark elements, the queue begins to write data elements to disk. It does so until it reaches the low water mark elements. At this point, it stops writing until either high water mark is reached again or the on-disk queue becomes empty, in which case the queue reverts back to in-memory mode, only. While holding at the low watermark, new elements are actually enqueued in memory. They are eventually written to disk, but only if the high water mark is ever reached again. If it isn't, these items never touch the disk. So even when a queue runs disk-assisted, there is in-memory data present (this is a big difference to pure disk queues!).

This algorithm prevents unnecessary disk writes, but also leaves some additional buffer space for message bursts. Remember that creating disk files and writing to them is a lengthy operation. It is too lengthy to e.g. block receiving UDP messages. Doing so would result in message loss. Thus, the queue initiates DA mode, but still is able to receive messages and enqueue them - as long as the maximum queue size is not reached. The number of elements between the high water mark and the maximum queue size serves as this "emergency buffer". Size it according to your needs, if traffic is very bursty you will probably need a large buffer here. Keep in mind, though, that under normal operations these queue elements will probably never be used. Setting the high water mark too low will cause disk-assistance to be turned on more often than actually needed.

The water marks can be set via the "*$<object>QueueHighWatermark*" and "*$<object>QueueLowWatermark*" configuration file directives. Note that these are actual numbers, not percentages. Be sure they make sense (also in respect to "*$<object>QueueSize*"), as rsyslodg does currently not perform any checks on the numbers provided. It is easy to screw up the system here (yes, a feature enhancement request is filed ;)).

Limiting the Queue Size

All queues, including disk queues, have a limit of the number of elements they can enqueue. This is set via the "*$<object>QueueSize*" config parameter. Note that the size is specified in number of enqueued elements, not their actual memory size. Memory size limits can not be set. A conservative assumption is that a single syslog messages takes up 512 bytes on average (in-memory, NOT on the wire, this *is* a difference).

Disk assisted queues are special in that they do **not** have any size limit. The enqueue an unlimited amount of elements. To prevent running out of space, disk and disk-assisted queues can be size-limited via the "*$<object>QueueMaxDiskSpace*" configuration parameter. If it is not set, the limit is only available free space (and reaching this limit is currently not very gracefully handled, so avoid running into it!). If a limit is set, the queue can not grow larger than it. Note, however, that the limit is approximate. The engine always writes complete records. As such, it is possible that slightly more than the set limit is used (usually less than 1k, given the average message size). Keeping strictly on the limit would be a performance hurt, and thus the design decision was to favour performance. If you don't like that policy, simply specify a slightly lower limit (e.g. 999,999K instead of 1G).

In general, it is a good idea to limit the physical disk space even if you dedicate a whole disk to rsyslog. That way, you prevent it from running out of space (future version will have an auto-size-limit logic, that then kicks in in such situations).

Worker Thread Pools

Each queue (except in "direct" mode) has an associated pool of worker threads. Worker threads carry out the action to be performed on the data elements enqueued. As an actual sample, the main message queue's worker task is to apply filter logic to each incoming message and enqueue them to the relevant output queues (actions).

Worker threads are started and stopped on an as-needed basis. On a system without activity, there may be no worker at all running. One is automatically started when a message comes in. Similarly, additional workers are started if the queue grows above a specific size. The "*$<object>QueueWorkerThreadMinimumMessages*" config parameter controls worker startup. If it is set to the minimum number of elements that must be enqueued in order to justify a new worker startup. For example, let's assume it is set to 100. As long as no more than 100 messages are in the queue, a single worker will be used. When more than 100 messages arrive, a new worker thread is automatically started. Similarly, a third worker will be started when there are at least 300 messages, a forth when reaching 400 and so on.

It, however, does not make sense to have too many worker threads running in parallel. Thus, the upper limit can be set via "*$<object>QueueWorkerThreads*". If it, for example, is set to four, no more than four workers will ever be started, no matter how many elements are enqueued.

Worker threads that have been started are kept running until an inactivity timeout happens. The timeout can be set via "*$<object>QueueWorkerTimeoutThreadShutdown*" and is specified in milliseconds. If you do not like to keep the workers running, simply set it to 0, which means immediate timeout and thus immediate shutdown. But consider that creating threads involves some overhead, and this is why we keep them running. If you would like to never shutdown any worker threads, specify -1 for this parameter.

Discarding Messages

If the queue reaches the so called "discard watermark" (a number of queued elements), less important messages can automatically be discarded. This is in an effort to save queue space for more important messages, which you even less like to lose. Please note that whenever there are more than "discard watermark" messages, both newly incoming as well as already enqueued low-priority messages are discarded. The algorithm discards messages newly coming in and those at the front of the queue.

The discard watermark is a last resort setting. It should be set sufficiently high, but low enough to allow for large message burst. Please note that it take effect immediately and thus shows effect promptly - but that doesn't help if the burst mainly consist of high-priority messages...

The discard watermark is set via the "*$<object>QueueDiscardMark*" directive. The priority of messages to be discarded is set via "*$<object>QueueDiscardSeverity*". This directive accepts both the usual textual severity as well as a numerical one. To understand it, you must be aware of the numerical severity values. They are defined in RFC 3164:

Code	Severity
0	Emergency: system is unusable
1	Alert: action must be taken immediately
2	Critical: critical conditions
3	Error: error conditions
4	Warning: warning conditions
5	Notice: normal but significant condition
6	Informational: informational messages
7	Debug: debug-level messages

Anything of the specified severity and (numerically) above it is discarded. To turn message discarding off, simply specify the discard watermark to be higher than the queue size. An alternative is to specify the numerical value 8 as

DiscardSeverity. This is also the default setting to prevent unintentional message loss. So if you would like to use message discarding, you need to set" *$<object>QueueDiscardSeverity*" to an actual value.

An interesting application is with disk-assisted queues: if the discard watermark is set lower than the high watermark, message discarding will start before the queue becomes disk-assisted. This may be a good thing if you would like to switch to disk-assisted mode only in cases where it is absolutely unavoidable and you prefer to discard less important messages first.

Filled-Up Queues

If the queue has either reached its configured maximum number of entries or disk space, it is finally full. If so, rsyslogd throttles the data element submitter. If that, for example, is a reliable input (TCP, local log socket), that will slow down the message originator which is a good resolution for this scenario.

During throttling, a disk-assisted queue continues to write to disk and messages are also discarded based on severity as well as regular dequeuing and processing continues. So chances are good the situation will be resolved by simply throttling. Note, though, that throttling is highly undesirable for unreliable sources, like UDP message reception. So it is not a good thing to run into throttling mode at all.

We can not hold processing infinitely, not even when throttling. For example, throttling the local log socket too long would cause the system at whole come to a standstill. To prevent this, rsyslogd times out after a configured period ("*$<object>QueueTimeoutEnqueue*", specified in milliseconds) if no space becomes available. As a last resort, it then discards the newly arrived message.

If you do not like throttling, set the timeout to 0 - the message will then immediately be discarded. If you use a high timeout, be sure you know what you do. If a high main message queue enqueue timeout is set, it can lead to something like a complete hang of the system. The same problem does not apply to action queues.

Rate Limiting

Rate limiting provides a way to prevent rsyslogd from processing things too fast. It can, for example, prevent overrunning a receiver system.

Currently, there are only limited rate-limiting features available. The "*$<object>QueueDequeueSlowdown*" directive allows to specify how long (in microseconds) dequeueing should be delayed. While simple, it still is powerful. For example, using a DequeueSlowdown delay of 1,000 microseconds on a UDP send action ensures that no more than 1,000 messages can be sent within a second (actually less, as there is also some time needed for the processing itself).

Processing Timeframes

Queues can be set to dequeue (process) messages only during certain timeframes. This is useful if you, for example, would like to transfer the bulk of messages only during off-peak hours, e.g. when you have only limited bandwidth on the network path the the central server.

Currently, only a single timeframe is supported and, even worse, it can only be specified by the hour. It is not hard to extend rsyslog's capabilities in this regard - it was just not requested so far. So if you need more fine-grained control, let us know and we'll probably implement it. There are two configuration directives, both should be used together or results are unpredictable:" *$<object>QueueDequeueTimeBegin <hour>*" and "*$<object>QueueDequeueTimeEnd <hour>*". The hour parameter must be specified in 24-hour format (so 10pm is 22). A use case for this parameter can be found in the rsyslog wiki.

Performance

The locking involved with maintaining the queue has a potentially large performance impact. How large this is, and if it exists at all, depends much on the configuration and actual use case. However, the queue is able to work on so-called "batches" when dequeueing data elements. With batches, multiple data elements are dequeued at once (with a single locking call). The queue dequeues all available elements up to a configured upper limit (*<object>DequeueBatchSize <number>*). It is important to note that the actual upper limit is dictated by availability. The queue engine will never wait for a batch to fill. So even if a high upper limit is configured, batches may consist of fewer elements, even just one, if there are no more elements waiting in the queue.

Batching can improve performance considerably. Note, however, that it affects the order in which messages are passed to the queue worker threads, as each worker now receive as batch of messages. Also, the larger the batch size and the higher the maximum number of permitted worker threads, the more main memory is needed. For a busy server, large batch sizes (around 1,000 or even more elements) may be useful. Please note that with batching, the main memory must hold BatchSize * NumOfWorkers objects in memory (worst-case scenario), even if running in disk-only mode. So if you use the default 5 workers at the main message queue and set the batch size to 1,000, you need to be prepared that the main message queue holds up to 5,000 messages in main memory **in addition** to the configured queue size limits!

The queue object's default maximum batch size is eight, but there exists different defaults for the actual parts of rsyslog processing that utilize queues. So you need to check these object's defaults.

Terminating Queues

Terminating a process sounds easy, but can be complex. Terminating a running queue is in fact the most complex operation a queue object can perform. You don't see that from a user's point of view, but its quite hard work for the developer to do everything in the right order.

The complexity arises when the queue has still data enqueued when it finishes. Rsyslog tries to preserve as much of it as possible. As a first measure, there is a regular queue time out ("*$<object>QueueTimeoutShutdown*", specified in milliseconds): the queue workers are given that time period to finish processing the queue.

If after that period there is still data in the queue, workers are instructed to finish the current data element and then terminate. This essentially means any other data is lost. There is another timeout ("*$<object>QueueTimeoutActionCompletion*", also specified in milliseconds) that specifies how long the workers have to finish the current element. If that timeout expires, any remaining workers are cancelled and the queue is brought down.

If you do not like to lose data on shutdown, the "*$<object>QueueSaveOnShutdown*" parameter can be set to "on". This requires either a disk or disk-assisted queue. If set, rsyslogd ensures that any queue elements are saved to disk before it terminates. This includes data elements there were begun being processed by workers that needed to be cancelled due to too-long processing. For a large queue, this operation may be lengthy. No timeout applies to a required shutdown save.

The Janitor Process

The janitor process carries out periodic cleanup tasks. For example, it is used by *omfile* to close files after a timeout has expired.

The janitor runs periodically. As such, all tasks carried out via the janitor will be activated based on the interval at which it runs. This means that all janitor-related times set are approximate and should be considered as "no earlier than" (NET). If, for example, you set a timeout to 5 minutes and the janitor is run in 10-minute intervals, the timeout may actually happen after 5 minutes, but it may also take up to 20 minutes for it to be detected.

In general (see note about HUP below), janitor based activities scheduled to occur after *n* minutes will occur after *n* and *(n + 2*janitorInterval)* minutes.

To reduce the potential delay caused by janitor invocation, *the interval at which the janitor runs can be be adjusted*. If high precision is required, it should be set to one minute. Janitor-based activities will still be NET times, but the time frame will be much smaller. In the example with the file timeout, it would be between 5 and 6 minutes if the janitor is run at a one-minute interval.

Note that the more frequent the janitor is run, the more frequent the system needs to wakeup from potential low power state. This is no issue for data center machines (which usually always run at full speed), but it may be an issue for power-constrained environments like notebooks. For such systems, a higher janitor interval may make sense.

As a special case, sending a HUP signal to rsyslog also activate the janitor process. This can lead to too-frequent wakeups of janitor-related services. However, we don't expect this to cause any issues. If it does, it could be solved by creating a separate thread for the janitor. But as this takes up some system resources and is not not considered useful, we have not implemented it that way. If the HUP/janitor interaction causes problems, let the rsyslog team know and we can change the implementation.

Message parsers in rsyslog

Written by Rainer Gerhards (2009-11-06)

Intro

Message parsers are a feature of rsyslog 5.3.4 and above. In this article, I describe what message parsers are, what they can do and how they relate to the relevant standards. I will also describe what you can not do with time. Finally, I give some advice on implementing your own custom parser.

What are message parsers?

Well, the quick answer is that message parsers are the component of rsyslog that parses the syslog message after it is being received. Prior to rsyslog 5.3.4, message parsers where built in into the rsyslog core itself and could not be modified (other than by modifying the rsyslog code).

In 5.3.4, we changed that: message parsers are now loadable modules (just like input and output modules). That means that new message parsers can be added without modifying the rsyslog core, even without contributing something back to the project.

But that doesn't answer what a message parser really is. What does it mean to "parse a message" and, maybe more importantly, what is a message? To answer these questions correctly, we need to dig down into the relevant standards. RFC5424 specifies a layered architecture for the syslog protocol:

For us important is the distinction between the syslog transport and the upper layers. The transport layer specifies how a stream of messages is assembled at the sender side and how this stream of messages is disassembled into the individual messages at the receiver side. In networking terminology, this is called "framing". The core idea is that each message is put into a so-called "frame", which then is transmitted over the communications link.

The framing used is depending on the protocol. For example, in UDP the "frame"-equivalent is a packet that is being sent (this also means that no two messages can travel within a single UDP packet). In "plain tcp syslog", the industry standard, LF is used as a frame delimiter (which also means that no multi-line message can properly be transmitted, a "design" flaw in plain tcp syslog). In RFC5425 there is a header in front of each frame that contains the size of the message. With this framing, any message content can properly be transferred.

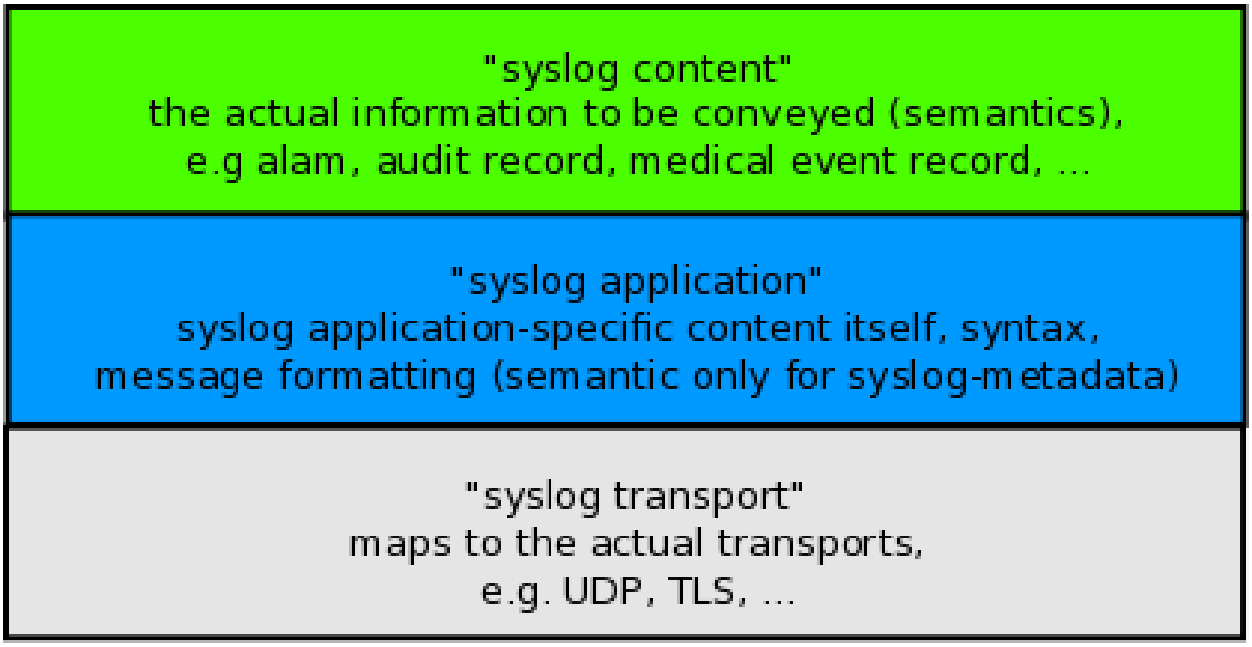


Fig. 1.3: RFC5424 syslog protocol layers

And now comes the important part: **message parsers do NOT operate at the transport layer**, they operate, as their name implies, on messages. So we can not use message parsers to change the underlying framing. For example, if a sender splits (for whatever reason) a single message into two and encapsulates these into two frames, there is no way a message parser could undo that.

A typical example may be a multi-line message: let's assume some originator has generated a message for the format "A\nB" (where \n means LF). If that message is being transmitted via plain tcp syslog, the frame delimiter is LF. So the sender will delimit the frame with LF, but otherwise send the message unmodified onto the wire (because that is how things are -unfortunately- done in plain tcp syslog...). So wire will see "A\nB\n". When this arrives at the receiver, the transport layer will undo the framing. When it sees the LF after A, it thinks it finds a valid frame delimiter (in fact, this is the correct view!). So the receive will extract one complete message A and one complete message B, not knowing that they once were both part of a large multi-line message. These two messages are then passed to the upper layers, where the message parsers receive them and extract information. However, the message parsers never know (or even have a chance to see) that A and B belonged together. Even further, in rsyslog there is no guarantee that A will be parsed before B - concurrent operations may cause the reverse order (and do so very validly).

The important lesson is: **message parsers can not be used to fix a broken framing**. You need a full protocol implementation to do that, what is the domain of input and output modules.

I have now told you what you can not do with message parsers. But what they are good for? Thankfully, broken framing is not the primary problem of the syslog world. A wealth of different formats is. Unfortunately, many real-world implementations violate the relevant standards in one way or another. That makes it often very hard to extract meaningful information from a message or to process messages from different sources by the same rules. In my article syslog parsing in rsyslog I have elaborated on all the real-world evil that you can usually see. So I won't repeat that here. But in short, the real problem is not the framing, but how to make malformed messages well-looking.

This is what message parsers permit you to do: take a (well-known) malformed message, parse it according to its semantics and generate perfectly valid internal message representations from it. So as long as messages are consistently in the same wrong format (and they usually are!), a message parser can look at that format, parse it, and make the message processable just like it were well formed in the first place. Plus, one can abuse the interface to do some other "interesting" tricks, but that would take us to far.

While this functionality may not sound exciting, it actually solves a very big issue (that you only really understand if

you have managed a system with various different syslog sources). Note that we were often able to process malformed messages in the past with the help of the property replacer and regular expressions. While this is nice, it has a performance hit. A message parser is a C code, compiled to native language, and thus typically much faster than any regular expression based method (depending, of course, on the quality of the implementation...).

How are message parsers used?

In a simplified view, rsyslog

1. first receives messages (via the input module),
2. *then parses them (at the message level!)* and
3. then processes them (operating on the internal message representation).

Message parsers are utilized in the second step (written in italics). Thus, they take the raw message (NOT frame!) received from the remote system and create the internal structure out of it that the other parts of rsyslog need in order to perform their processing. Parsing is vital, because an unparsed message can not be processed in the third stage, the actual application-level processing (like forwarding or writing to files).

Parser Chains and how they Operate

Rsyslog chains parsers together to provide flexibility. A **parser chain** contains all parsers that can potentially be used to parse a message. It is assumed that there is some way a parser can detect if the message it is being presented is supported by it. If so, the parser will tell the rsyslog engine and parse the message. The rsyslog engine now calls each parser inside the chain (in sequence!) until the first parser is able to parse the message. After one parser has been found, the message is considered parsed and no others parsers are called on that message.

Side-note: this method implies there are some "not-so-dirty" tricks available to modify the message by a parser module that declares itself as "unable to parse" but still does some message modification. This was not a primary design goal, but may be utilized, and the interface probably extended, to support generic filter modules. These would need to go to the root of the parser chain. As mentioned, the current system already supports this.

The position inside the parser chain can be thought of as a priority: parser sitting earlier in the chain take precedence over those sitting later in it. So more specific parser should go earlier in the chain. A good example of how this works is the default parser set provided by rsyslog: rsyslog.rfc5424 and rsyslog.rfc3164, each one parses according to the rfc that has named it. RFC5424 was designed to be distinguishable from RFC3164 message by the sequence "1 " immediately after the so-called PRI-part (don't worry about these words, it is sufficient if you understand there is a well-defined sequence used to identify RFC5424 messages). In contrary, RFC3164 actually permits everything as a valid message. Thus the RFC3164 parser will always parse a message, sometimes with quite unexpected outcome (there is a lot of guesswork involved in that parser, which unfortunately is unavoidable due to existing technology limits). So the default parser chain is to try the RFC5424 parser first and after it the RFC3164 parser. If we have a 5424-formatted message, that parser will identify and parse it and the rsyslog engine will stop processing. But if we receive a legacy syslog message, the RFC5424 will detect that it can not parse it, return this status to the engine which then calls the next parser inside the chain. That usually happens to be the RFC3164 parser, which will always process the message. But there could also be any other parser inside the chain, and then each one would be called unless one that is able to parse can be found.

If we reversed the parser order, RFC5424 messages would incorrectly parsed. Why? Because the RFC3164 parser will always parse every message, so if it were asked first, it would parse (and misinterpret) the 5424-formatted message, return it did so and the rsyslog engine would never call the 5424 parser. So order of sequence is very important.

What happens if no parser in the chain could parse a message? Well, then we could not obtain the in-memory representation that is needed to further process the message. In that case, rsyslog has no other choice than to discard the message. If it does so, it will emit a warning message, but only in the first 1,000 incidents. This limit is a safety measure against message-loops, which otherwise could quickly result from a parser chain misconfiguration. **If you**

do not tolerate loss of unparsable messages, you must ensure that each message can be parsed. You can easily achieve this by always using the "rsyslog-rfc3164" parser as the *last* parser inside parser chains. That may result in invalid parsing, but you will have a chance to see the invalid message (in debug mode, a warning message will be written to the debug log each time a message is dropped due to inability to parse it).

Where are parser chains used?

We now know what parser chains are and how they operate. The question is now how many parser chains can be active and how it is decided which parser chain is used on which message. This is controlled via *rsyslog's rulesets*. In short, multiple rulesets can be defined and there always exist at least one ruleset. A parser chain is bound to a specific ruleset. This is done by virtue of defining parsers via the *$RulesetParser* configuration directive (for specifics, see there). If no such directive is specified, the default parser chain is used. As of this writing, the default parser chain always consists of "rsyslog.rfc5424", "rsyslog.rfc3164", in that order. As soon as a parser is configured, the default list is cleared and the new parser is added to the end of the (initially empty) ruleset's parser chain.

The important point to know is that parser chains are defined on a per-ruleset basis.

Can I use different parser chains for different devices?

The correct answer is: generally yes, but it depends. First of all, remember that input modules (and specific listeners) may be bound to specific rulesets. As parser chains "reside" in rulesets, binding to a ruleset also binds to the parser chain that is bound to that ruleset. As a number one prerequisite, the input module must support binding to different rulesets. Not all do, but their number is growing. For example, the important imudp and imtcp input modules support that functionality. Those that do not (for example im3195) can only utilize the default ruleset and thus the parser chain defined in that ruleset.

If you do not know if the input module in question supports ruleset binding, check its documentation page. Those that support it have the required directives.

Note that it is currently under evaluation if rsyslog will support binding parser chains to specific inputs directly, without depending on the ruleset. There are some concerns that this may not be necessary but adds considerable complexity to the configuration. So this may or may not be possible in the future. In any case, if we decide to add it, input modules need to support it, so this functionality would require some time to implement.

The cookbook recipe for using different parsers for different devices is given as an actual in-depth example in the *$RulesetParser* configuration directive doc page. In short, it is accomplished by defining specific rulesets for the required parser chains, defining different listener ports for each of the devices with different format and binding these listeners to the correct ruleset (and thus parser chains). Using that approach, a variety of different message formats can be supported via a single rsyslog instance.

Which message parsers are available

As of this writing, there exist only two message parsers, one for RFC5424 format and one for legacy syslog (loosely described in RFC3164). These parsers are built-in and must not be explicitly loaded. However, message parsers can be added with relative ease by anyone knowing to code in C. Then, they can be loaded via $ModLoad just like any other loadable module. It is expected that the rsyslog project will be contributed additional message parsers over time, so that at some point there hopefully is a rich choice of them (I intend to add a browsable repository as soon as new parsers pop up).

How to write a message parser?

As a prerequisite, you need to know the exact format that the device is sending. Then, you need moderate C coding skills, and a little bit of rsyslog internals. I guess the rsyslog specific part should not be that hard, as almost all information can be gained from the existing parsers. They are rather simple in structure and can be found under the "./tools" directory. They are named pmrfc3164.c and pmrfc5424.c. You need to follow the usual loadable module guidelines. It is my expectation that writing a parser should typically not take longer than a single day, with maybe a day more to get acquainted with rsyslog. Of course, I am not sure if the number is actually right.

If you can not program or have no time to do it, Adiscon can also write a message parser for you as part of the rsyslog professional services offering.

Conclusion

Malformed syslog messages are a pain and unfortunately often seen in practice. Message parsers provide a fast and efficient solution for this problem. Different parsers can be defined for different devices, and they all convert message information into rsyslog's well-defined internal format. Message parsers were first introduced in rsyslog 5.3.4 and also offer some interesting ideas that may be explored in the future - up to full message normalization capabilities. It is strongly recommended that anyone with a heterogeneous environment take a look at message parser capabilities.

Multiple Rulesets in rsyslog

Starting with version 4.5.0 and 5.1.1, rsyslog supports multiple rulesets within a single configuration. This is especially useful for routing the reception of remote messages to a set of specific rules. Note that the input module must support binding to non-standard rulesets, so the functionality may not be available with all inputs.

In this document, I am using *imtcp*, an input module that supports binding to non-standard rulesets since rsyslog started to support them.

What is a Ruleset?

If you have worked with (r)syslog.conf, you know that it is made up of what I call rules (others tend to call them selectors, a sysklogd term). Each rule consist of a filter and one or more actions to be carried out when the filter evaluates to true. A filter may be as simple as a traditional syslog priority based filter (like "*.*" or "mail.info" or a as complex as a script-like expression. Details on that are covered in the config file documentation. After the filter come action specifiers, and an action is something that does something to a message, e.g. write it to a file or forward it to a remote logging server.

A traditional configuration file is made up of one or more of these rules. When a new message arrives, its processing starts with the first rule (in order of appearance in rsyslog.conf) and continues for each rule until either all rules have been processed or a so-called "discard" action happens, in which case processing stops and the message is thrown away (what also happens after the last rule has been processed).

The **multi-ruleset** support now permits to specify more than one such rule sequence. You can think of a traditional config file just as a single default rule set, which is automatically bound to each of the inputs. This is even what actually happens. When rsyslog.conf is processed, the config file parser looks for the directive

```
ruleset(name="rulesetname");
```

Where name is any name the user likes (but must not start with "RSYSLOG_", which is the name space reserved for rsyslog use). If it finds this directive, it begins a new rule set (if the name was not yet know) or switches to an already-existing one (if the name was known). All rules defined between this $RuleSet directive and the next one are appended to the named ruleset. Note that the reserved name "RSYSLOG_DefaultRuleset" is used to specify rsyslogd's default ruleset. You can use that name wherever you can use a ruleset name, including when binding an input to it.

Inside a ruleset, messages are processed as described above: they start with the first rule and rules are processed in the order of appearance of the configuration file until either there are no more rules or the discard action is executed. Note that with multiple rulesets no longer **all** rsyslog.conf rules are executed but **only** those that are contained within the specific ruleset.

Inputs must explicitly bind to rulesets. If they don't do, the default ruleset is bound.

This brings up the next question:

What does "To bind to a Ruleset" mean?

This term is used in the same sense as "to bind an IP address to an interface": it means that a specific input, or part of an input (like a tcp listener) will use a specific ruleset to "pass its messages to". So when a new message arrives, it will be processed via the bound ruleset. Rule from all other rulesets are irrelevant and will never be processed.

This makes multiple rulesets very handy to process local and remote message via separate means: bind the respective receivers to different rule sets, and you do not need to separate the messages by any other method.

Binding to rulesets is input-specific. For imtcp, this is done via the

```
input(type="imptcp" port="514" ruleset="rulesetname");
```

directive. Note that "name" must be the name of a ruleset that is already defined at the time the bind directive is given. There are many ways to make sure this happens, but I personally think that it is best to define all rule sets at the top of rsyslog.conf and define the inputs at the bottom. This kind of reverses the traditional recommended ordering, but seems to be a really useful and straightforward way of doing things.

Why are rulesets important for different parser configurations?

Custom message parsers, used to handle different (and potentially otherwise-invalid) message formats, can be bound to rulesets. So multiple rulesets can be a very useful way to handle devices sending messages in different malformed formats in a consistent way. Unfortunately, this is not uncommon in the syslog world. An in-depth explanation with configuration sample can be found at the *$RulesetParser* configuration directive.

Can I use a different Ruleset as the default?

This is possible by using the

```
$DefaultRuleset <name>
```

Directive. Please note, however, that this directive is actually global: that is, it does not modify the ruleset to which the next input is bound but rather provides a system-wide default rule set for those inputs that did not explicitly bind to one. As such, the directive can not be used as a work-around to bind inputs to non-default rulesets that do not support ruleset binding.

Rulesets and Queues

By default, rulesets do not have their own queue. It must be activated via the $RulesetCreateMainQueue directive, or if using rainerscript format, by specifying queue parameters on the ruleset directive, e.g.

```
ruleset(name="whatever" queue.type="fixedArray" queue. ...)
```

See *http://www.rsyslog.com/doc/master/rainerscript/queue_parameters.html* for more details.

Please note that when a ruleset uses its own queue, processing of the ruleset happens **asynchronously** to the rest of processing. As such, any modifications made to the message object (e.g. message or local variables that are set) or discarding of the message object **have no effect outside that ruleset**. So if you want to modify the message object inside the ruleset, you **cannot** define a queue for it. Most importantly, you cannot call it and expect the modified properties to be present when the call returns. Even more so, the call will most probably return before the message is even begun to be processed by the ruleset in question.

Note that in RainerScript format specifying any "queue.*" can cause the creation of a dedicated queue and as such asynchronous processing. This is because queue parameters cannot be specified without a queue. Note, though, that the actual creation is **guaranteed** only if "queue.type" is specified as above. So if you intentionally want to assing a separate queue to the ruleset, do so as shown above.

Examples

Split local and remote logging

Let's say you have a pretty standard system that logs its local messages to the usual bunch of files that are specified in the default rsyslog.conf. As an example, your rsyslog.conf might look like this:

```
# ... module loading ...
# The authpriv file has restricted access.
authpriv.*  /var/log/secure
# Log all the mail messages in one place.
mail.*      /var/log/maillog
# Log cron stuff
cron.*      /var/log/cron
# Everybody gets emergency messages
*.emerg     *
... more ...
```

Now, you want to add receive messages from a remote system and log these to a special file, but you do not want to have these messages written to the files specified above. The traditional approach is to add a rule in front of all others that filters on the message, processes it and then discards it:

```
# ... module loading ...
# process remote messages
if $fromhost-ip == '192.168.152.137' then {
        action(type="omfile" file="/var/log/remotefile02")
        stop
    }


# only messages not from 192.0.21 make it past this point

# The authpriv file has restricted access.
authpriv.*                              /var/log/secure
# Log all the mail messages in one place.
mail.*                                  /var/log/maillog
# Log cron stuff
cron.*                                  /var/log/cron
# Everybody gets emergency messages
*.emerg                                 *
... more ...
```

Note that "stop" is the discard action!. Also note that we assume that 192.0.2.1 is the sole remote sender (to keep it simple).

With multiple rulesets, we can simply define a dedicated ruleset for the remote reception case and bind it to the receiver. This may be written as follows:

```
# ... module loading ...
# process remote messages
# define new ruleset and add rules to it:
ruleset(name="remote"){
    action(type="omfile" file="/var/log/remotefile")
}
# only messages not from 192.0.21 make it past this point

# bind ruleset to tcp listener and activate it:
input(type="imptcp" port="10514" ruleset="remote")
```

Split local and remote logging for three different ports

This example is almost like the first one, but it extends it a little bit. While it is very similar, I hope it is different enough to provide a useful example why you may want to have more than two rulesets.

Again, we would like to use the "regular" log files for local logging, only. But this time we set up three syslog/tcp listeners, each one listening to a different port (in this example 10514, 10515, and 10516). Logs received from these receivers shall go into different files. Also, logs received from 10516 (and only from that port!) with "mail.*" priority, shall be written into a specific file and **not** be written to 10516's general log file.

This is the config:

```
# ... module loading ...
# process remote messages

ruleset(name="remote10514"){
    action(type="omfile" file="/var/log/remote10514")
}

ruleset(name="remote10515"){
    action(type="omfile" file="/var/log/remote10515")
}

ruleset(name="remote10516"){
    if prifilt("mail.*") then {
        /var/log/mail10516
        stop
        # note that the stop-command will prevent this message from
        # being written to the remote10516 file - as usual...
    }
    /var/log/remote10516
}


# and now define listeners bound to the relevant ruleset
input(type="imptcp" port="10514" ruleset="remote10514")
input(type="imptcp" port="10515" ruleset="remote10515")
input(type="imptcp" port="10516" ruleset="remote10516")
```

Performance

Fewer Filters

No rule processing can be faster than not processing a rule at all. As such, it is useful for a high performance system to identify disjunct actions and try to split these off to different rule sets. In the example section, we had a case where three different tcp listeners need to write to three different files. This is a perfect example of where multiple rule sets are easier to use and offer more performance. The performance is better simply because there is no need to check the reception service - instead messages are automatically pushed to the right rule set and can be processed by very simple rules (maybe even with "*.*"-filters, the fastest ones available).

Partitioning of Input Data

Starting with rsyslog 5.3.4, rulesets permit higher concurrency. They offer the ability to run on their own "main" queue. What that means is that a own queue is associated with a specific rule set. That means that inputs bound to that ruleset do no longer need to compete with each other when they enqueue a data element into the queue. Instead, enqueue operations can be completed in parallel.

An example: let us assume we have three TCP listeners. Without rulesets, each of them needs to insert messages into the main message queue. So if each of them wants to submit a newly arrived message into the queue at the same time, only one can do so while the others need to wait. With multiple rulesets, its own queue can be created for each ruleset. If now each listener is bound to its own ruleset, concurrent message submission is possible. On a machine with a sufficiently large number of cores, this can result in dramatic performance improvement.

It is highly advised that high-performance systems define a dedicated ruleset, with a dedicated queue for each of the inputs.

By default, rulesets do **not** have their own queue. It must be activated via the *$RulesetCreateMainQueue* directive.

See Also

Legacy Format Samples for Multiple Rulesets

This chapter complements rsyslog's documentation of *rulesets*. While the base document focusses on RainerScript format, it does not provide samples in legacy format. These are included in this document.

Important: do **not** use legacy ruleset defintions for new configurations. Especially with rulesets, legacy format is extremely hard to get right. The information in this page is included in order to help you understand already existing configurations using the ruleset feature. We even recommend to convert any such configs to RainerScript format because of its increased robustness and simplicity.

Legacy ruleset support was available starting with version 4.5.0 and 5.1.1.

Split local and remote logging

Let's say you have a pretty standard system that logs its local messages to the usual bunch of files that are specified in the default rsyslog.conf. As an example, your rsyslog.conf might look like this:

```
# ... module loading ...
# The authpriv file has restricted access.
authpriv.*  /var/log/secure
# Log all the mail messages in one place.
mail.*      /var/log/maillog
```

```
# Log cron stuff
cron.*      /var/log/cron
# Everybody gets emergency messages
*.emerg     *
... more ...
```

Now, you want to add receive messages from a remote system and log these to a special file, but you do not want to have these messages written to the files specified above. The traditional approach is to add a rule in front of all others that filters on the message, processes it and then discards it:

```
# ... module loading ...
# process remote messages
:fromhost-ip, isequal, "192.0.2.1"    /var/log/remotefile
& ~
# only messages not from 192.0.21 make it past this point

# The authpriv file has restricted access.
authpriv.*                            /var/log/secure
# Log all the mail messages in one place.
mail.*                                /var/log/maillog
# Log cron stuff
cron.*                                /var/log/cron
# Everybody gets emergency messages
*.emerg                               *
... more ...
```

Note the tilde character, which is the discard action!. Also note that we assume that 192.0.2.1 is the sole remote sender (to keep it simple).

With multiple rulesets, we can simply define a dedicated ruleset for the remote reception case and bind it to the receiver. This may be written as follows:

```
# ... module loading ...
# process remote messages
# define new ruleset and add rules to it:
$RuleSet remote
*.*           /var/log/remotefile
# only messages not from 192.0.21 make it past this point

# bind ruleset to tcp listener
$InputTCPServerBindRuleset remote
# and activate it:
$InputTCPServerRun 10514

# switch back to the default ruleset:
$RuleSet RSYSLOG_DefaultRuleset
# The authpriv file has restricted access.
authpriv.*    /var/log/secure
# Log all the mail messages in one place.
mail.*        /var/log/maillog
# Log cron stuff
cron.*        /var/log/cron
# Everybody gets emergency messages
*.emerg       *
... more ...
```

Here, we need to switch back to the default ruleset after we have defined our custom one. This is why I recommend a different ordering, which I find more intuitive. The sample below has it, and it leads to the same results:

```
# ... module loading ...
# at first, this is a copy of the unmodified rsyslog.conf
# The authpriv file has restricted access.
authpriv.*    /var/log/secure
# Log all the mail messages in one place.
mail.*        /var/log/maillog
# Log cron stuff
cron.*        /var/log/cron
# Everybody gets emergency messages
*.emerg       *
... more ...
# end of the "regular" rsyslog.conf. Now come the new definitions:

# process remote messages
# define new ruleset and add rules to it:
$RuleSet remote
*.*           /var/log/remotefile

# bind ruleset to tcp listener
$InputTCPServerBindRuleset remote
# and activate it:
$InputTCPServerRun 10514
```

Here, we do not switch back to the default ruleset, because this is not needed as it is completely defined when we begin the "remote" ruleset.

Now look at the examples and compare them to the single-ruleset solution. You will notice that we do **not** need a real filter in the multi-ruleset case: we can simply use "*.*" as all messages now means all messages that are being processed by this rule set and all of them come in via the TCP receiver! This is what makes using multiple rulesets so much easier.

Split local and remote logging for three different ports

This example is almost like the first one, but it extends it a little bit. While it is very similar, I hope it is different enough to provide a useful example why you may want to have more than two rulesets.

Again, we would like to use the "regular" log files for local logging, only. But this time we set up three syslog/tcp listeners, each one listening to a different port (in this example 10514, 10515, and 10516). Logs received from these receivers shall go into different files. Also, logs received from 10516 (and only from that port!) with "mail.*" priority, shall be written into a specif file and **not** be written to 10516's general log file.

This is the config:

```
# ... module loading ...
# at first, this is a copy of the unmodified rsyslog.conf
# The authpriv file has restricted access.
authpriv.* /var/log/secure
# Log all the mail messages in one place.
mail.*  /var/log/maillog
# Log cron stuff
cron.*  /var/log/cron
# Everybody gets emergency messages
*.emerg       *
... more ...
# end of the "regular" rsyslog.conf. Now come the new definitions:

# process remote messages
```

```
#define rulesets first
$RuleSet remote10514
*.*     /var/log/remote10514

$RuleSet remote10515
*.*     /var/log/remote10515

$RuleSet remote10516
mail.*  /var/log/mail10516
&       ~
# note that the discard-action will prevent this messag from
# being written to the remote10516 file - as usual...
*.*     /var/log/remote10516

# and now define listeners bound to the relevant ruleset
$InputTCPServerBindRuleset remote10514
$InputTCPServerRun 10514

$InputTCPServerBindRuleset remote10515
$InputTCPServerRun 10515

$InputTCPServerBindRuleset remote10516
$InputTCPServerRun 10516
```

Note that the "mail.*" rule inside the "remote10516" ruleset does not affect processing inside any other rule set, including the default rule set.

This documentation is part of the rsyslog project. Copyright © 2009-2014 by Rainer Gerhards and Adiscon. Released under the GNU GPL version 3 or higher.

NetStream Drivers

Network stream drivers are a layer between various parts of rsyslogd (e.g. the imtcp module) and the transport layer. They provide sequenced delivery, authentication and confidentiality to the upper layers. Drivers implement different capabilities.

Users need to know about netstream drivers because they need to configure the proper driver, and proper driver properties, to achieve desired results (e.g. a *Encrypting Syslog Traffic with TLS (SSL) [short version]*).

Current Network Stream Drivers

ptcp Network Stream Driver

This network stream driver implement a plain tcp transport without security properties.

Supported Driver Modes

- 0 - unencrypted trasmission

Supported Authentication Modes

- "anon" - no authentication

gtls Network Stream Driver

This network stream driver implements a TLS protected transport via the GnuTLS library.

Available since: 3.19.0 (suggested minimum 3.19.8 and above)

Supported Driver Modes

- 0 - unencrypted trasmission (just like ptcp driver)
- 1 - TLS-protected operation

Note: mode 0 does not provide any benefit over the ptcp driver. This mode exists for technical reasons, but should not be used. It may be removed in the future.

Supported Authentication Modes

- anon - anonymous authentication as described in IETF's draft-ietf-syslog-transport-tls-12 Internet draft
- x509/fingerprint - certificate fingerprint authentication as described in IETF's draft-ietf-syslog-transport-tls-12 Internet draft
- x509/certvalid - certificate validation only
- x509/name - certificate validation and subject name authentication as described in IETF's draft-ietf-syslog-transport-tls-12 Internet draft

Note: "anon" does not permit to authenticate the remote peer. As such, this mode is vulnerable to man in the middle attacks as well as unauthorized access. It is recommended NOT to use this mode.

x509/certvalid is a nonstandard mode. It validates the remote peers certificate, but does not check the subject name. This is weak authentication that may be useful in scenarios where multiple devices are deployed and it is sufficient proof of authenticity when their certificates are signed by the CA the server trusts. This is better than anon authentication, but still not recommended. **Known Problems**

Even in x509/fingerprint mode, both the client and server certificate currently must be signed by the same root CA. This is an artifact of the underlying GnuTLS library and the way we use it. It is expected that we can resolve this issue in the future.

Example Use Cases

Receiving massive amounts of messages with high performance

Use Case

You are receiving syslog messages via UDP and or TCP at a very high data rate. You want to tune the system so that it can process as many messages as possible. All messages shall be written to a single output file.

Sample Configuration

```
# load required modules
module(load="imudp" threads="2"
       timeRequery="8" batchSize="128")
module(load="imptcp" threads="3")

# listeners
# repeat blocks if more listeners are needed
```

```
# alternatively, use array syntax:
# port=["514","515",...]
input(type="imudp" port="514"
      ruleset="writeRemoteData")
input(type="imptcp" port="10514"
      ruleset="writeRemoteData")

# now define our ruleset, which also includes
# threading and queue parameters.
ruleset(name="writeRemoteData"
        queue.type="fixedArray"
        queue.size="250000"
        queue.dequeueBatchSize="4096"
        queue.workerThreads="4"
        queue.workerThreadMinimumMessages="60000"
       ) {
    action(type="omfile" file="/var/log/remote.log"
           ioBufferSize="64k" flushOnTXEnd="off"
           asyncWriting="on")
}
```

Notes on the suggested config

It is highly suggested to use a recent enough Linux kernel that supports the **recvmmsg()** system call. This system call improves UDP reception speed and decreases overall system CPU utilization.

We use the **imptcp** module for tcp input, as it uses more optimal results. Note, however, that it is only available on Linux and does currently *not* support TLS. If **imptcp** cannot be used, use **imtcp** instead (this will be a bit slower).

When writing to the output file, we use buffered mode. This means that full buffers are written, but during processing file lines are not written until the buffer is full (and thus may be delayed) and also incomplete lines are written (at buffer boundary). When the file is closed (rsyslogd stop or HUP), the buffer is completely flushed. As this is a high-traffic use case, we assume that buffered mode does not cause any concerns.

Suggested User Performance Testing

Each environment is a bit different. Depending on circumstances, the **imudp** module parameters may not be optimal. In order to obtain best performance, it is suggested to measure performance level with two to four threads and somewhat lower and higher batchSize. Note that these parameters affect each other. The values given in the config above should usually work well in *high-traffic* environments. They are sub-optimal for low to medium traffic environments.

See Also

imptcp, imtcp, imudp, ruleset()

Copyright

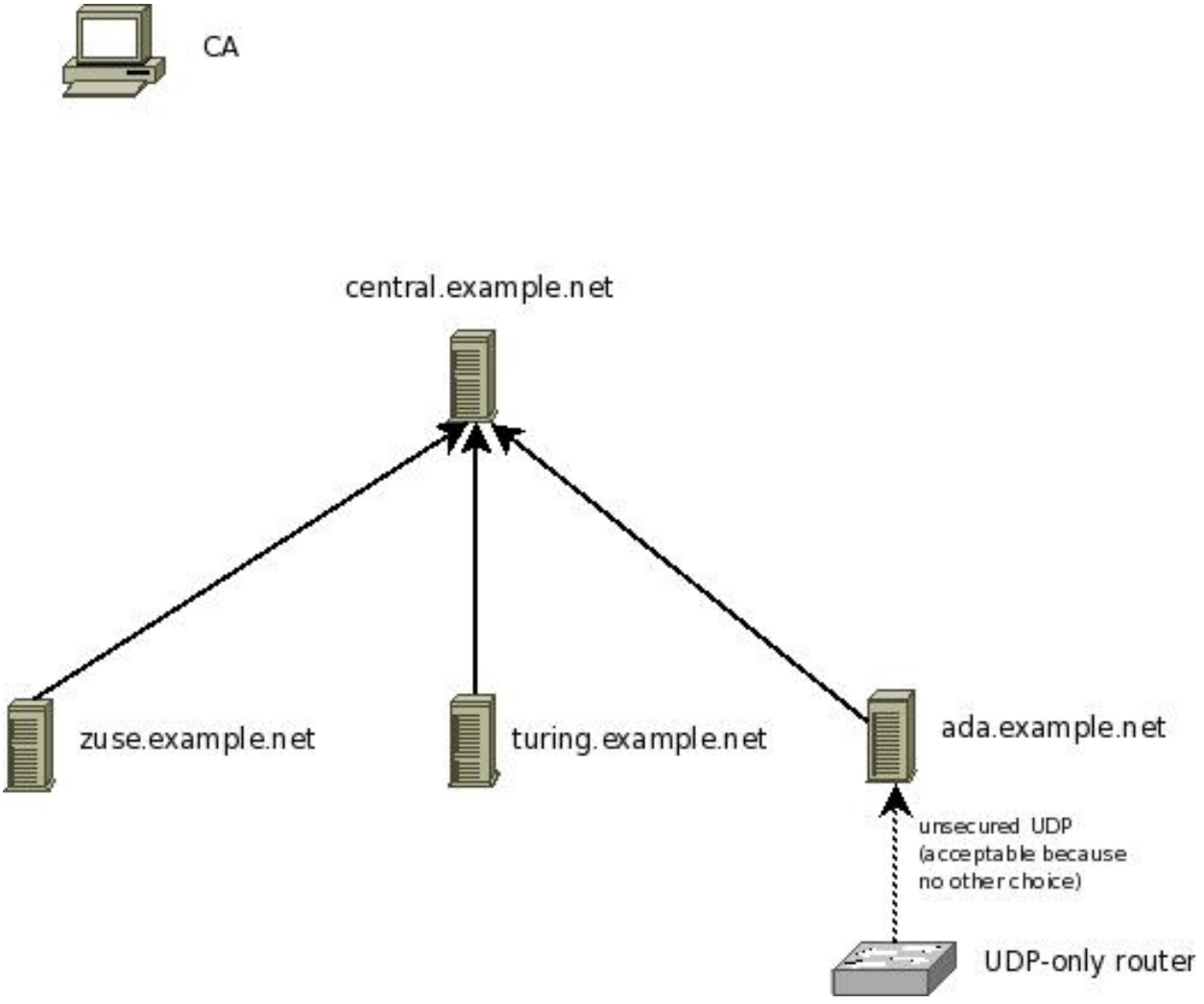
Tutorials

Encrypting Syslog Traffic with TLS (SSL)

Written by Rainer Gerhards (2008-07-03)

Sample Use Case: Single Central Log Server

We have a quite simple use case. There is one central syslog server, named central.example.net. These server is being reported to by two Linux machines with name zuse.example.net and turing.example.net. Also, there is a third client - ada.example.net - which send both its own messages to the central server but also forwards messages receive from an UDP-only capable router. We have decided to use ada.example.net because it is in the same local network segment as the router and so we enjoy TLS' security benefits for forwarding the router messages inside the corporate network. All systems (except the router) use rsyslog as the syslog software.



Please note that the CA must not necessarily be connected to the rest of the network. Actually, it may be considered a security plus if it is not. If the CA is reachable via the regular network, it should be sufficiently secured (firewal rules et al). Keep in mind that if the CA's security is breached, your overall system security is breached.

In case the CA is compromised, you need to regenerate the CA's certificate as well as all individual machines certificates.

Setting up the CA

The first step is to set up a certificate authority (CA). It must be maintained by a trustworthy person (or group) and approves the indentities of all machines. It does so by issuing their certificates. In a small setup, the administrator can provide the CA function. What is important is the the CA's private key is well-protocted and machine certificates are only issued if it is know they are valid (in a single-admin case that means the admin should not issue certificates to anyone else except himself).

The CA creates a so-called self-signed certificate. That is, it approves its own authenticy. This sounds useless, but the key point to understand is that every machine will be provided a copy of the CA's certificate. Accepting this certificate is a matter of trust. So by configuring the CA certificate, the administrator tells rsyslog which certificates to trust. This is the root of all trust under this model. That is why the CA's private key is so important - everyone getting hold of it is trusted by our rsyslog instances.

To create a self-signed certificate, use the following commands with GnuTLS (which is currently the only supported TLS library, what may change in the future). Please note that GnuTLS' tools are not installed by default on many platforms. Also, the tools do not necessarily come with the GnuTLS core package. If you do not have certtool on your system, check if there is package for the GnuTLS tools available (under Fedora, for example, this is named gnutls-utils-<version> and it is NOT installed by default).

1. generate the private key:

   ```
   certtool --generate-privkey --outfile ca-key.pem
   ```

   This takes a short while. Be sure to do some work on your workstation, it waits for radom input. Switching between windows is sufficient ;)

2. now create the (self-signed) CA certificate itself:

   ```
   certtool --generate-self-signed --load-privkey ca-key.pem --outfile ca.pem
   ```

   This generates the CA certificate. This command queries you for a number of things. Use appropriate responses. When it comes to certificate validity, keep in mind that you need to recreate all certificates when this one expires. So it may be a good idea to use a long period, eg. 3650 days (roughly 10 years). You need to specify that the certificates belongs to an authority. The certificate is used to sign other certificates.

Sample Screen Session

Text in red is user input. Please note that for some questions, there is no user input given. This means the default was accepted by simply pressing the enter key.

```
[root@rgf9dev sample]# certtool --generate-privkey --outfile ca-key.pem --bits 2048
Generating a 2048 bit RSA private key...
[root@rgf9dev sample]# certtool --generate-self-signed --load-privkey ca-key.pem --
↪outfile ca.pem
Generating a self signed certificate...
Please enter the details of the certificate's distinguished name. Just press enter to␣
↪ignore a field.
Country name (2 chars): US
Organization name: SomeOrg
Organizational unit name: SomeOU
Locality name: Somewhere
State or province name: CA
Common name: someName (not necessarily DNS!)
UID:
This field should not be used in new certificates.
E-mail:
Enter the certificate's serial number (decimal):


Activation/Expiration time.
The certificate will expire in (days): 3650


Extensions.
Does the certificate belong to an authority? (Y/N): y
Path length constraint (decimal, -1 for no constraint):
Is this a TLS web client certificate? (Y/N):
Is this also a TLS web server certificate? (Y/N):
Enter the e-mail of the subject of the certificate: someone@example.net
Will the certificate be used to sign other certificates? (Y/N): y
Will the certificate be used to sign CRLs? (Y/N):
Will the certificate be used to sign code? (Y/N):
Will the certificate be used to sign OCSP requests? (Y/N):
Will the certificate be used for time stamping? (Y/N):
Enter the URI of the CRL distribution point:
X.509 Certificate Information:
    Version: 3
    Serial Number (hex): 485a365e
    Validity:
        Not Before: Thu Jun 19 10:35:12 UTC 2008
        Not After: Sun Jun 17 10:35:25 UTC 2018
    Subject: C=US,O=SomeOrg,OU=SomeOU,L=Somewhere,ST=CA,CN=someName (not necessarily␣
↪DNS!)
    Subject Public Key Algorithm: RSA
        Modulus (bits 2048):
            d9:9c:82:46:24:7f:34:8f:60:cf:05:77:71:82:61:66
            05:13:28:06:7a:70:41:bf:32:85:12:5c:25:a7:1a:5a
            28:11:02:1a:78:c1:da:34:ee:b4:7e:12:9b:81:24:70
            ff:e4:89:88:ca:05:30:0a:3f:d7:58:0b:38:24:a9:b7
            2e:a2:b6:8a:1d:60:53:2f:ec:e9:38:36:3b:9b:77:93
            5d:64:76:31:07:30:a5:31:0c:e2:ec:e3:8d:5d:13:01
            11:3d:0b:5e:3c:4a:32:d8:f3:b3:56:22:32:cb:de:7d
            64:9a:2b:91:d9:f0:0b:82:c1:29:d4:15:2c:41:0b:97
```

```
        Exponent:
            01:00:01
    Extensions:
        Basic Constraints (critical):
            Certificate Authority (CA): TRUE
        Subject Alternative Name (not critical):
            RFC822name: someone@example.net
        Key Usage (critical):
            Certificate signing.
        Subject Key Identifier (not critical):
            fbfe968d10a73ae5b70d7b434886c8f872997b89
Other Information:
    Public Key Id:
        fbfe968d10a73ae5b70d7b434886c8f872997b89

Is the above information ok? (Y/N): y


Signing certificate...
[root@rgf9dev sample]# chmod 400 ca-key.pem
[root@rgf9dev sample]# ls -l
total 8
-r-------- 1 root root  887 2008-06-19 12:33 ca-key.pem
-rw-r--r-- 1 root root 1029 2008-06-19 12:36 ca.pem
[root@rgf9dev sample]#
```

Be sure to safeguard ca-key.pem! Nobody except the CA itself needs to have it. If some third party obtains it, you security is broken!

Generating the machine certificate

In this step, we generate certificates for each of the machines. Please note that both clients and servers need certificates. The certificate identifies each machine to the remote peer. The DNSName specified inside the certificate can

be specified inside the $<object>PermittedPeer config statements.

For now, we assume that a single person (or group) is responsible for the whole rsyslog system and thus it is OK if that single person is in posession of all machine's private keys. This simplification permits us to use a somewhat less complicated way of generating the machine certificates. So, we generate both the private and public key on the CA (which is NOT a server!) and then copy them over to the respective machines.

If the roles of machine and CA administrators are split, the private key must be generated by the machine administrator. This is done via a certificate request. This request is then sent to the CA admin, which in turn generates the certificate (containing the public key). The CA admin then sends back the certificate to the machine admin, who installs it. That way, the CA admin never get's hold of the machine's private key. Instructions for this mode will be given in a later revision of this document.

In any case, it is vital that the machine's private key is protected. Anybody able to obtain that private key can imporsonate as the machine to which it belongs, thus breaching your security.

Sample Screen Session

Text in red is user input. Please note that for some questions, there is no user input given. This means the default was accepted by simply pressing the enter key.

Please note: you need to substitute the names specified below with values that match your environment. Most importantly, machine.example.net must be replaced by the actual name of the machine that will be using this certificate. For example, if you generate a certificate for a machine named "server.example.com", you need to use that name. If you generate a certificate for "client.example.com", you need to use this name. Make sure that each machine certificate has a unique name. If not, you can not apply proper access control.

```
[root@rgf9dev sample]# certtool --generate-privkey --outfile key.pem --sec-param 2048
Generating a 2048 bit RSA private key...
[root@rgf9dev sample]# certtool --generate-request --load-privkey key.pem --outfile␣
↪request.pem
Generating a PKCS #10 certificate request...
Country name (2 chars): US
Organization name: SomeOrg
Organizational unit name: SomeOU
Locality name: Somewhere
State or province name: CA
Common name: machine.example.net
UID:
Enter a dnsName of the subject of the certificate:
Enter the IP address of the subject of the certificate:
Enter the e-mail of the subject of the certificate:
Enter a challange password:
Does the certificate belong to an authority? (y/N): n
Will the certificate be used for signing (DHE and RSA-EXPORT ciphersuites)? (y/N):
Will the certificate be used for encryption (RSA ciphersuites)? (y/N):
Is this a TLS web client certificate? (y/N): y
Is this also a TLS web server certificate? (y/N): y
[root@rgf9dev sample]# certtool --generate-certificate --load-request request.pem --
↪outfile cert.pem --load-ca-certificate ca.pem --load-ca-privkey ca-key.pem
Generating a signed certificate...
Enter the certificate's serial number (decimal):


Activation/Expiration time.
The certificate will expire in (days): 1000


Extensions.
Do you want to honour the extensions from the request? (y/N):
Does the certificate belong to an authority? (Y/N): n
Will the certificate be used for IPsec IKE operations? (y/N):
Is this a TLS web client certificate? (Y/N): y
Is this also a TLS web server certificate? (Y/N): y
Enter the dnsName of the subject of the certificate: machine.example.net {This is the␣
↪name of the machine that will use the certificate}
Enter the IP address of the subject of certificate:
Will the certificate be used for signing (DHE and RSA-EXPORT ciphersuites)? (Y/N):
Will the certificate be used for encryption (RSA ciphersuites)? (Y/N):
X.509 Certificate Information:
    Version: 3
    Serial Number (hex): 485a3819
    Validity:
        Not Before: Thu Jun 19 10:42:54 UTC 2008
        Not After: Wed Mar 16 10:42:57 UTC 2011
    Subject: C=US,O=SomeOrg,OU=SomeOU,L=Somewhere,ST=CA,CN=machine.example.net
    Subject Public Key Algorithm: RSA
        Modulus (bits 2048):
            b2:4e:5b:a9:48:1e:ff:2e:73:a1:33:ee:d8:a2:af:ae
```

```
            2f:23:76:91:b8:39:94:00:23:f2:6f:25:ad:c9:6a:ab
            2d:e6:f3:62:d8:3e:6e:8a:d6:1e:3f:72:e5:d8:b9:e0
            d0:79:c2:94:21:65:0b:10:53:66:b0:36:a6:a7:cd:46
            1e:2c:6a:9b:79:c6:ee:c6:e2:ed:b0:a9:59:e2:49:da
            c7:e3:f0:1c:e0:53:98:87:0d:d5:28:db:a4:82:36:ed
            3a:1e:d1:5c:07:13:95:5d:b3:28:05:17:2a:2b:b6:8e
            8e:78:d2:cf:ac:87:13:15:fc:17:43:6b:15:c3:7d:b9
        Exponent:
            01:00:01
    Extensions:
        Basic Constraints (critical):
            Certificate Authority (CA): FALSE
        Key Purpose (not critical):
            TLS WWW Client.
            TLS WWW Server.
        Subject Alternative Name (not critical):
            DNSname: machine.example.net
        Subject Key Identifier (not critical):
            0ce1c3dbd19d31fa035b07afe2e0ef22d90b28ac
        Authority Key Identifier (not critical):
            fbfe968d10a73ae5b70d7b434886c8f872997b89
Other Information:
    Public Key Id:
        0ce1c3dbd19d31fa035b07afe2e0ef22d90b28ac

Is the above information ok? (Y/N): y


Signing certificate...
[root@rgf9dev sample]# rm -f request.pem
[root@rgf9dev sample]# ls -l
total 16
-r-------- 1 root root  887 2008-06-19 12:33 ca-key.pem
-rw-r--r-- 1 root root 1029 2008-06-19 12:36 ca.pem
-rw-r--r-- 1 root root 1074 2008-06-19 12:43 cert.pem
-rw-r--r-- 1 root root  887 2008-06-19 12:40 key.pem
[root@rgf9dev sample]# # it may be a good idea to rename the files to indicate where␣
↪they belong to
[root@rgf9dev sample]# mv cert.pem machine-cert.pem
[root@rgf9dev sample]# mv key.pem machine-key.pem
[root@rgf9dev sample]#
```

Distributing Files

Provide the machine with:

- a copy of ca.pem
- cert.pem
- key.pem

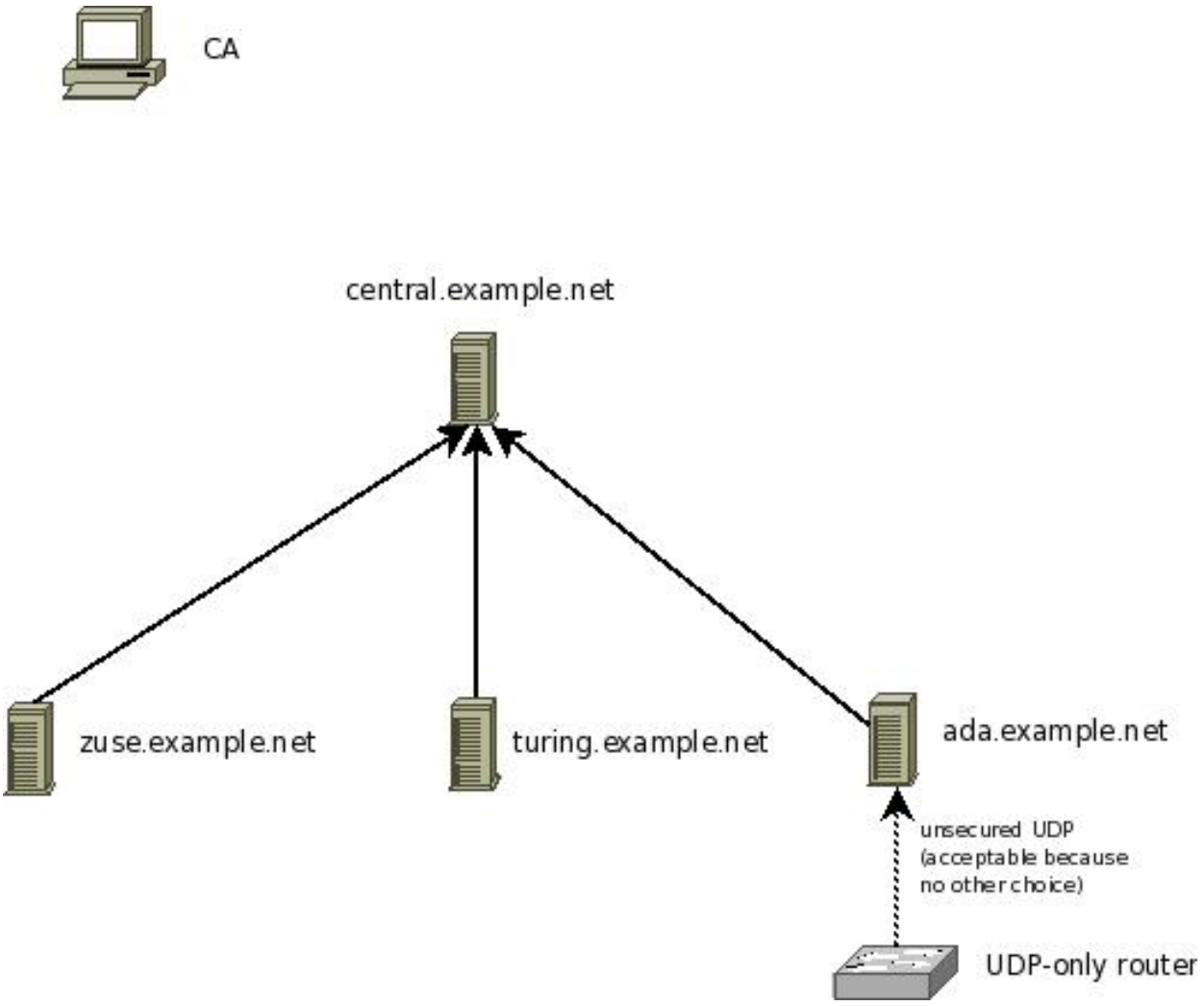
This is how the relevant part of rsyslog.conf looks on the target machine:

```
$DefaultNetstreamDriverCAFile /home/rger/proj/rsyslog/sample/ca.pem
$DefaultNetstreamDriverCertFile /home/rger/proj/rsyslog/sample/machine-cert.pem
$DefaultNetstreamDriverKeyFile /home/rger/proj/rsyslog/sample/machine-key.pem
```

Never provide anyone with ca-key.pem! Also, make sure nobody but the machine in question gets hold of key.pem.

Setting up the Central Server

In this step, we configure the central server. We assume it accepts messages only via TLS protected plain tcp based syslog from those peers that are explicitely permitted to send to it. The picture below show our configuration. This step configures the server central.example.net.



Important: *Keep in mind that the order of configuration directives is very important in rsyslog. As such, the samples given below do only work if the given order is preserved. Re-ordering the directives can break configurations and has broken them in practice. If you intend to re-order them, please be sure that you fully understand how the configuration language works and, most importantly, which statements form a block together. Please also note that we understand the the current configuration file format is ugly. However, there has been more important work in the way of enhancing it. If you would like to contribute some time to improve the config file language, please let us know. Any help is appreciated (be it doc or coding work!).*

Steps to do:

- make sure you have a functional CA (Setting up the CA)
- generate a machine certificate for central.example.net (follow instructions in Generating Machine Certificates)
- make sure you copy over ca.pem, machine-key.pem ad machine-cert.pem to the central server. Ensure that no user except root can access them (**even read permissions are really bad**).
- configure the server so that it accepts messages from all machines in the example.net domain that have certificates from your CA. Alternatively, you may also precisely define from which machine names messages are accepted. See sample rsyslog.conf below.

In this setup, we use wildcards to ease adding new systems. We permit the server to accept messages from systems whos names match *.example.net.

```
$InputTCPServerStreamDriverPermittedPeer *.example.net
```

This will match zuse.example.net and turing.example.net, but NOT pascal.otherdepartment.example.net. If the later would be desired, you can (and need) to include additional permitted peer config statments:

```
$InputTCPServerStreamDriverPermittedPeer *.example.net
$InputTCPServerStreamDriverPermittedPeer *.otherdepartment.example.net
$InputTCPServerStreamDriverPermittedPeer *.example.com
```

As can be seen with example.com, the different permitted peers need NOT to be in a single domain tree. Also, individual machines can be configured. For example, if only zuse, turing and ada should be able to talk to the server, you can achive this by:

```
$InputTCPServerStreamDriverPermittedPeer zuse.example.net
$InputTCPServerStreamDriverPermittedPeer turing.example.net
$InputTCPServerStreamDriverPermittedPeer ada.example.net
```

As an extension to the (upcoming) IETF syslog/tls standard, you can specify some text together with a domain component wildcard. So "*server.example.net", "server*.example.net" are valid permitted peers. However "server*Fix.example.net" is NOT a valid wildcard. The IETF standard permits no text along the wildcards.

The reason we use wildcards in the default setup is that it makes it easy to add systems without the need to change the central server's configuration. It is important to understand that the central server will accept names **only** (no exception) if the client certificate was signed by the CA we set up. So if someone tries to create a malicious certificate with a name "zuse.example.net", the server will **not** accept it. So a wildcard is safe as long as you ensure CA security is not breached. Actually, you authorize a client by issuing the certificate to it.

At this point, please be reminded once again that your security needs may be quite different from what we assume in this tutorial. Evaluate your options based on your security needs.

Sample syslog.conf

Keep in mind that this rsyslog.conf accepts messages via TCP, only. The only other source accepted is messages from the server itself.

```
$ModLoad imuxsock # local messages
$ModLoad imtcp # TCP listener

# make gtls driver the default
$DefaultNetstreamDriver gtls

# certificate files
```

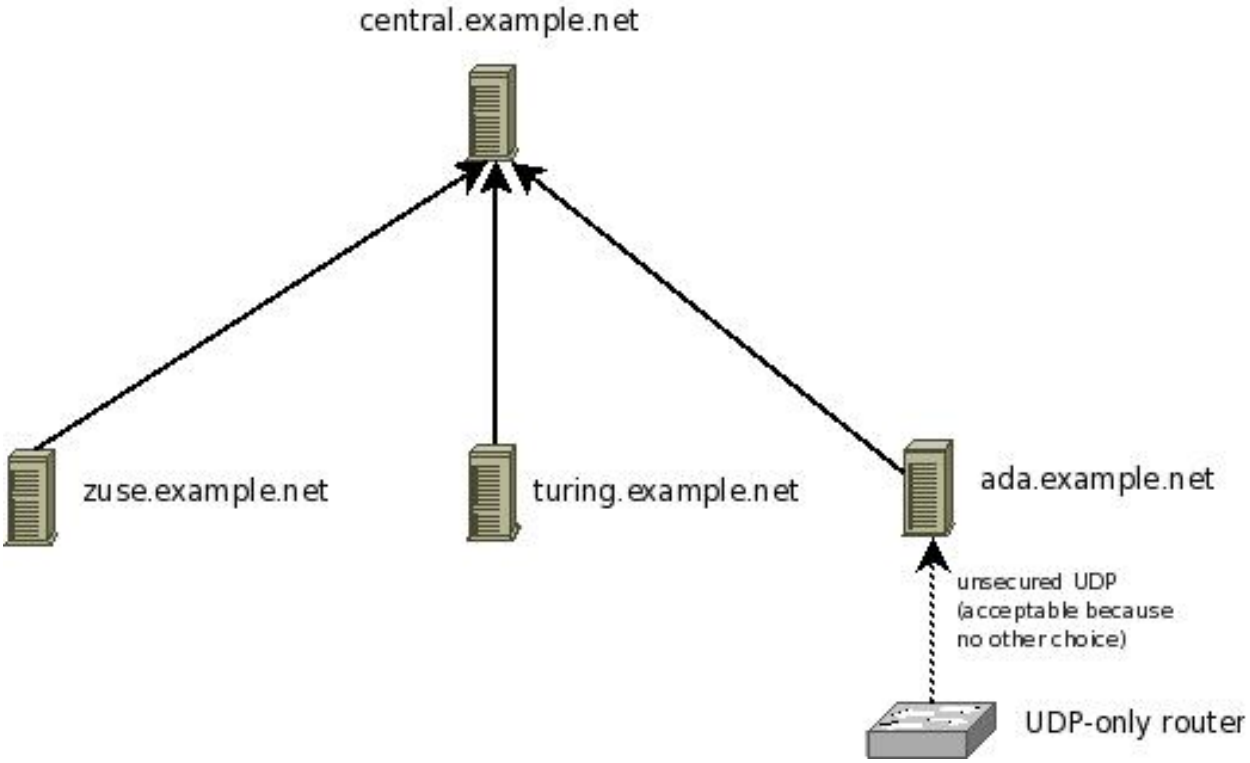
```
$DefaultNetstreamDriverCAFile /rsyslog/protected/ca.pem
$DefaultNetstreamDriverCertFile /rsyslog/protected/machine-cert.pem
$DefaultNetstreamDriverKeyFile /rsyslog/protected/machine-key.pem

$InputTCPServerStreamDriverAuthMode x509/name
$InputTCPServerStreamDriverPermittedPeer *.example.net
$InputTCPServerStreamDriverMode 1 # run driver in TLS-only mode
$InputTCPServerRun 10514 # start up listener at port 10514
```

Be sure to safeguard at least the private key (machine-key.pem)! If some third party obtains it, you security is broken!

Setting up a client

In this step, we configure a client machine. We from our scenario, we use zuse.example.net. You need to do the same steps for all other clients, too (in the example, that meanst turng.example.net). The client check's the server's identity and talks to it only if it is the expected server. This is a very important step. Without it, you would not detect man-in-the-middle attacks or simple malicious servers who try to get hold of your valuable log data.



Steps to do:

- make sure you have a functional CA (Setting up the CA)
- generate a machine certificate for zuse.example.net (follow instructions in Generating Machine Certificates)
- make sure you copy over ca.pem, machine-key.pem ad machine-cert.pem to the client. Ensure that no user except root can access them (**even read permissions are really bad**).
- configure the client so that it checks the server identity and sends messages only if the server identity is known. Please note that you have the same options as when configuring a server. However, we now use a single name only, because there is only one central server. No using wildcards make sure that we will exclusively talk to that server (otherwise, a compromised client may take over its role). If you load-balance to different server identies, you obviously need to allow all of them. It still is suggested to use explcit names.

At this point, please be reminded once again that your security needs may be quite different from what we assume in this tutorial. Evaluate your options based on your security needs.

Sample syslog.conf

Keep in mind that this rsyslog.conf sends messages via TCP, only. Also, we do not show any rules to write local files. Feel free to add them.

```
# make gtls driver the default
$DefaultNetstreamDriver gtls

# certificate files
$DefaultNetstreamDriverCAFile /rsyslog/protected/ca.pem
$DefaultNetstreamDriverCertFile /rsyslog/protected/machine-cert.pem
$DefaultNetstreamDriverKeyFile /rsyslog/protected/machine-key.pem

$ActionSendStreamDriverAuthMode x509/name
$ActionSendStreamDriverPermittedPeer central.example.net
$ActionSendStreamDriverMode 1 # run driver in TLS-only mode
*.* @@central.example.net:10514 # forward everything to remote server
```

Note: the example above forwards every message to the remote server. Of course, you can use the normal filters to restrict the set of information that is sent. Depending on your message volume and needs, this may be a smart thing to do.

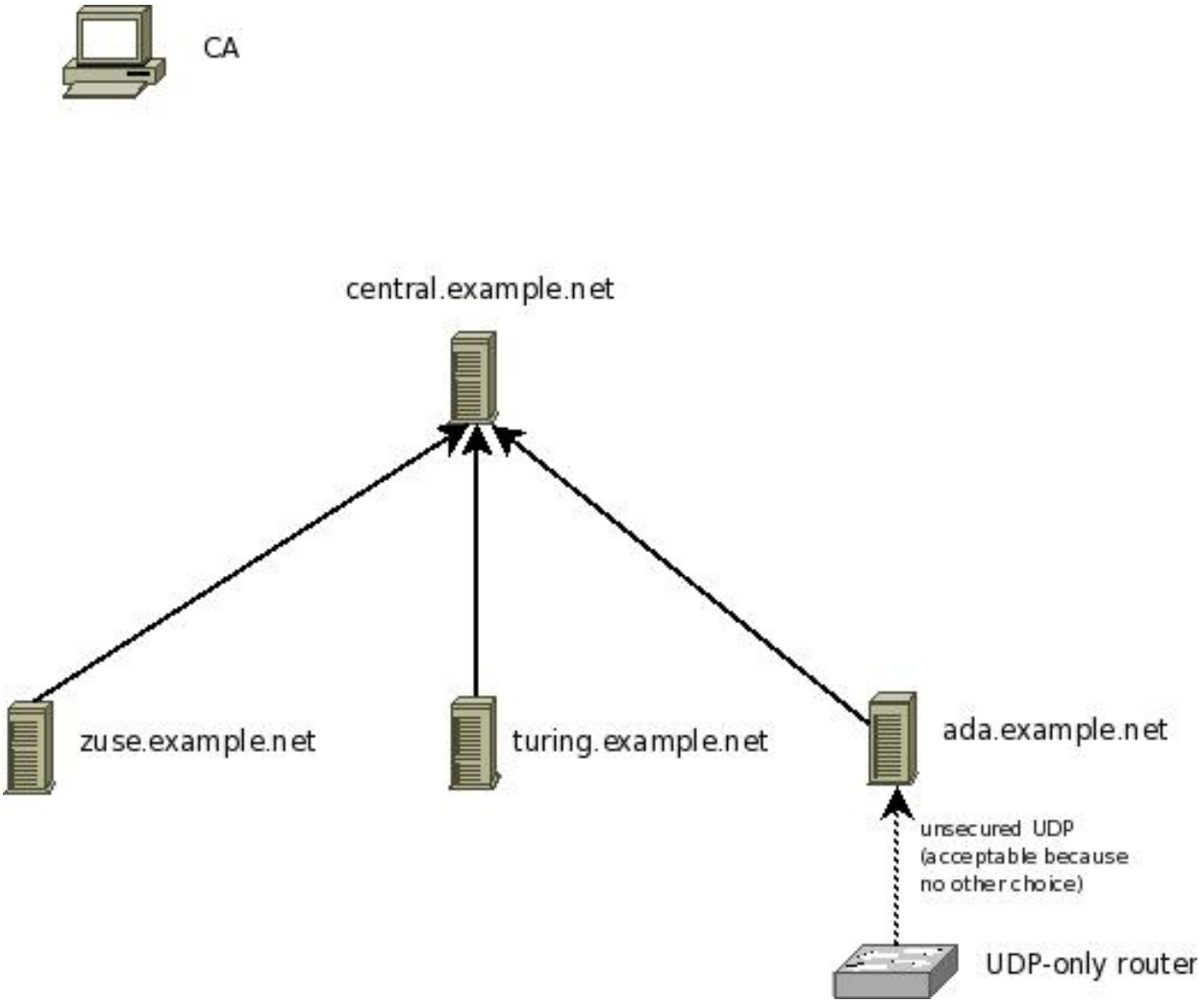
Be sure to safeguard at least the private key (machine-key.pem)! If some third party obtains it, you security is broken!

Setting up the UDP syslog relay

In this step, we configure the UDP relay ada.example.net. As a reminder, that machine relays messages from a local router, which only supports UDP syslog, to the central syslog server. The router does not talk directly to it, because we would like to have TLS protection for its sensitve logs. If the router and the syslog relay are on a sufficiently secure private network, this setup can be considered reasonable secure. In any case, it is the best alternative among the possible configuration scenarios.

Steps to do:

- make sure you have a functional CA (Setting up the CA)
- generate a machine certificate for ada.example.net (follow instructions in Generating Machine Certificates)
- make sure you copy over ca.pem, machine-key.pem ad machine-cert.pem to the client. Ensure that no user except root can access them (**even read permissions are really bad**).


CA
central.example.net
zuse.example.net
turing.example.net
ada.example.net
unsecured UDP
(acceptable because
no other choice)
UDP-only router

- configure the client so that it checks the server identity and sends messages only if the server identity is known.

These were essentially the same steps as for any TLS syslog client. We now need to add the capability to forward the router logs:

- make sure that the firewall rules permit message recpetion on UDP port 514 (if you use a non-standard port for UDP syslog, make sure that port number is permitted).
- you may want to limit who can send syslog messages via UDP. A great place to do this is inside the firewall, but you can also do it in rsyslog.conf via an $AllowedSender directive. We have used one in the sample config below. Please be aware that this is a kind of weak authentication, but definitely better than nothing...
- add the UDP input plugin to rsyslog's config and start a UDP listener
- make sure that your forwarding-filter permits to forward messages received from the remote router to the server. In our sample scenario, we do not need to add anything special, because all messages are forwarded. This includes messages received from remote hosts.

At this point, please be reminded once again that your security needs may be quite different from what we assume in this tutorial. Evaluate your options based on your security needs.

Sample syslog.conf

Keep in mind that this rsyslog.conf sends messages via TCP, only. Also, we do not show any rules to write local files. Feel free to add them.

```
# start a UDP listener for the remote router
$ModLoad imudp    # load UDP server plugin
$AllowedSender UDP, 192.0.2.1 # permit only the router
$UDPServerRun 514 # listen on default syslog UDP port 514

# make gtls driver the default
$DefaultNetstreamDriver gtls

# certificate files
$DefaultNetstreamDriverCAFile /rsyslog/protected/ca.pem
$DefaultNetstreamDriverCertFile /rsyslog/protected/machine-cert.pem
$DefaultNetstreamDriverKeyFile /rsyslog/protected/machine-key.pem

$ActionSendStreamDriverAuthMode x509/name
$ActionSendStreamDriverPermittedPeer central.example.net
$ActionSendStreamDriverMode 1 # run driver in TLS-only mode
*.* @@central.example.net:10514 # forward everything to remote server
```

Be sure to safeguard at least the private key (machine-key.pem)! If some third party obtains it, you security is broken!

Error Messages

This page covers error message you may see when setting up rsyslog with TLS. Please note that many of the message stem back to the TLS library being used. In those cases, there is not always a good explanation available in rsyslog alone.

A single error typically results in two or more message being emitted: (at least) one is the actual error cause, followed by usually one message with additional information (like certificate contents). In a typical system, these message should immediately follow each other in your log. Kepp in mind that they are reported as syslog.err, so you need to

capture these to actually see errors (the default rsyslog.conf's shipped by many systems will do that, recording them e.g. in /etc/messages).

certificate invalid

Sample:

```
not permitted to talk to peer, certificate invalid: insecure algorithm
```

This message may occur during connection setup. It indicates that the remote peer's certificate can not be accepted. The reason for this is given in the message part that is shown in red. Please note that this red part directly stems back to the TLS library, so rsyslog does acutally not have any more information about the reason.

With GnuTLS, the following reasons have been seen in practice:

insecure algorith

The certificate contains information on which encryption algorithms are to be used. This information is entered when the certificate is created. Some older alogrithms are no longer secure and the TLS library does not accept them. Thus the connection request failed. The cure is to use a certificate with sufficiently secure alogorithms.

Please note that noi encryption algorithm is totally secure. It only is secure based on our current knowledge AND on computing power available. As computers get more and more powerful, previously secure algorithms become insecure over time. As such, algorithms considered secure today may not be accepted by the TLS library in the future.

So in theory, after a system upgrade, a connection request may fail with the "insecure algorithm" failure without any change in rsyslog configuration or certificates. This could be caused by a new perception of the TLS library of what is secure and what not.

GnuTLS error -64

Sample:

```
unexpected GnuTLS error -64 in nsd_gtls.c:517: Error while reading file.
```

This error points to an encoding error witht the pem file in question. It means "base 64 encoding error". From my experience, it can be caused by a couple of things, some of them not obvious:

- You specified a wrong file, which is not actually in .pem format
- The file was incorrectly generated
- I think I have also seen this when I accidently swapped private key files and certificate files. So double-check the type of file you are using.
- It may even be a result of an access (permission) problem. In theory, that should lead to another error, but in practice it sometimes seems to lead to this -64 error.

info on invalid cert

Sample:

```
info on invalid cert: peer provided 1 certificate(s). Certificate 1 info: certificate␣
↪valid from Wed Jun 18 11:45:44 2008 to Sat Jun 16 11:45:53 2018; Certificate public␣
↪key: RSA; DN: C=US,O=Sample Corp,OU=Certs,L=Somehwere,ST=CA,CN=somename; Issuer DN:␣
↪C=US,O=Sample Corp,OU=Certs,L=Somewhere,ST=CA,CN=somename,EMAIL=xxx@example.com;␣
↪SAN:DNSname: machine.example.net;
```

This is **not** an error message in itself. It always follows the actual error message and tells you what is seen in the peer's certificate. This is done to give you a chance to evaluate the certificate and better understand why the initial error message was issued.

Please note that you can NOT diagnose problems based on this message alone. It follows in a number of error cases and does not pinpoint any problems by itself.

Overview

This document describes a secure way to set up rsyslog TLS. A secure logging environment requires more than just encrypting the transmission channel. This document provides one possible way to create such a secure system.

Rsyslog's TLS authentication can be used very flexible and thus supports a wide range of security policies. This section tries to give some advise on a scenario that works well for many environments. However, it may not be suitable for you - please assess you security needs before using the recommendations below. Do not blame us if it doesn't provide what you need ;)

Our policy offers these security benefits:

- syslog messages are encrypted while traveling on the wire
- the syslog sender authenticates to the syslog receiver; thus, the receiver knows who is talking to it
- the syslog receiver authenticates to the syslog sender; thus, the sender can check if it indeed is sending to the expected receiver
- the mutual authentication prevents man-in-the-middle attacks

Our secrity goals are achived via public/private key security. As such, it is vital that private keys are well protected and not accessible to third parties.

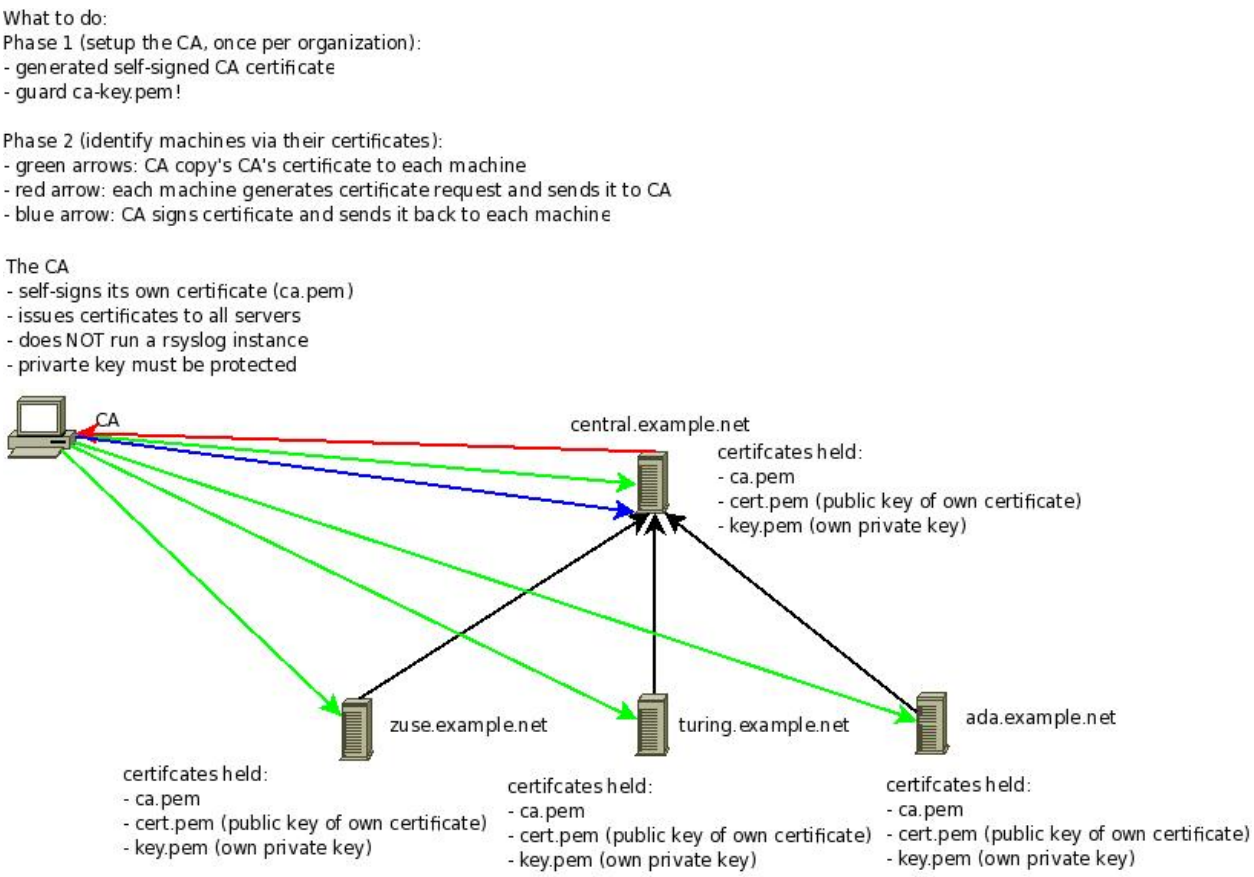
If private keys have become known to third parties, the system does not provide any security at all. Also, our solution bases on X.509 certificates and a (very limited) chain of trust. We have one instance (the CA) that issues all machine certificates. The machine certificate indentifies a particular machine. While in theory (and practice), there could be several "sub-CA" that issues machine certificates for a specific adminitrative domain, we do not include this in our "simple yet secure" setup. If you intend to use this, rsyslog supports it, but then you need to dig a bit more into the documentation (or use the forum to ask). In general, if you depart from our simple model, you should have good reasons for doing so and know quite well what you are doing - otherwise you may compromise your system security.

Please note that security never comes without effort. In the scenario described here, we have limited the effort as much as possible. What remains is some setup work for the central CA, the certificate setup for each machine as well as a few configuration commands that need to be applied to all of them. Proably the most important limiting factor in our setup is that all senders and receivers must support IETF's syslog-transport-tls standard (which is not finalized yet). We use mandatory-to-implement technology, yet you may have trouble finding all required features in some implementations. More often, unfortunately, you will find that an implementation does not support the upcoming IETF standard at all - especially in the "early days" (starting May 2008) when rsyslog is the only implementation of said standard.

Fortunately, rsyslog supports almost every protocol that is out there in the syslog world. So in cases where transport-tls is not available on a sender, we recommend to use rsyslog as the initial relay. In that mode, the not-capabe sender sends to rsyslog via another protocol, which then relays the message via transport-tls to either another interim relay or the final destination (which, of course, must by transport-tls capable). In such a scenario, it is best to try see what the sender support. Maybe it is possible to use industry-standard plain tcp syslog with it. Often you can even combine it with stunnel, which then, too, enables a secure delivery to the first rsyslog relay. If all of that is not possible, you can (and often must...) resort to UDP. Even though this is now lossy and insecure, this is better than not having the ability to listen to that device at all. It may even be reasonale secure if the uncapable sender and the first rsyslog relay communicate via a private channel, e.g. a dedicated network link.

One final word of caution: transport-tls protects the connection between the sender and the receiver. It does not necessarily protect against attacks that are present in the message itself. Especially in a relay environment, the message may have been originated from a malicious system, which placed invalid hostnames and/or other content into it. If there is no provisioning against such things, these records may show up in the receivers' repository. -transport-tls does not protect against this (but it may help, properly used). Keep in mind that syslog-transport-tls provides hop-by-hop security. It does not provide end-to-end security and it does not authenticate the message itself (just the last sender).

If you'd like to get all information very rapidly, the graphic below contains everything you need to know (from the certificate perspective) in a very condensed manner. It is no surprise if the graphic puzzles you. In this case, simply read on for full instructions.



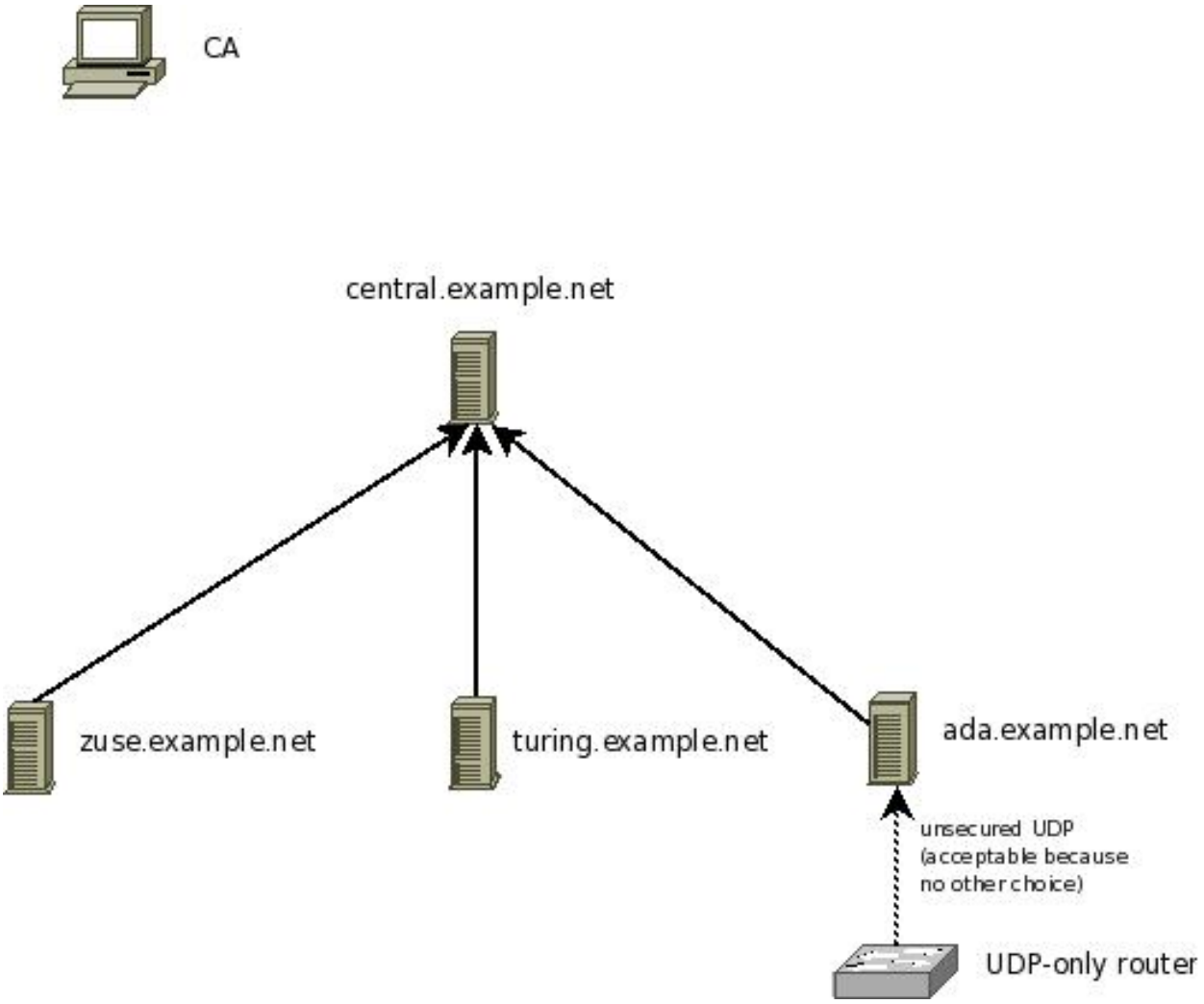
Summary

If you followed the steps outlined in this documentation set, you now have

a reasonable (for most needs) secure setup for the following environment:

You have learned about the security decisions involved and which we made in this example. **Be once again reminded that you must make sure yourself that whatever you do matches your security needs!** There is no guarantee that what we generally find useful actually is. It may even be totally unsuitable for your environment.

In the example, we created a rsyslog certificate authority (CA). Guard the CA's files. You need them whenever you need to create a new machine certificate. We also saw how to generate the machine certificates themselfs and distribute them to the individual machines. Also, you have found some configuration samples for a server, a client and a syslog relay. Hopefully, this will enable you to set up a similar system in many environments.


CA
central.example.net
zuse.example.net
turing.example.net
ada.example.net
unsecured UDP
(acceptable because
no other choice)
UDP-only router

Please be warned that you defined some expiration dates for the certificates. After they are reached, the certificates are no longer valid and rsyslog will NOT accept them. At that point, syslog messages will no longer be transmitted (and rsyslogd will heavily begin to complain). So it is a good idea to make sure that you renew the certificates before they expire. Recording a reminder somewhere is probably a good idea.

Encrypting Syslog Traffic with TLS (SSL) [short version]

Written by Rainer Gerhards *(2008-05-06)*

Abstract

In this paper, I describe how to encrypt syslog **messages on the network.** Encryption is vital to keep the confidiental content of syslog messages secure. I describe the overall approach and provide an HOWTO do it with rsyslog's TLS features.

Please note that TLS is the more secure successor of SSL. While people often talk about "SSL encryption" they actually mean "TLS encryption". So don't look any further if you look for how to SSL-encrypt syslog. You have found the right spot.

This is a quick guide. There is a more elaborate guide currently under construction which provides a much more secure environment. It is highly recommended to at least have a look at it.

Background

Traditional syslog is a clear-text protocol. That means anyone with a sniffer can have a peek at your data. In some environments, this is no problem at all. In others, it is a huge setback, probably even preventing deployment of syslog solutions. Thankfully, there are easy ways to encrypt syslog communication.

The traditional approach involves running a wrapper like stunnel around the syslog session. This works quite well and is in widespread use. However, it is not tightly coupled with the main syslogd and some, even severe, problems can result from this (follow a mailing list thread that describes total loss of syslog messages due to stunnel mode and the unreliability of TCP syslog).

Rsyslog supports syslog via GSSAPI since long to overcome these limitatinos. However, syslog via GSSAPI is a rsyslog-exclusive transfer mode and it requires a proper Kerberos environment. As such, it isn't a really universal solution. The IETF has begun standardizing syslog over plain tcp over TLS for a while now. While I am not fully satisfied with the results so far, this obviously has the potential to become the long-term solution. The Internet Draft in question, syslog-transport-tls has been dormant for some time but is now (May of 2008) again being worked on. I expect it to turn into a RFC within the next 12 month (but don't take this for granted ;)). I didn't want to wait for it, because there obviously is need for TLS syslog right now (and, honestly, I have waited long enough...). Consequently, I have implemented the current draft, with some interpretations I made (there will be a compliance doc soon). So in essence, a TLS-protected syslog transfer mode is available right now. As a side-note, Rsyslog is the world's first implementation of syslog-transport-tls.

Please note that in theory it should be compatible with other, non IETF syslog-transport-tls implementations. If you would like to run it with something else, please let us know so that we can create a compatibility list (and implement compatbility where it doesn't yet exist).

Overall System Setup

Encryption requires a reliable stream. So It will not work over UDP syslog. In rsyslog, network transports utilize a so-called "network stream layer" (netstream for short). This layer provides a unified view of the transport to the application layer. The plain TCP syslog sender and receiver are the upper layer. The driver layer currently consists of

the "ptcp" and "gtls" library plugins. "ptcp" stands for "plain tcp" and is used for unencrypted message transfer. It is also used internally by the gtls driver, so it must always be present on a system. The "gtls" driver is for GnutTLS, a TLS library. It is used for encrypted message transfer. In the future, additional drivers will become available (most importantly, we would like to include a driver for NSS).

What you need to do to build an encrypted syslog channel is to simply use the proper netstream drivers on both the client and the server. Client, in the sense of this document, is the rsyslog system that is sending syslog messages to a remote (central) loghost, which is called the server. In short, the setup is as follows:

Client

- forwards messages via plain tcp syslog using gtls netstream driver to central server on port 10514

Server

- accept incoming messages via plain tcp syslog using gtls netstream driver on port 10514

Setting up the system

Server Setup

At the server, you need to have a digital certificate. That certificate enables SSL operation, as it provides the necessary crypto keys being used to secure the connection. There is a set of default certificates in ./contrib/gnutls. These are key.pem and cert.pem. These are good for testing. If you use it in production, it is very easy to break into your secure channel as everybody is able to get hold of your private key. So it is a good idea to generate the key and certificate yourself.

You also need a root CA certificate. Again, there is a sample CA certificate in ./contrib/gnutls, named ca.cert. It is suggested to generate your own.

To configure the server, you need to tell it where are its certificate files, to use the gtls driver and start up a listener. This is done as follows:

```
# make gtls driver the default
$DefaultNetstreamDriver gtls

# certificate files
$DefaultNetstreamDriverCAFile /path/to/contrib/gnutls/ca.pem
$DefaultNetstreamDriverCertFile /path/to/contrib/gnutls/cert.pem
$DefaultNetstreamDriverKeyFile /path/to/contrib/gnutls/key.pem

$ModLoad imtcp # load TCP listener

$InputTCPServerStreamDriverMode 1 # run driver in TLS-only mode
$InputTCPServerStreamDriverAuthMode anon # client is NOT authenticated
$InputTCPServerRun 10514 # start up listener at port 10514
```

This is all you need to do. You can use the rest of your rsyslog.conf together with this configuration. The way messages are received does not interfer with any other option, so you are able to do anything else you like without any restrictions.

Restart rsyslogd. The server should now be fully operational.

Client Setup

The client setup is equally simple. You need less certificates, just the CA cert.

```
# certificate files - just CA for a client
$DefaultNetstreamDriverCAFile /path/to/contrib/gnutls/ca.pem

# set up the action
$DefaultNetstreamDriver gtls # use gtls netstream driver
$ActionSendStreamDriverMode 1 # require TLS for the connection
$ActionSendStreamDriverAuthMode anon # server is NOT authenticated
*.* @@(o)server.example.net:10514 # send (all) messages
```

Note that we use the regular TCP forwarding syntax (@@) here. There is nothing special, because the encryption is handled by the netstream driver. So I have just forwarded every message (*.*) for simplicity - you can use any of rsyslog's filtering capabilities (like epxression-based filters or regular expressions). Note that the "(o)" part is not strictly necessary. It selects octet-based framing, which provides compatiblity to IETF's syslog-transport-tls draft. Besides compatibility, this is also a more reliable transfer mode, so I suggest to always use it.

Done

After following these steps, you should have a working secure syslog forwarding system. To verify, you can type "logger test" or a similar "smart" command on the client. It should show up in the respective server log file. If you dig out your sniffer, you should see that the traffic on the wire is actually protected.

Limitations

The RELP transport can currently not be protected by TLS. A work-around is to use stunnel. TLS support for RELP will be added once plain TCP syslog has sufficiently matured and there either is some time left to do this or we find a sponsor ;).

Certificates

In order to be really secure, certificates are needed. This is a short summary on how to generate the necessary certificates with GnuTLS' certtool. You can also generate certificates via other tools, but as we currently support GnuTLS as the only TLS library, we thought it is a good idea to use their tools.

Note that this section aims at people who are not involved with PKI at all. The main goal is to get them going in a reasonable secure way.

CA Certificate

This is used to sign all of your other certificates. The CA cert must be trusted by all clients and servers. The private key must be well-protected and not given to any third parties. The certificate itself can (and must) be distributed. To generate it, do the following:

1. generate the private key:

   ```
   certtool --generate-privkey --outfile ca-key.pem
   ```

   This takes a short while. Be sure to do some work on your workstation, it waits for radom input. Switching between windows is sufficient ;)

2. now create the (self-signed) CA certificate itself:

```
certtool --generate-self-signed --load-privkey ca-key.pem --outfile ca.pem
```

This generates the CA certificate. This command queries you for a number of things. Use appropriate responses. When it comes to certificate validity, keep in mind that you need to recreate all certificates when this one expires. So it may be a good idea to use a long period, eg. 3650 days (roughly 10 years). You need to specify that the certificates belongs to an authrity. The certificate is used to sign other certificates.

3. You need to distribute this certificate to all peers and you need to point to it via the $DefaultNetstreamDriverCAFile config directive. All other certificates will be issued by this CA. Important: do only distribute the ca.pem, NOT ca-key.pem (the private key). Distributing the CA private key would totally breach security as everybody could issue new certificates on the behalf of this CA.

Individual Peer Certificate

Each peer (be it client, server or both), needs a certificate that conveys its identity. Access control is based on these certificates. You can, for example, configure a server to accept connections only from configured clients. The client ID is taken from the client instances certificate. So as a general rule of thumb, you need to create a certificate for each instance of rsyslogd that you run. That instance also needs the private key, so that it can properly decrypt the traffic. Safeguard the peer's private key file. If somebody gets hold of it, it can malicously pretend to be the compromised host. If such happens, regenerate the certificate and make sure you use a different name instead of the compromised one (if you use name-based authentication).

These are the steps to generate the indivudual certificates (repeat: you need to do this for every instance, do NOT share the certificates created in this step):

1. generate a private key (do NOT mistake this with the CA's private key - this one is different):

```
certtool --generate-privkey --outfile key.pem
```

Again, this takes a short while.

2. generate a certificate request:

```
certtool --generate-request --load-privkey key.pem --outfile request.pem
```

If you do not have the CA's private key (because you are not authorized for this), you can send the certificate request to the responsible person. If you do this, you can skip the remaining steps, as the CA will provide you with the final certificate. If you submit the request to the CA, you need to tell the CA the answers that you would normally provide in step 3 below.

3. Sign (validate, authorize) the certificate request and generate the instances certificate. You need to have the CA's certificate and private key for this:

```
certtool --generate-certificate --load-request request.pem --outfile cert.pem \ --
↪load-ca-certificate ca.pem --load-ca-privkey ca-key.pem
```

Answer questions as follows: Cert does not belogn to an authority; it is a TLS web server and client certificate; the dnsName MUST be the name of the peer in question (e.g. centralserver.example.net) - this is the name used for authenticating the peers. Please note that you may use an IP address in dnsName. This is a good idea if you would like to use default server authentication and you use selector lines with IP addresses (e.g. "*.* @@192.168.0.1") - in that case you need to select a dnsName of 192.168.0.1. But, of course, changing the server IP then requires generating a new certificate.

After you have generated the certificate, you need to place it onto the local machine running rsyslogd. Specify the certificate and key via the $DefaultNetstreamDriverCertFile /path/to/cert.pem and $DefaultNetstreamDriverKeyFile /path/to/key.pem configuration directives. Make sure that nobody has access to key.pem, as that would breach security.

And, once again: do NOT use these files on more than one instance. Doing so would prevent you from distinguising between the instances and thus would disable useful authentication.

Troubleshooting Certificates

If you experience trouble with your certificate setup, it may be useful to get some information on what is contained in a specific certificate (file). To obtain that information, do

```
$ certtool --certificate-info --infile cert.pem
```

where "cert.pem" can be replaced by the various certificate pem files (but it does not work with the key files).

Conclusion

With minumal effort, you can set up a secure logging infrastructure employing TLS encrypted syslog message transmission.

Feedback requested

I would appreciate feedback on this tutorial. If you have additional ideas, comments or find bugs (I *do* bugs - no way... ;)), please let me know.

Revision History

- 2008-05-06 * Rainer Gerhards * Initial Version created
- 2008-05-26 * Rainer Gerhards * added information about certificates

Copyright

Writing syslog messages to MySQL, PostgreSQL or any other supported Database

Written by Rainer Gerhards*with some additions by Marc Schiffbauer (2008-02-28)*

Abstract

In this paper, I describe how to writesyslog***messages to a***MySQL***or***PostgreSQL***database.****Having syslog messages in a database is often handy, especially when you intend to set up a front-end for viewing them. This paper describes an approach with*rsyslogd*, an alternative enhanced syslog daemon natively supporting MySQL and PostgreSQL. I describe the components needed to be installed and how to configure them. Please note that as of this writing, rsyslog supports a variety of databases. While this guide is still MySQL- and PostgreSQL-focused, you can probably use it together with other ones too. You just need to modify a few settings.*

Background

In many cases, syslog data is simply written to text files. This approach has some advantages, most notably it is very fast and efficient. However, data stored in text files is not readily accessible for real-time viewing and analysis. To do that, the messages need to be in a database. There are various ways to store syslog messages in a database. For example, some have the syslogd write text files which are later feed via a separate script into the database. Others have written scripts taking the data (via a pipe) from a non-database-aware syslogd and store them as they appear. Some others use database-aware syslogds and make them write the data directly to the database. In this paper, I use that "direct write" approach. I think it is superior, because the syslogd itself knows the status of the database connection and thus can handle it intelligently (well ... hopefully ;)). I use rsyslogd to acomplish this, simply because I have initiated the rsyslog project with database-awareness as one goal.

One word of caution: while message storage in the database provides an excellent foundation for interactive analysis, it comes at a cost. Database i/o is considerably slower than text file i/o. As such, directly writing to the database makes sense only if your message volume is low enough to allow a) the syslogd, b) the network, and c) the database server to catch up with it. Some time ago, I have written a paper on optimizing syslog server performance. While this paper talks about Window-based solutions, the ideas in it are generic enough to apply here, too. So it might be worth reading if you anticipate medium high to high traffic. If you anticipate really high traffic (or very large traffic spikes), you should seriously consider forgetting about direct database writes - in my opinion, such a situation needs either a very specialized system or a different approach (the text-file-to-database approach might work better for you in this case).

Overall System Setup

In this paper, I concentrate on the server side. If you are thinking about interactive syslog message review, you probably want to centralize syslog. In such a scenario, you have multiple machines (the so-called clients) send their data to a central machine (called server in this context). While I expect such a setup to be typical when you are interested in storing messages in the database, I do not describe how to set it up. This is beyond the scope of this paper. If you search a little, you will probably find many good descriptions on how to centralize syslog. If you do that, it might be a good idea to do it securely, so you might also be interested in my paper on ssl-encrypting syslog message transfer.

No matter how the messages arrive at the server, their processing is always the same. So you can use this paper in combination with any description for centralized syslog reporting.

As I already said, I use rsyslogd on the server. It has intrinsic support for talking to the supported databases. For obvious reasons, we also need an instance of MySQL or PostgreSQL running. To keep us focused, the setup of the database itself is also beyond the scope of this paper. I assume that you have successfully installed the database and also have a front-end at hand to work with it (for example, phpMyAdmin or phpPgAdmin. Please make sure that this is installed, actually working and you have a basic understanding of how to handle it.

Setting up the system

You need to download and install rsyslogd first. Obtain it from the rsyslog site. Make sure that you disable stock syslogd, otherwise you will experience some difficulties. On some distributions (Fedora 8 and above, for example), rsyslog may already by the default syslogd, in which case you obviously do not need to do anything specific. For many others, there are prebuild packages available. If you use either, please make sure that you have the required database plugins for your database available. It usually is a separate package and typically **not** installed by default.

It is important to understand how rsyslogd talks to the database. In rsyslogd, there is the concept of "templates". Basically, a template is a string that includes some replacement characters, which are called "properties" in rsyslog. Properties are accessed via the "Property Replacer". Simply said, you access properties by including their name between percent signs inside the template. For example, if the syslog message is "Test", the template "%msg%" would be expanded to "Test". Rsyslogd supports sending template text as a SQL statement to the database. As such, the template must be a valid SQL statement. There is no limit in what the statement might be, but there are some obvious and not so obvious choices. For example, a template "drop table xxx" is possible, but does not make an awful lot of sense. In practice, you will always use an "insert" statement inside the template.

An example: if you would just like to store the msg part of the full syslog message, you have probably created a table "syslog" with a single column "message". In such a case, a good template would be "insert into syslog(message) values ('%msg%')". With the example above, that would be expanded to "insert into syslog(message) values('Test')". This expanded string is then sent to the database. It's that easy, no special magic. The only thing you must ensure is that your template expands to a proper SQL statement and that this statement matches your database design.

Does that mean you need to create database schema yourself and also must fully understand rsyslogd's properties? No, that's not needed. Because we anticipated that folks are probably more interested in getting things going instead of designing them from scratch. So we have provided a default schema as well as build-in support for it. This schema also offers an additional benefit: rsyslog is part of Adiscon's MonitorWare product line (which includes open source and closed source members). All of these tools share the same default schema and know how to operate on it. For this reason, the default schema is also called the "MonitorWare Schema". If you use it, you can simply add phpLogCon, a GPLed syslog web interface, to your system and have instant interactive access to your database. So there are some benefits in using the provided schema.

The schema definition is contained in the file "createDB.sql". It comes with the rsyslog package and one can be found for each supported database type (in the plugins directory). Review it to check that the database name is acceptable for you. Be sure to leave the table and field names unmodified, because otherwise you need to customize rsyslogd's default sql template, which we do not do in this paper. Then, run the script with your favorite SQL client. Double-check that the table was successfully created.

It is important to note that the correct database encoding must be used so that the database will accept strings independend of the string encoding. This is an important part because it can not be guarantied that all syslog messages will have a defined character encoding. This is especially true if the rsyslog-Server will collect messages from different clients and different products.

For example PostgreSQL may refuse to accept messages if you would set the database encoding to "UTF8" while a client is sending invalid byte sequences for that encoding.

Database support in rsyslog is integrated via loadable plugin modules. To use the database functionality, the database plugin must be enabled in the config file BEFORE the first database table action is used. This is done by placing the

```
$ModLoad ommysql
```

directive at the begining of /etc/rsyslog.conf for MySQL and

```
$ModLoad ompgsql
```

for PostgreSQL.

For other databases, use their plugin name (e.g. omoracle).

Next, we need to tell rsyslogd to write data to the database. As we use the default schema, we do NOT need to define a template for this. We can use the hardcoded one (rsyslogd handles the proper template linking). So all we need to do e.g. for MySQL is add a simple selector line to /etc/rsyslog.conf:

```
*.*       :ommysql:database-server,database-name,database-userid,
database-password
```

Again, other databases have other selector names, e.g. ":ompgsql:" instead of ":ommysql:". See the output plugin's documentation for details.

In many cases, the database will run on the local machine. In this case, you can simply use "127.0.0.1" for *database-server*. This can be especially advisable, if you do not need to expose the database to any process outside of the local machine. In this case, you can simply bind it to 127.0.0.1, which provides a quite secure setup. Of course, rsyslog also supports remote database instances. In that case, use the remote server name (e.g. mydb.example.com) or IP-address. The *database-name* by default is "Syslog". If you have modified the default, use your name here. *Database-userid* and *-password* are the credentials used to connect to the database. As they are stored in clear text in rsyslog.conf, that user should have only the least possible privileges. It is sufficient to grant it INSERT privileges to the systemevents table, only. As a side note, it is strongly advisable to make the rsyslog.conf file readable by root only - if you make it world-readable, everybody could obtain the password (and eventually other vital information from it). In our example,

let's assume you have created a database user named "syslogwriter" with a password of "topsecret" (just to say it bluntly: such a password is NOT a good idea...). If your database is on the local machine, your rsyslog.conf line might look like in this sample:

```
*.*         :ommysql:127.0.0.1,Syslog,syslogwriter,topsecret
```

Save rsyslog.conf, restart rsyslogd - and you should see syslog messages being stored in the "systemevents" table!

The example line stores every message to the database. Especially if you have a high traffic volume, you will probably limit the amount of messages being logged. This is easy to accomplish: the "write database" action is just a regular selector line. As such, you can apply normal selector-line filtering. If, for example, you are only interested in messages from the mail subsystem, you can use the following selector line:

```
mail.*         :ommysql:127.0.0.1,syslog,syslogwriter,topsecret
```

Review the rsyslog.conf documentation for details on selector lines and their filtering.

You have now completed everything necessary to store syslog messages to the a database. If you would like to try out a front-end, you might want to look at phpLogCon, which displays syslog data in a browser. As of this writing, phpLogCon is not yet a powerful tool, but it's open source, so it might be a starting point for your own solution.

On Reliability...

Rsyslogd writes syslog messages directly to the database. This implies that the database must be available at the time of message arrival. If the database is offline, no space is left or something else goes wrong - rsyslogd can not write the database record. If rsyslogd is unable to store a message, it performs one retry. This is helpful if the database server was restarted. In this case, the previous connection was broken but a reconnect immediately succeeds. However, if the database is down for an extended period of time, an immediate retry does not help.

Message loss in this scenario can easily be prevented with rsyslog. All you need to do is run the database writer in queued mode. This is now described in a generic way and I do not intend to duplicate it here. So please be sure to read "Handling a massive syslog database insert rate with Rsyslog", which describes the scenario and also includes configuration examples.

Conclusion

With minimal effort, you can use rsyslogd to write syslog messages to a database. You can even make it absolutely fail-safe and protect it against database server downtime. Once the messages are arrived there, you can interactively review and analyze them. In practice, the messages are also stored in text files for longer-term archival and the databases are cleared out after some time (to avoid becoming too slow). If you expect an extremely high syslog message volume, storing it in real-time to the database may outperform your database server. In such cases, either filter out some messages or used queued mode (which in general is recommended with databases).

The method outlined in this paper provides an easy to setup and maintain solution for most use cases.

Feedback Requested

I would appreciate feedback on this paper. If you have additional ideas, comments or find bugs, please let me know.

References and Additional Material

- www.rsyslog.com - the rsyslog site
- Paper on Syslog Server Optimization

Revision History

- 2005-08-02 * Rainer Gerhards * initial version created
- 2005-08-03 * Rainer Gerhards * added references to demo site
- 2007-06-13 * Rainer Gerhards * removed demo site - was torn down because too expensive for usage count
- 2008-02-21 * Rainer Gerhards * updated reliability section, can now be done with on-demand disk queues
- 2008-02-28 * Rainer Gerhards * added info on other databases, updated syntax to more recent one
- 2010-01-29 * Marc Schiffbauer * added some PostgreSQL stuff, made wording more database generic, fixed some typos

Copyright

Handling a massive syslog database insert rate with Rsyslog

Written by Rainer Gerhards *(2008-01-31)*

Abstract

In this paper, I describe how log massive amounts of syslog **messages to a database.**This HOWTO is currently under development and thus a bit brief. Updates are promised ;).*

The Intention

Database updates are inherently slow when it comes to storing syslog messages. However, there are a number of applications where it is handy to have the message inside a database. Rsyslog supports native database writing via output plugins. As of this writing, there are plugins available for MySQL an PostgreSQL. Maybe additional plugins have become available by the time you read this. Be sure to check.

In order to successfully write messages to a database backend, the backend must be capable to record messages at the expected average arrival rate. This is the rate if you take all messages that can arrive within a day and divide it by 86400 (the number of seconds per day). Let's say you expect 43,200,000 messages per day. That's an average rate of 500 messages per second (mps). Your database server MUST be able to handle that amount of message per second on a sustained rate. If it doesn't, you either need to add an additional server, lower the number of message - or forget about it.

However, this is probably not your peak rate. Let's simply assume your systems work only half a day, that's 12 hours (and, yes, I know this is unrealistic, but you'll get the point soon). So your average rate is actually 1,000 mps during work hours and 0 mps during non-work hours. To make matters worse, workload is not divided evenly during the day. So you may have peaks of up to 10,000mps while at other times the load may go down to maybe just 100mps. Peaks may stay well above 2,000mps for a few minutes.

So how the hack you will be able to handle all of this traffic (including the peaks) with a database server that is just capable of inserting a maximum of 500mps?

The key here is buffering. Messages that the database server is not capable to handle will be buffered until it is. Of course, that means database insert are NOT real-time. If you need real-time inserts, you need to make sure your database server can handle traffic at the actual peak rate. But lets assume you are OK with some delay.

Buffering is fine. But how about these massive amounts of data? That can't be hold in memory, so don't we run out of luck with buffering? The key here is that rsyslog can not only buffer in memory but also buffer to disk (this may remind you of "spooling" which gets you the right idea). There are several queuing modes available, offering differnent throughput. In general, the idea is to buffer in memory until the memory buffer is exhausted and switch to disk-buffering when needed (and only as long as needed). All of this is handled automatically and transparently by rsyslog.

With our above scenario, the disk buffer would build up during the day and rsyslog would use the night to drain it. Obviously, this is an extreme example, but it shows what can be done. Please note that queue content survies rsyslogd restarts, so even a reboot of the system will not cause any message loss.

How To Setup

Frankly, it's quite easy. You just need to do is instruct rsyslog to use a disk queue and then configure your action. There is nothing else to do. With the following simple config file, you log anything you receive to a MySQL database and have buffering applied automatically.

```
$ModLoad ommysql # load the output driver (use ompgsql for PostgreSQL)
$ModLoad imudp # network reception
$UDPServerRun 514 # start a udp server at port 514
$ModLoad imuxsock # local message reception
$WorkDirectory /rsyslog/work # default location for work (spool) files
$MainMsgQueueFileName mainq # set file name, also enables disk mode
$ActionResumeRetryCount -1 # infinite retries on insert failure
#for PostgreSQL replace :ommysql: by :ompgsql: below: *.* :ommysql:hostname,dbname,
↪userid,password;
```

The simple setup above has one drawback: the write database action is executed together with all other actions. Typically, local files are also written. These local file writes are now bound to the speed of the database action. So if the database is down, or threre is a large backlog, local files are also not (or late) written.

There is an easy way to avoid this with rsyslog. It involves a slightly more complicated setup. In rsyslog, each action can utilize its own queue. If so, messages are simply pulled over from the main queue and then the action queue handles action processing on its own. This way, main processing and the action are de-coupled. In the above example, this means that local file writes will happen immediately while the database writes are queued. As a side-note, each action can have its own queue, so if you would like to more than a single database or send messages reliably to another host, you can do all of this on their own queues, de-coupling their processing speeds.

The configuration for the de-coupled database write involves just a few more commands:

```
$ModLoad ommysql # load the output driver (use ompgsql for PostgreSQL)
$ModLoad imudp # network reception
$UDPServerRun 514 # start a udp server at port 514
$ModLoad imuxsock # local message reception
$WorkDirectory /rsyslog/work # default location for work (spool) files
$ActionQueueType LinkedList # use asynchronous processing
$ActionQueueFileName dbq # set file name, also enables disk mode
$ActionResumeRetryCount -1 # infinite retries on insert failure
# for PostgreSQL replace :ommysql: by :ompgsql: below: *.* :ommysql:hostname,dbname,
↪userid,password;
```

This is the recommended configuration for this use case. It requires rsyslog 3.11.0 or above.

In this example, the main message queue is NOT disk-assisted (there is no $MainMsgQueueFileName directive). We still could do that, but have not done it because there seems to be no need. The only slow running action is the database writer and it has its own queue. So there is no real reason to use a large main message queue (except, of course, if you expect *really* heavy traffic bursts).

Note that you can modify a lot of queue performance parameters, but the above config will get you going with default values. If you consider using this on a real busy server, it is strongly recommended to invest some time in setting the tuning parameters to appropriate values.

Feedback requested

I would appreciate feedback on this tutorial. If you have additional ideas, comments or find bugs (I *do* bugs - no way... ;)), please let me know.

Revision History

- 2008-01-28 * Rainer Gerhards * Initial Version created
- 2008-01-28 * Rainer Gerhards * Updated to new v3.11.0 capabilities

Copyright

Reliable Forwarding of syslog Messages with Rsyslog

Written by Rainer Gerhards *(2008-06-27)*

Abstract

In this paper, I describe how to forward syslog **messages (quite) reliable to a central rsyslog server.** This depends on rsyslog being installed on the client system and it is recommended to have it installed on the server system. Please note that industry-standard plain TCP syslog protocol is not fully reliable (thus the "quite reliable"). If you need a truely reliable solution, you need to look into RELP (natively supported by rsyslog).*

The Intention

Whenever two systems talk over a network, something can go wrong. For example, the communications link may go down, or a client or server may abort. Even in regular cases, the server may be offline for a short period of time because of routine maintenance.

A logging system should be capable of avoiding message loss in situations where the server is not reachable. To do so, unsent data needs to be buffered at the client while the server is offline. Then, once the server is up again, this data is to be sent.

This can easily be acomplished by rsyslog. In rsyslog, every action runs on its own queue and each queue can be set to buffer data if the action is not ready. Of course, you must be able to detect that "the action is not ready", which means the remote server is offline. This can be detected with plain TCP syslog and RELP, but not with UDP. So you need to use either of the two. In this howto, we use plain TCP syslog.

Please note that we are using rsyslog-specific features. The are required on the client, but not on the server. So the client system must run rsyslog (at least version 3.12.0), while on the server another syslogd may be running, as long as it supports plain tcp syslog.

The rsyslog queueing subsystem tries to buffer to memory. So even if the remote server goes offline, no disk file is generated. File on disk are created only if there is need to, for example if rsyslog runs out of (configured) memory queue space or needs to shutdown (and thus persist yet unsent messages). Using main memory and going to the disk when needed is a huge performance benefit. You do not need to care about it, because, all of it is handled automatically and transparently by rsyslog.

How To Setup

First, you need to create a working directory for rsyslog. This is where it stores its queue files (should need arise). You may use any location on your local system.

Next, you need to do is instruct rsyslog to use a disk queue and then configure your action. There is nothing else to do. With the following simple config file, you forward anything you receive to a remote server and have buffering applied automatically when it goes down. This must be done on the client machine.

```
$ModLoad imuxsock # local message reception
$WorkDirectory /rsyslog/work # default location for work (spool) files
$ActionQueueType LinkedList # use asynchronous processing
$ActionQueueFileName srvrfwd # set file name, also enables disk mode
$ActionResumeRetryCount -1 # infinite retries on insert failure
$ActionQueueSaveOnShutdown on # save in-memory data if rsyslog shuts down
*.* @@server:port
```

The port given above is optional. It may not be specified, in which case you only provide the server name. The "$ActionQueueFileName" is used to create queue files, should need arise. This value must be unique inside rsyslog.conf. No two rules must use the same queue file. Also, for obvious reasons, it must only contain those characters that can be used inside a valid file name. Rsyslog possibly adds some characters in front and/or at the end of that name when it creates files. So that name should not be at the file size name length limit (which should not be a problem these days).

Please note that actual spool files are only created if the remote server is down **and** there is no more space in the in-memory queue. By default, a short failure of the remote server will never result in the creation of a disk file as a couple of hundered messages can be held in memory by default. [These parameters can be fine-tuned. However, then you need to either fully understand how the queue works (read elaborate doc) or use professional services to have it done based on your specs ;) - what that means is that fine-tuning queue parameters is far from being trivial...]

If you would like to test if your buffering scenario works, you need to stop, wait a while and restart you central server. Do **not** watch for files being created, as this usually does not happen and never happens immediately.

Forwarding to More than One Server

If you have more than one server you would like to forward to, that's quickly done. Rsyslog has no limit on the number or type of actions, so you can define as many targets as you like. What is important to know, however, is that the full set of directives make up an action. So you can not simply add (just) a second forwarding rule, but need to duplicate the rule configuration as well. Be careful that you use different queue file names for the second action, else you will mess up your system.

A sample for forwarding to two hosts looks like this:

```
$ModLoad imuxsock # local message reception
$WorkDirectory /rsyslog/work # default location for work (spool) files

# start forwarding rule 1
```

```
$ActionQueueType LinkedList # use asynchronous processing
$ActionQueueFileName srvrfwd1 # set file name, also enables disk mode
$ActionResumeRetryCount -1 # infinite retries on insert failure
$ActionQueueSaveOnShutdown on # save in-memory data if rsyslog shuts down
*.* @@server1:port
# end forwarding rule 1

# start forwarding rule 2
$ActionQueueType LinkedList # use asynchronous processing
$ActionQueueFileName srvrfwd2 # set file name, also enables disk mode
$ActionResumeRetryCount -1 # infinite retries on insert failure
$ActionQueueSaveOnShutdown on # save in-memory data if rsyslog shuts down
*.* @@server2
# end forwarding rule 2
```

Note the filename used for the first rule it is "srvrfwd1" and for the second it is "srvrfwd2". I have used a server without port name in the second forwarding rule. This was just to illustrate how this can be done. You can also specify a port there (or drop the port from server1).

When there are multiple action queues, they all work independently. Thus, if server1 goes down, server2 still receives data in real-time. The client will **not** block and wait for server1 to come back online. Similarily, server1's operation will not be affected by server2's state.

Some Final Words on Reliability ...

Using plain TCP syslog provides a lot of reliability over UDP syslog. However, plain TCP syslog is **not** a fully reliable transport. In order to get full reliability, you need to use the RELP protocol.

Folow the next link to learn more about the problems you may encounter with plain tcp syslog.

Feedback requested

I would appreciate feedback on this tutorial. If you have additional ideas, comments or find bugs (I *do* bugs - no way... ;)), please let me know.

Revision History

- 2008-06-27 * Rainer Gerhards * Initial Version created

Copyright

Recording the Priority of Syslog Messages

Written by Rainer Gerhards *(2007-06-18)*

Abstract

The so-called priority (PRI) is very important in syslog messages, because almost all filtering in syslog.conf is based on it. However, many syslogds (including the Linux stock sysklogd) do not provide a way to record that value. In this article, I'll give a brief overview of how PRI can be written to a log file.

Background

The PRI value is a combination of so-called severity and facility. The facility indicates where the message originated from (e.g. kernel, mail subsystem) while the severity provides a glimpse of how important the message might be (e.g. error or informational). Be careful with these values: they are in no way consistent across applications (especially severity). However, they still form the basis of most filtering in syslog.conf. For example, the directive (aka "selector line)

```
mail.* /var/log/mail.log
```

means that messages with the mail facility should be stored to /var/log/mail.log, no matter which severity indicator they have (that is telling us the asterisk). If you set up complex conditions, it can be annoying to find out which PRI value a specific syslog message has. Most stock syslogds do not provide any way to record them.

How is it done?

With rsyslog, PRI recording is simple. All you need is the correct template. Even if you do not use rsyslog on a regular basis, it might be a handy tool for finding out the priority.

Rsyslog provides a flexible system to specify the output formats. It is template-based. A template with the traditional syslog format looks as follows:

```
$template TraditionalFormat,"%timegenerated% %HOSTNAME% %syslogtag%%msg:::drop-last-lf
↪%\n"
```

The part in quotes is the output formats. Things between percent-signs are so-called messages properties. They are replaced with the respective content from the syslog message when output is written. Everything outside of the percent signs is literal text, which is simply written as specified.

Thankfully, rsyslog provides message properties for the priority. These are called "PRI", "syslogfacility" and "syslogpriority" (case is important!). They are numerical values. Starting with rsyslog 1.13.4, there is also a property "pri-text", which contains the priority in friendly text format (e.g. "local0.err<133>"). For the rest of this article, I assume that you run version 1.13.4 or higher.

Recording the priority is now a simple matter of adding the respective field to the template. It now looks like this:

```
$template TraditionalFormatWithPRI,"%pri-text%: %timegenerated% %HOSTNAME% %syslogtag%
↪%msg:::drop-last-lf%\n"
```

Now we have the right template - but how to write it to a file? You probably have a line like this in your syslog.conf:

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

It does not specify a template. Consequently, rsyslog uses the traditional format. In order to use some other format, simply specify the template after the semicolon:

```
*.* -/var/log/messages.log;TraditionalFormatWithPRI
```

That's all you need to do. There is one common pitfall: you need to define the template before you use it in a selector line. Otherwise, you will receive an error.

Once you have applied the changes, you need to restart rsyslogd. It will then pick the new configuration.

What if I do not want rsyslogd to be the standard syslogd?

If you do not want to switch to rsyslog, you can still use it as a setup aid. A little bit of configuration is required.

1. Download, make and install rsyslog
2. copy your syslog.conf over to rsyslog.conf
3. add the template described above to it; select the file that should use it
4. stop your regular syslog daemon for the time being
5. run rsyslogd (you may even do this interactively by calling it with the -n additional option from a shell)
6. stop rsyslogd (press ctrl-c when running interactively)
7. restart your regular syslogd

That's it - you can now review the priorities.

Some Sample Data

Below is some sample data created with the template specified above. Note the priority recording at the start of each line.

```
kern.info<6>: Jun 15 18:10:38 host kernel: PCI: Sharing IRQ 11 with 00:04.0
kern.info<6>: Jun 15 18:10:38 host kernel: PCI: Sharing IRQ 11 with 01:00.0
kern.warn<4>: Jun 15 18:10:38 host kernel: Yenta IRQ list 06b8, PCI irq11
kern.warn<4>: Jun 15 18:10:38 host kernel: Socket status: 30000006
kern.warn<4>: Jun 15 18:10:38 host kernel: Yenta IRQ list 06b8, PCI irq11
kern.warn<4>: Jun 15 18:10:38 host kernel: Socket status: 30000010
kern.info<6>: Jun 15 18:10:38 host kernel: cs: IO port probe 0x0c00-0x0cff: clean.
kern.info<6>: Jun 15 18:10:38 host kernel: cs: IO port probe 0x0100-0x04ff: excluding␣
↪0x100-0x107 0x378-0x37f 0x4d0-0x4d7
kern.info<6>: Jun 15 18:10:38 host kernel: cs: IO port probe 0x0a00-0x0aff: clean.
local7.notice<189>: Jun 15 18:17:24 host dd: 1+0 records out
local7.notice<189>: Jun 15 18:17:24 host random: Saving random seed: succeeded
local7.notice<189>: Jun 15 18:17:25 host portmap: portmap shutdown succeeded
local7.notice<189>: Jun 15 18:17:25 host network: Shutting down interface eth1:␣
↪succeeded
local7.notice<189>: Jun 15 18:17:25 host network: Shutting down loopback interface:␣
↪succeeded
local7.notice<189>: Jun 15 18:17:25 host pcmcia: Shutting down PCMCIA services:␣
↪cardmgr
user.notice<13>: Jun 15 18:17:25 host /etc/hotplug/net.agent: NET unregister event␣
↪not supported
local7.notice<189>: Jun 15 18:17:27 host pcmcia: modules.
local7.notice<189>: Jun 15 18:17:29 host rc: Stopping pcmcia: succeeded
local7.notice<189>: Jun 15 18:17:30 host rc: Starting killall: succeeded
syslog.info<46>: Jun 15 18:17:33 host [origin software="rsyslogd" swVersion="1.13.3"␣
↪x-pid="2464"] exiting on signal 15.
syslog.info<46>: Jun 18 10:55:47 host [origin software="rsyslogd" swVersion="1.13.3"␣
↪x-pid="2367"][x-configInfo udpReception="Yes" udpPort="514" tcpReception="Yes"␣
↪tcpPort="1470"] restart
```

```
user.notice<13>: Jun 18 10:55:50 host rger: test
syslog.info<46>: Jun 18 10:55:52 host [origin software="rsyslogd" swVersion="1.13.3"␣
↪x-pid="2367"] exiting on signal 2.``
```

Feedback Requested

I would appreciate feedback on this paper. If you have additional ideas, comments or find bugs, please let me know.

References and Additional Material

- www.rsyslog.com - the rsyslog site

Revision History

- 2007-06-18 * Rainer Gerhards * initial version created

Copyright

Failover Syslog Server

There are often situations where syslog data from the local system should be sent to a central syslogd (for consolidation, archival and whatever other reasons). A common problem is that messages are lost when the central syslogd goes down. Rsyslog has the capability to work with failover servers to prevent message loss. A perquisite is that TCP based syslog forwarding is used to sent to the central server. The reason is that with UDP there is no reliable way to detect the remote system has gone away. Let's assume you have a primary and two secondary central servers. Then, you can use the following config file excerpt to send data to them: rsyslog.conf:

```
*.* @@primary-syslog.example.com
$ActionExecOnlyWhenPreviousIsSuspended on
& @@secondary-1-syslog.example.com
& @@secondary-2-syslog.example.com
& /var/log/localbuffer
$ActionExecOnlyWhenPreviousIsSuspended off
```

This selector processes all messages it receives (.). It tries to forward every message to primary-syslog.example.com (via tcp). If it can not reach that server, it tries secondary-1-syslog.example.com, if that fails too, it tries secondary-2-syslog.example.com. If neither of these servers can be connected, the data is stored in /var/log/localbuffer. Please note that the secondaries and the local log buffer are only used if the one before them does not work. So ideally, /var/log/localbuffer will never receive a message. If one of the servers resumes operation, it automatically takes over processing again.

Log rotation with rsyslog

Written by Michael Meckelein

Situation

Your environment does not allow you to store tons of logs? You have limited disc space available for logging, for example you want to log to a 124 MB RAM usb stick? Or you do not want to keep all the logs for months, logs from the last days is sufficient? Think about log rotation.

Log rotation based on a fixed log size

This small but hopefully useful article will show you the way to keep your logs at a given size. The following sample is based on rsyslog illustrating a simple but effective log rotation with a maximum size condition.

Use Output Channels for fixed-length syslog files

Lets assume you do not want to spend more than 100 MB hard disc space for you logs. With rsyslog you can configure Output Channels to achieve this. Putting the following directive

```
# start log rotation via outchannel
# outchannel definition
$outchannel log_rotation,/var/log/log_rotation.log, 52428800,/home/me/./log_rotation_
↪script
#  activate the channel and log everything to it
*.* :omfile:$log_rotation
# end log rotation via outchannel
```

to ryslog.conf instruct rsyslog to log everything to the destination file '/var/log/log_rotation.log' until the give file size of 50 MB is reached. If the max file size is reached it will perform an action. In our case it executes the script /home/me/log_rotation_script which contains a single command:

```
mv -f /var/log/log_rotation.log /var/log/log_rotation.log.1
```

This moves the original log to a kind of backup log file. After the action was successfully performed rsyslog creates a new /var/log/log_rotation.log file and fill it up with new logs. So the latest logs are always in log_rotation.log.

Conclusion

With this approach two files for logging are used, each with a maximum size of 50 MB. So we can say we have successfully configured a log rotation which satisfies our requirement. We keep the logs at a fixed-size level of 100 MB.

This documentation is part of the rsyslog project.

GELF forwarding in rsyslog

Written by Florian Riedl

Situation

The current setup has a system with rsyslog as the central syslog server and a system with Graylog for storage and analyzing the log messages. Graylog expects the log messages to arrive in GELF (Graylog Extended Log Format).

Changing the default log format to GELF

To make rsyslog send GELF we basically need to create a custom template. This template will define the format in which the log messages will get sent to Graylog.

```
template(name="gelf" type="list") {
    constant(value="{\"version\":\"1.1\",")
    constant(value="\"host\":\"")
    property(name="hostname")
    constant(value="\",\"short_message\":\"")
    property(name="msg" format="json")
    constant(value="\",\"timestamp\":\"")
    property(name="timegenerated" dateformat="unixtimestamp")
    constant(value="\",\"level\":\"")
    property(name="syslogseverity")
    constant(value="\"}")
}
```

This is a typical representation in the list format with all the necessary fields and format definitions that Graylog expects.

Applying the template to a syslog action

The next step is applying the template to our output action. Since we are forwarding log messages to Graylog, this is usually a syslog sending action.

```
# syslog forwarder via UDP
action(type="omfwd" target="graylogserver" port="514" protocol="udp" template="gelf")
```

We now have a syslog forwarding action. This uses the omfwd module. Please note that the case above only works for UDP transport. When using TCP, Graylog expects a Nullbyte as message delimiter. This is currently not possible with rsyslog.

Conclusion

With this quick and easy setup you can feed Graylog with the correct log message format so it can do its work. This case can be applied to a lot of different scenarios as well, but with different templates.

This documentation is part of the rsyslog project.

Development

The rsyslog config data model

This document describes the config data model on a high layer. For details, it is suggested to review the actual source code. The aim of this document is to provide general understanding for both rsyslog developers as well as developers

writing config management systems.

Objects

Most config objects life in a flat space and are global to rsyslog. However, actual rule processing is done via a script-like language. These config scripts need to be represented via a tree structure.

Note that the language as currently implemented is Turing-complete if the user makes use of very tricky constructs. It was never our intention to provide a Turing-complete language and we will probably try to disable these tricks in the future. However, this is not a priority for us, as these users get what they deserve. For someone involved with the config, it probably is sufficient to know that loops are **not** supported by the config language (even though you can create loop-like structures). Thus, a tree is fully sufficient to represent any configuration.

In the following sections, we'll quickly describe variables/properties, flat structure elements and the execution tree.

Variables/Properties

Rsyslog supports

- traditional syslog (RFC-based) message properties
- structured data content, including any non-syslog properties
- Variables
  - global
  - local
  - message-enhancing (like message properties)

A description of these properties and variables is available elsewhere. As far as a config processor is concerned, the important thing to know is that they be used during template definitions and script operations.

Flat Elements

Global Parameters

This element must contain all global parameters settable by rsyslog. This includes elements from the global() as well as main_queue() config statements. As of this writing, some global parameter can only be set by legacy statements.

Note that main_queue() actually is a full queue defintion.

Modules

This contains all loaded modules, among others:

- input modules
- output modules
- message modification modules
- message parsers

Note that for historical reasons some output modules are directly linked into rsyslog and must not be specified.

Each module must be given only once. The data object must contain all module-global parameters.

Inputs

Describes all defined inputs with their parameters. Is build from the input() statement or its legacy equivalent (ugly). Contains links to

- module used for input
- ruleset used for processing

Rulesets

They contain the tree-like execution structure. However, rulesets itself are flat and cannot be nested. Note that there exists statements that permit rulesets to call into each other, but all rulesets are in the same flat top-level space.

Note that a ruleset has an associated queue object. In most cases, it needs not to be configured as a real queue (not one in direct mode) is only required in special cases.

Hierarchical Elements

These are used for rule exection. They are somewhat hard to fit into a traditional config scheme, as they provide full tree-like branching structure.

Basically, a tree consists of statements and evaluations. Consider the ruleset to be the root of the execution tree. It is rather common that the tree's main level is a long linked list, with only actions being branched out. This, for example, happens with a traditional rsyslog.conf setting, which only contains files to be written based on some priority filters. However, one must not be tricked into thinking that this basic case is sufficient to support as enterprise users typically create far more complex cases.

In essence, rsyslog walks the tree, and executes statements while it does so. Usually, a filter needs to be evaluated and execution branches based on the filter outcome. The tree actually **is** an AST.

Execution Statements

These are most easy to implement as they are end nodes (and as such nothing can be nested under them). They are most importantly created by the action() config object, but also with statements like "set" and "unset". Note that "call" is also considered a terminal node, even though it executes *another* ruleset.

Note that actions have associated queues, so a queue object and its parameter need to be present. When building configurations interactively, it is suggested that the default is either not to configure queue parameters by default or to do this only for actions where it makes sense (e.g. connection to remote systems which may go offline).

Expression Evaluation

A full expression evaluation engine is available who does the typical programming-language type of expression processing. The usual mathematical, boolean and string operations are supported, as well as functions. As of this writing, functions are hardcoded into rsyslog but may in the future be part of a loadable module. Evaluations can access all rsyslog properties and variables. They may be nested arbitrarily deep.

Control-of-Flow Statements

Remember that rsyslog does intentionally not support loop statements. So control-of-flow boils down to

- conditional statements
  - "if ... then ... else ..."
  - syslog PRI-based filters
  - property-based filters
- stop

Where "stop" terminates processing of this message. The conditional statements contain subbranches, where "if" contains both "then" and "else" subbranches and the other two only the "then" subranch (Note: inside the execution engine, the others may also have "else" branches, but these are result of the rsyslog config optimizer run and cannot configured by the user).

When executing a config script, rsyslog executes the subbranch in question and then continues to evaluate the next statement in the currently executing branch that contained the conditional statement. If there is no next statement, it goes up one layer. This is continued until the last statement of the root statement list is reached. At that point execution of the message is terminated and the message object destructed. Again, think AST, as this is exactly what it is.

Note on Queue Objects

Queue objects are **not** named objects inside the rsyslog configuration. So their data is always contained with the object that uses the queue (action(), ruleset(), main_queue()). From a UI perspective, this unfortunately tends to complicate a config builder a bit.

Debugging

Author: Pascal Withopf <pascalwithopf1@gmail.com>

Target audience are developers and users who need to debug an error with tests. For debugging with rsyslog.conf see *troubleshooting*.

Debugging with tests

When you want to solve a specific problem you will probably create a test
and want to debug with it instead of configuring rsyslog. If you want to
write a debug log you need to open the file **../rsyslog/tests/diag.sh**
and delete the **#** in front of the two lines:

export RSYSLOG_DEBUG="debug nologfuncflow noprintmutexaction nostdout"
export RSYSLOG_DEBUGLOG="log"

A debug log will be writen now, but remember to put the **#** back again
before commiting your changes. Otherwise it won't work.

Memory debugging

You can't use multiple memory debugger at the same time. This will resort
in errors. Also remember to undo all changes in diag.sh after you are done,
because it will also resort in errors if you commit them with your work.

Valgrind

If you want to use Valgrind you need to enable it for tests.
To do that open the file **../rsyslog/tests/diag.sh** and delete the **#**
in front of the line:
valgrind="valgrind –malloc-fill=ff –free-fill=fe –log-fd=1"
This will enable valgrind and you will have extra debugging in your test-suite.log file.

Address sanitizer

If you want to use adress sanitizer you need to set your CFLAGS. Use this command:
export CFLAGS="-g -fsanitizer=address"
After this is done you need to configure and build rsyslog again, otherwise it won't work.

rsylog code style

Note: code style is still under construction. This guide lists some basic style requirements.

Code that does not match the code style guide will not pass CI testing.

The following is required right now:

- we use ANSCI C99
- indention is done with tabs, not spaces
- trailing whitespace in lines is not permitted
- lines longer than 140 characters are not permitted; also, it is advised to keep lines shorter than 120 characters unless there is a good reason. The 120 char rule is not enforced, but everything over 140 chars is rejected and must be reformatted.

Writing Rsyslog Output Plugins

This page is the begin of some developer documentation for writing output plugins. Doing so is quite easy (and that was a design goal), but there currently is only sparse documentation on the process available. I was tempted NOT to write this guide here because I know I will most probably not be able to write a complete guide.

However, I finally concluded that it may be better to have some information and pointers than to have nothing.

Getting Started and Samples

The best to get started with rsyslog plugin development is by looking at existing plugins. All that start with "om" are **o**utput **m**odules. That means they are primarily thought of being message sinks. In theory, however, output plugins may aggergate other functionality, too. Nobody has taken this route so far so if you would like to do that, it is highly suggested to post your plan on the rsyslog mailing list, first (so that we can offer advise).

The rsyslog distribution tarball contains the omstdout plugin which is extremely well targeted for getting started. Just note that this plugin itself is not meant for production use. But it is very simplistic and so a really good starting point to grasp the core ideas.

In any case, you should also read the comments in ./runtime/module-template.h. Output plugins are build based on a large set of code-generating macros. These macros handle most of the plumbing needed by the interface. As long as

no special callback to rsyslog is needed (it typically is not), an output plugin does not really need to be aware that it is executed by rsyslog. As a plug-in programmer, you can (in most cases) "code as usual". However, all macros and entry points need to be provided and thus reading the code comments in the files mentioned is highly suggested.

For testing, you need rsyslog's debugging support. Some useful information is given in "troubleshooting rsyslog from the doc set.

Special Topics

Threading

Rsyslog uses massive parallel processing and multithreading. However, a plugin's entry points are guaranteed to be never called concurrently **for the same action**. That means your plugin must be able to be called concurrently by two or more threads, but you can be sure that for the same instance no concurrent calls happen. This is guaranteed by the interface specification and the rsyslog core guards against multiple concurrent calls. An instance, in simple words, is one that shares a single instanceData structure.

So as long as you do not mess around with global data, you do not need to think about multithreading (and can apply a purely sequential programming methodology).

Please note that during the configuration parsing stage of execution, access to global variables for the configuration system is safe. In that stage, the core will only call sequentially into the plugin.

Getting Message Data

The doAction() entry point of your plugin is provided with messages to be processed. It will only be activated after filtering and all other conditions, so you do not need to apply any other conditional but can simply process the message.

Note that you do NOT receive the full internal representation of the message object. There are various (including historical) reasons for this and, among others, this is a design decision based on security.

Your plugin will only receive what the end user has configured in a $template statement. However, starting with 4.1.6, there are two ways of receiving the template content. The default mode, and in most cases sufficient and optimal, is to receive a single string with the expanded template. As I said, this is usually optimal, think about writing things to files, emailing content or forwarding it.

The important philosophy is that a plugin should **never** reformat any of such strings - that would either remove the user's ability to fully control message formats or it would lead to duplicating code that is already present in the core. If you need some formatting that is not yet present in the core, suggest it to the rsyslog project, best done by sending a patch ;), and we will try hard to get it into the core (so far, we could accept all such suggestions - no promise, though).

If a single string seems not suitable for your application, the plugin can also request access to the template components. The typical use case seems to be databases, where you would like to access properties via specific fields. With that mode, you receive a char ** array, where each array element points to one field from the template (from left to right). Fields start at array index 0 and a NULL pointer means you have reached the end of the array (the typical Unix "poor man's linked list in an array" design). Note, however, that each of the individual components is a string. It is not a date stamp, number or whatever, but a string. This is because rsyslog processes strings (from a high-level design look at it) and so this is the natural data type. Feel free to convert to whatever you need, but keep in mind that malformed packets may have lead to field contents you'd never expected...

If you like to use the array-based parameter passing method, think that it is only available in rsyslog 4.1.6 and above. If you can accept that your plugin will not be working with previous versions, you do not need to handle pre 4.1.6 cases. However, it would be "nice" if you shut down yourself in these cases - otherwise the older rsyslog core engine will pass you a string where you expect the array of pointers, what most probably results in a segfault. To check whether or not the core supports the functionality, you can use this code sequence:

```
BEGINmodInit()
    rsRetVal localRet;
    rsRetVal (*pomsrGetSupportedTplOpts)(unsigned long *pOpts);
    unsigned long opts;
    int bArrayPassingSupported;     /* does core support template passing as an array?
↪ */
CODESTARTmodInit
    *ipIFVersProvided = CURR_MOD_IF_VERSION; /* we only support the current interface␣
↪specification */
CODEmodInit_QueryRegCFSLineHdlr
    /* check if the rsyslog core supports parameter passing code */
    bArrayPassingSupported = 0;
    localRet = pHostQueryEtryPt((uchar*)"OMSRgetSupportedTplOpts", &
↪pomsrGetSupportedTplOpts);
    if(localRet == RS_RET_OK) {
        /* found entry point, so let's see if core supports array passing */
        CHKiRet((*pomsrGetSupportedTplOpts)(&opts));
        if(opts & OMSR_TPL_AS_ARRAY)
            bArrayPassingSupported = 1;
    } else if(localRet != RS_RET_ENTRY_POINT_NOT_FOUND) {
        ABORT_FINALIZE(localRet); /* Something else went wrong, what is not␣
↪acceptable */
    }
    DBGPRINTF("omstdout: array-passing is %ssupported by rsyslog core.\n",␣
↪bArrayPassingSupported ? "" : "not ");

    if(!bArrayPassingSupported) {
        DBGPRINTF("rsyslog core too old, shutting down this plug-in\n");
        ABORT_FINALIZE(RS_RET_ERR);
    }
```

The code first checks if the core supports the OMSRgetSupportedTplOpts() API (which is also not present in all versions!) and, if so, queries the core if the OMSR_TPL_AS_ARRAY mode is supported. If either does not exits, the core is too old for this functionality. The sample snippet above then shuts down, but a plugin may instead just do things different. In omstdout, you can see how a plugin may deal with the situation.

In any case, it is recommended that at least a graceful shutdown is made and the array-passing capability not blindly be used. In such cases, we can not guard the plugin from segfaulting and if the plugin (as currently always) is run within rsyslog's process space, that results in a segfault for rsyslog. So do not do this.

Another possible mode is OMSR_TPL_AS_JSON, where instead of the template a json-c memory object tree is passed to the module. The module can extract data via json-c API calls. It MUST NOT modify the provided structure. This mode is primarily aimed at plugins that need to process tree-like data, as found for example in MongoDB or ElasticSearch.

Batching of Messages

Starting with rsyslog 4.3.x, batching of output messages is supported. Previously, only a single-message interface was supported.

With the **single message** plugin interface, each message is passed via a separate call to the plugin. Most importantly, the rsyslog engine assumes that each call to the plugin is a complete transaction and as such assumes that messages be properly committed after the plugin returns to the engine.

With the **batching** interface, rsyslog employs something along the line of "transactions". Obviously, the rsyslog core can not make non-transactional outputs to be fully transactional. But what it can is support that the output tells the core which messages have been committed by the output and which not yet. The core can than take care of those

uncommitted messages when problems occur. For example, if a plugin has received 50 messages but not yet told the core that it committed them, and then returns an error state, the core assumes that all these 50 messages were **not** written to the output. The core then requeues all 50 messages and does the usual retry processing. Once the output plugin tells the core that it is ready again to accept messages, the rsyslog core will provide it with these 50 not yet committed messages again (actually, at this point, the rsyslog core no longer knows that it is re-submitting the messages). If, in contrary, the plugin had told rsyslog that 40 of these 50 messages were committed (before it failed), then only 10 would have been requeued and resubmitted.

In order to provide an efficient implementation, there are some (mild) constraints in that transactional model: first of all, rsyslog itself specifies the ultimate transaction boundaries. That is, it tells the plugin when a transaction begins and when it must finish. The plugin is free to commit messages in between, but it **must** commit all work done when the core tells it that the transaction ends. All messages passed in between a begin and end transaction notification are called a batch of messages. They are passed in one by one, just as without transaction support. Note that batch sizes are variable within the range of 1 to a user configured maximum limit. Most importantly, that means that plugins may receive batches of single messages, so they are required to commit each message individually. If the plugin tries to be "smarter" than the rsyslog engine and does not commit messages in those cases (for example), the plugin puts message stream integrity at risk: once rsyslog has notified the plugin of transaction end, it discards all messages as it considers them committed and save. If now something goes wrong, the rsyslog core does not try to recover lost messages (and keep in mind that "goes wrong" includes such uncontrollable things like connection loss to a database server). So it is highly recommended to fully abide to the plugin interface details, even though you may think you can do it better. The second reason for that is that the core engine will have configuration settings that enable the user to tune commit rate to their use-case specific needs. And, as a relief: why would rsyslog ever decide to use batches of one? There is a trivial case and that is when we have very low activity so that no queue of messages builds up, in which case it makes sense to commit work as it arrives. (As a side-note, there are some valid cases where a timeout-based commit feature makes sense. This is also under evaluation and, once decided, the core will offer an interface plus a way to preserve message stream integrity for properly-crafted plugins).

The second restriction is that if a plugin makes commits in between (what is perfectly legal) those commits must be in-order. So if a commit is made for message ten out of 50, this means that messages one to nine are also committed. It would be possible to remove this restriction, but we have decided to deliberately introduce it to simplify things.

Output Plugin Transaction Interface

In order to keep compatible with existing output plugins (and because it introduces no complexity), the transactional plugin interface is build on the traditional non-transactional one. Well... actually the traditional interface was transactional since its introduction, in the sense that each message was processed in its own transaction.

So the current `doAction() entry point can be considered to have this structure (from the transactional interface point of view):`

```
doAction()
    {
    beginTransaction()
    ProcessMessage()
    endTransaction()
    }
```

For the **transactional interface**, we now move these implicit `beginTransaction()` and `endTransaction(()` call out of the message processing body, resulting is such a structure:

```
beginTransaction()
    {
    /* prepare for transaction */
    }

doAction()
```

```
    {
    ProcessMessage()
    /* maybe do partial commits */
    }

endTransaction()
    {
    /* commit (rest of) batch */
    }
```

And this calling structure actually is the transactional interface! It is as simple as this. For the new interface, the core calls a `beginTransaction()` entry point inside the plugin at the start of the batch. Similarly, the core call `endTransaction()` at the end of the batch. The plugin must implement these entry points according to its needs.

But how does the core know when to use the old or the new calling interface? This is rather easy: when loading a plugin, the core queries the plugin for the `beginTransaction()` and `endTransaction()` entry points. If the plugin supports these, the new interface is used. If the plugin does not support them, the old interface is used and rsyslog implies that a commit is done after each message. Note that there is no special "downlevel" handling necessary to support this. In the case of the non-transactional interface, rsyslog considers each completed call to `doAction` as partial commit up to the current message. So implementation inside the core is very straightforward.

Actually, **we recommend that the transactional entry points only be defined by those plugins that actually need them**. All others should not define them in which case the default commit behaviour inside rsyslog will apply (thus removing complexity from the plugin).

In order to support partial commits, special return codes must be defined for `doAction`. All those return codes mean that processing completed successfully. But they convey additional information about the commit status as follows:

RS_RET_OK	The record and all previous inside the batch has been commited. *Note:* this definition is what makes integrating plugins without the transaction being/end calls so easy - this is the traditional "success" return state and if every call returns it, there is no need for actually calling `endTransaction()`, because there is no transaction open).
RS_RET_DEFER_COMMIT	The record has been processed, but is not yet commited. This is the expected state for transactional-aware plugins.
RS_RET_PREVIOUS_COMMITTED	The previous record inside the batch has been committed, but the current one not yet. This state is introduced to support sources that fill up buffers and commit once a buffer is completely filled. That may occur halfway in the next record, so it may be important to be able to tell the engine the everything up to the previouos record is commited

Note that the typical **calling cycle** is `beginTransaction()`, followed by *n* times `doAction()` followed by `endTransaction()`. However, if either `beginTransaction()` or `doAction()` return back an error state (including RS_RET_SUSPENDED), then the transaction is considered aborted. In result, the remaining calls in this cycle (e.g. `endTransaction()`) are never made and a new cycle (starting with `beginTransaction()` is begun when processing resumes. So an output plugin must expect and handle those partial cycles gracefully.

The question remains how can a plugin know if the core supports batching? First of all, even if the engine would not know it, the plugin would return with RS_RET_DEFER_COMMIT, what then would be treated as an error by the engine. This would effectively disable the output, but cause no further harm (but may be harm enough in itself).

The real solution is to enable the plugin to query the rsyslog core if this feature is supported or not. At the time of the introduction of batching, no such query-interface exists. So we introduce it with that release. What the means is if a rsyslog core can not provide this query interface, it is a core that was build before batching support was available. So the absence of a query interface indicates that the transactional interface is not available. One might now be tempted to think there is no need to do the actual check, but is is recommended to ask the rsyslog engine explicitly if the transactional interface is present and will be honored. This enables us to create versions in the future which have, for whatever reason we do not yet know, no support for this interface.

The logic to do these checks is contained in the `INITChkCoreFeature` macro, which can be used as follows:

```
INITChkCoreFeature(bCoreSupportsBatching, CORE_FEATURE_BATCHING);
```

Here, bCoreSupportsBatching is a plugin-defined integer which after execution is 1 if batches (and thus the transactional interface) is supported and 0 otherwise. CORE_FEATURE_BATCHING is the feature we are interested in. Future versions of rsyslog may contain additional feature-test-macros (you can see all of them in ./runtime/rsyslog.h).

Note that the ompsql output plugin supports transactional mode in a hybrid way and thus can be considered good example code.

Open Issues

- Processing errors handling
- reliable re-queue during error handling and queue termination

Licensing

From the rsyslog point of view, plugins constitute separate projects. As such, we think plugins are not required to be compatible with GPLv3. However, this is no legal advise. If you intend to release something under a non-GPLV3 compatible license it is probably best to consult with your lawyer.

Most importantly, and this is definite, the rsyslog team does not expect or require you to contribute your plugin to the rsyslog project (but of course we are happy if you do).

Copyright

The rsyslog queue object

This page reflects the status as of 2008-01-17. The documentation is still incomplete. Target audience is developers and users who would like to get an in-depth understanding of queues as used in rsyslog.

Please note that this document is outdated and does not longer reflect the specifics of the queue object. However, I have decided to leave it in the doc set, as the overall picture provided still is quite OK. I intend to update this document somewhat later when I have reached the "store-and-forward" milestone.

Some definitions

A queue is DA-enabled if it is configured to use disk-assisted mode when there is need to. A queue is in DA mode (or DA run mode), when it actually runs disk assisted.

Implementation Details

Disk-Assisted Mode

Memory-Type queues may utilize disk-assisted (DA) mode. DA mode is enabled whenever a queue file name prefix is provided. This is called DA-enabled mode. If DA-enabled, the queue operates as a regular memory queue until a high water mark is reached. If that happens, the queue activates disk assistance (called "runs disk assisted" or "runs DA" - you can find that often in source file comments). To do so, it creates a helper queue instance (the DA queue). At that point, there are two queues running - the primary queue's consumer changes to a shuffle-to-DA-queue consumer and the original primary consumer is assigned to the DA queue. Existing and new messages are spooled to the disk queue, where the DA worker takes them from and passes them for execution to the actual consumer. In essence, the primary queue has now become a memory buffer for the DA queue. The primary queue will be drained until a low water mark is reached. At that point, processing is held. New messages enqueued to the primary queue will not be processed but kept in memory. Processing resumes when either the high water mark is reached again or the DA queue indicates it is empty. If the DA queue is empty, it is shut down and processing of the primary queue continues as a regular in-memory queue (aka "DA mode is shut down"). The whole thing iterates once the high water mark is hit again.

There is one special case: if the primary queue is shut down and could not finish processing all messages within the configured timeout periods, the DA queue is instantiated to take up the remaining messages. These will be preserved and be processed during the next run. During that period, the DA queue runs in "enqueue-only" mode and does not execute any consumer. Draining the primary queue is typically very fast. If that behaviour is not desired, it can be turned of via parameters. In that case, any remaining in-memory messages are lost.

Due to the fact that when running DA two queues work closely together and worker threads (including the DA worker) may shut down at any time (due to timeout), processing synchronization and startup and shutdown is somewhat complex. I'll outline the exact conditions and steps down here. I also do this so that I know clearly what to develop to, so please be patient if the information is a bit too in-depth ;)

DA Run Mode Initialization

Three cases:

1. any time during queueEnqObj() when the high water mark is hit
2. at queue startup if there is an on-disk queue present (presence of QI file indicates presence of queue data)
3. at queue shutdown if remaining in-memory data needs to be persisted to disk

In **case 1**, the worker pool is running. When switching to DA mode, all regular workers are sent termination commands. The DA worker is initiated. Regular workers may run in parallel to the DA worker until they terminate. Regular workers shall terminate as soon as their current consumer has completed. They shall not execute the DA consumer.

In **case 2**, the worker pool is not yet running and is NOT started. The DA worker is initiated.

In **case 3**, the worker pool is already shut down. The DA worker is initiated. The DA queue runs in enqueue-only mode.

In all cases, the DA worker starts up and checks if DA mode is already fully initialized. If not, it initializes it, what most importantly means construction of the queue.

Then, regular worker processing is carried out. That is, the queue worker will wait on empty queue and terminate after an timeout. However, If any message is received, the DA consumer is executed. That consumer checks the low water mark. If the low water mark is reached, it stops processing until either the high water mark is reached again or the DA queue indicates it is empty (there is a pthread_cond_t for this synchronization).

In theory, a **case-2** startup could lead to the worker becoming inactive and terminating while waiting on the primary queue to fill. In practice, this is highly unlikely (but only for the main message queue) because rsyslog issues a startup

message. HOWEVER, we can not rely on that, it would introduce a race. If the primary rsyslog thread (the one that issues the message) is scheduled very late and there is a low inactivity timeout for queue workers, the queue worker may terminate before the startup message is issued. And if the on-disk queue holds only a few messages, it may become empty before the DA worker is re-initiated again. So it is possible that the DA run mode termination criteria occurs while no DA worker is running on the primary queue.

In cases 1 and 3, the DA worker can never become inactive without hitting the DA shutdown criteria. In **case 1**, it either shuffles messages from the primary to the DA queue or it waits because it has the hit low water mark.

In **case 3**, it always shuffles messages between the queues (because, that's the sole purpose of that run). In order for this to happen, the high water mark has been set to the value of 1 when DA run mode has been initialized. This ensures that the regular logic can be applied to drain the primary queue. To prevent a hold due to reaching the low water mark, that mark must be changed to 0 before the DA worker starts.

DA Run Mode Shutdown

In essence, DA run mode is terminated when the DA queue is empty and the primary worker queue size is below the high water mark. It is also terminated when the primary queue is shut down. The decision to switch back to regular (non-DA) run mode is typically made by the DA worker. If it switches, the DA queue is destructed and the regular worker pool is restarted. In some cases, the queue shutdown process may initiate the "switch" (in this case more or less a clean shutdown of the DA queue).

One might think that it would be more natural for the DA queue to detect being idle and shut down itself. However, there are some issues associated with that. Most importantly, all queue worker threads need to be shut down during queue destruction. Only after that has happened, final destruction steps can happen (else we would have a myriad of races). However, it is the DA queues worker thread that detects it is empty (empty queue detection always happens at the consumer side and must so). That would lead to the DA queue worker thread to initiate DA queue destruction which in turn would lead to that very same thread being canceled (because workers must shut down before the queue can be destructed). Obviously, this does not work out (and I didn't even mention the other issues - so let's forget about it). As such, the thread that enqueues messages must destruct the queue - and that is the primary queue's DA worker thread.

There are some subtleties due to thread synchronization and the fact that the DA consumer may not be running (in a **case-2 startup**). So it is not trivial to reliably change the queue back from DA run mode to regular run mode. The priority is a clean switch. We accept the fact that there may be situations where we cleanly shut down DA run mode, just to re-enable it with the very next message being enqueued. While unlikely, this will happen from time to time and is considered perfectly legal. We can't predict the future and it would introduce too great complexity to try to do something against that (that would most probably even lead to worse performance under regular conditions).

The primary queue's DA worker thread may wait at two different places:

1. after reaching the low water mark and waiting for either high water or DA queue empty
2. at the regular pthread_cond_wait() on an empty primary queue

Case 2 is unlikely, but may happen (see info above on a case 2 startup).

The DA worker may also not wait at all, because it is actively executing and shuffling messages between the queues. In that case, however, the program flow passes both of the two wait conditions but simply does not wait.

Finally, the DA worker may be inactive(again, with a case-2 startup). In that case no work(er) at all is executed. Most importantly, without the DA worker being active, nobody will ever detect the need to change back to regular mode. If we have this situation, the very next message enqueued will cause the switch, because then the DA run mode shutdown criteria is met. However, it may take close to eternal for this message to arrive. During that time, disk and memory resources for the DA queue remain allocated. This also leaves processing in a sub-optimal state and it may take longer than necessary to switch back to regular queue mode when a message burst happens. In extreme cases, this could even lead to shutdown of DA run mode, which takes so long that the high water mark is passed and DA

run mode is immediately re-initialized - while with an immediate switch, the message burst may have been able to be processed by the in-memory queue without DA support.

So in short, it is desirable switch to regular run mode as soon as possible. To do this, we need an active DA worker. The easy solution is to initiate DA worker startup from the DA queue's worker once it detects empty condition. To do so, the DA queue's worker must call into a "*DA worker startup initiation*" routine inside the main queue. As a reminder, the DA worker will most probably not receive the "DA queue empty" signal in that case, because it will be long sent (in most cases) before the DA worker even waits for it. So **it is vital that DA run mode termination checks be done in the DA worker before it goes into any wait condition**.

Please note that the "*DA worker startup initiation*" routine may be called concurrently from multiple initiators. **To prevent a race, it must be guarded by the queue mutex**and return without any action (and no error code!) if the DA worker is already initiated.

All other cases can be handled by checking the termination criteria immediately at the start of the worker and then once again for each run. The logic follows this simplified flow diagram:

Some of the more subtle aspects of worker processing (e.g. enqueue thread signaling and other fine things) have been left out in order to get the big picture. What is called "check DA mode switchback..." right after "worker init" is actually a check for the worker's termination criteria. Typically, **the worker termination criteria is a shutdown request**. However, **for a DA worker, termination is also requested if the queue size is below the high water mark AND the DA queue is empty**. There is also a third termination criteria and it is not even on the chart: that is the inactivity timeout, which exists in all modes. Note that while the inactivity timeout shuts down a thread, it logically does not terminate the worker pool (or DA worker): workers are restarted on an as-needed basis. However, inactivity timeouts are very important because they require us to restart workers in some situations where we may expect a running one. So always keep them on your mind.

Queue Destruction

Now let's consider **the case of destruction of the primary queue.**During destruction, our focus is on loosing as few messages as possible. If the queue is not DA-enabled, there is nothing but the configured timeouts to handle that situation. However, with a DA-enabled queue there are more options.

If the queue is DA-enabled, it may be *configured to persist messages to disk before it is terminated*. In that case, loss of messages never occurs (at the price of a potentially lengthy shutdown). Even if that setting is not applied, the queue should drain as many messages as possible to the disk. For that reason, it makes no sense to wait on a low water mark. Also, if the queue is already in DA run mode, it does not make any sense to switch back to regular run mode during termination and then try to process some messages via the regular consumer. It is much more appropriate the try completely drain the queue during the remaining timeout period. For the same reason, it is preferred that no new consumers be activated (via the DA queue's worker), as they only cost valuable CPU cycles and, more importantly, would potentially be long(er)-running and possibly be needed to be cancelled. To prevent all of that, **queue parameters are changed for DA-enabled queues:** the high water mark is to 1 and the low water mark to 0 on the primary queue. The DA queue is commanded to run in enqueue-only mode. If the primary queue is *configured to persist messages to disk before it is terminated*, its SHUTDOWN timeout is changed to to eternal. These parameters will cause the queue to drain as much as possible to disk (and they may cause a case 3 DA run mode initiation). Please note that once the primary queue has been drained, the DA queue's worker will automatically switch back to regular (non-DA) run mode. **It must be ensured that no worker cancellation occurs during that switchback**. Please note that the queue may not switch back to regular run mode if it is not *configured to persist messages to disk before it is terminated*. In order to apply the new parameters, **worker threads must be awakened.** Remember we may not be in DA run mode at this stage. In that case, the regular workers must be awakened, which then will switch to DA run mode. No worker may be active, in that case one must be initiated. If in DA run mode and the DA worker is inactive, the "*DA worker startup initiation*" must be called to activate it. That routine ensures only one DA worker is started even with multiple concurrent callers - this may be the case here. The DA queue's worker may have requested DA worker startup in order to terminate on empty queue (which will probably not be honored as we have changed the low water mark).

After all this is done, the queue destructor requests termination of the queue's worker threads. It will use the normal timeouts and potentially cancel too-long running worker threads. **The shutdown process must ensure that all workers reach running state before they are commanded to terminate**. Otherwise it may run into a race condition that could lead to a false shutdown with workers running asynchronously. As a few workers may have just been started to initialize (to apply new parameter settings), the probability for this race condition is extremely high, especially on single-CPU systems.

After all workers have been shut down (or cancelled), the queue may still be in DA run mode. If so, this must be terminated, which now can simply be done by destructing the DA queue object. This is not a real switchback to regular run mode, but that doesn't matter because the queue object will soon be gone away.

Finally, the queue is mostly shut down and ready to be actually destructed. As a last try, the queuePersists() entry point is called. It is used to persists a non-DA-enabled queue in whatever way is possible for that queue. There may be no implementation for the specific queue type. Please note that this is not just a theoretical construct. This is an extremely important code path when the DA queue itself is destructed. Remember that it is a queue object in its own right. The DA queue is obviously not DA-enabled, so it calls into queuePersists() during its destruction - this is what enables us to persist the disk queue!

After that point, left over queue resources (mutexes, dynamic memory, ...) are freed and the queue object is actually destructed.

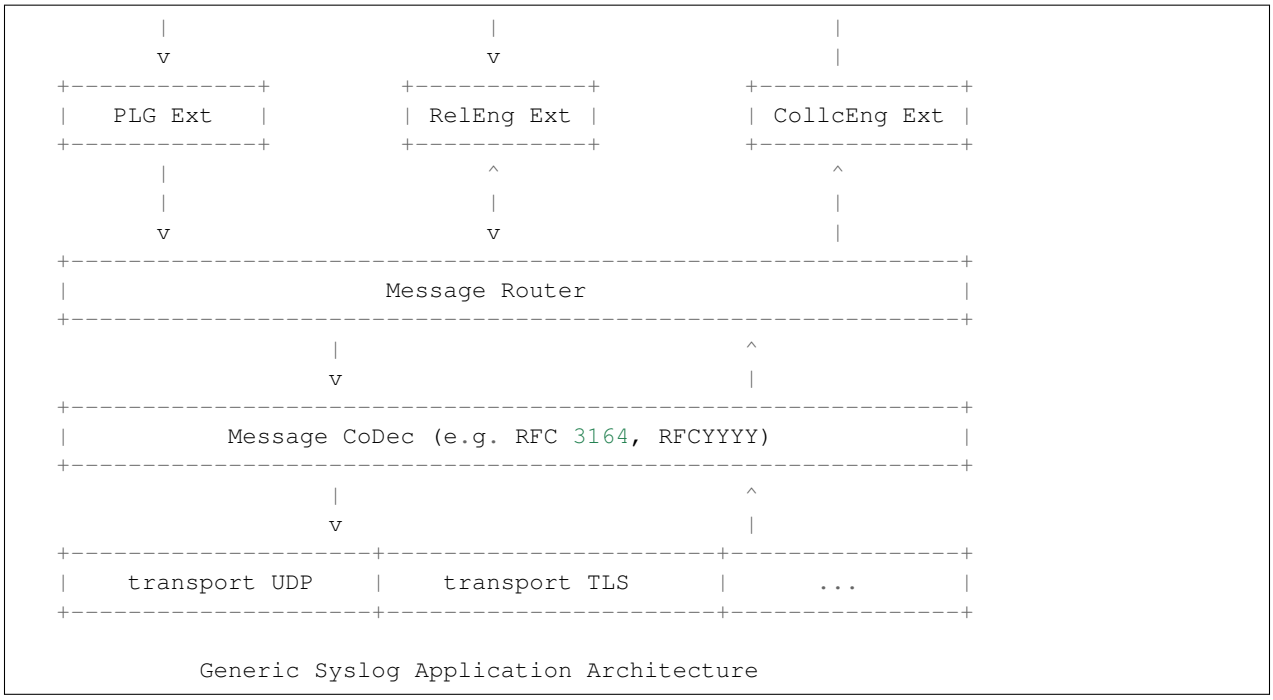
Copyright

Generic design of a syslogd

Written 2007-04-10 by Rainer Gerhards

The text below describes a generic approach on how a syslogd can be implemented. I created this description for some other project, where it was not used. Instead of throwing it away, I thought it would be a good addition to the rsyslog documentation. While rsyslog differs in details from the description below, it is sufficiently close to it. Further development of rsyslog will probably match it even closer to the description.

If you intend to read the rsyslog source code, I recommend reading this document here first. You will not find the same names and not all of the concepts inside rsyslog. However, I think your understanding will benefit from knowing the generic architecture.



Generic Syslog Application Architecture

- A "syslog application" is an application whose purpose is the processing of syslog messages. It may be part of a larger application with a broader purpose. An example: a database application might come with its own syslog send subsystem and not go through a central syslog application. In the sense of this document, that application is called a "syslog application" even though a casual observer might correctly call it a database application and may not even know that it supports sending of syslog messages.
- Payload is the information that is to be conveyed. Payload by itself may have any format and is totally independent from to format specified in this document. The "Message CoDec" of the syslog application will bring it into the required format.
- Payload Originators ("PLOrig") are the original creators of payload. Typically, these are application programs.
- A "Remote PLOrig" is a payload originator residing in a different application than the syslog application itself. That application may reside on a different machine and may talk to the syslog application via RPC.
- A "PLOrig" is a payload originator residing within the syslog application itself. Typically, this PLOrig emits syslog application startup, shutdown, error and status log messages.
- A "GWI" is a inbound gateway. For example, a SNMP-to-syslog gateway may receive SNMP messages and translate them into syslog.
- The ellipsis after "GWI" indicates that there are potentially a variety of different other ways to originally generate payload.
- A "PLGenerator" is a payload generator. It takes the information from the payload-generating source and integrates it into the syslog subsystem of the application. This is a highly theoretical concept. In practice, there may not actually be any such component. Instead, the payload generators (or other parts like the GWI) may talk directly to the syslog subsystem. Conceptually, the "PLGenerator" is the first component where the information is actually syslog content.
- A "PLG Ext" is a payload generator extension. It is used to modify the syslog information. An example of a "PLG Ext" might be the addition of cryptographic signatures to the syslog information.
- A "Message Router" is a component that accepts in- and outbound syslog information and routes it to the proper next destination inside the syslog application. The routing information itself is expected to be learnt by operator configuration.

- A "Message CoDec" is the message encoder/decoder. The encoder takes syslog information and encodes them into the required format for a syslog message. The decoder takes a syslog message and decodes it into syslog information. Codecs for multiple syslog formats may be present inside a single syslog application.
- A transport (UDP, TLS, yet-to-be-defined ones) sends and receives syslog messages. Multiple transports may be used by a single syslog application at the same time. A single transport instance may be used for both sending and receiving. Alternatively, a single instance might be used for sending and receiving exclusively. Multiple instances may be used for different listener ports and receivers.
- A "RelayEng" is the relaying engine. It provides functionality necessary for receiving syslog information and sending it to another syslog application.
- A "RelEng Ext" is an extension that processes syslog information as it enters or exits a RelayEng. An example of such a component might be a relay cryptographically signing received syslog messages. Such a function might be useful to guarantee authenticity starting from a given point inside a relay chain.
- A "CollectorEng" is a collector engine. At this component, syslog information leaves the syslog system and is translated into some other form. After the CollectorEng, the information is no longer defined to be of native syslog type.
- A "CollcEng Ext" is a collector engine extension. It modifies syslog information before it is passed on to the CollectorEng. An example for this might be the verification of cryptographically signed syslog message information. Please note that another implementation approach would be to do the verification outside of the syslog application or in a stage after "CollectorEng".
- A "GWO" is an outbound gateway. An example of this might be the forwarding of syslog information via SNMP or SMTP. Please note that when a GWO directly connects to a GWI on a different syslog application, no native exchange of syslog information takes place. Instead, the native protocol of these gateways (e.g. SNMP) is used. The syslog information is embedded inside that protocol. Depending on protocol and gateway implementation, some of the native syslog information might be lost.
- A "Store" is any way to persistently store the extracted syslog information, e.g. to the file system or to a data base.
- "Disc" means the discarding of messages. Operators often find it useful to discard noise messages and so most syslog applications contain a way to do that.
- The ellipsis after "Disc" indicates that there are potentially a variety of different other ways to consume syslog information.
- There may be multiple instances of each of the described components in a single syslog application.
- A syslog application is made up of all or some of the above mentioned components.

Historical Documents

This part of the documentation set contains historical documents which are still of interest or may be useful in some more exotic environments.

Using php-syslog-ng with rsyslog

Written by Rainer Gerhards *(2005-08-04)*

Note: it has been reported that this guide is somewhat outdated. Most importantly, this guide is for the **original** php-syslog-ng and **cannot** be used for it successor logzilla. Please use the guide with care. Also, please note that **rsyslog's "native" web frontend is** Adiscon LogAnalyzer, which provides best integration and a lot of extra functionality.

Abstract

In this paper, I describe how to use php-syslog-ng **with** rsyslogd. Php-syslog-ng is a popular web interface to syslog data. Its name stem from the fact that it usually picks up its data from a database created by syslog-ng and some helper scripts. However, there is nothing syslog-ng specific in the database. With rsyslogd's high customizability, it is easy to write to a syslog-ng like schema. I will tell you how to do this, enabling you to use php-syslog-ng as a front-end for rsyslogd - or save the hassle with syslog-ng database configuration and simply go ahead and use rsyslogd instead.*

Overall System Setup

The setup is pretty straightforward. Basically, php-syslog-ng's interface to the syslogd is the database. We use the schema that php-syslog-ng expects and make rsyslogd write to it in its format. Because of this, php-syslog-ng does not even know there is no syslog-ng present.

Setting up the system

For php-syslog-ng, you can follow its usual setup instructions. Just skip any steps refering to configure syslog-ng. Make sure you create the database schema in MySQL. As of this writing, the expected schema can be created via this script:

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

Please note that at the time you are reading this paper, the schema might have changed. Check for any differences. As we customize rsyslogd to the schema, it is vital to have the correct one. If this paper is outdated, let me know so that I can fix it.

Once this schema is created, we simply instruct rsyslogd to store received data in it. I wont go into too much detail here. If you are interested in some more details, you might find my paper "Writing syslog messages to MySQL" worth reading. For this article, we simply modify rsyslog.confso that it writes to the database. That is easy. Just these two lines are needed:

```
$template syslog-ng,"insert into logs(host, facility, priority, tag, date, time, msg)␣
↪values ('%HOSTNAME%', %syslogfacility%, %syslogpriority%, '%syslogtag%', '
↪%timereported:::date-mysql%', '%timereported:::date-mysql%', '%msg%')", SQL
*.*,           mysql-server,syslog,user,pass;syslog-ng
```

These are just **two** lines. I have color-coded them so that you see what belongs together (the colors have no other meaning). The green line is the actual SQL statement being used to take care of the syslog-ng schema. Rsyslogd allows you to fully control the statement sent to the database. This allows you to write to any database format, including your homegrown one (if you so desire). Please note that there is a small inefficiency in our current usage: the `'%timereported:::date-mysql%'` property is used for both the time and the date (if you wonder about what all these funny characters mean, see the rsyslogd property replacer manual) . We could have extracted just the date and time parts of the respective properties. However, this is more complicated and also adds processing time to rsyslogd's processing (substrings must be extracted). So we take a full mysql-formatted timestamp and supply it to MySQL. The sql engine in turn discards the unneeded part. It works pretty well. As of my understanding, the inefficiency of discarding the unneeded part in MySQL is lower than the effciency gain from using the full timestamp in rsyslogd. So it is most probably the best solution.

Please note that rsyslogd knows two different timestamp properties: one is timereported, used here. It is the timestamp from the message itself. Sometimes that is a good choice, in other cases not. It depends on your environment. The other one is the timegenerated property. This is the time when rsyslogd received the message. For obvious reasons, that timestamp is consistent, even when your devices are in multiple time zones or their clocks are off. However, it is not "the real thing". It's your choice which one you prefer. If you prefer timegenerated ... simply use it ;)

The line in red tells rsyslogd which messages to log and where to store it. The "*.*" selects all messages. You can use standard syslog selector line filters here if you do not like to see everything in your database. The ">" tells rsyslogd that a MySQL connection must be established. Then, "mysql-server" is the name or IP address of the server machine, "syslog" is the database name (default from the schema) and "user" and "pass" are the logon credentials. Use a user with low privileges, insert into the logs table is sufficient. "syslog-ng" is the template name and tells rsyslogd to use the SQL statement shown above.

Once you have made the changes, all you need to do is restart rsyslogd. Then, you should see syslog messages flow into your database - and show up in php-syslog-ng.

Conclusion

With minumal effort, you can use php-syslog-ng together with rsyslogd. For those unfamiliar with syslog-ng, this configuration is probably easier to set up then switching to syslog-ng. For existing rsyslogd users, php-syslog-ng might be a nice add-on to their logging infrastructure.

Please note that the MonitorWare family (to which rsyslog belongs) also offers a web-interface: Adiscon LogAnalyzer. From my point of view, obviously, **phpLogCon is the more natural choice for a web interface to be used together with rsyslog**. It also offers superb functionality and provides, for example,native display of Windows event log entries. I have set up a demo server., You can have a peek at it without installing anything.

Feedback Requested

I would appreciate feedback on this paper. If you have additional ideas, comments or find bugs, please let me know.

References and Additional Material

- php-syslog-ng

Revision History

- 2005-08-04 * Rainer Gerhards * initial version created

Copyright

SSL Encrypting Syslog with Stunnel

Written by Rainer Gerhards *(2005-07-22)*

HISTORICAL DOCUMENT

Note: this is an outdated HISTORICAL document. A much better description on securing syslog with TLS is available.

Abstract

In this paper, I describe how to encrypt syslog **messages on the network.** Encryption is vital to keep the confidiental content of syslog messages secure. I describe the overall approach and provide an HOWTO do it with the help of rsyslogd and stunnel.*

Please note that starting with rsyslog 3.19.0, rsyslog provides native TLS/SSL encryption without the need of stunnel. I strongly recomend to use that feature instead of stunnel. The stunnel documentation here is mostly provided for backwards compatibility. New deployments are advised to use native TLS mode.**

Background

Syslog is a clear-text protocol. That means anyone with a sniffer can have a peek at your data. In some environments, this is no problem at all. In others, it is a huge setback, probably even preventing deployment of syslog solutions. Thankfully, there is an easy way to encrypt syslog communication. I will describe one approach in this paper.

The most straightforward solution would be that the syslogd itself encrypts messages. Unfortuantely, encryption is only standardized in RFC 3195. But there is currently no syslogd that implements RFC 3195's encryption features, so this route leads to nothing. Another approach would be to use vendor- or project-specific syslog extensions. There are a few around, but the problem here is that they have compatibility issues. However, there is one surprisingly easy and interoperable solution: though not standardized, many vendors and projects implement plain tcp syslog. In a nutshell, plain tcp syslog is a mode where standard syslog messages are transmitted via tcp and records are separated by newline characters. This mode is supported by all major syslogd's (both on Linux/Unix and Windows) as well as log sources (for example, EventReporter for Windows Event Log forwarding). Plain tcp syslog offers reliability, but it does not offer encryption in itself. However, since it operates on a tcp stream, it is now easy to add encryption. There are various ways to do that. In this paper, I will describe how it is done with stunnel (an other alternative would be IPSec, for example).

Stunnel is open source and it is available both for Unix/Linux and Windows. It provides a way to use ssl communication for any non-ssl aware client and server - in this case, our syslogd.

Stunnel works much like a wrapper. Both on the client and on the server machine, tunnel portals are created. The non-ssl aware client and server software is configured to not directly talk to the remote partner, but to the local (s)tunnel portal instead. Stunnel, in turn, takes the data received from the client, encrypts it via ssl, sends it to the remote tunnel portal and that remote portal sends it to the recipient process on the remote machine. The transfer to the portals is done via unencrypted communication. As such, it is vital that the portal and the respective program that is talking to it are on the same machine, otherwise data would travel partly unencrypted. Tunneling, as done by stunnel, requires connection oriented communication. This is why you need to use tcp-based syslog. As a side-note, you can also encrypt a plain-text RFC 3195 session via stunnel, though this definitely is not what the protocol designers had on their mind ;)

In the rest of this document, I assume that you use rsyslog on both the client and the server. For the samples, I use Debian. Interestingly, there are some annoying differences between stunnel implementations. For example, on Debian a comment line starts with a semicolon (';'). On Red Hat, it starts with a hash sign ('#'). So you need to watch out for subtle issues when setting up your system.

Overall System Setup

In ths paper, I assume two machines, one named "client" and the other named "server". It is obvious that, in practice, you will probably have multiple clients but only one server. Syslog traffic shall be transmitted via stunnel over the network. Port 60514 is to be used for that purpose. The machines are set up as follows:

Client

- rsyslog forwards message to stunnel local portal at port 61514
- local stunnel forwards data via the network to port 60514 to its remote peer

Server

- stunnel listens on port 60514 to connections from its client peers
- all connections are forwarded to the locally-running rsyslog listening at port 61514

Setting up the system

For Debian, you need the "stunnel4" package. The "stunnel" package is the older 3.x release, which will not support the configuration I describe below. Other distributions might have other names. For example, on Red Hat it is just "stunnel". Make sure that you install the appropriate package on both the client and the server. It is also a good idea to check if there are updates for either stunnel or openssl (which stunnel uses) - there are often security fixes available and often the latest fixes are not included in the default package.

In my sample setup, I use only the bare minimum of options. For example, I do not make the server check client cerficiates. Also, I do not talk much about certificates at all. If you intend to really secure your system, you should probably learn about certificates and how to manage and deploy them. This is beyond the scope of this paper. For additional information, http://www.stunnel.org/faq/certs.html is a good starting point.

You also need to install rsyslogd on both machines. Do this before starting with the configuration. You should also familarize yourself with its configuration file syntax, so that you know which actions you can trigger with it. Rsyslogd can work as a drop-in replacement for stock sysklogd. So if you know the standard syslog.conf syntax, you do not need to learn any more to follow this paper.

Server Setup

At the server, you need to have a digital certificate. That certificate enables SSL operation, as it provides the necessary crypto keys being used to secure the connection. Many versions of stunnel come with a default certificate, often found in /etc/stunnel/stunnel.pem. If you have it, it is good for testing only. If you use it in production, it is very easy to

break into your secure channel as everybody is able to get hold of your private key. I didn't find an stunnel.pem on my Debian machine. I guess the Debian folks removed it because of its insecurity.

You can create your own certificate with a simple openssl tool - you need to do it if you have none and I highly recommend to create one in any case. To create it, cd to /etc/stunnel and type:

```
openssl req -new -x509 -days 3650 -nodes -out stunnel.pem -keyout
stunnel.pem
```

That command will ask you a number of questions. Provide some answer for them. If you are unsure, read http://www.stunnel.org/faq/certs.html. After the command has finished, you should have a usable stunnel.pem in your working directory.

Next is to create a configuration file for stunnel. It will direct stunnel what to do. You can use the following basic file:

```
; Certificate/key is needed in server modecert = /etc/stunnel/stunnel.pem; Some␣
↪debugging stuff useful for troubleshootingdebug = 7foreground=yes
```

[ssyslog] accept = 60514 connect = 61514

Save this file to e.g. /etc/stunnel/syslog-server.conf. Please note that the settings in *italics* are for debugging only. They run stunnel with a lot of debug information in the foreground. This is very valuable while you setup the system - and very useless once everything works well. So be sure to remove these lines when going to production.

Finally, you need to start the stunnel daemon. Under Debian, this is done via "stunnel /etc/stunnel/syslog.server.conf". If you have enabled the debug settings, you will immediately see a lot of nice messages.

Now you have stunnel running, but it obviously unable to talk to rsyslog - because it is not yet running. If not already done, configure it so that it does everything you want. If in doubt, you can simply copy /etc/syslog.conf to /etc/rsyslog.conf and you probably have what you want. The really important thing in rsyslogd configuration is that you must make it listen to tcp port 61514 (remember: this is where stunnel send the messages to). Thankfully, this is easy to achive: just add "-t 61514" to the rsyslogd startup options in your system startup script. After done so, start (or restart) rsyslogd.

The server should now be fully operational.

Client Setup

The client setup is simpler. Most importantly, you do not need a certificate (of course, you can use one if you would like to authenticate the client, but this is beyond the scope of this paper). So the basic thing you need to do is create the stunnel configuration file.

```
; Some debugging stuff useful for troubleshootingdebug = 7foreground=yes

client=yes

[ssyslog]
accept  = 127.0.0.1:61514
connect = 192.0.2.1:60514
```

Again, the text in *italics* is for debugging purposes only. I suggest you leave it in during your initial testing and then remove it. The most important difference to the server configuration outlined above is the "client=yes" directive. It is what makes this stunnel behave like a client. The accept directive binds stunnel only to the local host, so that it is protected from receiving messages from the network (somebody might fake to be the local sender). The address "192.0.2.1" is the address of the server machine. You must change it to match your configuration. Save this file to /etc/stunnel/syslog-client.conf.

Then, start stunnel via "stunnel4 /etc/stunnel/syslog-client.conf". Now you should see some startup messages. If no errors appear, you have a running client stunnel instance.

Finally, you need to tell rsyslogd to send data to the remote host. In stock syslogd, you do this via the "@host" forwarding directive. The same works with rsyslog, but it suppports extensions to use tcp. Add the following line to your /etc/rsyslog.conf:

```
*.*       @@127.0.0.1:61514
```

Please note the double at-sign (@@). This is no typo. It tells rsyslog to use tcp instead of udp delivery. In this sample, all messages are forwarded to the remote host. Obviously, you may want to limit this via the usual rsyslog.conf settings (if in doubt, use man rsyslog.con).

You do not need to add any special startup settings to rsyslog on the client. Start or restart rsyslog so that the new configuration setting takes place.

Done

After following these steps, you should have a working secure syslog forwarding system. To verify, you can type "logger test" or a similar smart command on the client. It should show up in the respective server log file. If you dig out you sniffer, you should see that the traffic on the wire is actually protected. In the configuration use above, the two stunnel endpoints should be quite chatty, so that you can follow the action going on on your system.

If you have only basic security needs, you can probably just remove the debug settings and take the rest of the configuration to production. If you are security-sensitve, you should have a look at the various stunnel settings that help you further secure the system.

Preventing Systems from talking directly to the rsyslog Server

It is possible that remote systems (or attackers) talk to the rsyslog server by directly connecting to its port 61514. Currently (July of 2005), rsyslog does not offer the ability to bind to the local host, only. This feature is planned, but as long as it is missing, rsyslog must be protected via a firewall. This can easily be done via e.g iptables. Just be sure not to forget it.

Conclusion

With minumal effort, you can set up a secure logging infrastructure employing ssl encrypted syslog message transmission. As a side note, you also have the benefit of reliable tcp delivery which is far less prone to message loss than udp.

Feedback requested

I would appreciate feedback on this tutorial. If you have additional ideas, comments or find bugs (I *do* bugs - no way... ;)), please let me know.

Revision History

- 2005-07-22 * Rainer Gerhards * Initial Version created
- 2005-07-26 * Rainer Gerhards * Some text brush-up, hyperlinks added
- 2005-08-03 * Rainer Gerhards * license added
- 2008-05-05 * Rainer Gerhards * updated to reflect native TLS capability of rsyslog 3.19.0 and above

Copyright

Developing rsyslog modules (outdated)

*Written by 'Rainer Gerhards <*http://www.adiscon.com/en/people/rainer-gerhards.php*>'_ (2007-07-28)*

This document is outdated and primarily contains historical information. Do not trust it to build code. It currently is under review.

This document is incomplete. The module interface is also quite incomplete and under development. Do not currently use it! You may want to visit Rainer's blog to learn what's going on.

Overview

In theory, modules provide input and output, among other functions, in rsyslog. In practice, modules are only utilized for output in the current release. The module interface is not yet completed and a moving target. We do not recommend to write a module based on the current specification. If you do, please be prepared that future released of rsyslog will probably break your module.

A goal of modularization is to provide an easy to use plug-in interface. However, this goal is not yet reached and all modules must be statically linked.

Module "generation"

There is a lot of plumbing that is always the same in all modules. For example, the interface definitions, answering function pointer queries and such. To get rid of these laborious things, I generate most of them automatically from a single file. This file is named module-template.h. It also contains the current best description of the interface "specification".

One thing that can also be achieved with it is the capability to cope with a rapidly changing interface specification. The module interface is evolving. Currently, it is far from being finished. As I moved the monolithic code to modules, I needed (and still need) to make many "non-clean" code hacks, just to get it working. These things are now gradually being removed. However, this requires frequent changes to the interfaces, as things move in and out while working towards a clean interface. All the interim is necessary to reach the goal. This volatility of specifications is the number one reasons I currently advise against implementing your own modules (hint: if you do, be sure to use module-template.h and be prepared to fix newly appearing and disappearing data elements).

Naming Conventions

Source

Output modules, and only output modules, should start with a file name of "om" (e.g. "omfile.c", "omshell.c"). Similarly, input modules will use "im" and filter modules "fm". The third character shall not be a hyphen.

Module Security

Modules are directly loaded into rsyslog's address space. As such, any module is provided a big level of trust. Please note that further module interfaces might provide a way to load a module into an isolated address space. This, however, is far from being completed. So the number one rule about module security is to run only code that you know you can trust.

To minimize the security risks associated with modules, rsyslog provides only the most minimalistic access to data structures to its modules. For that reason, the output modules do not receive any direct pointers to the selector_t structure, the syslogd action structures and - most importantly - the msg structure itself. Access to these structures would enable modules to access data that is none of their business, creating a potential security weakness.

Not having access to these structures also simplifies further queueing and error handling cases. As we do not need to provide e.g. full access to the msg object itself, we do not need to serialize and cache it. Instead, strings needed by the module are created by syslogd and then the final result is provided to the module. That, for example, means that in a queued case $NOW is the actual timestamp of when the message was processed, which may be even days before it being dequeued. Think about it: If we wouldn't cache the resulting string, $NOW would be the actual date if the action were suspended and messages queued for some time. That could potentially result in big confusion.

It is thought that if an output module actually needs access to the while msg object, we will (then) introduce a way to serialize it (e.g. to XML) in the property replacer. Then, the output module can work with this serialized object. The key point is that output modules never deal directly with msg objects (and other internal structures). Besides security, this also greatly simplifies the job of the output module developer.

Action Selectors

Modules (and rsyslog) need to know when they are called. For this, there must a an action identification in selector lines. There are two syntaxes: the single-character syntax, where a single characters identifies a module (e.g. "*" for a wall message) and the modules designator syntax, where the module name is given between colons (e.g. ":ommysql:"). The single character syntax is depreciated and should not be used for new plugins.

An in-depth discussion of module designation in action selectors can be found in this forum thread:

http://www.rsyslog.com/index.php?name=PNphpBB2&file=viewtopic&p=678#678

Copyright

CHAPTER 2

Reference

RSyslog - History

Rsyslog is a GPL-ed, enhanced syslogd. Among others, it offers support for reliable syslog over TCP, writing to MySQL databases and fully configurable output formats (including great timestamps).

Rsyslog was initiated by Rainer Gerhards. If you are interested to learn why Rainer initiated the project, you may want to read his blog posting on "why the world needs another syslogd".

Rsyslog was forked in **2004** from the sysklogd standard package. The goal of the rsyslog project is to provide a feature-richer and reliable syslog daemon while retaining drop-in replacement capabilities to stock syslogd. By "reliable", we mean support for reliable transmission modes like TCP or RFC 3195 (syslog-reliable). We do NOT imply that the sysklogd package is unreliable.

The name "rsyslog" stems back to the planned support for syslog-reliable. Ironically, the initial release of rsyslog did NEITHER support syslog-reliable NOR tcp based syslog. Instead, it contained enhanced configurability and other enhancements (like database support). The reason for this is that full support for RFC 3195 would require even more changes and especially fundamental architectural changes. Also, questions asked on the loganalysis list and at other places indicated that RFC3195 is NOT a prime priority for users, but rather better control over the output format. So there we were, with a rsyslogd that covers a lot of enhancements, but not a single one of these that made its name ;) Since version 0.9.2, receiving syslog messages via plain tcp is finally supported, a bit later sending via TCP, too. Starting with 1.11.0, RFC 3195 is finally supported at the receiving side (a.k.a. "listener"). Support for sending via RFC 3195 is still due. Anyhow, rsyslog has come much closer to what it name promises.

The database support was initially included so that our web-based syslog interface could be used. This is another open source project which can be found under http://www.phplogcon.org. We highly recommend having a look at it. It might not work for you if you expect thousands of messages per second (because your database won't be able to provide adequate performance), but in many cases it is a very handy analysis and troubleshooting tool. In the mean time, of course, lots of people have found many applications for writing to databases, so the prime focus is no longer on phpLogcon.

Rsyslogd supports an enhanced syslog.conf file format, and also works with the standard syslog.conf. In theory, it should be possible to simply replace the syslogd binary with the one that comes with rsyslog. Of course, in order to use any of the new features, you must re-write your syslog.conf. To learn how to do this, please review our commented sample.conf file. It outlines the enhancements over stock syslogd. Discussion has often arisen of whether having an

"old syslogd" logfile format is good or evil. So far, this has not been solved (but Rainer likes the idea of a new format), so we need to live with it for the time being. It is planned to be reconsidered in the 3.x release time frame.

If you are interested in the IHE environment, you might be interested to hear that rsyslog supports message with sizes of 32k and more. This feature has been tested, but by default is turned off (as it has some memory footprint that we didn't want to put on users not actually requiring it). Search the file syslogd.c and search for "IHE" - you will find easy and precise instructions on what you need to change (it's just one line of code!). Please note that RFC 3195/COOKED supports 1K message sizes only. It'll probably support longer messages in the future, but it is our believe that using larger messages with current RFC 3195 is a violation of the standard.

In **February 2007**, 1.13.1 was released and served for quite a while as a stable reference. Unfortunately, it was not later released as stable, so the stable build became quite outdated.

In **June 2007**, Peter Vrabec from Red Hat helped us to create RPM files for Fedora as well as supporting IPv6. There also seemed to be some interest from the Red Hat community. This interest and new ideas resulted in a very busy time with many great additions.

In **July 2007**, Andrew Pantyukhin added BSD ports files for rsyslog and liblogging. We were strongly encouraged by this too. It looks like rsyslog is getting more and more momentum. Let's see what comes next...

Also in **July 2007** (and beginning of August), Rainer remodeled the output part of rsyslog. It got a clean object model and is now prepared for a plug-in architecture. During that time, some base ideas for the overall new object model appeared.

In **August 2007** community involvement grew more and more. Also, more packages appeared. We were quite happy about that. To facilitate user contributions, we set up a wiki on August 10th, 2007. Also in August 2007, rsyslog 1.18.2 appeared, which is deemed to be quite close to the final 2.0.0 release. With its appearance, the pace of changes was deliberately reduced, in order to allow it to mature (see Rainers's blog post on this topic, written a bit early, but covering the essence).

In **November 2007**, rsyslog became the default syslogd in Fedora 8. Obviously, that was something we *really* liked. Community involvement also is still growing. There is one sad thing to note: ever since summer, there is an extremely hard to find segfault bug. It happens on very rare occasions only and never in lab. We are hunting this bug for month now, but still could not get hold of it. Unfortunately, this also affects the new features schedule. It makes limited sense to implement new features if problems with existing ones are not really understood.

December 2007 showed the appearance of a postgres output module, contributed by sur5r. With 1.20.0, December is also the first time since the bug hunt that we introduce other new features. It has been decided that we carefully will add features in order to not affect the overall project by these rare bugs. Still, the bug hunt is top priority, but we need to have more data to analyze. At then end of December, it looked like the bug was found (a race condition), but further confirmation from the field is required before declaring victory. December also brings the initial development on **rsyslog v3**, resulting in loadable input modules, now running on a separate thread each.

On **January, 2nd 2008**, rsyslog 1.21.2 is re-released as rsyslog v2.0.0 stable. This is a major milestone as far as the stable build is concerned. v3 is not yet officially announced. Other than the stable v2 build, v3 will not be backwards compatibile (including missing compatibility to stock sysklogd) for quite a while. Config file changes are required and some command line options do no longer work due to the new design.

On January, 31st 2008 the new massively-multithreaded queue engine was released for the first time. It was a major milestone, implementing a feature I dreamed of for more than a year.

End of February 2008 saw the first note about RainerScript, a way to configure rsyslogd via a script-language like configuration format. This effort evolved out of the need to have complex expression support, which was also the first use case. On February, 28th rsyslog 3.12.0 was released, the first version to contain expression support. This also meant that rsyslog from that date on supported all syslog-ng major features, but had a number of major features exclusive to it. With 3.12.0, I consider rsyslog fully superior to syslog-ng (except for platform support).

Be sure to visit Rainer's syslog blog to get some more insight into the development and futures of rsyslog and syslog in general. Don't be shy to post to either the blog or the rsyslog forums.

Licensing

If you intend to use rsyslog inside a GPLv3 compatible project, you are free to do so. You don't even need to continue reading. If you intend to use rsyslog inside a non-GPLv3 compatible project, rsyslog offers you some liberties to do that, too. However, you then need to study the licensing details in depth.

The project hopes this is a good compromise, which also gives a boost to fellow free software developers who release under GPLv3.

And now on to the dirty and boring license details, still on a executive summary level. For the real details, check source files and the files COPYING and COPYING.LESSER inside the distribution.

The rsyslog package contains several components:

- the rsyslog core programs (like rsyslogd)
- plugins (like imklog, omrelp, ...)
- the rsyslog runtime library

Each of these components can be thought of as individual projects. In fact, some of the plugins have different main authors than the rest of the rsyslog package. All of these components are currently put together into a single "rsyslog" package (tarball) for convenience: this makes it easier to distribute a consistent version where everything is included (and in the right versions) to build a full system. Platform package maintainers in general take the overall package and split off the individual components, so that users can install only what they need. In source installations, this can be done via the proper ./configure switches.

However, while it is convenient to package all parts in a single tarball, it does not imply all of them are necessarily covered by the same license. Traditionally, GPL licenses are used for rsyslog, because the project would like to provide free software. GPLv3 has been used since around 2008 to help fight for our freedom. All rsyslog core programs are released under GPLv3. But, from the beginning on, plugins were separate projects and we did not impose and license restrictions on them. So even though all plugins that currently ship with the rsyslog package are also placed under GPLv3, this can not taken for granted. You need to check each plugins license terms if in question - this is especially important for plugins that do NOT ship as part of the rsyslog tarball.

In order to make rsyslog technology available to a broader range of applications, the rsyslog runtime is, at least partly, licensed under LGPL. If in doubt, check the source file licensing comments. As of now, the following files are licensed under LGPL:

- queue.c/.h
- wti.c/.h
- wtp.c/.h
- vm.c/.h
- vmop.c/.h
- vmprg.c/.h
- vmstk.c/.h
- expr.c/.h
- sysvar.c/.h
- ctok.c/.h
- ctok_token.c/.h
- regexp.c/.h
- sync.c/.h

- stream.c/.h
- var.c/.h

This list will change as time of the runtime modularization. At some point in the future, there will be a well-designed set of files inside a runtime library branch and all of these will be LGPL. Some select extras will probably still be covered by GPL. We are following a similar licensing model in GnuTLS, which makes effort to reserve some functionality exclusively to open source projects.

How you can Help

You like rsyslog and would like to lend us a helping hand? This page tells you how easy it is to help a little bit. You can contribute to the project even with a single mouse click! If you could pick a single item from the wish list, that would be awfully helpful!

This is our wish list:

- let others know how great rsyslog is
  - spread word about rsyslog in forums and newsgroups
  - place a link to www.rsyslog.com from your home page
- let us know about rsyslog - we are eager for feedback
  - tell us what you like and what you not like - so that we can include that into development
  - tell us what you use rsyslog for - especially if you have high traffic volume or an otherwise "uncommon" deployment. We are looking for case studies and experience how rsyslog performs in unusual scenarios.
  - allow us to post your thoughts and experiences as a "user story" on the web site (so far, none are there ;))
- if you know how to create packages (rpm, deb, ...)
  - we would very much appreciate your help with package creation. We know that it is important to have good binary packages for a product to spread widely. Yet, we do not have the knowledge to do it all ourselves. Drop Rainer a noteif you could help us out.
- if you have configured a device for sending syslog data, and that device is not in our syslog configuration database, you might want to tell us how to configure it.
- if you are a corporate user
  - you might consider Adiscon's commercial MonitorWare products for Windows, e.g. to deliver Windows Event Log data to rsyslogd (sales of the commercial products funds the open source development - and they also work very well).
  - you might be interested in purchasing professional support or add-on development for rsyslog

We appreciate your help very much. A big thank you for anything you might do!

Community Resources

You can also browse the following online resources:

- the rsyslog wiki, a community resource which includes rsyslog configuration examples
- rsyslog discussion forum - use this for technical support
- rsyslog video tutorials

- rsyslog FAQ
- syslog device configuration guide (off-site)
- deutsches rsyslog forum (forum in German language)

And don't forget about the rsyslog mailing list. If you are interested in the "backstage", you may find Rainer's blog an interesting read (filter on syslog and rsyslog tags). Or meet Rainer Gerhards at Facebook or Google+. If you would like to use rsyslog source code inside your open source project, you can do that without any restriction as long as your license is GPLv3 compatible. If your license is incompatible to GPLv3, you may even be still permitted to use rsyslog source code. However, then you need to look at the way rsyslog is licensed.

Feedback is always welcome, but if you have a support question, please do not mail Rainer directly (why not?) - use the rsyslog mailing list or rsyslog forum instead.

RSyslog - Features

This page lists both current features as well as those being considered for future versions of rsyslog. If you think a feature is missing, drop Rainer a note. Rsyslog is a vital project. Features are added each few days. If you would like to keep up of what is going on, you can also subscribe to the rsyslog mailing list.

A better structured feature list is now contained in our rsyslog vs. syslog-ng comparison. Probably that page will replace this one in the future.

Current Features

- native support for writing to MySQL databases
- native support for writing to Postgres databases
- direct support for Firebird/Interbase, OpenTDS (MS SQL, Sybase), SQLLite, Ingres, Oracle, and mSQL via libdbi, a database abstraction layer (almost as good as native)
- native support for sending mail messages (first seen in 3.17.0)
- support for (plain) tcp based syslog - much better reliability
- support for sending and receiving compressed syslog messages
- support for on-demand on-disk spooling of messages that can not be processed fast enough (a great feature for writing massive amounts of syslog messages to a database)
- support for selectively processing messages only during specific timeframes and spooling them to disk otherwise
- ability to monitor text files and convert their contents into syslog messages (one per line)
- ability to configure backup syslog/database servers - if the primary fails, control is switched to a prioritized list of backups
- support for receiving messages via reliable RFC 3195 delivery (a bit clumsy to build right now...)
- ability to generate file names and directories (log targets) dynamically, based on many different properties
- control of log output format, including ability to present channel and priority as visible log data
- good timestamp format control; at a minimum, ISO 8601/RFC 3339 second-resolution UTC zone
- ability to reformat message contents and work with substrings
- support for log files larger than 2gb
- support for file size limitation and automatic rollover command execution

- support for running multiple rsyslogd instances on a single machine
- support for TLS-protected syslog (both natively and via stunnel)
- ability to filter on any part of the message, not just facility and severity
- ability to use regular expressions in filters
- support for discarding messages based on filters
- ability to execute shell scripts on received messages
- control of whether the local hostname or the hostname of the origin of the data is shown as the hostname in the output
- ability to preserve the original hostname in NAT environments and relay chains
- ability to limit the allowed network senders
- powerful BSD-style hostname and program name blocks for easy multi-host support
- massively multi-threaded with dynamic work thread pools that start up and shut themselves down on an as-needed basis (great for high log volume on multicore machines)
- very experimental and volatile support for syslog-protocol compliant messages (it is volatile because standardization is currently underway and this is a proof-of-concept implementation to aid this effort)
- world's first implementation of syslog-transport-tls
- the sysklogd's klogd functionality is implemented as the *imklog* input plug-in. So rsyslog is a full replacement for the sysklogd package
- support for IPv6
- ability to control repeated line reduction ("last message repeated n times") on a per selector-line basis
- supports sub-configuration files, which can be automatically read from directories. Includes are specified in the main configuration file
- supports multiple actions per selector/filter condition
- MySQL and Postgres SQL functionality as a dynamically loadable plug-in
- modular design for inputs and outputs - easily extensible via custom plugins
- an easy-to-write to plugin interface
- ability to send SNMP trap messages
- ability to filter out messages based on sequence of arrival
- support for comma-separated-values (CSV) output generation (via the "csv" property replace option). The CSV format supported is that from RFC 4180.
- support for arbitrary complex boolean, string and arithmetic expressions in message filters

World's first

Rsyslog has an interesting number of "world's firsts" - things that were implemented for the first time ever in rsyslog. Some of them are still features not available elsewhere.

- world's first implementation of IETF I-D syslog-protocol (February 2006, version 1.12.2 and above), now RFC5424
- world's first implementation of dynamic syslog on-the-wire compression (December 2006, version 1.13.0 and above)

- world's first open-source implementation of a disk-queueing syslogd (January 2008, version 3.11.0 and above)
- world's first implementation of IETF I-D syslog-transport-tls (May 2008, version 3.19.0 and above)

Upcoming Features

The list below is something like a repository of ideas we'd like to implement. Features on this list are typically NOT scheduled for immediate inclusion.

Note that we also maintain a 'list of features that are looking for sponsors <http://www.rsyslog.com/sponsor_feature>'_. If you are interested in any of these features, or any other feature, you may consider sponsoring the implementation. This is also a great way to show your commitment to the open source community. Plus, it can be financially attractive: just think about how much less it may be to sponsor a feature instead of purchasing a commercial implementation. Also, the benefit of being recognised as a sponsor may even drive new customers to your business!

- port it to more *nix variants (eg AIX and HP UX) - this needs volunteers with access to those machines and knowledge
- pcre filtering - maybe (depending on feedback) - simple regex already partly added. So far, this seems sufficient so that there is no urgent need to do pcre. If done, it will be a loadable RainerScript function.
- support for RFC 3195 as a sender - this is currently unlikely to happen, because there is no real demand for it. Any work on RFC 3195 has been suspend until we see some real interest in it. It is probably much better to use TCP-based syslog, which is interoperable with a large number of applications. You may also read my blog post on the future of liblogging, which contains interesting information about the future of RFC 3195 in rsyslog.

To see when each feature was added, see the rsyslog change log (online only).

Proposals

Version Naming

This is the proposal on how versions should be named in the future:

Rsyslog version naming has undergone a number of changes in the past. Our sincere hopes is that the scheme outlined here will serve us well for the future. In general, a three-number versioning scheme with a potential development state indication is used. It follows this pattern:

major.minor.patchlevel[-devstate]

where devstate has some forther structure: -<releaseReason><releaseNumber>

All stable builds come without the devstate part. All unstable development version come with it.

The major version is incremented whenever something really important happens. A single new feature, even if important, does not justify an increase in the major version. There is no hard rule when the major version needs an increment. It mostly is a soft factor, when the developers and/or the community think there has been sufficient change to justify that. Major version increments are expected to happen quite infrequently, maybe around once a year. A major version increment has important implications from the support side: without support contracts, the current major version's last stable release and the last stable release of the version immediately below it are supported (Adiscon, the rsyslog sponsor, offers support contracts covering all other versions).

The minor version is incremented whenever a non-trivial new feature is planned to be added. Triviality of a feature is simply determined by time estimated to implement a feature. If that's more than a few days, it is considered a non-trivial feature. Whenever a new minor version is begun, the desired feature is identified and will be the primary focus of that major.minor version. Trivial features may justify a new minor version if they either do not look trivial

from the user's point of view or change something quite considerable (so we need to alert users). A minor version increment may also be done for some other good reasons that the developers have.

The patchlevel is incremented whenever there is a bugfix or very minor feature added to a (stable or development) release.

The devstate is important during development of a feature. It helps the developers to release versions with new features to the general public and in the hope that this will result in some testing. To understand how it works, we need to look at the release cycle: As already said, at the start of a new minor version, a new non-trivial feature to be implemented in that version is selected. Development on this feature begins. At the current pace of development, getting initial support for such a non-trivial feature typically takes between two and four weeks. During this time, new feature requests come in. Also, we may find out that it may be just the right time to implement some not yet targeted feature requests. A reason for this is that the minor release's feature focus is easier to implement if the other feature is implemented first. This is a quite common thing to happen. So development on the primary focus may hold for a short period while we implement something else. Even unrelated, but very trivial feature requests (maybe an hour's worth of time to implement), may be done in between. Once we have implemented these things, we would like to release as quickly as possible (even more if someone has asked for the feature). So we do not like to wait for the original focus feature to be ready (what could take maybe three more weeks). As a result, we release the new features. But that version will also include partial code of the focus feature. Typically this doesn't hurt as long as noone tries to use it (what of course would miserably fail). But still, part of the new code is already in it. When we release such a "minor-feature enhanced" but "focus-feature not yet completed" version, we need a way to flag it. In current thinking, that is using a "-mf<version>" devstate in the version number ("mf" stands for "minor feature"). Version numbers for -mf releases start at 0 for the first release and are monotonically incremented. Once the focus feature has been fully implemented, a new version now actually supporting that feature will be released. Now, the release reason is changed to the well-know "-rc<version>" where "rc" stands for release candidate. For the first release candidate, the version starts at 0 again and is incremented monotonically for each subsequent release. Please note that a -rc0 may only have bare functionality but later -rc's have a richer one. If new minor features are implemented and released once we have reached rc stage, still a new rc version is issued. The difference between "mf" and "rc" is simply the presence of the desired feature. No support is provided for -mf versions once the first -rc version has been released. And only the most current -rc version is supported.

The -rc is removed and the version declared stable when we think it has undergone sufficient testing and look sufficiently well. Then, it'll turn into a stable release. Stable minor releases never receive non-trivial new features. There may be more than one -rc releases without a stable release present at the same time. In fact, most often we will work on the next minor development version while the previous minor version is still a -rc because it is not yet considered sufficiently stable.

Note: the absence of the -devstate part indicates that a release is stable. Following the same logic, any release with a -devstate part is unstable.

A quick sample:

4.0.0 is the stable release. We begin to implement relp, moving to major.minor to 4.1. While we develop it, someone requests a trivial feature, which we implement. We need to release, so we will have 4.1.0-mf0. Another new feature is requested, move to 4.1.0-mf2. A first version of RELP is implemented: 4.1.0-rc0. A new trivial feature is implemented: 4.1.0-rc1. Relp is being enhanced: 4.1.0-rc2. We now feel RELP is good enough for the time being and begin to implement TLS on plain /Tcp syslog: logical increment to 4.2. Now another new feature in that tree: 4.2.0-mf0. Note that we now have 4.0.0 (stable) and 4.1.0-rc2 and 4.1.0-mf0 (both devel). We find a big bug in RELP coding. Two new releases: 4.1.0-rc3, 4.2.0-mf1 (the bug fix acts like a non-focus feature change). We release TLS: 4.2.0-rc0. Another RELP bug fix 4.1.0-rc4, 4.2.0-rc1. After a while, RELP is matured: 4.1.0 (stable). Now support for 4.0.x stable ends. It, however, is still provided for 3.x.x (in the actual case 2.x.x, because v3 was under the old naming scheme and now stable v3 was ever released).

This is how it is done so far:

This document briefly outlines the strategy for naming versions. It applies to versions 1.0.0 and above. Versions below that are all unstable and have a different naming schema.

Please note that version naming is currently being changed. There is a 'blog post about future rsyslog versions <http://blog.gerhards.net/2007/08/on-rsyslog-versions.html>'_.

The major version is incremented whenever a considerate, major features have been added. This is expected to happen quite infrequently.

The minor version number is incremented whenever there is "sufficient need" (at the discretion of the developers). There is a notable difference between stable and unstable branches. The **stable branch** always has a minor version number in the range from 0 to 9. It is expected that the stable branch will receive bug and security fixes only. So the range of minor version numbers should be quite sufficient.

For the **unstable branch**, minor version numbers always start at 10 and are incremented as needed (again, at the discretion of the developers). Here, new minor versions include both fixes as well as new features (hopefully most of the time). They are expected to be released quite often.

The patch level (third number) is incremented whenever a really minor thing must be added to an existing version. This is expected to happen quite infrequently.

In general, the unstable branch carries all new development. Once it concludes with a sufficiently-enhanced, quite stable version, a new major stable version is assigned.

Lookup Tables

NOTE: this is proposed functionality, which is NOT YET IMPLEMENTED!

Lookup tables are a powerful construct to obtain "class" information based on message content (e.g. to build log file names for different server types, departments or remote offices).

The base idea is to use a message variable as an index into a table which then returns another value. For example, $fromhost-ip could be used as an index, with the table value representing the type of server or the department or remote office it is located in. A main point with lookup tables is that the lookup is very fast. So while lookup tables can be emulated with if-elseif constructs, they are generally much faster. Also, it is possible to reload lookup tables during rsyslog runtime without the need for a full restart.

The lookup tables itself exists in a separate configuration file (one per table). This file is loaded on rsyslog startup and when a reload is requested.

There are different types of lookup tables:

- **string** - the value to be looked up is an arbitrary string. Only exact some strings match.
- **array** - the value to be looked up is an integer number from a consequtive set. The set does not need to start at zero or one, but there must be no number missing. So, for example 5,6,7,8,9 would be a valid set of index values, while 1,2,4,5 would not be (due to missing 2). A match happens if the requested number is present.
- **sparseArray** - the value to be looked up is an integer value, but there may be gaps inside the set of values (usually there are large gaps). A typical use case would be the matching of IPv4 address information. A match happens on the first value that is less than or equal to the requested value.

Note that index integer numbers are represented by unsigned 32 bits.

Lookup tables can be access via the lookup() built-in function. The core idea is to set a local variable to the lookup result and later on use that local variable in templates.

More details on usage now follow.

Lookup Table File Format

Lookup table files contain a single JSON object. This object contains of a header and a table part.

Header

The header is the top-level json. It has parameters "version", "nomatch", and "type". The version parameter must be given and must always be one for this version of rsyslog. The nomatch parameter is optional. If specified, it contains the value to be used if lookup() is provided an index value for which no entry exists. The default for "nomatch" is the empty string. Type specifies the type of lookup to be done.

Table

This must be an array of elements, even if only a single value exists (for obvious reasons, we do not expect this to occur often). Each array element must contain two fields "index" and "value".

Example

This is a sample of how an ip-to-office mapping may look like:

```
{ "version":1, "nomatch":"unk", "type":"string",
  "table":[ {"index":"10.0.1.1", "value":"A" },
          {"index":"10.0.1.2", "value":"A" },
          {"index":"10.0.1.3", "value":"A" },
          {"index":"10.0.2.1", "value":"B" },
          {"index":"10.0.2.2", "value":"B" },
          {"index":"10.0.2.3", "value":"B" }
        ]
}
```

Note: if a different IP comes in, the value "unk" is returend thanks to the nomatch parameter in the first line.

RainerScript Statements

lookup_table() Object

This statement defines and intially loads a lookup table. Its format is as follows:

```
lookup_table(name="name" file="/path/to/file" reloadOnHUP="on|off")
```

Parameters

- **name** (mandatory)

  Defines the name of lookup table for further reference inside the configuration. Names must be unique. Note that it is possible, though not advisible, to have different names for the same file.

- **file** (mandatory)

  Specifies the full path for the lookup table file. This file must be readable for the user rsyslog is run under (important when dropping privileges). It must point to a valid lookup table file as described above.

- **reloadOnHUP** (optional, default "on")

  Specifies if the table shall automatically be reloaded as part of HUP processing. For static tables, the default is "off" and specifying "on" triggers an error message. Note that the default of "on" may be somewhat suboptimal

performance-wise, but probably is what the user intuitively expects. Turn it off if you know that you do not need the automatic reload capability.

lookup() Function

This function is used to actually do the table lookup. Format:

```
lookup("name", indexvalue)
```

Parameters

- **return value**

  The function returns the string that is associated with the given indexvalue. If the indexvalue is not present inside the lookup table, the "nomatch" string is returned (or an empty string if it is not defined).

- **name** (constant string)

  The lookup table to be used. Note that this must be specificed as a constant. In theory, variable table names could be made possible, but their runtime behaviour is not as good as for static names, and we do not (yet) see good use cases where dynamic table names could be useful.

- **indexvalue** (expression)

  The value to be looked up. While this is an arbitrary RainerScript expression, it's final value is always converted to a string in order to conduct the lookup. For example, "lookup(table, 3+4)" would be exactly the same as "lookup(table, "7")". In most cases, indexvalue will probably be a single variable, but it could also be the result of all RainerScript-supported expression types (like string concatenation or substring extraction). Valid samples are "lookup(name, $fromhost-ip & $hostname)" or "lookup(name, substr($fromhost-ip, 0, 5))" as well as of course the usual "lookup(table, $fromhost-ip)".

load_lookup_table Statement

Note: in the final implementation, this MAY be implemented as an action. This is a low-level decesion that must be made during the detail development process. Parameters and semantics will remain the same of this happens.

This statement is used to reload a lookup table. It will fail if the table is static. While this statement is executed, lookups to this table are temporarily blocked. So for large tables, there may be a slight performance hit during the load phase. It is assume that always a triggering condition is used to load the table.

```
load_lookup_table(name="name" errOnFail="on|off" valueOnFail="value")
```

Parameters

- **name** (string)

  The lookup table to be used.

- **errOnFail** (boolean, default "on")

  Specifies whether or not an error message is to be emitted if there are any problems reloading the lookup table.

- **valueOnFail** (optional, string)

  This parameter affects processing if the lookup table cannot be loaded for some reason: If the parameter is not present, the previous table will be kept in use. If the parameter is given, the previous table will no longer be used, and instead an empty table be with nomath=valueOnFail be generated. In short, that means when the parameter is set and the reload fails, all matches will always return what is specified in valueOnFail.

Usage example

For clarity, we show only those parts of rsyslog.conf that affect lookup tables. We use the remote office example that an example lookup table file is given above for.

```
lookup_table(name="ip2office" file="/path/to/ipoffice.lu"
             reloadOnHUP="off")


template(name="depfile" type="string"
         string="/var/log/%$usr.dep%/messages")

set $usr.dep = lookup("ip2office", $fromhost-ip);
action(type="omfile" dynfile="depfile")

# support for reload "commands"
if $fromhost-ip == "10.0.1.123"
   and $msg contains "reload office lookup table"
   then
   load_lookup_table(name="ip2office" errOnFail="on")
```

Note: for performance reasons, it makes sense to put the reload command into a dedicated ruleset, bound to a specific listener - which than should also be sufficiently secured, e.g. via TLS mutual auth.

Implementation Details

The lookup table functionality is implemented via highly efficient algorithms. The string lookup has O(log n) time complexity. The array lookup is O(1). In case of sparseArray, we have O(log n).

To preserve space and, more important, increase cache hit performance, equal data values are only stored once, no matter how often a lookup index points to them.

Rsyslog Whitepapers

These are a collection of white papers written by Rainer Gerhards. They detail logging comparisons and realities based on his experience. They also cover observations regarding the development of rsyslog and the community around syslog messaging

White Papers

syslog parsing in rsyslog

Written by Rainer Gerhards *(2008-09-23)*

We regularly receive messages asking why rsyslog **parses this or that message incorrectly.** Of course, it turns out that rsyslog does the right thing, but the message sender does not. And also of course, this is not even of the slightest help to the end user experiencing the problem ;). So I thought I write this paper. It describes the problem source and shows potential solutions (aha!).

Syslog Standardization

The syslog protocol has not been standardized until relatively recently.The first document "smelling" a bit like a standard is **RFC 3164**, which dates back to August 2001. The problem is that this document is no real standard. It has assigned "informational" status by the IETF which means it provides some hopefully useful information but does not demand anything. It is impossible to "comply" to an informational document. This, of course, doesn't stop marketing guys from telling they comply to RFC3164 and it also does not stop some techs to tell you "this and that does not comply to RFC3164, so it is <anybody else but them>'s fault".

Then, there is **RFC 3195**, which is a real standard. In it's section 3 it makes (a somewhat questionable) reference to (informational) RFC 3164 which may be interpreted in a way that RFC3195 standardizes the format layed out in RFC 3164 by virtue of referencing them. So RFC3195 seems to extend its standardization domain to the concepts layed out in RFC 3164 (which is why I tend to find that refrence questionable). In that sense, RFC3195 standardizes the format informationally described in RFC3164, Section 4. But it demands it only for the scope of RFC3195, which is syslog over BEEP - and NOT syslog over UDP. So one may argue whether or not the RFC3164 format could be considered a standard for any non-BEEP (including UDP) syslog, too. In the strict view I tend to have, it does not. Refering to the RFC3195 context usually does not help, because there are virtually no RFC3195 implementations available (at this time, I would consider this RFC a failure).

Now let's for a short moment assume that RFC3195 would somehow be able to demand RFC3164 format for non-BEEP syslog. So we could use RFC3164 format as a standard. But does that really help? Let's cite RFC 3164, right at the begining of section 4 (actually, this is the first sentence):

```
   The payload of any IP packet that has a UDP destination port of 514
   MUST be treated as a syslog message.


Think a bit about it: this means that whatever is send to port 514 must be␣
↪considered
a valid syslog message. No format at all is demanded. So if "this is junk"␣
↪is sent to
UDP port 514 - voila, we have a valid message (interestingly, it is no␣
↪longer a syslog
message if it is sent to port 515 ;)). You may now argue that I am overdoing.
↪ So let's
cite RFC 3164, Section 5.4, Example 2:


  Example 2

        Use the BFG!

   While this is a valid message, it has extraordinarily little useful
   information.
```

As you can see, RFC3164 explicitely states that no format at all is required.

Now a side-note is due: all of this does not mean that the RFC3164 authors did not know what they were doing. No, right the contrary is true: RFC3164 mission is to describe what has been seen in practice as syslog messages and the conclusion is quite right that there is no common understanding on the message format. This is also the reason why RFC3164 is an informational document: it provides useful information, but does not precisely specify anything.

After all of this bashing, I now have to admit that RFC3164 has some format recommendations layed out in section 4. The format described has quite some value in it and implementors recently try to follow it. This format is usually meant when someone tells you that a software is "RFC3164 compliant" or expects "RFC3164 compliant messages". I also have to admit that rsyslog also uses this format and, in the sense outlined here, expects messages received to be "RFC3164 compliant" (knowingly that such a beast does not exist - I am simply lying here ;)).

Please note that there is some relief of the situation in reach. There is a new normative syslog RFC series upcoming, and it specifies a standard message format. At the time of this writing, the main documents are sitting in the RFC editor queue waiting for a transport mapping to be completed. I personally expect them to be assigned RFC numbers in 2009.

Update: the numbers are now assigned and the base RFC is **RFC 5424**.

Practical Format Requirements

From a practical point of view, the message format expected (and generated by default in legacy mode) is:

```
<PRI>TIMESTAMP SP HOST SP TAG MSG(Freetext)
```

SP is the ASCII "space" character and the definition of the rest of the fields can be taken from RFC3164. Please note that there also is a lot of confusion on what syntax and semantics the TAG actually has. This format is called "legacy syslog" because it is not well specified (as you know by now) and has been "inherited from the real world".

Rsyslog offers two parsers: one for the upcoming RFC series and one for legacy format. We concentrate on the later. That parser applies some logic to detect missing hostnames, is able to handle various ways the TIMESTAMP is typically malformed. In short it applies a lot of guesswork in trying to figure out what a message really means. I am sure the guessing algorithm can be improved, and I am always trying that when I see new malformed messages (and there is an ample set of them...). However, this finds its limits where it is not possible to differentiate between two entities which could be either. For example, look at this message:

```
<144>Tue Sep 23 11:40:01 taghost sample message
```

Does it contain a hostname? Mabye. The value "taghost" is a valid hostname. Of course, it is also a valid tag. If it is a hostname, the tag's value is "sample" and the msg value is "message". Or is the hostname missing, the tag is "taghost" and msg is "sample message"? As a human, I tend to say the later interpretation is correct. But that's hard to tell the message parser (and, no, I do not intend to apply artificial intelligence just to guess what the hostname value is...).

One approach is to configure the parser so that it never expects hostnames. This becomes problematic if you receive messages from multiple devices. Over time, I may implement parser conditionals, but this is not yet available and I am not really sure if it is needed comlexity...

Things like this, happen. Even more scary formats happen in practice. Even from mainstream vendors. For example, I was just asked about this message (which, btw, finally made me write this article here):

```
"<130> [ERROR] iapp_socket_task.c 399: iappSocketTask: iappRecvPkt returned error"
```

If you compare it with the format RFC3164 "suggests", you'll quickly notice that the message is "a bit" malformed. Actually, even my human intelligence is not sufficient to guess if there is a TAG or not (is "[ERROR]" a tag or part of the message). I may not be the smartest guy, but don't expect me to program a parser that is smarter than me.

To the best of my konwledge, these vendor's device's syslog format can be configured, so it would proabably be a good idea to include a (sufficiently well-formed) timestamp, the sending hostname and (maybe?) a tag to make this message well parseable. I will also once again take this sample and see if we can apply some guesswork. For example, "[" can not be part of a well-formed TIMESTAMP, so logic can conclude there is not TIMESTAMP. Also, "[" can not be used inside a valid hostname, so logic can conclude that the message contains no hostname. Even if I implement this logic (which I will probably do), this is a partial solution: it is impossible to guess if there is a tag or not (honestly!). And,

even worse, it is a solution only for those set of messages that can be handled by the logic described. Now consider this hypothetical message:

```
"<130> [ERROR] host.example.net 2008-09-23 11-40-22 PST iapp_socket_task.c 399:␣
↪iappSocketTask: iappRecvPkt returned error"
```

Obviously, it requires additional guesswork. If we iterate over all the cases, we can very quickly see that it is impossible to guess everything correct. In the example above we can not even surely tell if PST should be a timezone or some other message property.

A potential solution is to generate a parser-table based parser, but this requires considerable effort and also has quite some runtime overhead. I try to avoid this for now (but I may do it, especially if someone sponsors this work ;)). Side-note: if you want to be a bit scared about potential formats, you may want to have a look at my paper "On the Nature of Syslog Data".

Work-Around

The number one work-around is to configure your devices so that they emit (sufficiently) well-formed messages. You should by now know what these look like.

If that cure is not available, there are some things you can do in rsyslog to handle the situation. First of all, be sure to read about *rsyslog.conf format* and the *property replacer* specifically. You need to understand that everything is configured in rsyslog. And that the message is parsed into properties. There are also properties available which do not stem back directly to parsing. Most importantly, %fromhost% property holds the name of the system rsyslog received the message from. In non-relay cases, this can be used instead of hostname. In relay cases, there is no cure other than to either fix the orginal sender or at least one of the relays in front of the rsyslog instance in question. Similarly, you can use %timegenerated% instead of %timereported%. Timegenerated is the time the message hit rsyslog for the first time. For non-relayed, locally connected peers, Timegenerated should be a very close approximation of the actual time a message was formed at the sender (depending, of course, on potential internal queueing inside the sender). Also, you may use the %rawmsg% property together with the several extraction modes the property replacer supports. Rawmsg contains the message as it is received from the remote peer. In a sense, you can implement a post-parser with this method.

To use these properties, you need to define your own templates and assign them. Details can be found in the above-quoted documentation. Just let's do a quick example. Let's say you have the horrible message shown above and can not fix the sending device for some good reason. In rsyslog.conf, you used to say:

```
*.* /var/log/somefile
```

Of course, things do not work out well with that ill-formed message. So you decide to dump the rawmsg to the file and pull the remote host and time of message generation from rsyslog's internal properties (which, btw, is clever, because otherwise there is no indication of these two properties...). So you need to define a template for that and make sure the template is used with your file logging action. This is how it may look:

```
$template, MalfromedMsgFormater,"%timegenerated% %fromhost% %rawmsg:::drop-last-lf%\n"
*.* /var/log/somefile;MalformedMsgFormatter
```

This will make your log much nicer, but not look perfect. Experiment a bit with the available properties and replacer extraction options to fine-tune it to your needs.

The Ultimate Solution...

Is available with rsyslog 5.3.4 and above. Here, we can define so-called custom parsers. These are plugin modules, written in C and adapted to a specific message format need. The big plus of custom parsers is that they offer excellent

performance and unlimited possibilities - far better than any work-around could do. Custom parsers can be bound to specific rule sets (and thus listening) ports with relative ease. The only con is that they must be written. However, if you are lucky, a parser for your device may already exist. If not, you can opt to write it yourself, what is not too hard if you know some C. Alternatively, Adiscon can program one for you as part of the rsyslog professional services offering. In any case, you should seriously consider custom parsers as an alternative if you can not reconfigure your device to send decent message format.

Wrap-Up

Syslog message format is not sufficiently standardized. There exists a weak "standard" format, which is used by a good number of implementations. However, there exist many others, including mainstream vendor implementations, which have a (sometimes horribly) different format. Rsyslog tries to deal with anomalies but can not guess right in all instances. If possible, the sender should be configured to submit well-formed messages. If that is not possible, you can work around these issues with rsyslog's property replacer and template system. Or you can use a suitable message parser or write one for your needs.

I hope this is a useful guide. You may also have a look at the rsyslog troubleshooting guide for further help and places where to ask questions.

syslog-protocol support in rsyslog

rsyslog **provides a trial implementation of the proposed** syslog-protocol **standard.** The intention of this implementation is to find out what inside syslog-protocol is causing problems during implementation. As syslog-protocol is a standard under development, its support in rsyslog is highly volatile. It may change from release to release. So while it provides some advantages in the real world, users are cautioned against using it right now. If you do, be prepared that you will probably need to update all of your rsyslogds with each new release. If you try it anyhow, please provide feedback as that would be most beneficial for us.

Currently supported message format

Due to recent discussion on syslog-protocol, we do not follow any specific revision of the draft but rather the candidate ideas. The format supported currently is:

"<PRI>VERSION SP TIMESTAMP SP HOSTNAME SP APP-NAME SP PROCID SP MSGID SP [SD-ID]s SP MSG"

Field syntax and semantics are as defined in IETF I-D syslog-protocol-15.

Capabilities Implemented

- receiving message in the supported format (see above)
- sending messages in the supported format
- relaying messages
- receiving messages in either legacy or -protocol format and transforming them into the other one
- virtual availability of TAG, PROCID, APP-NAME, MSGID, SD-ID no matter if the message was received via legacy format, API or syslog-protocol format (non-present fields are being emulated with great success)
- maximum message size is set via preprocessor #define
- syslog-protocol messages can be transmitted both over UDP and plain TCP with some restrictions on compliance in the case of TCP

Findings

This lists what has been found during implementation:

- The same receiver must be able to support both legacy and syslog-protocol syslog messages. Anything else would be a big inconvenience to users and would make deployment much harder. The detection must be done automatically (see below on how easy that is).
- **NUL characters inside MSG** cause the message to be truncated at that point. This is probably a major point for many C-based implementations. No measures have yet been taken against this. Modifying the code to "cleanly" support NUL characters is non-trivial, even though rsyslogd already has some byte-counted string library (but this is new and not yet available everywhere).
- **character encoding in MSG**: is is problematic to do the right UTF-8 encoding. The reason is that we pick up the MSG from the local domain socket (which got it from the syslog(3) API). The text obtained does not include any encoding information, but it does include non US-ASCII characters. It may also include any other encoding. Other than by guessing based on the provided text, I have no way to find out what it is. In order to make the syslogd do anything useful, I have now simply taken the message as is and stuffed it into the MSG part. Please note that I think this will be a route that other implementors would take, too.
- A minimal parser is easy to implement. It took me roughly 2 hours to add it to rsyslogd. This includes the time for restructuring the code to be able to parse both legacy syslog as well as syslog-protocol. The parser has some restrictions, though
  - STRUCTURED-DATA field is extracted, but not validated. Structured data "[test]]" is not caught as an error. Nor are any other errors caught. For my needs with this syslogd, that level of structured data processing is probably sufficient. I do not want to parse/validate it in all cases. This is also a performance issue. I think other implementors could have the same view. As such, we should not make validation a requirement.
  - MSG is not further processed (e.g. Unicode not being validated)
  - the other header fields are also extracted, but no validation is performed right now. At least some validation should be easy to add (not done this because it is a proof-of-concept and scheduled to change).
- Universal access to all syslog fields (missing ones being emulated) was also quite easy. It took me around another 2 hours to integrate emulation of non-present fields into the code base.
- The version at the start of the message makes it easy to detect if we have legacy syslog or syslog-protocol. Do NOT move it to somewhere inside the middle of the message, that would complicate things. It might not be totally fail-safe to just rely on "1 " as the "cookie" for a syslog-protocol. Eventually, it would be good to add some more uniqueness, e.g. "@#1 ".
- I have no (easy) way to detect truncation if that happens on the UDP stack. All I see is that I receive e.g. a 4K message. If the message was e.g. 6K, I received two chunks. The first chunk (4K) is correctly detected as a syslog-protocol message, the second (2K) as legacy syslog. I do not see what we could do against this. This questions the usefulness of the TRUNCATE bit. Eventually, I could look at the UDP headers and see that it is a fragment. I have looked at a network sniffer log of the conversation. This looks like two totally-independent messages were sent by the sender stack.
- The maximum message size is currently being configured via a preprocessor #define. It can easily be set to 2K or 4K, but more than 4K is not possible because of UDP stack limitations. Eventually, this can be worked around, but I have not done this yet.
- rsyslogd can accept syslog-protocol formatted messages but is able to relay them in legacy format. I find this a must in real-life deployments. For this, I needed to do some field mapping so that APP-NAME/PROCID are mapped into a TAG.
- rsyslogd can also accept legacy syslog message and relay them in syslog-protocol format. For this, I needed to apply some sub-parsing of the TAG, which on most occasions provides correct results. There might be some

misinterpretations but I consider these to be mostly non-intrusive.

- Messages received from the syslog API (the normal case under *nix) also do not have APP-NAME and PROCID and I must parse them out of TAG as described directly above. As such, this algorithm is absolutely vital to make things work on *nix.
- I have an issue with messages received via the syslog(3) API (or, to be more precise, via the local domain socket this API writes to): These messages contain a timestamp, but that timestamp does neither have the year nor the high-resolution time. The year is no real issue, I just take the year of the reception of that message. There is a very small window of exposure for messages read from the log immediately after midnight Jan 1st. The message in the domain socket might have been written immediately before midnight in the old year. I think this is acceptable. However, I can not assign a high-precision timestamp, at least it is somewhat off if I take the timestamp from message reception on the local socket. An alternative might be to ignore the timestamp present and instead use that one when the message is pulled from the local socket (I am talking about IPC, not the network - just a reminder...). This is doable, but eventually not advisable. It looks like this needs to be resolved via a configuration option.
- rsyslogd already advertised its origin information on application startup (in a syslog-protocol-14 compatible format). It is fairly easy to include that with any message if desired (not currently done).
- A big problem I noticed are malformed messages. In -syslog-protocol, we recommend/require to discard malformed messages. However, in practice users would like to see everything that the syslogd receives, even if it is in error. For the first version, I have not included any error handling at all. However, I think I would deliberately ignore any "discard" requirement. My current point of view is that in my code I would eventually flag a message as being invalid and allow the user to filter on this invalidness. So these invalid messages could be redirected into special bins.
- The error logging recommendations (those I insisted on;)) are not really practical. My application has its own error logging philosophy and I will not change this to follow a draft.
- Relevance of support for leap seconds and senders without knowledge of time is questionable. I have not made any specific provisions in the code nor would I know how to handle that differently. I could, however, pull the local reception timestamp in this case, so it might be useful to have this feature. I do not think any more about this for the initial proof-of-concept. Note it as a potential problem area, especially when logging to databases.
- The HOSTNAME field for internally generated messages currently contains the hostname part only, not the FQDN. This can be changed inside the code base, but it requires some thinking so that thinks are kept compatible with legacy syslog. I have not done this for the proof-of-concept, but I think it is not really bad. Maybe an hour or half a day of thinking.
- It is possible that I did not receive a TAG with legacy syslog or via the syslog API. In this case, I can not generate the APP-NAME. For consistency, I have used "-" in such cases (just like in PROCID, MSGID and STRUCTURED-DATA).
- As an architectural side-effect, syslog-protocol formatted messages can also be transmitted over non-standard syslog/raw tcp. This implementation uses the industry-standard LF termination of tcp syslog records. As such, syslog-protocol messages containing a LF will be broken invalidly. There is nothing that can be done against this without specifying a TCP transport. This issue might be more important than one thinks on first thought. The reason is the wide deployment of syslog/tcp via industry standard.

Some notes on syslog-transport-udp-06

- I did not make any low-level modifications to the UDP code and think I am still basically covered with this I-D.
- I deliberately violate section 3.3 insofar as that I do not necessarily accept messages destined to port 514. This feature is user-required and a must. The same applies to the destination port. I am not sure if the "MUST" in section 3.3 was meant that this MUST be an option, but not necessarily be active. The wording should be clarified.

- section 3.6: I do not check checksums. See the issue with discarding messages above. The same solution will probably be applied in my code.

Conlusions/Suggestions

These are my personal conclusions and suggestions. Obviously, they must be discussed ;)

- NUL should be disallowed in MSG
- As it is not possible to definitely know the character encoding of the application-provided message, MSG should **not** be specified to use UTF-8 exclusively. Instead, it is suggested that any encoding may be used but UTF-8 is preferred. To detect UTF-8, the MSG should start with the UTF-8 byte order mask of "EF BB BF" if it is UTF-8 encoded (see section 155.9 of http://www.unicode.org/versions/Unicode4.0.0/ch15.pdf)
- Requirements to drop messages should be reconsidered. I guess I would not be the only implementor ignoring them.
- Logging requirements should be reconsidered and probably be removed.
- It would be advisable to specify "-" for APP-NAME is the name is not known to the sender.
- The implications of the current syslog/tcp industry standard on syslog-protocol should be further evaluated and be fully understood

Turning Lanes and Rsyslog Queues

If there is a single object absolutely vital to understanding the way rsyslog works, this object is queues. Queues offer a variety of services, including support for multithreading. While there is elaborate in-depth documentation on the ins and outs of rsyslog queues, some of the concepts are hard to grasp even for experienced people. I think this is because rsyslog uses a very high layer of abstraction which includes things that look quite unnatural, like queues that do **not** actually queue...

With this document, I take a different approach: I will not describe every specific detail of queue operation but hope to be able to provide the core idea of how queues are used in rsyslog by using an analogy. I will compare the rsyslog data flow with real-life traffic flowing at an intersection.

But first let's set the stage for the rsyslog part. The graphic below describes the data flow inside rsyslog:

Note that there is a video tutorial available on the data flow. It is not perfect, but may aid in understanding this picture.

For our needs, the important fact to know is that messages enter rsyslog on "the left side" (for example, via UDP), are being preprocessed, put into the so-called main queue, taken off that queue, be filtered and be placed into one or several action queues (depending on filter results). They leave rsyslog on "the right side" where output modules (like the file or database writer) consume them.

So there are always **two** stages where a message (conceptually) is queued - first in the main queue and later on in *n* action specific queues (with *n* being the number of actions that the message in question needs to be processed by, what is being decided by the "Filter Engine"). As such, a message will be in at least two queues during its lifetime (with the exception of messages being discarded by the queue itself, but for the purpose of this document, we will ignore that possibility).

Also, it is vitally important to understand that **each** action has a queue sitting in front of it. If you have dug into the details of rsyslog configuration, you have probably seen that a queue mode can be set for each action. And the default queue mode is the so-called "direct mode", in which "the queue does not actually enqueue data". That sounds silly, but is not. It is an important abstraction that helps keep the code clean.

To understand this, we first need to look at who is the active component. In our data flow, the active part always sits to the left of the object. For example, the "Preprocessor" is being called by the inputs and calls itself into the main message queue. That is, the queue receiver is called, it is passive. One might think that the "Parser & Filter Engine"

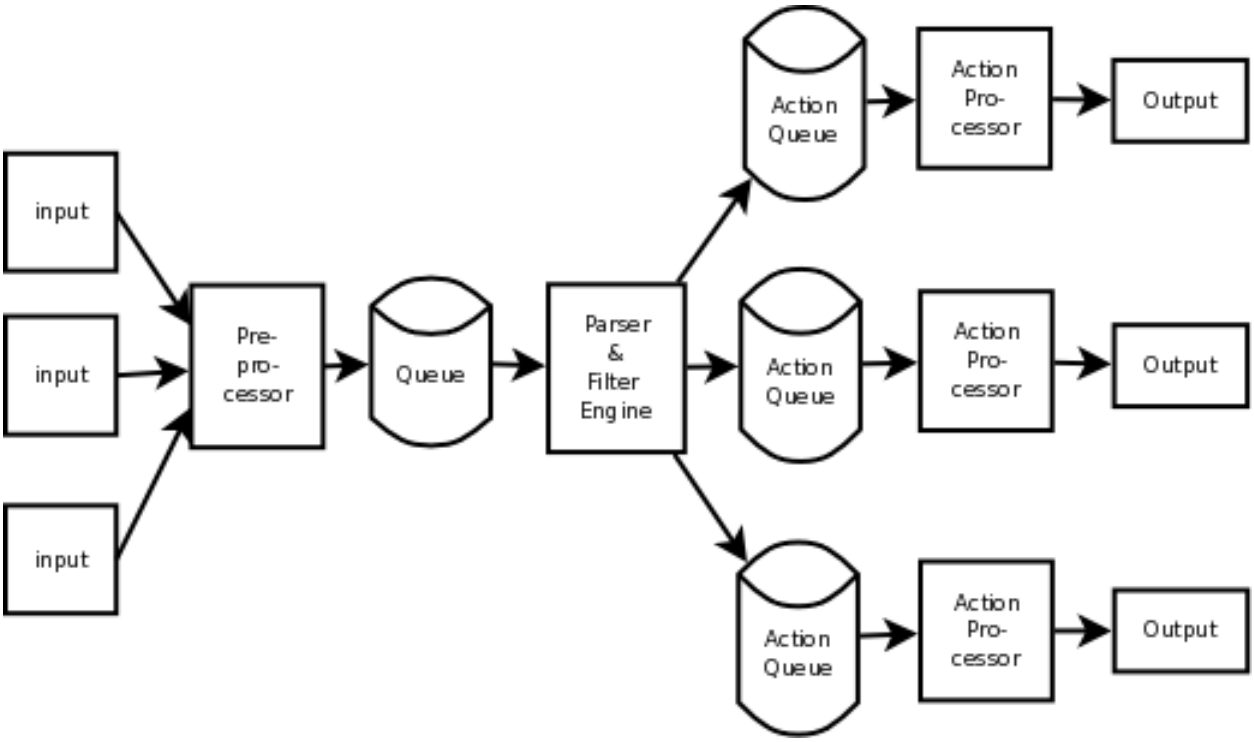


Fig. 2.1: rsyslog data flow

is an active component that actively pulls messages from the queue. This is wrong! Actually, it is the queue that has a pool of worker threads, and these workers pull data from the queue and then call the passively waiting Parser and Filter Engine with those messages. So the main message queue is the active part, the Parser and Filter Engine is passive.

Let's now try an analogy analogy for this part: Think about a TV show. The show is produced in some TV studio, from there sent (actively) to a radio tower. The radio tower passively receives from the studio and then actively sends out a signal, which is passively received by your TV set. In our simplified view, we have the following picture:

The lower part of the picture lists the equivalent rsyslog entities, in an abstracted way. Every queue has a producer (in the above sample the input) and a consumer (in the above sample the Parser and Filter Engine). Their active and passive functions are equivalent to the TV entities that are listed on top of the rsyslog entity. For example, a rsyslog consumer can never actively initiate reception of a message in the same way a TV set cannot actively "initiate" a TV show - both can only "handle" (display or process) what is sent to them.

Now let's look at the action queues: here, the active part, the producer, is the Parser and Filter Engine. The passive part is the Action Processor. The later does any processing that is necessary to call the output plugin, in particular it processes the template to create the plugin calling parameters (either a string or vector of arguments). From the action queue's point of view, Action Processor and Output form a single entity. Again, the TV set analogy holds. The Output **does not** actively ask the queue for data, but rather passively waits until the queue itself pushes some data to it.

Armed with this knowledge, we can now look at the way action queue modes work. My analogy here is a junction, as shown below (note that the colors in the pictures below are **not** related to the colors in the pictures above!):

This is a very simple real-life traffic case: one road joins another. We look at traffic on the straight road, here shown by blue and green arrows. Traffic in the opposing direction is shown in blue. Traffic flows without any delays as long as nobody takes turns. To be more precise, if the opposing traffic takes a (right) turn, traffic still continues to flow without delay. However, if a car in the red traffic flow intends to do a (left, then) turn, the situation changes:

The turning car is represented by the green arrow. It cannot turn unless there is a gap in the "blue traffic stream". And as this car blocks the roadway, the remaining traffic (now shown in red, which should indicate the block condition),

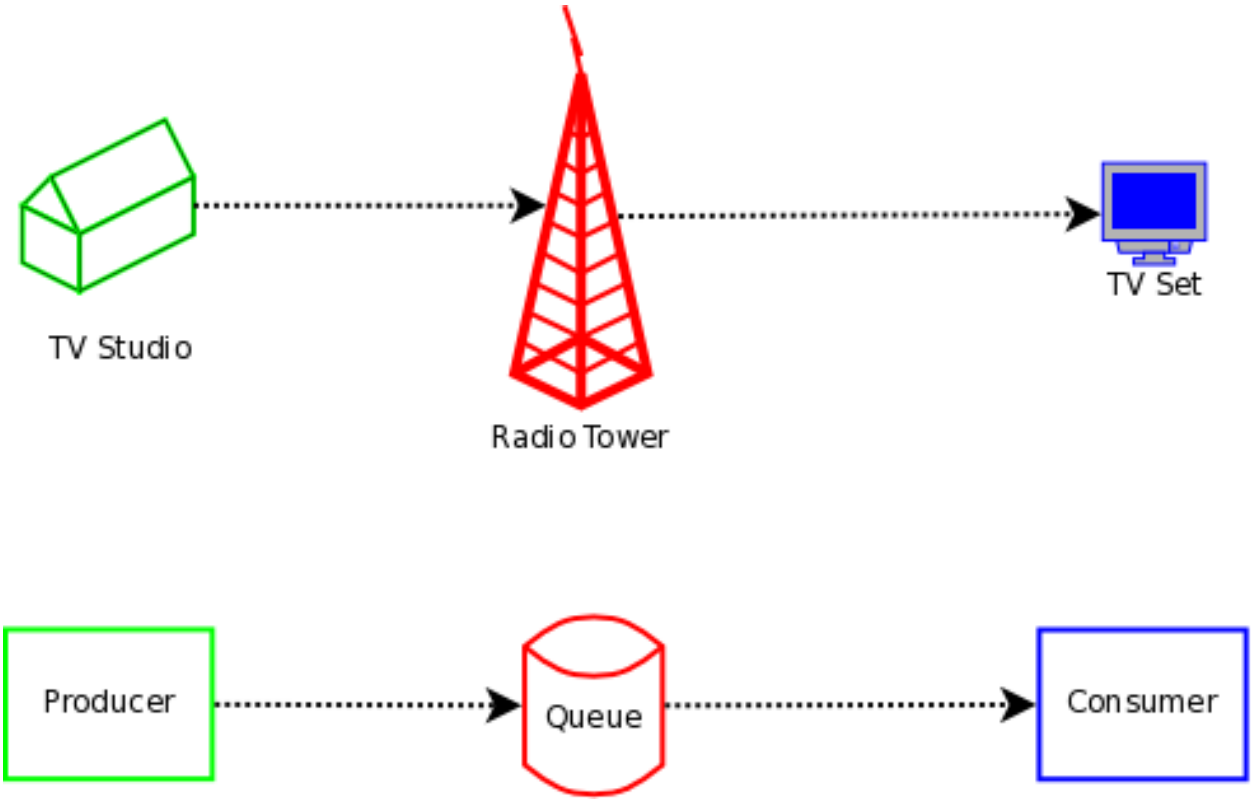
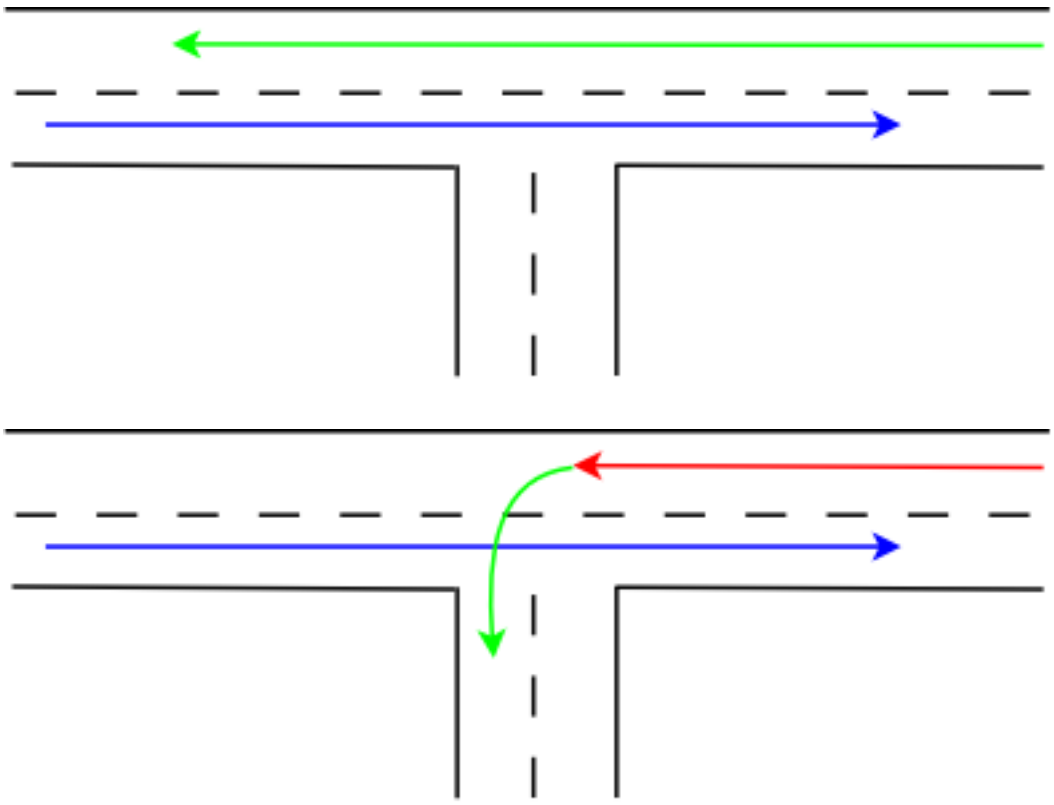
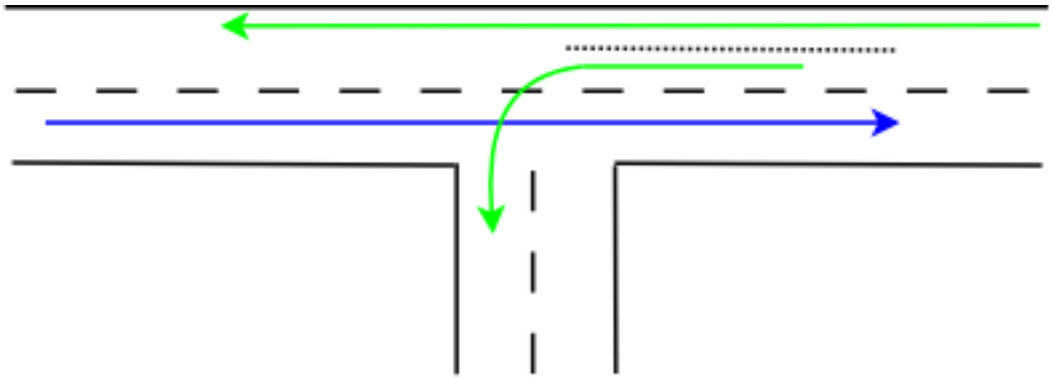


Fig. 2.2: rsyslog queues and TV analogy



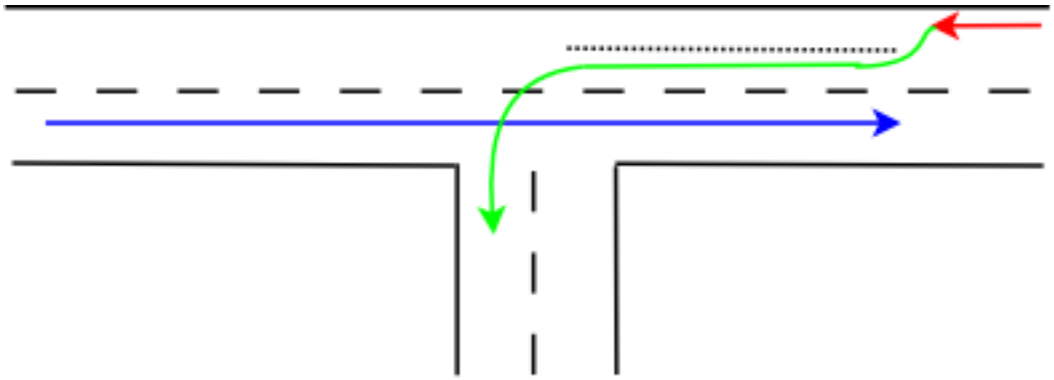
must wait until the "green" car has made its turn. So a queue will build up on that lane, waiting for the turn to be completed. Note that in the examples below I do not care that much about the properties of the opposing traffic. That is, because its structure is not really important for what I intend to show. Think about the blue arrow as being a traffic stream that most of the time blocks left-turners, but from time to time has a gap that is sufficiently large for a left-turn to complete.

Our road network designers know that this may be unfortunate, and for more important roads and junctions, they came up with the concept of turning lanes:



Now, the car taking the turn can wait in a special area, the turning lane. As such, the "straight" traffic is no longer blocked and can flow in parallel to the turning lane (indicated by a now-green-again arrow).

However, the turning lane offers only finite space. So if too many cars intend to take a left turn, and there is no gap in the "blue" traffic, we end up with this well-known situation:



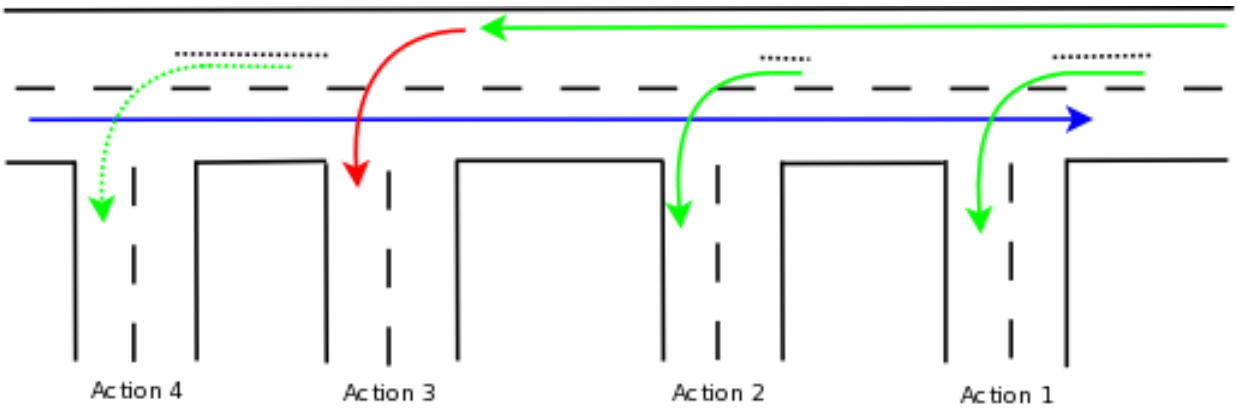
The turning lane is now filled up, resulting in a tailback of cars intending to left turn on the main driving lane. The end result is that "straight" traffic is again being blocked, just as in our initial problem case without the turning lane. In essence, the turning lane has provided some relief, but only for a limited amount of cars. Street system designers now try to weight cost vs. benefit and create (costly) turning lanes that are sufficiently large to prevent traffic jams in most, but not all cases.

Now let's dig a bit into the mathematical properties of turning lanes. We assume that cars all have the same length. So, units of cars, the length is always one (which is nice, as we don't need to care about that factor any longer ;)). A turning lane has finite capacity of n cars. As long as the number of cars wanting to take a turn is less than or eqal to n, "straigth traffic" is not blocked (or the other way round, traffic is blocked if at least $n + 1$ cars want to take a turn!). We can now find an optimal value for n: it is a function of the probability that a car wants to turn and the cost of the turning lane (as well as the probability there is a gap in the "blue" traffic, but we ignore this in our simple sample). If we start from some finite upper bound of n, we can decrease n to a point where it reaches zero. But let's first look at $n = 1$, in which case exactly one car can wait on the turning lane. More than one car, and the rest of the traffic is blocked. Our everyday logic indicates that this is actually the lowest boundary for n.

In an abstract view, however, *n* can be zero and that works nicely. There still can be *n* cars at any given time on the turning lane, it just happens that this means there can be no car at all on it. And, as usual, if we have at least *n + 1* cars wanting to turn, the main traffic flow is blocked. True, but *n + 1 = 0 + 1 = 1* so as soon as there is any car wanting to take a turn, the main traffic flow is blocked (remember, in all cases, I assume no sufficiently large gaps in the opposing traffic).

This is the situation our everyday perception calls "road without turning lane". In my math model, it is a "road with turning lane of size 0". The subtle difference is important: my math model guarantees that, in an abstract sense, there always is a turning lane, it may just be too short. But it exists, even though we don't see it. And now I can claim that even in my small home village, all roads have turning lanes, which is rather impressive, isn't it? ;)

And now we finally have arrived at rsyslog's queues! Rsyslog action queues exists for all actions just like all roads in my village have turning lanes! And as in this real-life sample, it may be hard to see the action queues for that reason. In rsyslog, the "direct" queue mode is the equivalent to the 0-sized turning lane. And actions queues are the equivalent to turning lanes in general, with our real-life *n* being the maximum queue size. The main traffic line (which sometimes is blocked) is the equivalent to the main message queue. And the periods without gaps in the opposing traffic are equivalent to execution time of an action. In a rough sketch, the rsyslog main and action queues look like in the following picture.



We need to read this picture from right to left (otherwise I would need to redo all the graphics ;)). In action 3, you see a 0-sized turning lane, aka an action queue in "direct" mode. All other queues are run in non-direct modes, but with different sizes greater than 0.

Let us first use our car analogy: Assume we are in a car on the main lane that wants to take turn into the "action 4" road. We pass action 1, where a number of cars wait in the turning lane and we pass action 2, which has a slightly smaller, but still not filled up turning lane. So we pass that without delay, too. Then we come to "action 3", which has no turning lane. Unfortunately, the car in front of us wants to turn left into that road, so it blocks the main lane. So, this time we need to wait. An observer standing on the sidewalk may see that while we need to wait, there are still some cars in the "action 4" turning lane. As such, even though no new cars can arrive on the main lane, cars still turn into the "action 4" lane. In other words, an observer standing in "action 4" road is unable to see that traffic on the main lane is blocked.

Now on to rsyslog: Other than in the real-world traffic example, messages in rsyslog can - at more or less the same time - "take turns" into several roads at once. This is done by duplicating the message if the road has a non-zero-sized "turning lane" - or in rsyslog terms a queue that is running in any non-direct mode. If so, a deep copy of the message object is made, that placed into the action queue and then the initial message proceeds on the "main lane". The action queue then pushes the duplicates through action processing. This is also the reason why a discard action inside a non-direct queue does not seem to have an effect. Actually, it discards the copy that was just created, but the original message object continues to flow.

In action 1, we have some entries in the action queue, as we have in action 2 (where the queue is slightly shorter). As we have seen, new messages pass action one and two almost instantaneously. However, when a messages reaches

action 3, its flow is blocked. Now, message processing must wait for the action to complete. Processing flow in a direct mode queue is something like a U-turn:

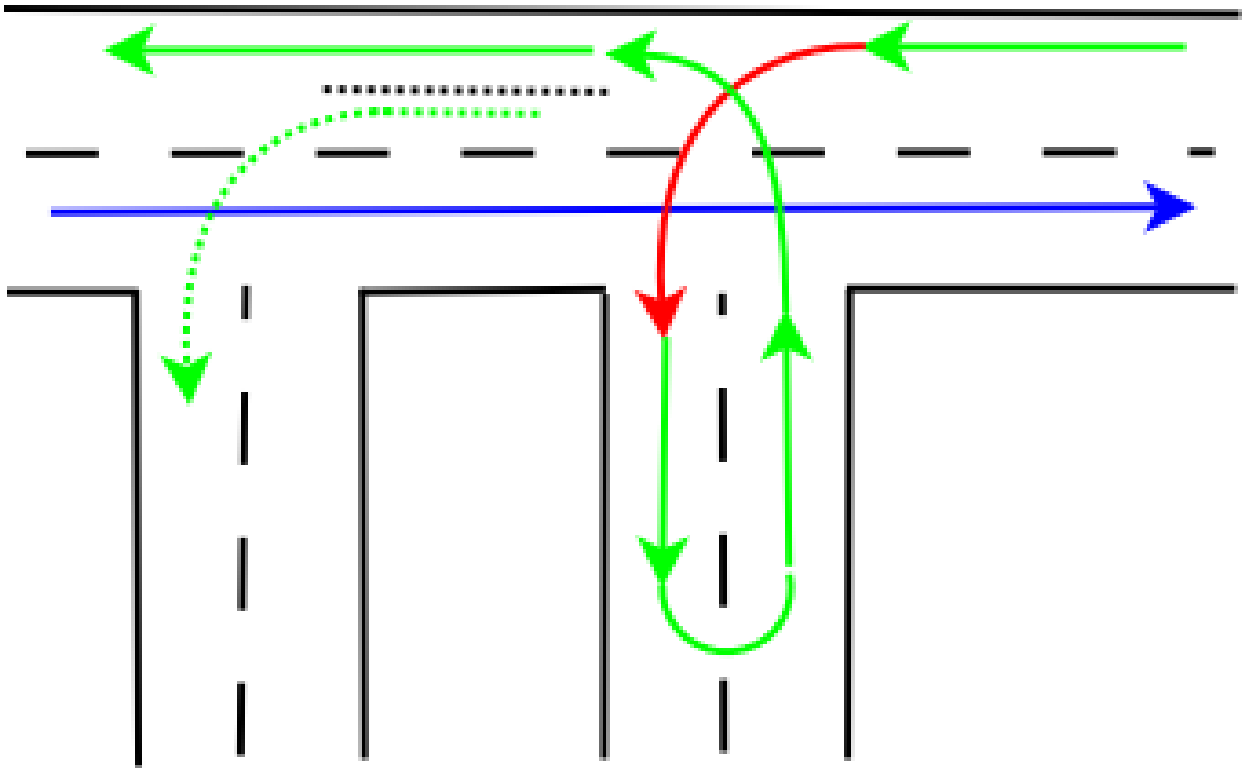


Fig. 2.3: message processing in an rsyslog action queue in direct mode

The message starts to execute the action and once this is done, processing flow continues. In a real-life analogy, this may be the route of a delivery man who needs to drop a parcel in a side street before he continues driving on the main route. As a side-note, think of what happens with the rest of the delivery route, at least for today, if the delivery truck has a serious accident in the side street. The rest of the parcels won't be delivered today, will they? This is exactly how the discard action works. It drops the message object inside the action and thus the message will no longer be available for further delivery - but as I said, only if the discard is done in a direct mode queue (I am stressing this example because it often causes a lot of confusion).

Back to the overall scenario. We have seen that messages need to wait for action 3 to complete. Does this necessarily mean that at the same time no messages can be processed in action 4? Well, it depends. As in the real-life scenario, action 4 will continue to receive traffic as long as its action queue ("turn lane") is not drained. In our drawing, it is not. So action 4 will be executed while messages still wait for action 3 to be completed.

Now look at the overall picture from a slightly different angle:

The number of all connected green and red arrows is four - one each for action 1, 2 and 4 (this one is dotted as action 4 was a special case) and one for the "main lane" as well as action 3 (this one contains the sole red arrow). **This number is the lower bound for the number of threads in rsyslog's output system ("right-hand part" of the main message queue)!** Each of the connected arrows is a continuous thread and each "turn lane" is a place where processing is forked onto a new thread. Also, note that in action 3 the processing is carried out on the main thread, but not in the non-direct queue modes.

I have said this is "the lower bound for the number of threads...". This is with good reason: the main queue may have more than one worker thread (individual action queues currently do not support this, but could do in the future - there are good reasons for that, too but exploring why would finally take us away from what we intend to see). Note that you configure an upper bound for the number of main message queue worker threads. The actual number

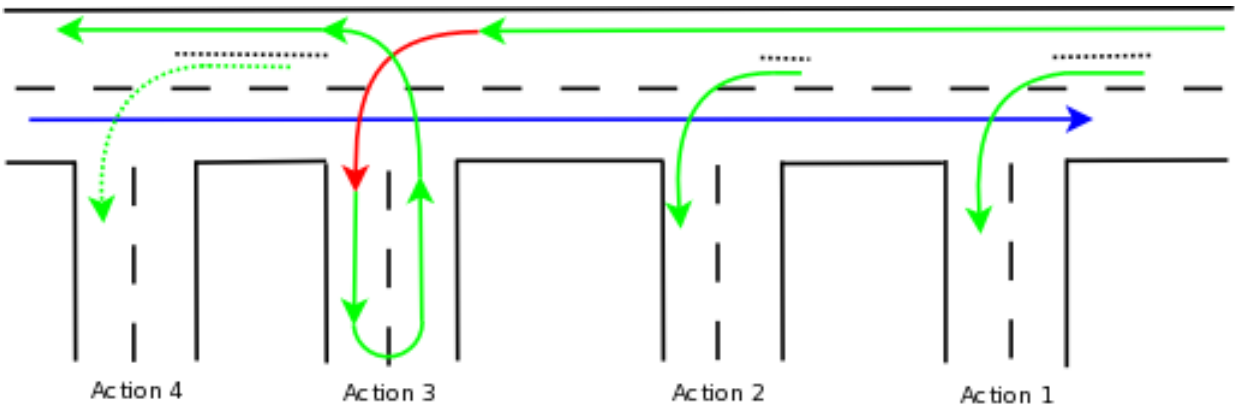


Fig. 2.4: message processing in an rsyslog action queue in direct mode

varies depending on a lot of operational variables, most importantly the number of messages inside the queue. The number *t_m* of actually running threads is within the integer-interval [0,confLimit] (with confLimit being the operator configured limit, which defaults to 5). Output plugins may have more than one thread created by themselves. It is quite unusual for an output plugin to create such threads and so I assume we do not have any of these. Then, the overall number of threads in rsyslog's filtering and output system is *t_total = t_m + number of actions in non-direct modes*. Add the number of inputs configured to that and you have the total number of threads running in rsyslog at a given time (assuming again that inputs utilize only one thread per plugin, a not-so-safe assumption).

A quick side-note: I gave the lower bound for *t_m* as zero, which is somewhat in contrast to what I wrote at the begin of the last paragraph. Zero is actually correct, because rsyslog stops all worker threads when there is no work to do. This is also true for the action queues. So the ultimate lower bound for a rsyslog output system without any work to carry out actually is zero. But this bound will never be reached when there is continuous flow of activity. And, if you are curios: if the number of workers is zero, the worker wakeup process is actually handled within the threading context of the "left-hand-side" (or producer) of the queue. After being started, the worker begins to play the active queue component again. All of this, of course, can be overridden with configuration directives.

When looking at the threading model, one can simply add n lanes to the main lane but otherwise retain the traffic analogy. This is a very good description of the actual process (think what this means to the "turning lanes"; hint: there still is only one per action!).

Let's try to do a warp-up: I have hopefully been able to show that in rsyslog, an action queue "sits in front of" each output plugin. Messages are received and flow, from input to output, over various stages and two level of queues to the outputs. Actions queues are always present, but may not easily be visible when in direct mode (where no actual queuing takes place). The "road junction with turning lane" analogy well describes the way - and intent - of the various queue levels in rsyslog.

On the output side, the queue is the active component, **not** the consumer. As such, the consumer cannot ask the queue for anything (like n number of messages) but rather is activated by the queue itself. As such, a queue somewhat resembles a "living thing" whereas the outputs are just tools that this "living thing" uses.

Note that I left out a couple of subtleties, especially when it comes to error handling and terminating a queue (you hopefully have now at least a rough idea why I say "terminating **a queue**" and not "terminating an action" - *who is the "living thing"?*). An action returns a status to the queue, but it is the queue that ultimately decides which messages can finally be considered processed and which not. Please note that the queue may even cancel an output right in the middle of its action. This happens, if configured, if an output needs more than a configured maximum processing time and is a guard condition to prevent slow outputs from deferring a rsyslog restart for too long. Especially in this case re-queuing and cleanup is not trivial. Also, note that I did not discuss disk-assisted queue modes. The basic rules apply, but there are some additional constraints, especially in regard to the threading model. Transitioning between actual disk-assisted mode and pure-in-memory-mode (which is done automatically when needed) is also far from trivial and a real joy for an implementer to work on ;).

If you have not done so before, it may be worth reading the rsyslog queue user's guide, which most importantly lists all the knobs you can turn to tweak queue operation.

Preserving syslog sender over NAT

Question: I have a number of syslog clients behind a NAT device. The receiver receives syslog messages that travelled over the NAT device. This leads the receiver to believe that all messages originated from the same IP address. With stock syslogd, I can not differentiate between the senders. Is there any way to record the correct sender of the message with rsyslog?

Answer: OK, I've now had some real lab time. The good news in short: if you use rsyslog both on the senders as well as on the receiver, you do NOT have any problems with NAT.

To double-check (and out of curiosity), I also tried with stock syslogd. I used the ones that came with RedHat and FreeBSD. Neither of them reports the sending machine correctly, they all report the NAT address. Obviously, this is what made this thread appear, but it is a good verification for the correctness of my lab. Next, I tried rsyslogd on the sender and stock syslogd on the receiver (just RedHat this time). The machine was still incorrectly displayed as the NAT address. However, now the real machine name immediately followed the NAT address, so you could differentiate the different machines – but in a inconsistent way.

Finally, I tried to run the stock syslogds against rsyslogd. Again, the host was not properly displayed. Actually, this time the host was not displayed at all (with the default rsyslogd template). Instead, the tag showed up in the host field. So this configuration is basically unusable.

The root cause of the NAT issue with stock syslogd obviously is that it does NOT include the HOST header that should be sent as of RFC 3164. This requires the receiver to take the host from the socket, which – in a NATed environment – can only hold the mangled NAT address. Rsyslog instead includes the HOST header, so the actual host name can be taken from that (this is the way rsyslog works with the default templates).

I barely remember seeing this in code when I initially forked rsyslog from sysklogd. I have not verified it once again. I have also not tested with syslog-ng, simply because that is not my prime focus and a lab would have required too much time.

To make a long story short: If you use rsyslog on both the senders and receivers, NAT is no issue for you.

How reliable should reliable logging be?

With any logging, you need to decide what you want to do if the log cannot be written

- do you want the application to stop because it can't write a log message

or

- do you want the application to continue, but not write the log message

Note that this decision is still there even if you are not logging remotely, your local disk partition where you are writing logs can fill up, become read-only, or have other problems.

The RFC for syslog (dating back a couple of decades, well before rsyslog started) specify that the application writing the log message should block and wait for the log message to be processed. Rsyslog (like every other modern syslog daemon) fudges this a bit and buffers the log data in RAM rather than following the original behavior of writing the data to disk and doing a fsync before acknowledging the log message.

If you have a problem with your output from rsyslog, your application will keep running until rsyslog fills it's queues, and then it will stop.

When you configure rsyslog to send the logs to another machine (either to rsyslog on another machine or to some sort of database), you introduce a significant new set of failure modes for the output from rsyslog.

You can configure the size of the rsyslog memory queues (I had one machine dedicated to running rsyslog where I created queues large enough to use >100G of ram for logs)

You can configure rsyslog to spill from it's memory queues to disk queues (disk assisted queue mode) when it fills it's memory queues.

You can create a separate set of queues for the action that has a high probability of failing (sending to a remote machine via TCP in this case), but this doesn't buy you more time, it just means that other logs can continue to be written when the remote system is down.

You can configure rsyslog to have high/low watermark levels, when the queue fills past the high watermark, rsyslog will start discarding logs below a specified severity, and stop doing so when it drops below the low watermark level

For rsyslog -> *syslog, you can use UDP for your transport so that the logs will get dropped at the network layer if the remote system is unresponsive.

You have lots of options.

If you are really concerned with reliability, I should point out that using TCP does not eliminate the possibility of loosing logs when a remote system goes down. When you send a message via TCP, the sender considers it sent when it's handed to the OS to send it. The OS has a window of how much data it allows to be outstanding (sent without acknowledgement from the remote system), and when the TCP connection fails (due to a firewall or a remote machine going down), the sending OS has no way to tell the application what data what data is outstanding, so the outstanding data will be lost. This is a smaller window of loss than UDP, which will happily keep sending your data forever, but it's still a potential for loss. Rsyslog offers the RELP (Reliable Event Logging Protocol), which addresses this problem by using application level acknowledgements so no messages can get lost due to network issues. That just leaves memory buffering (both in rsyslog and in the OS after rsyslog tells the OS to write the logs) as potential data loss points. Those failures will only trigger if the system crashes or rsyslog is shutdown (and yes, there are ways to address these as well)

The reason why nothing today operates without the possibility of loosing log messages is that making the logs completely reliable absolutely kills performance. With buffering, rsyslog can handle 400,000 logs/sec on a low-mid range machine. With utterly reliable logs and spinning disks, this rate drops to <100 logs/sec. With a $5K PCI SSD card, you can get up to ~4,000 logs/sec (in both cases, at the cost of not being able to use the disk for anything else on the system (so if you do use the disk for anything else, performance drops from there, and pretty rapidly). This is why traditional syslog had a reputation for being very slow.

See Also

- http://blog.gerhards.net/2008/04/on-unreliability-of-plain-tcp-syslog.html

Free Services for Rsyslog

A personal word from Rainer, the lead developer of rsyslog:

The rsyslog community provides ample free support resources. Please see our troubleshooting guide **to get started.**

Every now and then I receive private mail with support questions. I appreciate any feedback, but I must limit my resources so that I can help driver a great logging system forward.

To do so, I have decided not to reply to unsolicited support emails, at least not with a solution (but rather a link to this page ;)). I hope this does not offend you. The reason is quite simple: If I do personal support, you gain some advantage without contributing something back. Think about it: if you ask your question on the public forum or mailing list, others with the same problem can help you and, most importantly, even years later find your post (and the answer) and get the problem solved. So by solving your issue in public, you help create a great community resource and also help

your fellow users find solutions quicker. In the long term, this also contributes to improved code because the more questions users can find solutions to themselves, the fewer I need to look at.

But it becomes even better: the rsyslog community is much broader than Rainer ;) - there are other helpful members hanging around the public places. They often answer questions, so that I do not need to look at them (btw, once again a big "thank you", folks!). And, more important, those folks have a different background than me. So they often either know better how to solve your problem (e.g. because it is distro-specific) or they know how to better phrase it (after all, I like abstract terms and concepts ;)). So you do yourself a favor if you use the public places.

An excellent place to go to is the rsyslog forum inside the knowledge base (which in itself is a great place to visit!). For those used to mailing lists, the rsyslog mailing list also offers excellent advise.

Don't like to post your question in a public place? Well, then you should consider purchasing rsyslog professional support. The fees are very low and help fund the project. If you use rsyslog seriously inside a corporate environment, there is no excuse for not getting one of the support packages ;)

Of course, things are different when I ask you to mail me privately. I'll usually do that when I think it makes sense, for example when we exchange debug logs.

I hope you now understand the free support options and the reasoning for them. I hope I haven't offended you with my words - this is not my intension. I just needed to make clear why there are some limits on my responsiveness. Happy logging!

Compatibility

Compatibility Notes for rsyslog v8

This document describes things to keep in mind when moving from v7 to v8. It does not list enhancements nor does it talk about compatibility concerns introduced by earlier versions (for this, see their respective compatibility documents). Its focus is primarily on what you need to know if you used v7 and want to use v8 without hassle.

Version 8 offers a completely rewritten core rsyslog engine. This resulted in a number of changes that are visible to users and (plugin) developers. Most importantly, pre-v8 plugins **do not longer work** and need to be updated to support the new calling interfaces. If you developed a plugin, be sure to review the developer section below.

Mark Messages

In previous versions, mark messages were by default only processed if an action was not executed for some time. The default has now changed, and mark messages are now always processed. Note that this enables faster processing inside rsyslog. To change to previous behaviour, you need to add action.writeAllMarkMessages="off" to the actions in question.

Untested Modules

The following modules have been updated and successfully build, but no "real" test were conducted. Users of these modules should use extra care.

- mmsequence
- plugins/omgssapi
- omsnmp
- mmfields
- mmpstrucdata

- plugins/mmaudit
- ommongodb - larger changes still outstanding
- ompgsql
- plugins/omrabbitmq - not project supported
- plugins/omzmq3 - not project supported
- plugins/omhdfs (transaction support should be improved, requires sponsor)
- omuxsock

In addition to bug reports, success reports are also appreciated for these modules (this may save us testing).

What Developers need to Know

output plugin interface

To support the new core engine, the output interface has been considerably changed. It is suggested to review some of the project-provided plugins for full details. In this doc, we describe the most important changes from a high level perspective.

Multi-thread awareness required

The new engine activates one **worker**instance of output actions on each worker thread. This means an action has now three types of data:

- global
- action-instance - previously known pData, one for each action inside the config
- worker-action-instance - one for each worker thread (called pWrkrData), note that this is specific to exactly one pData

The plugin **must** now by multi-threading aware. It may be called by multiple threads concurrently, but it is guaranteed that each call is for a unique pWrkrData structure. This still permits to write plugins easily, but enables the engine to work with much higher performance. Note that plugin developers should assume it is the norm that multiple concurrent worker action instances are active a the some time.

New required entry points

In order to support the new threading model, new entry points are required. Most importantly, only the plugin knows which data must be present in pData and pWrkrData, so it must created and destroy these data structures on request of the engine. Note that pWrkrData may be destroyed at any time and new ones re-created later. Depending on workload structure and configuration, this can happen frequently.

New entry points are:

- createWrkrInstance
- freeWrkrInstance

The calling interface for these entry points has changed. Basically, they now receive a pWrkrData object instead pData. It is assumed that createWrkrInstance populates pWrkrData->pData appropriately.

- beginTransaction
- doAction
- endTransaction

Changed entry points

Some of the existing entry points have been changed.

The **doAction** entry point formerly got a variable `iMsgOpts` which is no longer provided. This variable was introduced in early days and exposed some internal message state information to the output module. Review of all known existing plugins showed that none except omfile ever used that variable. And omfile only did so to do some no longer required legacy handling.

In essence, it is highly unlikely that you ever accessed this variable. So we expect that nobody actually notices that the variable has gone away.

Removal of the variable provides a slight performance gain, as we do no longer need to maintain it inside the output system (which leads to less CPU instructions and better cache hits).

RS_RET_SUSPENDED is no longer supported when creating an action instance

This means a plugin must not try to establish any connections or the like before any of its processing entry points (like beginTransaction or doAction) is called. This was generally also the case von v7, but was not enforced in all cases. In v8, creating action creation fails if anything but RS_RET_OK is returned.

string generator interface

Bottom line: string generators need to be changed or will abort.

The BEGINstrgen() entry point has greatly changed. Instead of two parameters for the output buffers, they now receive a single `iparam` pointer, which contains all data items needed. Also, the message pointer is now const to "prevent" (accidential) changes to the message via the strgen interface.

Note that strgen modules must now maintain the iparam->lenStr field, which must hold the length of the generated string on exit. This is necessary as we cache the string sizes in order to reduced strlen() calls. Also, the numerical parameters are now unsigned and no longer size_t. This permits us to store them directly into optimized heap structures.

Specifics for Version 8.3 and 8.4

Unsupported Command Line Options Removed

The command line options a, h, m, o, p, g, r, t and c were not supported since many versions. However, they spit out an error message that they were unsupported. This error message now no longer appears, instead the regular usage() display happens. This should not have any effect to users.

Specifics for Version 8.5 and 8.6

imfile changes

Starting with 8.5.0, imfile supports wildcards in file names, but does do so only in inotify mode. In order to support wildcards, the handling of statefile needed to be changed. Most importantly, the *statefile* input parameter has been deprecated. See *imfile module documentation* for more details.

Command Line Options

There is a small set of configuration command line options available dating back to the dark ages of syslog technology. Setting command-line options is distro specific and a hassle for most users. As such, we are phasing out these options, and will do so rather quickly.

Some of them (most notably -l, -s) will completely be removed, as feedback so far indicated they are no longer in use. Others will be replaced by proper configuration objects.

Expect future rsyslog versions to no longer accept those configuration command line options.

Please see this table to see what to use as a replacement for the current options:

Option	replacement
-4	global(net.ipprotocol="ipv4-only")
-6	global(net.ipprotocol="ipv6-only")
-A	omfwd input parameter "udp.sendToAll"
-l	dropped, currently no replacement
-q	global(net.aclAddHostnameOnFail="on")
-Q	global(net.aclResolveHostname="off")
-s	dropped, currently no replacement
-S	omrelp action parameter "localclientip"
-w	global(net.permitACLWarning="off")
-x	global(net.enableDNS="off")

Compatibility Notes for rsyslog v7

This document describes things to keep in mind when moving from v6 to v7. It does not list enhancements nor does it talk about compatibility concerns introduced by earlier versions (for this, see their respective compatibility documents). Its focus is primarily on what you need to know if you used v6 and want to use v7 without hassle.

Version 7 builds on the new config language introduced in v6 and extends it. Other than v6, it not just only extends the config language, but provides considerable changes to core elements as well. The result is much more power and ease of use as well (this time that is not contradictionary).

BSD-Style blocks

BSD style blocks are no longer supported (for good reason). See the rsyslog BSD blocks info page for more information and how to upgrade your config.

CEE-Properties

In rsyslog v6, CEE properties could not be used across disk-based queues. If this was done, their content was reset. This was a missing feature in v6. In v7, this feature has been implemented. Consequently, situations where the previous behaviour were desired need now to be solved differently. We do not think that this will cause any problems to anyone, especially as in v6 this was announced as a missing feature.

omusrmsg: using just a username or "*" is deprecated

In legacy config format, the asterisk denotes writing the message to all users. This is usually used for emergency messages and configured like this:

```
*.emerg  *
```

Unfortunately, the use of this single character conflicts with other uses, for example with the multiplication operator. While rsyslog up to versions v7.4 preserves the meaning of asterisk as an action, it is deprecated and will probably be removed in future versions. Consequently, a warning message is emitted. To make this warning go away, the action must be explicitly given, as follows:

```
*.emerg  :omusrmsg:*
```

The same holds true for user names. For example

```
*.emerg  john
```

at a minimum should be rewritten as

```
*.emerg  :omusrmsg:john
```

Of course, for even more clarity the new RainerScript style of action can also be used:

```
*.emerg  action(type="omusrmsg" users="john")
```

In Rainer's blog, there is more background information on why omusrmsg needed to be changed available.

omruleset and discard (~) action are deprecated

Both continue to work, but have been replaced by better alternatives.

The discard action (tilde character) has been replaced by the "stop" RainerScript directive. It is considered more intuitive and offers slightly better performance.

The omruleset module has been replaced by the "call" RainerScript directive. Call permits to execute a ruleset like a subroutine, and does so with much higher performance than omruleset did. Note that omruleset could be run off an async queue. This was more a side than a desired effect and is not supported by the call statement. If that effect was needed, it can simply be simulated by running the called rulesets actions asynchronously (what in any case is the right way to handle this).

Note that the deprecated modules emit warning messages when being used. They tell that the construct is deprecated and which statement is to be used as replacement. This does **not** affect operations: both modules are still fully operational and will not be removed in the v7 timeframe.

Retries of output plugins that do not do proper replies

Some output plugins may not be able to detect if their target is capable of accepting data again after an error (technically, they always return OK when TryResume is called). Previously, the rsyslog core engine suspended such an action after 1000 successive failures. This lead to potentially a large amount of errors and error messages. Starting with 7.2.1, this has been reduced to 10 successive failures. This still gives the plugin a chance to recover. In extreme cases, a plugin may now enter suspend mode where it previously did not do so. In practice, we do NOT expect that.

omfile: file creation time

Originally, omfile created files immediately during startup, no matter if they were written to or not. In v7, this has changed. Files are only created when they are needed for the first time.

Also, in pre-v7 files were created *before* privileges were dropped. This meant that files could be written to locations where the actual desired rsyslog user was *not* permitted to. In v7, this has been fixed. This is fix also the prime reason that files are now created on demand (which is later in the process and after the privilege drop).

Notes for the 7.3/7.4 branch

"last message repeated n times" Processing

This processing has been optimized and moved to the input side. This results in usually far better performance and also de-couples different sources from the same processing. It is now also integrated in to the more generic rate-limiting processing.

User-Noticable Changes

The code works almost as before, with two exceptions:

- The suppression amount can be different, as the new algorithm precisely check's a single source, and while that source is being read. The previous algorithm worked on a set of mixed messages from multiple sources.
- The previous algorithm wrote a "last message repeated n times" message at least every 60 seconds. For performance reasons, we do no longer do this but write this message only when a new message arrives or rsyslog is shut down.

Note that the new algorithms needs support from input modules. If old modules which do not have the necessary support are used, duplicate messages will most probably not be detected. Upgrading the module code is simple, and all rsyslog-provided plugins support the new method, so this should not be a real problem (crafting a solution would result in rather complex code - for a case that most probably would never happen).

Performance Implications

In general, the new method enables far faster output processing. However, it needs to be noted that the "last message repeated n" processing needs parsed messages in order to detect duplicated. Consequently, if it is enabled the parser step cannot be deferred to the main queue processing thread and thus must be done during input processing. The changes workload distribution and may have (good or bad) effect on the overall performance. If you have a very high performance installation, it is suggested to check the performance profile before deploying the new version.

Note: for high-performance environments it is highly recommended NOT to use "last message repeated n times" processing but rather the other (more efficient) rate-limiting methods. These also do NOT require the parsing step to be done during input processing.

Stricter string-template Processing

Previously, no error message for invalid string template parameters was generated. Rather a malformed template was generated, and error information emitted at runtime. However, this could be quite confusing. Note that the new code changes user experience: formerly, rsyslog and the affected actions properly started up, but the actions did not produce proper data. Now, there are startup error messages and the actions are NOT executed (due to missing template due to template error).

Compatibility Notes for rsyslog v6

This document describes things to keep in mind when moving from v5 to v6. It does not list enhancements nor does it talk about compatibility concerns introduced by earlier versions (for this, see their respective compatibility documents). Its focus is primarily on what you need to know if you used a previous version and want to use the current one without hassle.

Version 6 offers a better config language and some other improvements. As the config system has many ties into the rsyslog engine AND all plugins, the changes are somewhat intrusive. Note, however, that core processing has not been changed much in v6 and will not. So once the configuration is loaded, the stability of v6 is quite comparable to v5.

Property "pri-text"

Traditionally, this property did not only return the textual form of the pri ("local0.err"), but also appended the numerical value to it ("local0.err<133>"). This sounds odd and was left unnoticed for some years. In October 2011, this odd behaviour was brought up on the rsyslog mailing list by Gregory K. Ruiz-Ade. Code review showed that the behaviour was intentional, but no trace of what the intention was when it was introduced could be found. The documentation was also unclear, it said no numerical value was present, but the samples had it. We agreed that the additional numerical value is of disadvantage. We also guessed that this property is very rarely being used, otherwise the problem should have been raised much earlier. However, we didn't want to change behaviour in older builds. So v6 was set to clean up the situation. In v6, text-pri will always return the textual part only ("local0.err") and the numerical value will not be contained any longer inside the string. If you actually need that value, it can fairly easily be added via the template system. **If you have used this property previously and relied on the numerical part, you need to update your rsyslog configuration files.**

Plugin ABI

The plugin interface has considerably been changed to support the new config language. All plugins need to be upgraded. This usually does not require much coding. However, if the new config language shall be supported, more changes must be made to plugin code. All project-supported plugins have been upgraded, so this compatibility issue is only of interest for you if you have custom plugins or use some user-contributed plugins from the rsyslog project that are not maintained by the project itself (omoracle is an example). Please expect some further plugin instability during the initial v6 releases.

RainerScript based rsyslog.conf

A better config format was the main release target for rsyslog v6. It comes in the flavor of so-called RainerScript (why the name RainerScript?) RainerScript supports legacy syslog.conf format, much as you know it from other syslogds (like sysklogd or the BSD syslogd) as well as previous versions of rsyslog. Initial work on RainerScript began in v4, and the if-construct was already supported in v4 and v5. Version 6 has now taken this further. After long discussions we decided to use the legacy format as a basis, and lightly extend it by native RainerScript constructs. The main goal was to make sure that previous knowledge and config systems could still be used while offering a much more intuitive and powerful way of configuring rsyslog.

RainerScript has been implemented from scratch and with new tools (flex/bison, for those in the know). Starting with 6.3.3, this new config file processor replaces the legacy one. Note that the new processor handles all formats, extended RainerScript as well as legacy syslog.conf format. There are some legacy construct that were especially hard to translate. You'll read about them in other parts of this document (especially outchannels, which require a format change).

In v6, all legacy formats are supported. In the long term, we may remove some of the ugly rsyslog-specific constructs. Good candidates are all configuration commands starting with a dollar sign, like "$ActionFileDefaultTemplate").

However, this will not be the case before rsyslog v7 or (much more likely) v8/9. Right now, you also need to use these commands, because not all have already been converted to the new RainerScript format.

In 6.3.3, the new parser is used, but almost none of the extended RainerScript capabilities are available. They will incrementally be introduced with the following releases. Note that for some features (most importantly if-then-else nested blocks), the v6 core engine is not capable enough. It is our aim to provide a much better config language to as many rsyslog users as quickly as possible. As such, we refrain from doing big engine changes in v6. This in turn means we cannot introduce some features into RainerScript that we really want to see. These features will come up with rsyslog v7, which will have even better flow control capabilities inside the core engine. Note that v7 will fully support v6 RainerScript. Let us also say that the v6 version is not a low-end quick hack: it offers full-fledged syslog message processing control, capable of doing the best you can find inside the industry. We just say that v7 will come up with even more advanced capabilities.

Please note that we tried hard to make the RainerScript parser compatible with all legacy config files. However, we may have failed in one case or another. So if you experience problems during config processing, chances are there may be a problem on the rsyslog side. In that case, please let us know.

Please see the blog post about rsyslog 6.3.3 config format for details of what is currently supported.

compatibility mode

Compatibility mode (specified via -c option) has been removed. This was a migration aid from sysklogd and very early versions of rsyslog. As all major distros now have rsyslog as their default, and thus ship rsyslog-compliant config files, there is no longer a need for compatibility mode. Removing it provides easier to maintain code. Also, practice has shown that many users were confused by compatibility mode (and even some package maintainers got it wrong). So this not only cleans up the code but rather removes a frequent source of error.

It must be noted, though, that this means rsyslog is no longer a 100% drop-in replacement for sysklogd. If you convert an extremely old system, you need to checks its config and probably need to apply some very mild changes to the config file.

abort on config errors

Previous versions accepted some malformedness inside the config file without aborting. This could lead to some uncertainty about which configuration was actually running. In v6 there are some situations where config file errors can not be ignored. In these cases rsyslog emits error messages to stderr, and then exists with a non-zero exit code. It is important to check for those cases as this means log data is potentially lost. Please note that the root problem is the same for earlier versions as well. With them, it was just harder to spot why things went wrong (and if at all).

Default Batch Sizes

Due to their positive effect on performance and comparatively low overhead, default batch sizes have been increased. Starting with 6.3.4, the action queues have a default batch size of 128 messages.

Default action queue enqueue timeout

This timeout previously was 2 seconds, and has been reduced to 50ms (starting with 6.5.0). This change was made as a long timeout will caused delays in the associated main queue, something that was quite unexpected to users. Now, this can still happen, but the effect is much less harsh (but still considerable on a busy system). Also, 50ms should be fairly enough for most output sources, except when they are really broken (like network disconnect). If they are really broken, even a 2second timeout does not help, so we hopefully get the best of both worlds with the new timeout. A specific timeout can of course still be configured, it is just the timeout that changed.

outchannels

Outchannels are a to-be-removed feature of rsyslog, at least as far as the config syntax is concerned. Nevertheless, v6 still supports it, but a new syntax is required for the action. Let's assume your outchannel is named "channel". The previous syntax was

```
*.* $channel
```

This was deprecated in v5 and no longer works in v6. Instead, you need to specify

```
*.* :omfile:$channel
```

Note that this syntax is available starting with rsyslog v4. It is important to keep on your mind that future versions of rsyslog will require different syntax and/or drop outchannel support completely. So if at all possible, avoid using this feature. If you must use it, be prepared for future changes and watch announcements very carefully.

ompipe default template

Starting with 6.5.0, ompipe does no longer use the omfile default template. Instead, the default template must be set via the module load statement. An example is

```
module(load="builtin:ompipe" template="myDefaultTemplate")
```

For obvious reasons, the default template must be defined somewhere in the config file, otherwise errors will happen during the config load phase.

omusrmsg

The omusrmsg module is used to send messages to users. In legacy-legacy config format (that is the very old sysklogd style), it was sufficient to use just the user name to call this action, like in this example:

```
*.* rgerhards
```

This format is very ambiguous and causes headache (see blog post on omusrmsg for details). Thus the format has been superseded by this syntax (which is legacy format ;-)):

```
*.* :omusrmsg:rgerhards
```

That syntax is supported since later subversions of version 4.

Rsyslog v6 still supports the legacy-legacy format, but in a very strict sense. For example, if multiple users or templates are given, no spaces must be included in the action line. For example, this works up to v5, but no longer in v6:

```
*.* rgerhards, bgerhards
```

To fix it in a way that is compatible with pre-v4, use (note the removed space!):

```
*.* rgerhards,bgerhards
```

Of course, it probably is better to understand in native v6 format:

```
*.* action(type="omusrmsg" users="rgerhards, bgerhards")
```

As you see, here you may include spaces between user names.

In the long term, legacy-legacy format will most probably totally disappear, so it is a wise decision to change config files at least to the legacy format (with ":omusrmsg:" in front of the name).

Escape Sequences in Script-Based Filters

In v5, escape sequences were very simplistic. Inside a string, "x" meant "x" with x being any character. This has been changed so that the usual set of escapes is supported, must importantly "n", "t", "xhh" (with hh being hex digits) and "ooo" with (o being octal digits). So if one of these sequences was used previously, results are obviously different. However, that should not create any real problems, because it is hard to envision why someone should have done that (why write "n" when you can also write "n"?).

Copyright

Compatibility Notes for rsyslog v5

Written by Rainer Gerhards *(2009-07-15)*

The changes introduced in rsyslog v5 are numerous, but not very intrusive. This document describes things to keep in mind when moving from v4 to v5. It does not list enhancements nor does it talk about compatibility concerns introduced by earlier versions (for this, see their respective compatibility documents).

HUP processing

The $HUPisRestart directive is supported by some early v5 versions, but has been removed in 5.1.3 and above. That means that restart-type HUP processing is no longer available. This processing was redundant and had a lot a drawbacks. For details, please see the rsyslog v4 compatibility notes which elaborate on the reasons and the (few) things you may need to change.

Please note that starting with 5.8.11 HUP will also requery the local hostname.

Queue on-disk format

The queue information file format has been changed. When upgrading from v4 to v5, make sure that the queue is emptied and no on-disk structure present. We did not go great length in understanding the old format, as there was too little demand for that (and it being quite some effort if done right).

Queue Worker Thread Shutdown

Previous rsyslog versions had the capability to "run" on zero queue worker if no work was required. This was done to save a very limited number of resources. However, it came at the price of great complexity. In v5, we have decided to let a minimum of one worker run all the time. The additional resource consumption is probably not noticeable at all, however, this enabled us to do some important code cleanups, resulting in faster and more reliable code (complex code is hard to maintain and error-prone). From the regular user's point of view, this change should be barely noticeable. I am including the note for expert users, who will notice it in rsyslog debug output and other analysis tools. So it is no error if each queue in non-direct mode now always runs at least one worker thread.

Compatibility Notes for rsyslog v4

Written by Rainer Gerhards *(2009-07-15)*

The changes introduced in rsyslog v4 are numerous, but not very intrusive. This document describes things to keep in mind when moving from v3 to v4. It does not list enhancements nor does it talk about compatibility concerns introduced by v3 (for this, see the rsyslog v3 compatibility notes).

HUP processing

With v3 and below, rsyslog used the traditional HUP behaviour. That meant that all output files are closed and the configuration file is re-read and the new configuration applied.

With a program as simple and static as sysklogd, this was not much of an issue. The most important config settings (like udp reception) of a traditional syslogd can not be modified via the configuration file. So a config file reload only meant setting up a new set of filters. It also didn't account as problem that while doing so messages may be lost - without any threading and queuing model, a traditional syslogd will potentially always loose messages, so it is irrelevant if this happens, too, during the short config re-read phase.

In rsyslog, things are quite different: the program is more or less a framework into which loadable modules are loaded as needed for a particular configuration. The software that will actually be running is tailored via the config file. Thus, a re-read of the config file requires a full, very heavy restart, because the software actually running with the new config can be totally different from what ran with the old config.

Consequently, the traditional HUP is a very heavy operation and may even cause some data loss because queues must be shut down, listeners stopped and so on. Some of these operations (depending on their configuration) involve intentional message loss. The operation also takes up a lot of system resources and needs quite some time (maybe seconds) to be completed. During this restart period, the syslog subsystem is not fully available.

From the software developer's point of view, the full restart done by a HUP is rather complex, especially if user-timeout limits set on action completion are taken into consideration (for those in the know: at the extreme ends this means we need to cancel threads as a last resort, but than we need to make sure that such cancellation does not happen at points where it would be fatal for a restart). A regular restart, where the process is actually terminated, is much less complex, because the operating system does a full cleanup after process termination, so rsyslogd does not need to take care for exotic cleanup cases and leave that to the OS. In the end result, restart-type HUPs clutter the code, increase complexity (read: add bugs) and cost performance.

On the contrary, a HUP is typically needed for log rotation, and the real desire is to close files. This is a non-disruptive and very lightweight operation.

Many people have said that they are used to HUP the syslogd to apply configuration changes. This is true, but it is questionable if that really justifies all the cost that comes with it. After all, it is the difference between typing

```
$ kill -HUP `cat /var/run/rsyslogd.pid`
```

versus

```
$ /etc/init.d/rsyslog restart
```

Semantically, both is mostly the same thing. The only difference is that with the restart command rsyslogd can spit config error message to stderr, so that the user is able to see any problems and fix them. With a HUP, we do not have access to stderr and thus can log error messages only to their configured destinations; experience tells that most users will never find them there. What, by the way, is another strong argument against restarting rsyslogd by HUPing it.

So a restart via HUP is not strictly necessary and most other daemons require that a restart command is typed in if a restart is required.

Rsyslog will follow this paradigm in the next versions, resulting in many benefits. In v4, we provide some support for the old-style semantics. We introduced a setting $HUPisRestart which may be set to "on" (traditional, heavy operation) or "off" (new, lightweight "file close only" operation). The initial versions had the default set to traditional behavior, but starting with 4.5.1 we are now using the new behavior as the default.

Most importantly, **this may break some scripts**, but my sincere belief is that there are very few scripts that automatically **change** rsyslog's config and then do a HUP to reload it. Anyhow, if you have some of these, it may be a good idea to change them now instead of turning restart-type HUPs on. Other than that, one mainly needs to change the habit of how to restart rsyslog after a configuration change.

Please note that restart-type HUP is deprecated and will go away in rsyslog v5. So it is a good idea to become ready for the new version now and also enjoy some of the benefits of the "real restart", like the better error-reporting capability.

Note that code complexity reduction (and thus performance improvement) needs the restart-type HUP code to be removed, so these changes can (and will) only happen in version 5.

outchannels

Note: as always documented, outchannels are an experimental feature that may be removed and/or changed in the future. There is one concrete change done starting with 4.6.7: let's assume an outchannel "mychannel" was defined. Then, this channel could be used inside an

. $mychannel

This is still supported and will remain to be supported in v4. However, there is a new variant which explicitly tells this is to be handled by omfile. This new syntax is as follows:

. :omfile:$mychannel

Note that future versions, specifically starting with v6, the older syntax is no longer supported. So users are strongly advised to switch to the new syntax. As an aid to the conversion process, rsyslog 4.7.4 and above issue a warning message if the old-style directive is seen – but still accept the old syntax without any problems.

Compatibility Notes for rsyslog v3

Written by Rainer Gerhards *(2008-03-28)*

Rsyslog aims to be a drop-in replacement for sysklogd. However, version 3 has some considerable enhancements, which lead to some backward compatibility issues both in regard to sysklogd and rsyslog v1 and v2. Most of these issues are avoided by default by not specifying the -c option on the rsyslog command line. That will enable backwards-compatibility mode. However, please note that things may be suboptimal in backward compatibility mode, so the advise is to work through this document, update your rsyslog.conf, remove the no longer supported startup options and then add -c3 as the first option to the rsyslog command line. That will enable native mode.

Please note that rsyslogd helps you during that process by logging appropriate messages about compatibility mode and backwards-compatibility statements automatically generated. You may want your syslogd log for those. They immediately follow rsyslogd's startup message.

Inputs

With v2 and below, inputs were automatically started together with rsyslog. In v3, inputs are optional! They come in the form of plug-in modules. **At least one input module must be loaded to make rsyslog do any useful work.** The config file directives doc briefly lists which config statements are available by which modules.

It is suggested that input modules be loaded in the top part of the config file. Here is an example, also highlighting the most important modules:

```
$ModLoad immark # provides --MARK-- message capability
$ModLoad imudp # provides UDP syslog reception
$ModLoad imtcp # provides TCP syslog reception
$ModLoad imgssapi # provides GSSAPI syslog reception
$ModLoad imuxsock # provides support for local system logging (e.g. via logger␣
↪command)
$ModLoad imklog # provides kernel logging support (previously done by rklogd)
```

Command Line Options

A number of command line options have been removed. New config file directives have been added for them. The -h and -e option have been removed even in compatibility mode. They are ignored but an informative message is logged. Please note that -h was never supported in v2, but was silently ignored. It disappeared some time ago in the final v1 builds. It can be replaced by applying proper filtering inside syslog.conf.

-c option / Compatibility Mode

The -c option is new and tells rsyslogd about the desired backward compatibility mode. It must always be the first option on the command line, as it influences processing of the other options. To use the rsyslog v3 native interface, specify -c3. To use compatibility mode , either do not use -c at all or use -c<vers> where vers is the rsyslog version that it shall be compatible to. Use -c0 to be command-line compatible to sysklogd.

Please note that rsyslogd issues warning messages if the -c3 command line option is not given. This is to alert you that your are running in compatibility mode. Compatibility mode interferes with your rsyslog.conf commands and may cause some undesired side-effects. It is meant to be used with a plain old rsyslog.conf - if you use new features, things become messy. So the best advise is to work through this document, convert your options and config file and then use rsyslog in native mode. In order to aid you in this process, rsyslog logs every compatibility-mode config file directive it has generated. So you can simply copy them from your logfile and paste them to the config.

-e Option

This option is no longer supported, as the "last message repeated n times" feature is now turned off by default. We changed this default because this feature is causing a lot of trouble and we need to make it either go away or change the way it works. For more information, please see our dedicated forum thread on "last message repeated n times". This thread also contains information on how to configure rsyslogd so that it continues to support this feature (as long as it is not totally removed).

-m Option

The -m command line option is emulated in compatibility mode. To replace it, use the following config directives (compatibility mode auto-generates them):

```
$ModLoad immark
$MarkMessagePeriod 1800 # 30 minutes
```

-r Option

Is no longer available in native mode. However, it is understood in compatibility mode (if no -c option is given). Use the `$UDPSeverRun <port>` config file directives. You can now also set the local address the server should listen to via `$UDPServerAddress <ip>` config directive.

The following example configures an UDP syslog server at the local address 192.0.2.1 on port 514:

```
$ModLoad imudp
$UDPServerAddress 192.0.2.1 # this MUST be before the $UDPServerRun directive!
$UDPServerRun 514
```

`$UDPServerAddress \*` means listen on all local interfaces. This is the default if no directive is specified.

Please note that now multiple listeners are supported. For example, you can do the following:

```
$ModLoad imudp
$UDPServerAddress 192.0.2.1 # this MUST be before the $UDPServerRun directive!
$UDPServerRun 514
$UDPServerAddress \* # all local interfaces
$UDPServerRun 1514
```

These config file settings run two listeners: one at 192.0.2.1:514 and one on port 1514, which listens on all local interfaces.

Default port for UDP (and TCP) Servers

Please note that with pre-v3 rsyslogd, a service database lookup was made when a UDP server was started and no port was configured. Only if that failed, the IANA default of 514 was used. For TCP servers, this lookup was never done and 514 always used if no specific port was configured. For consistency, both TCP and UDP now use port 514 as default. If a lookup is desired, you need to specify it in the "Run" directive, e.g. "*$UDPServerRun syslog*".

klogd

klogd has (finally) been replaced by a loadable input module. To enable klogd functionality, do

```
$ModLoad imklog
```

Note that this can not be handled by the compatibility layer, as klogd was a separate binary. A limited set of klogd command line settings is now supported via rsyslog.conf. That set of configuration directives is to be expanded.

Output File Syncing

Rsyslogd tries to keep as compatible to stock syslogd as possible. As such, it retained stock syslogd's default of syncing every file write if not specified otherwise (by placing a dash in front of the output file name). While this was a useful feature in past days where hardware was much less reliable and UPS seldom, this no longer is useful in today's world. Instead, the syncing is a high performance hit. With it, rsyslogd writes files around 50 **times** slower than without it. It also affects overall system performance due to the high IO activity. In rsyslog v3, syncing has been turned off by default. This is done via a specific configuration directive

:: $ActionFileEnableSync on/off

which is off by default. So even if rsyslogd finds sync selector lines, it ignores them by default. In order to enable file syncing, the administrator must specify `$ActionFileEnableSync on` at the top of rsyslog.conf. This ensures that syncing only happens in some installations where the administrator actually wanted that (performance-intense) feature. In the fast majority of cases (if not all), this dramatically increases rsyslogd performance without any negative effects.

Output File Format

Rsyslog supports high precision RFC 3339 timestamps and puts these into local log files by default. This is a departure from previous syslogd behaviour. We decided to sacrifice some backward-compatibility in an effort to provide a better logging solution. Rsyslog has been supporting the high-precision timestamps for over three years as of this writing, but nobody used them because they were not default (one may also assume that most people didn't even know about them). Now, we are writing the great high-precision time stamps, which greatly aid in getting the right sequence of logging events. If you do not like that, you can easily turn them off by placing

```
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat
```

right at the start of your rsyslog.conf. This will use the previous format. Please note that the name is case-sensitive and must be specified exactly as shown above. Please also note that you can of course use any other format of your liking. To do so, simply specify the template to use or set a new default template via the $ActionFileDefaultTemplate directive. Keep in mind, though, that templates must be defined before they are used.

Keep in mind that when receiving messages from remote hosts, the timestamp is just as precise as the remote host provided it. In most cases, this means you will only a receive a standard timestamp with second precision. If rsyslog is running at the remote end, you can configure it to provide high-precision timestamps (see below).

Forwarding Format

When forwarding messages to remote syslog servers, rsyslogd by default uses the plain old syslog format with second-level resolution inside the timestamps. We could have made it emit high precision timestamps. However, that would have broken almost all receivers, including earlier versions of rsyslog. To avoid this hassle, high-precision timestamps need to be explicitly enabled. To make this as painless as possible, rsyslog comes with a canned template that contains everything necessary. To enable high-precision timestamps, just use:

```
$ActionForwardDefaultTemplate RSYSLOG_ForwardFormat # for plain TCP and UDP
$ActionGSSForwardDefaultTemplate RSYSLOG_ForwardFormat # for GSS-API
```

And, of course, you can always set different forwarding formats by just specifying the right template.

If you are running in a system with only rsyslog 3.12.5 and above in the receiver roles, it is suggested to add one (or both) of the above statements to the top of your rsyslog.conf (but after the $ModLoad's!) - that will enable you to use the best in timestamp support available. Please note that when you use this format with other receivers, they will probably become pretty confused and not detect the timestamp at all. In earlier rsyslog versions, for example, that leads to duplication of timestamp and hostname fields and disables the detection of the original hostname in a relayed/NATed environment. So use the new format with care.

Queue Modes for the Main Message Queue

Either "FixedArray" or "LinkedList" is recommended. "Direct" is available, but should not be used except for a very good reason ("Direct" disables queueing and will potentially lead to message loss on the input side).

CHAPTER 3

Sponsors and Community

Please visit the rsyslog Sponsor's Page[4] to honor the project sponsors or become one yourself! We are very grateful for any help towards the project goals.

Visit the Rsyslog Status Page[2] to obtain current version information and project status.

If you like rsyslog, you might want to lend us a helping hand. It doesn't require a lot of time - even a single mouse click helps. Learn *how to help the rsyslog project*.

[4] Regular expression checker/generator tool for rsyslog
[2] rsyslog Sponsor's Page

CHAPTER 4

Related Links

Index