napalm-logs Documentation

Release Not installed

Mircea Ulinic

Mar 17, 2018

Contents

Python library to parse syslog messages from network devices and produce JSON serializable Python objects, in a vendor agnostic shape. The output objects are structured following the OpenConfig or IETF YANG models.

For example, the following syslog message from a Juniper device:

```
<149>Jun 21 14:03:12  vmx01 rpd[2902]: BGP_PREFIX_THRESH_EXCEEDED: 192.168.140.254␣
↪(External AS 4230): Configured maximum prefix-limit threshold(140) exceeded for␣
↪inet4-unicast nlri: 141 (instance master)
```

Will produce the following object:

```
{
  "yang_message": {
    "bgp": {
      "neighbors": {
        "neighbor": {
          "192.168.140.254": {
            "state": {
              "peer_as": "4230"
            },
            "afi_safis": {
              "afi_safi": {
                "inet4": {
                  "state": {
                    "prefixes": {
                      "received": 141
                    }
                  },
                  "ipv4_unicast": {
                    "prefix_limit": {
                      "state": {
                        "max_prefixes": 140
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  },
  "message_details": {
    "processId": "2902",
    "severity": 5,
    "facility": 18,
    "hostPrefix": null,
    "pri": "149",
    "processName": "rpd",
    "host": "vmx01",
    "tag": "BGP_PREFIX_THRESH_EXCEEDED",
    "time": "14:03:12",
    "date": "Jun 21",
    "message": "192.168.140.254 (External AS 4230): Configured maximum prefix-limit␣
↪threshold(140) exceeded for inet4-unicast nlri: 141 (instance master)"
  },
  "timestamp": 1498053792,
  "facility": 18,
```

```
  "ip": "127.0.0.1",
  "host": "vmx01",
  "yang_model": "openconfig-bgp",
  "error": "BGP_PREFIX_THRESH_EXCEEDED",
  "os": "junos",
  "severity": 5
}
```


The library is provided with a command line program which acts as a daemon, running in background and listening to syslog messages continuously, then publishing them over secured channels, where multiple clients can subscribe.

It is flexible to listen to the syslog messages via UDP or TCP, but also from brokers such as Apache Kafka. Similarly, the output objects can be published via various channels such as ZeroMQ, Kafka, or remote server logging. It is also pluggable enough to extend these capabilities and listen or publish to other services, depending on the needs.

The messages are published over a secured channel, encrypted and signed. Although the security can be disabled, this is highly discouraged.

CHAPTER 1

Output data

The objects published by napalm-logs are structured data, with the hierarchy standardized in the OpenConfig and IETF models. To check what models are used for each message type, together with examples of raw syslog messages and sample output objects, please check the *Structured Messages* section.

CHAPTER 2

Install

napalm-logs is available on PyPi and can easily be installed using the following command:

```
$ pip install napalm-logs
```

For advanced installation notes, see *Installation*.

CHAPTER 3

How to use napalm-logs

3.1 Basic Configuration

Firstly you need to decide if you would like all messages between *napalm-logs* and the clients to be encrypted. If you do want them to be encrypted you will require a certificate and key, which you can generate using the following command:

```
openssl req -nodes -x509 -newkey rsa:4096 -keyout /var/cache/napalm-logs.key -out /
↪var/cache/napalm-logs.crt -days 365
```

This will provide a self-signed certificate `napalm-logs.crt` and key `napalm-logs.key` under the `/var/cache` directory.

If you do not require the messages to be encrypted you can ignore the above step and just use the command line argument `--disable-security` when starting *napalm-logs*.

Each of the other config options come with defaults, so you can now start *napalm-logs* with default options and your chosen security options.

3.2 Starting napalm-logs

Napalm-logs will need to be run with root privileges if you want it to be able to listen on `udp` port `514` - the standard syslog port. If you need to run it via sudo and it has been installed in a virtual env, you will need to include the full path. In these examples I will run as root.

To start napalm-logs using the crt and key generated above you should run the following command:

```
napalm-logs --certificate /var/cache/napalm-logs.crt --keyfile /var/cache/napalm-logs.
↪key
```

This will start napalm-logs listening for incoming syslog messages on `0.0.0.0` port `514`. It will also start to listen for incoming client requests on `0.0.0.0` port `49017`, and incoming authentication requests on `0.0.0.0` port `49018`. For more information on authentication please see the *Client Authentication* section.

3.3 Further Configuration

It is possible to change the address and ports that napalm-logs will use, let's take a look at these options:

```
-a ADDRESS, --address=ADDRESS
                                              Listener address. Default: 0.0.0.0
-p PORT, --port=PORT  Listener bind port. Default: 514
--publish-address=PUBLISH_ADDRESS
                                              Publisher bind address. Default: 0.0.0.0
--publish-port=PUBLISH_PORT
                                              Publisher bind port. Default: 49017
--auth-address=AUTH_ADDRESS
                                              Authenticator bind address. Default: 0.
↪0.0.0
--auth-port=AUTH_PORT
                                              Authenticator bind port. Default: 49018
```

There are several `plugable` parts to napalm-logs, two of which are the `listener` and the `publisher`. The listener is the part that ingests the incoming syslog messages, and the publisher is the part that outputs them to the client.

You can chose which listener to use, and which publisher to use by using the following arguments:

```
--listener=LISTENER   Listener type. Default: udp
-t TRANSPORT, --transport=TRANSPORT
                                              Publish transport. Default: zmq
```

There are more configuration options, please see *Configuration Options* for more details.

3.4 Configuration file example

The napalm-logs server can be started without any CLI aguments, as long as they are correctly specified under the configuration file. The default path of the configuration file is under `/etc/napalm/logs`. To select a different filepath, we can use the `-c` option:

```
napalm-logs -c /home/admin/napalm/logs
```

The configuration file is formatted as YAML, which makes it more human readable. In general, any configuration option available on the CLI can be specified in the configuration file, with the mention that hyphen is replaced by underscore, e.g.: the CLI option `auth-address` becomes `auth_address` in the *napalm-logs* configuration file.

```
address: 172.17.17.1
port: 5514
publish_address: 172.17.17.2
publish_port: 49017
transport: zmq
listener:
  kafka:
    bootstrap_servers:
      - 10.10.10.1
      - 10.10.10.2
      - 10.10.10.3
```

The configuration above listens to the syslog messages from the Kafka bootstrap servers `10.10.10.1`, `10.10.10.2` and `10.10.10.3` then publishes the structured objects encrypted and serialized via ZeroMQ, serving them at the address 172.17.17.2, port 49017.

Check the complete list of configuration options under *Configuration Options*.

3.5 Starting a Client

The client structure depends on how you start the napalm-logs daemon. If the security is disabled (via the CLI option --disable-security or through the configuration file, where the disable_security field is set as false), the client script is as simple as:

```
#!/usr/bin/env python

import zmq
import napalm_logs.utils

server_address = '127.0.0.1'
server_port = 49017

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect('tcp://{address}:{port}'.format(address=server_address,
                                               port=server_port))
socket.setsockopt(zmq.SUBSCRIBE, '')

while True:
    raw_object = socket.recv()
    print(napalm_logs.utils.unserialize(raw_object))
```

Which subscribes to the ZeroMQ bus and deserializes messages using the napalm_logs.utils.unserialise helper. The server_address and the server_port of the client represent the --publish-address and the --publish-port of the napalm-logs daemon.

When the program is started with security enabled (**recommended**), the clients can use the napalm_logs.utils.ClientAuth class, which executes the handshake to retrieve the encryption key and hex of the verification key. This class requires the certificate (the same certificate specified when starting the napalm-logs daemon), as well as the authentication address and port (corresponding to the --auth-address and --auth-port CLI arguments or auth_address and auth_port configuration fields sent to the napalm-logs daemon):

```
#!/usr/bin/env python

import napalm_logs.utils
import zmq

server_address = '127.0.0.1'
server_port = 49017
auth_address = '127.0.0.1'
auth_port = 49018

certificate = '/var/cache/napalm-logs.crt' # This is the server crt generated earlier

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect('tcp://{address}:{port}'.format(address=server_address,
                                               port=server_port))
socket.setsockopt(zmq.SUBSCRIBE, '')
```

```
auth = napalm_logs.utils.ClientAuth(certificate,
                                    address=auth_address,
                                    port=auth_port)

while True:
    raw_object = socket.recv()
    decrypted = auth.decrypt(raw_object)
    print(decrypted)
```

3.5.1 Installation

Creating a Virtualenv

It is recommended to install all the modules required for a new program into a *Virtual Environment*. This ensures that the project dependencies are kept in its own environment, making sure that you don't have any versioning issues when other programs have the same dependencies.

```
virtualenv napalm-logs
```

This will create a directory called `napalm-logs` in the directory that you are currently in.

Now you need to activate the virtualenv:

```
source napalm-logs/bin/activate
```

Installing Napalm-logs

Now install napalm-logs using pip:

```
pip install napalm-logs
```

Docker

Napalm-logs can also be deployed via a Docker container.

A Dockerfile has been made available in the GitHub repository allowing the configuration of the container to be customized.

Alternatively, a pre-built image is available on Docker Hub which uses the UDP listener and publishes to Kafka by default. The pre-built image is recommended for testing only as Napalm-logs is executed with security disabled by default.

Usage:

```
docker run -d -p 6000:514/udp -i nathancatania/napalm-logs:latest
```

The above command will run the Napalm-logs container and listen on port 6000 (UDP) for incoming messages. By default, the pre-built container attempts to connect to a Kafka broker located at *127.0.0.1:9092* and will publish data to the *syslog.net* topic.

These defaults can be changed by specifying ENV variables at container runtime. For example:

```
docker run -d -e KAFKA_BROKER_HOST=192.168.1.200 -e KAFKA_BROKER_PORT=9094 -e KAFKA_
↪TOPIC=my_topic -p 55555:514/udp -i nathancatania/napalm-logs:latest
```

In this example:

- The container will listen on port 55555 for incoming messages.
- Napalm-logs will connect to a Kafka broker located at *192.168.1.200:9094*.
- Data will be published to the Kafka topic *my_topic*.

A list of available variables which can be changed is included below:

```
PUBLISH_PORT: 49017           # Source port of the host to publish data to Kafka on
KAFKA_BROKER_HOST: 127.0.0.1  # Hostname or IP of the Kafka broker to publish to
KAFKA_BROKER_PORT: 9092       # Port of the Kafka broker to publish to
KAFKA_TOPIC: syslog.net       # The Kafka topic to push data to.
SEND_RAW: true                # Publish messages where the OS, but NOT the message␣
↪could be identified.
SEND_UNKNOWN: false           # Publish messages where both OS and message could␣
↪not be identified.
WORKER_PROCESSES: 1           # Increasing this increases memory consumption but␣
↪is better for higher loads.
```

3.5.2 Supported devices and configuration

Napalm-logs can process syslog messages from the following network operating systems:

Junos

The following will configure Junos to send the syslog messages, over UDP, to the IP Address `10.10.10.1`, port 10101:

```
set system syslog host 10.10.10.1 port 10101 any any
```

Cisco IOS-XR

The following will configure IOS-XR to send the syslog messages, over UDP, to the IP Address `10.10.10.1`, port 10101:

```
logging 10.10.10.1 port 10101
```

To correctly send the hostname information, is is also recommended to explicitly configure the following:

```
logging hostnameprefix <hostname of the device>
```

Otherwise the device won't send this information.

Arista EOS

The following will configure EOS to send the syslog messages, over UDP, to the IP Address `10.10.10.1`, port 10101:

```
logging host 10.10.10.1 10101
```

To correctly send the hostname information, is is also recommended to explicitly configure the following:

```
logging format hostname fqdn
```

Cisco NX-OS

The following will configure NX-OS to send the syslog messages, over UDP, to the IP Address `10.10.10.1`, port 10101:

```
logging server 10.10.10.1 port 10101
```

To see how to configure the network device, check the documents referenced above. Note that the examples in each case represents the configuration used to send the syslog messages over UDP to a certain IP address and port. Remember that napalm-logs is able to receive the messages over UDP (by default), as well as via other channels - see *Listener*. While napalm-logs can be the UDP endpoint configured to receive the messages straight from the network device, there is no standard configuration, setup or architecture for the rest of the Listeners, but rather it depends very much on how you want to design your own use case.

Napalm-logs is able to publish messages from unidentified operating systems (or partially parsed messages), but this behaviour is disabled by default. To allow publishing messages from operating systems that are not supported yet by napalm-logs (but they will not be parsed at all), you can configure the *send_unknown: False* option on the publisher (i.e., `send_unknow: true`). To publish partially parsed messages from supported operating systems, but without a mapping for a certain class of messages, you can use the *send_raw: False* option.

3.5.3 Configuration Options

Here we will list all options and what they do.

Command Line

All of the command line arguments can also be added to a config file.

address

The IP address to use to listen for all incoming syslog messages. When using multiple listeners, it is recommended to specify this option for each listener.

Default: `0.0.0.0`.

CLI usage example:

```
$ napalm-logs -a 172.17.17.1
$ napalm-logs --address 172.17.17.1
```

Configuration file example:

```
address: 172.17.17.1
```

auth-address

The IP address to listen on for incoming authorisation requests.

Default: `0.0.0.0`.

CLI usage example:

```
$ napalm-logs --auth-address 172.17.17.2
```

Configuration file example:

```
auth_address: 172.17.17.2
```

auth-port

The port to listen on for incoming authorisation requests.

Default: `49018`

CLI usgae example:

```
$ napalm-logs --auth-port 2022
```

Configuration file example:

```
auth_port: 2022
```

certificate

The certificate to use for the authorisation process. This will be presented to incoming clients during the TLS handshake.

CLI usage example:

```
$ napalm-logs --certificate /var/cache/server.crt
```

Configuration file example:

```
certificate: /var/cache/server.crt
```

config-file

Specifies the file where further configuration options can be found.

Default: `/etc/napalm/logs`.

CLI usage example:

```
$ napalm-logs -c /srv/napalm-logs
$ napalm-logs --config-file /srv/napalm-logs
```

config-path

The directory path where device configuration files can be found. These are the files that contain the syslog message format for each device.

CLI usage example:

```
$ napalm-logs --config-path /home/admin/napalm-logs/
```

Configuration file example:

```
config_path: /home/admin/napalm-logs/
```

device-worker-processes: 1

New in version 0.3.0.

This option configures the number of worker processes to be started for each platform class. For better performances and higher capacity, it is recommended to increase this number, which defaults to 1, i.e., by default there will be started a single process per platform.

Note: Increasing the number of processes, will imply higher memory consumption.

For fine-tunning, consider increasing this number, and at the same time exclude (or include) the appropriate platforms, using the following options: *device_blacklist* and *device_whitelist*.

disable-security

If set no encryption or message signing will take place. All messages will be in plain text. The client will not be able to verify that a message was generated by the server.

It is not recommended to use this in a production environment.

CLI usage example:

```
$ napalm-logs --disable-security
```

Configuration file example:

```
disable_security: true
```

Note: Starting with release release-0.4.0, it is possible to specify this option for each Publisher individually. See *disable_security: False*.

extension-config-path

A path where you can specify further device configuration files that contain the syslog message format for devices.

CLI usage example:

```
$ napalm-logs --extension-config-path /home/admin/napalm-logs/
```

Configuration file example:

```
extension_config_path: /home/admin/napalm-logs/
```

`hwm`: 1000

New in version 0.3.0.

This option controls the ZeroMQ high water mark (the hard limit on the maximum number of outstanding messages ZeromMQ shall queue in memory). If this limit has been reached the internal sockets enter an exceptional state, and ZeroMQ blocks the reception of further messages. This option can be used to tune the performances of the napalm-logs, in terms of total messages processed. While the default limit should be generally enough, in environments with extremely high density of syslog messages to be processed, it is recommended to increase this value. Keep in mind that a higher queue implies higher memory consumption. For maximum capacity, this option can be set to `0`, i.e., inifinite queue.

CLI usage example:

```
$ napalm-logs --hwm 0
```

Configuration file example:

```
hwm: 0
```

`keyfile`

The private key for the certificate specified by the `certificate` option. This will be used to generate a key to encrypt messages.

CLI usage example:

```
$ napalm-logs --keyfile /var/cache/server.key
```

Configuration file example:

```
keyfile: /var/cache/server.key
```

`listener`: `udp`

The module to use when listening for incoming syslog messages. For more details, see *Listener*.

Starting with the release-0.4.0, you are able to listen to the syslog messages over multiple concomitant channels. This capability is available only from the configuration file. For more configuration options for the listener interface, please check the *Listener* section.

Default: `udp`.

CLI usage example:

```
$ napalm-logs --listener kafka
```

Configuration file example:

```
listener: kafka
```

Multiple listeners configuration example (file):

New in version 0.4.0.

```
listener:
  - kafka: {}
  - udp:
      address: 1.2.3.4
      port: 5514
      buffer_size: 2048
  - tcp:
      address: 1.2.3.4
      port: 5515
```

log-file

The file where to send log messages.

If you want log messages to be outputted to the command line you can specify `--log-file cli`.

Default: `/var/log/napalm/logs`.

CLI usage example:

```
$ napalm-logs --log-file /var/log/napalm-logs
```

Configuration file example:

```
log_file: /var/log/napalm-logs
```

log-format

The format of the log messages.

Default: `%(asctime)s,%(msecs)03.0f [%(name)-17s][%(levelname)-8s] %(message)s`.

Example: `2017-07-03 11:54:25,300,301 [napalm_logs.listener.tcp][INFO ] Stopping listener process`

CLI usage example:

```
$ napalm-logs --log-format '%(asctime)s,%(msecs)03.0f [%(levelname)] %(message)s'
```

Configuration file example:

```
log_format: '%(asctime)s,%(msecs)03.0f [%(levelname)] %(message)s'
```

log-level: WARNING

The level at which to log messages. Possible options are `CRITIAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`.

Default: `WARNING`.

CLI usage example:

```
$ napalm-logs -l debug
$ napalm-logs --log-level info
```

Configuration file example:

```
log_level: info
```

port: 514

The port to use to listen for all incoming syslog messages. This can be assigned using the CLI argument `-p`. When working with multiple listeners, it is recommended to specify the `port` argument for each listener to avoid confusions.

Default: `514`.

CLI usage example:

code-block:: bash

$ napalm-logs -p 1024 $ napalm-logs –port 1024

Configuration file example:

```
port: 1024
```

publisher: zmq

The channel(s) to be used when publishing the structured napalm-logs documents. Starting with release-0.4.0, it is possible to publish the messages over multiple channels. Each publisher has it's separate set of configuration options, for more details see *Publisher*.

Default: `zmq` (ZeroMQ)

CLI usage example:

```
$ napalm-logs --publisher zmq
```

Configuration file example:

```
publisher: zmq
```

Multiple publishers configuration example (file):

New in version 0.4.0.

```
publisher:
  - zmq:
      address: 1.2.3.4
      port: 1234
  - kafka:
      bootstrap_servers:
        - kk1.brokers.example.org
        - 192.168.0.1
        - 192.168.0.2:5678
      topic: napalm-logs-out
  - http:
      address: https://example.com/webhook
```

publish-address: 0.0.0.0

The IP address to use to output the processed message. When publishing the structured napalm-logs documents over multiple transports, it is recommended to specify the `address` field per publisher. For more examples, see *publisher: zmq* and *Publisher*.

Default: `0.0.0.0`.

CLI usage example:

```
$ napalm-logs --publish-address 172.17.17.3
```

Configuration file example:

```
publish_address: 172.17.17.3
```

publish-port: 49017

The port to use to output the processes message. When publishing the structured napalm-logs documents over multiple transports, it is recommended to specify the `port` field per publisher. For more examples, see *publisher: zmq* and *Publisher*.

Default: `49017`.

CLI usage example:

```
$ napalm-logs --publish-port 2048
```

Configuration file example:

```
publish_port: 2048
```

serializer: msgpack

The name of the serializer to be used when publishing the napalm-logs structured documents. When working with multiple publishers it is possible to control their serialization method individually, using the *serializer: msgpack* option.

Default: `msgpack`

CLI Example:

```
$ napalm-logs -s json
$ napalm-logs --serializer yaml
```

Configuration file example:

```
serializer: json
```

transport: zmq

The module to use to output the processed message information. For more details, see *Publisher*.

Warning: This option is no longer supported as of release-0.4.0. Use *publisher: zmq* instead.

Default: `zmq` (ZeroMQ).

CLI usage example:

```
$ napalm-logs -t kafka
$ napalm-logs --transport kafka
```

Configuration file example:

```
transport: kafka
```

Or:

```
transport: kafka
```

Config File Only Options

The options to be used inside of the pluggable modules are not provided via the command line, they need to be provided in the config file.

`device_whitelist`

List of platforms to be supported. By default this is an empty list, thus everything will be accepted. This is useful to control the number of sub-processes started.

Example:

```
device_whitelist:
  - junos
  - iosxr
```

`device_blacklist`

List of platforms to be ignored. By default this list is empty, thus nothing will be ignored. This is also useful to control the number of sub-processes started.

Example:

```
device_blacklist:
  - eos
```

3.5.4 Clients

The messages published by napalm-logs can be used in a variety of applications, as there are no restrictions regarding the channel (see *Publisher*).

The capabilities are already embedded in well known *Frameworks*, or the user can consume the structured messages using custom *Example Scripts*.

Frameworks

Salt

The structured messages published by napalm-logs can be imported into the Salt event bus using the napalm-logs Engine introduced in the 2017.7 release (Nitrogen).

Configuration

The `address` and `port` fields on the napalm-syslog Salt engine side must correspond to the values configured for *publish-address: 0.0.0.0* and *publish-address: 0.0.0.0* on the napalm-logs side. Similarly, `auth_address`, `auth_port`, `certificate`, and `transport` would have the values specified for *auth-address* and *auth-port*, *certificate*, and *transport: zmq*.

Note: Do not conflate the `address` and the `port` arguments on the napalm-logs side with `address` and `port` napalm-syslog Salt Engine fields: they are *not* the same!

For more configuration options and usage examples of the napalm-syslog Salt Egine, please check the documentation.

Configuration example:

When the napalm-logs engine is started usign the command line `$ napalm-logs -a 1.2.3.4 -p 1234 --publish-address 5.6.7.8 --publish-port 5678 --disable-security`, or using the configuration file:

```
address: 1.2.3.4
port: 1234
publish_address: 5.6.7.8
publish_port: 5678
disable_security: true
```

The napalm-syslog engine is configured under the Salt Master or Minion:

```
engines:
  - napalm_syslog:
      transport: zmq
      address: 5.6.7.8
      port: 5678
      disable_security: true
```

StackStorm

Example Scripts

For simplicity, the examples below assume that napalm-logs is started using *–disable-security*, and *ZeroMQ* is used as publisher.

Python

Receive the messages from napalm-logs and print on the command line:

```python
import zmq
import napalm_logs.utils

server_address = '127.0.0.1'  # --publish-address
server_port = 49017           # --publish-port

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect('tcp://{address}:{port}'.format(address=server_address,
                                               port=server_port))
socket.setsockopt(zmq.SUBSCRIBE, '')

while True:
    raw_object = socket.recv()
    print(napalm_logs.utils.unserialize(raw_object))
```

JavaScript (Node.js)

Receive the napalm-logs messages into a Node.js app, which only logs on the console. This assumes zeromq.js and msgpack-lite bindings are installed (`npm install zeromq` and `npm install msgpack-lite`).

```javascript
var zmq = require('zeromq')
var msgpack = require('msgpack-lite');
var sock = zmq.socket('sub');
sock.connect('tcp://127.0.0.1:49017');
sock.subscribe('');
sock.on('message', function(msg){
    var data = msgpack.decode(msg);
    console.log('Received message:');
    console.log(data);
});
```

3.5.5 Structured Messages

Each message has a certain identification tag which is unique and cross-platform.

For example, the following syslog message:

```
<28>Jul  4 13:40:55 vmx2 rpd[2942]: BGP_PREFIX_LIMIT_EXCEEDED: 10.0.0.31 (Internal AS␣
↪65001): Configured maximum prefix-limit(1) exceeded for inet-unicast nlri: 7␣
↪(instance master)
```

`napalm-logs` identifies that it was produced by a Junos device and assigns the error tag `BGP_PREFIX_LIMIT_EXCEEDED` and then will try to map the information into the OpenConfig model `openconfig_bgp`:

```json
{
  "yang_message": {
        "bgp": {
          "neighbors": {
                "neighbor": {
                  "192.168.140.254": {
                        "state": {
                          "peer_as": "65001"
```

```
            },
            "afi_safis": {
             "afi_safi": {
                "inet4": {
                 "state": {
                    "prefixes": {
                     "received": "141"
                    }
                 },
                 "ipv4_unicast": {
                    "prefix_limit": {
                     "state": {
                        "max_prefixes": "140"
                     }
                    }
                 }
                }
             }
            }
         }
        }
     }
    }
 },
 "message_details": {
    "processId": "2902",
    "hostPrefix": null,
    "pri": "149",
    "processName": "rpd",
    "host": "vmx01",
    "tag": "BGP_PREFIX_THRESH_EXCEEDED",
    "time": "14:03:12",
    "date": "Jun 21",
    "message": "192.168.140.254 (External AS 65001): Configured maximum prefix-
↪limit threshold(140) exceeded for inet4-unicast nlri: 141 (instance master)"
 },
 "timestamp": 1498050192,
 "facility": 18,
 "ip": "127.0.0.1",
 "host": "vmx01",
 "yang_model": "openconfig_bgp",
 "error": "BGP_PREFIX_THRESH_EXCEEDED",
 "os": "junos",
 "severity": 5
}
```

Under this section, we present the possible error tags, together with their corresponding YANG model and examples.

BGP_MD5_INCORRECT

This error tag corresponds to syslog messages notifying that the authentication for a BGP neighbor is incorrect.

Maps to the `openconfig-bgp` YANG model.

Implemented for

- junos

Syslog message example

```
<4>Jul 20 21:23:00 vmx01 /kernel: tcp_auth_ok: Packet from 192.168.140.254:61664␣
↪wrong MD5 digest
```

Structured message example


```
{
  "error": "BGP_MD5_INCORRECT",
  "facility": 0,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 20",
      "facility": 0,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "Packet from 192.168.140.254:61664 wrong MD5 digest",
      "pri": "4",
      "processId": null,
      "processName": "kernel",
      "severity": 4,
      "tag": "tcp_auth_ok",
      "time": "21:23:00"
  },
  "os": "junos",
  "severity": 4,
  "timestamp": 1500585780,
  "yang_message": {
      "bgp": {
          "neighbors": {
              "neighbor": {
                  "192.168.140.254": {
                      "state": {
                          "session_state": "CONNECT"
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-bgp"
}
```


BGP_NEIGHBOR_STATE_CHANGED

This error tag corresponds to syslog messages notifying that the configured bgp neighbor has changed state

Maps to the `openconfig-bgp` YANG model.

Implemented for

- junos

Syslog message example

```
<28>Jun 21 14:03:12  vmx01 rpd[2902]: RPD_BGP_NEIGHBOR_STATE_CHANGED: BGP peer 1.1.1.
↪1 (External AS 2222) changed state from OpenConfirm to Established (event␣
↪RecvKeepAlive) (instance master)
```

Structured message example

```
{
  "error": "BGP_NEIGHBOR_STATE_CHANGED",
  "facility": 3,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jun 21",
      "facility": 3,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "BGP peer 1.1.1.1 (External AS 2222) changed state from OpenConfirm␣
↪to Established (event RecvKeepAlive) (instance master)",
      "pri": "28",
      "processId": "2902",
      "processName": "rpd",
      "severity": 4,
      "tag": "RPD_BGP_NEIGHBOR_STATE_CHANGED",
      "time": "14:03:12"
  },
  "os": "junos",
  "severity": 4,
  "timestamp": 1498053792,
  "yang_message": {
      "bgp": {
          "neighbors": {
              "neighbor": {
                  "1.1.1.1": {
                      "state": {
                          "peer_as": "2222",
                          "session-state": "ESTABLISHED",
                          "session-state-change-event": "RecvKeepAlive",
                          "session-state-old": "OPEN_CONFIRM"
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-bgp"
}
```

BGP_PEER_NOT_CONFIGURED

This error tag corresponds to syslog messages notifying that the configured peer sent a BGP notification code 6 subcode 5, which idicates that the peer does not have the session configured.

Maps to the `openconfig-bgp` YANG model.

Implemented for

- junos

Syslog message example

```
<87>Jul  5 05:52:44  vmx01 rpd[1848]: bgp_read_message:2764: NOTIFICATION received␣
↪from 1.2.3.4 (External AS 1234): code 6 (Cease) subcode 5 (Connection Rejected)
```

Structured message example


```
{
  "error": "BGP_PEER_NOT_CONFIGURED",
  "facility": 10,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul  5",
      "facility": 10,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "2764: NOTIFICATION received from 1.2.3.4 (External AS 1234): code 6␣
↪(Cease) subcode 5 (Connection Rejected)",
      "pri": "87",
      "processId": "1848",
      "processName": "rpd",
      "severity": 7,
      "tag": "bgp_read_message",
      "time": "05:52:44"
  },
  "os": "junos",
  "severity": 7,
  "timestamp": 1499233964,
  "yang_message": {
      "bgp": {
          "neighbors": {
              "neighbor": {
                  "1.2.3.4": {
                      "state": {
                          "peer_as": "1234",
                          "session_state": "ACTIVE"
                      }
                  }
              }
          }
      }
```

```
  },
  "yang_model": "openconfig-bgp"
}
```

BGP_PREFIX_LIMIT_EXCEEDED

This error tag corresponds to syslog messages notifying that the prefix limit for a BGP neighbor has been exceeded, without tearing it down.

Maps to the `openconfig-bgp` YANG model.

Implemented for

- eos
- junos

Syslog message example

```
<149>Apr 16 11:04:17 edge01 Rib: %BGP-3-NOTIFICATION: received from neighbor 194.53.
↪172.97 (AS 2611) 6/1 (Cease/maximum number of prefixes reached) 0 bytes
```

Structured message example

```
{
  "error": "BGP_PREFIX_LIMIT_EXCEEDED",
  "facility": 18,
  "host": "edge01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Apr 16",
      "facility": 18,
      "host": "edge01",
      "message": ": received from neighbor 194.53.172.97 (AS 2611) 6/1 (Cease/maximum␣
↪number of prefixes reached) 0 bytes",
      "pri": "149",
      "processName": "Rib",
      "severity": 5,
      "tag": "BGP-3-NOTIFICATION",
      "time": "11:04:17"
  },
  "os": "eos",
  "severity": 5,
  "timestamp": 1492340657,
  "yang_message": {
      "bgp": {
          "neighbors": {
              "neighbor": {
                  "194.53.172.97": {
                      "state": {
                          "peer_as": "2611",
                          "session_state": "IDLE"
```

```
                        }
                    }
                }
            }
        }
    },
    "yang_model": "openconfig-bgp"
}
```


BGP_PREFIX_THRESH_EXCEEDED

This error tag corresponds to syslog messages notifying that the prefix limit threshhold for a BGP neighbor has been exceeded and the neighbor has been torn down.

Maps to the `openconfig-bgp` YANG model.

Implemented for

- iosxr
- junos

Syslog message example

```
<149>2647599: vmx01 RP/0/RSP1/CPU0:Mar 28 15:08:30.941 UTC: bgp[1051]: %ROUTING-BGP-5-
↪MAXPFX : No. of IPv6 Unicast prefixes received from fc00::1001 has reached 94106,␣
↪max 125000
```

Structured message example


```
{
  "error": "BGP_PREFIX_THRESH_EXCEEDED",
  "facility": 18,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Mar 28",
      "facility": 18,
      "host": "vmx01",
      "message": "No. of IPv6 Unicast prefixes received from fc00::1001 has reached␣
↪94106, max 125000",
      "messageId": "2647599",
      "milliseconds": ".941",
      "nodeId": "RP/0/RSP1/CPU0",
      "pri": "149",
      "processId": "1051",
      "processName": "bgp",
      "severity": 5,
      "tag": "ROUTING-BGP-5-MAXPFX",
      "time": "15:08:30",
      "timeZone": "UTC"
  },
```

```
  "os": "iosxr",
  "severity": 5,
  "timestamp": 1490713710,
  "yang_message": {
      "bgp": {
          "neighbors": {
              "neighbor": {
                  "fc00::1001": {
                      "afi_safis": {
                          "afi_safi": {
                              "inet6": {
                                  "ipv6_unicast": {
                                      "prefix_limit": {
                                          "state": {
                                              "max_prefixes": 125000
                                          }
                                      }
                                  },
                                  "state": {
                                      "prefixes": {
                                          "received": 94106
                                      }
                                  }
                              }
                          }
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-bgp"
}
```


BGP_SESSION_NOT_CONFIGURED

This error tag corresponds to syslog messages notifying that this router sent a BGP notification code 6 subcode 5 to another router, which idicates that the peer is trying to esablish a session, but this router does not have the session configured.

Maps to the `openconfig-bgp` YANG model.

Implemented for

- junos

Syslog message example

```
<28>Nov 20 16:58:04 re0-gw2.fin1 rpd[3167]: bgp_listen_accept:4984: NOTIFICATION sent␣
↪to 2001:e8:124::f1:12:1+51528 (proto): code 6 (Cease) subcode 5 (Connection␣
↪Rejected), Reason: Connection attempt from unconfigured neighbor:␣
↪2001:e8:124::f1:12:1+51528
```

Structured message example


```
{
  "error": "BGP_SESSION_NOT_CONFIGURED",
  "facility": 3,
  "host": "gw2.fin1",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Nov 20",
      "facility": 3,
      "host": "gw2.fin1",
      "hostPrefix": "re0-",
      "message": "4984: NOTIFICATION sent to 2001:e8:124::f1:12:1+51528 (proto): code␣
↪6 (Cease) subcode 5 (Connection Rejected), Reason: Connection attempt from␣
↪unconfigured neighbor: 2001:e8:124::f1:12:1+51528",
      "pri": "28",
      "processId": "3167",
      "processName": "rpd",
      "severity": 4,
      "tag": "bgp_listen_accept",
      "time": "16:58:04"
  },
  "os": "junos",
  "severity": 4,
  "timestamp": 1511197084,
  "yang_message": {
      "bgp": {
          "neighbors": {
              "neighbor": {
                  "2001:e8:124::f1:12:1": {
                      "state": {
                          "session_state": "IDLE"
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-bgp"
}
```


BPDU_BLOCK_INTERFACE_DISABLED

This error tag corresponds to syslog messages notifying that the configured interface has been disabled due to a bpdu block

Maps to the `openconfig-interface` YANG model.

Implemented for

- junos

Syslog message example

```
<25>Jun 21 14:03:12  vmx01 eswd[2902]: ESWD_BPDU_BLOCK_ERROR_DISABLED: ge-0/0/17.0:␣
↪bpdu-block disabled port
```

Structured message example


```
{
  "error": "BPDU_BLOCK_INTERFACE_DISABLED",
  "facility": 3,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jun 21",
      "facility": 3,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "ge-0/0/17.0: bpdu-block disabled port",
      "pri": "25",
      "processId": "2902",
      "processName": "eswd",
      "severity": 1,
      "tag": "ESWD_BPDU_BLOCK_ERROR_DISABLED",
      "time": "14:03:12"
  },
  "os": "junos",
  "severity": 1,
  "timestamp": 1498053792,
  "yang_message": {
      "interfaces": {
          "interface": {
              "ge-0/0/17.0": {
                  "state": {
                      "oper_status": "DOWN"
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-interface"
}
```


CONFIGURATION_COMMIT_COMPLETED

This error tag corresponds to syslog messages notifying that the a configuration commit is complete

There is no YANG model available yet to map this class of messages. Please check the *Structured message example* section to see the structure.

Implemented for

- iosxr

- junos

Syslog message example

```
<190>40: xrv RP/0/RP0/CPU0:Oct  4 22:52:47.441 : cfgmgr_trial_confirm[67310]: %MGBL-
↪CONFIG-6-DB_COMMIT : Configuration committed by user 'vagrant'. Use 'show␣
↪configuration commit changes 1000000093' to view the changes.
```

Structured message example


```
{
  "error": "CONFIGURATION_COMMIT_COMPLETED",
  "facility": 23,
  "host": "xrv",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Oct  4",
      "facility": 23,
      "host": "xrv",
      "message": "Configuration committed by user 'vagrant'. Use 'show configuration␣
↪commit changes 1000000093' to view the changes.",
      "messageId": "40",
      "milliseconds": ".441",
      "nodeId": "RP/0/RP0/CPU0",
      "pri": "190",
      "processId": "67310",
      "processName": "cfgmgr_trial_confirm",
      "severity": 6,
      "tag": "MGBL-CONFIG-6-DB_COMMIT",
      "time": "22:52:47",
      "timeZone": null
  },
  "os": "iosxr",
  "severity": 6,
  "timestamp": 1507157567,
  "yang_message": {
      "system": {
          "operations": {
              "commit_complete": true
          }
      }
  },
  "yang_model": "NO_MODEL"
}
```


CONFIGURATION_COMMIT_REQUESTED

This error tag corresponds to syslog messages notifying that the a user has requested a configuration commit

There is no YANG model available yet to map this class of messages. Please check the *Structured message example* section to see the structure.

Implemented for

- junos

Syslog message example

```
<189>Jul 20 21:44:00 vmx01 mgd[7729]: UI_COMMIT: User 'luke' requested 'commit'␣
↪operation (comment: hello)
```

Structured message example


```
{
  "error": "CONFIGURATION_COMMIT_REQUESTED",
  "facility": 23,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 20",
      "facility": 23,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "User 'luke' requested 'commit' operation (comment: hello)",
      "pri": "189",
      "processId": "7729",
      "processName": "mgd",
      "severity": 5,
      "tag": "UI_COMMIT",
      "time": "21:44:00"
  },
  "os": "junos",
  "severity": 5,
  "timestamp": 1500587040,
  "yang_message": {
      "users": {
          "user": {
              "luke": {
                  "action": {
                      "comment": "hello",
                      "requested_commit": true
                  }
              }
          }
      }
  },
  "yang_model": "NO_MODEL"
}
```


CONFIGURATION_ERROR

This error tag corresponds to syslog messages notifying that there is an error in the configuration

There is no YANG model available yet to map this class of messages. Please check the *Structured message example* section to see the structure.

Implemented for

- junos

Syslog message example

```
<187>Jul 20 21:44:00 vmx01 mgd[7729]: UI_CONFIGURATION_ERROR: Process: mgd, path:␣
↪[edit vlans VLANTEST l3-interface], statement: l3-interface vlan.666, Interface␣
↪must already be defined under [edit interfaces]
```

Structured message example


```
{
  "error": "CONFIGURATION_ERROR",
  "facility": 23,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 20",
      "facility": 23,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "Process: mgd, path: [edit vlans VLANTEST l3-interface], statement:␣
↪l3-interface vlan.666, Interface must already be defined under [edit interfaces]",
      "pri": "187",
      "processId": "7729",
      "processName": "mgd",
      "severity": 3,
      "tag": "UI_CONFIGURATION_ERROR",
      "time": "21:44:00"
  },
  "os": "junos",
  "severity": 3,
  "timestamp": 1500587040,
  "yang_message": {
      "system": {
          "configuration": {
              "error": true,
              "message": "Interface must already be defined under [edit interfaces]",
              "path": "[edit vlans VLANTEST l3-interface]",
              "statement": "l3-interface vlan.666"
          }
      }
  },
  "yang_model": "NO_MODEL"
}
```


CONFIGURATION_ROLLBACK

This error tag corresponds to syslog messages notifying that the a user has requested a configuration rollback

There is no YANG model available yet to map this class of messages. Please check the *Structured message example* section to see the structure.

Implemented for

- junos

Syslog message example

```
<189>Jul 20 21:44:00 vmx01 mgd[7729]: UI_LOAD_EVENT: User 'luke' is performing a
↪'rollback'
```

Structured message example


```
{
  "error": "CONFIGURATION_ROLLBACK",
  "facility": 23,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 20",
      "facility": 23,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "User 'luke' is performing a 'rollback'",
      "pri": "189",
      "processId": "7729",
      "processName": "mgd",
      "severity": 5,
      "tag": "UI_LOAD_EVENT",
      "time": "21:44:00"
  },
  "os": "junos",
  "severity": 5,
  "timestamp": 1500587040,
  "yang_message": {
      "users": {
          "user": {
              "luke": {
                  "action": {
                      "configuration_rollback": true
                  }
              }
          }
      }
  },
  "yang_model": "NO_MODEL"
}
```


INTERFACE_DOWN

Maps to the `openconfig-interfaces` YANG model.

Implemented for

- iosxr
- junos

Syslog message example

```
<187>94307: gw2.acy1 LC/0/2/CPU0:Jul  7 20:16:14.834 : ifmgr[214]: %PKT_INFRA-LINK-3-
↪UPDOWN : Interface TenGigE0/2/0/4, changed state to Down
```

Structured message example

```
{
  "error": "INTERFACE_DOWN",
  "facility": 23,
  "host": "gw2.acy1",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul  7",
      "facility": 23,
      "host": "gw2.acy1",
      "message": "Interface TenGigE0/2/0/4, changed state to Down",
      "messageId": "94307",
      "milliseconds": ".834",
      "nodeId": "LC/0/2/CPU0",
      "pri": "187",
      "processId": "214",
      "processName": "ifmgr",
      "severity": 3,
      "tag": "PKT_INFRA-LINK-3-UPDOWN",
      "time": "20:16:14",
      "timeZone": null
  },
  "os": "iosxr",
  "severity": 3,
  "timestamp": 1499458574,
  "yang_message": {
      "interfaces": {
          "interface": {
              "TenGigE0/2/0/4": {
                  "state": {
                      "oper_status": "DOWN"
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-interfaces"
}
```

INTERFACE_MAC_LIMIT_REACHED

This error tag corresponds to syslog messages notifying that the configured interface mac learning limit has been reached

Maps to the `openconfig-interface` YANG model.

Implemented for

- junos

Syslog message example

```
<149>Jun 21 14:03:12  vmx01 l2ald[2902]: L2ALD_MAC_LIMIT_REACHED_IF: Limit on learned␣
↪MAC addresses reached for ge-1/0/23.0; current count is 3
```

Structured message example


```
{
  "error": "INTERFACE_MAC_LIMIT_REACHED",
  "facility": 18,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jun 21",
      "facility": 18,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "Limit on learned MAC addresses reached for ge-1/0/23.0; current␣
↪count is 3",
      "pri": "149",
      "processId": "2902",
      "processName": "l2ald",
      "severity": 5,
      "tag": "L2ALD_MAC_LIMIT_REACHED_IF",
      "time": "14:03:12"
  },
  "os": "junos",
  "severity": 5,
  "timestamp": 1498053792,
  "yang_message": {
      "interfaces": {
          "interface": {
              "ge-1/0/23.0": {
                  "ethernet": {
                      "state": {
                          "learned-mac-addresses": "3"
                      }
                  }
              }
          }
      }
  },
```

```
  "yang_model": "openconfig-interface"
}
```

INTERFACE_UP

Maps to the `openconfig-interfaces` YANG model.

Implemented for

- iosxr

Syslog message example

```
<187>94307: gw1.dev1 LC/0/2/CPU0:Jul  7 20:16:14.834 : ifmgr[214]: %PKT_INFRA-LINK-3-
↪UPDOWN : Interface TenGigE0/2/0/4, changed state to Up
```

Structured message example


```
{
  "error": "INTERFACE_UP",
  "facility": 23,
  "host": "gw1.dev1",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul  7",
      "facility": 23,
      "host": "gw1.dev1",
      "message": "Interface TenGigE0/2/0/4, changed state to Up",
      "messageId": "94307",
      "milliseconds": ".834",
      "nodeId": "LC/0/2/CPU0",
      "pri": "187",
      "processId": "214",
      "processName": "ifmgr",
      "severity": 3,
      "tag": "PKT_INFRA-LINK-3-UPDOWN",
      "time": "20:16:14",
      "timeZone": null
  },
  "os": "iosxr",
  "severity": 3,
  "timestamp": 1499458574,
  "yang_message": {
      "interfaces": {
          "interface": {
              "TenGigE0/2/0/4": {
                  "state": {
                      "oper_status": "UP"
                  }
              }
          }
```

```
        }
    },
    "yang_model": "openconfig-interfaces"
}
```

ISIS_NEIGHBOR_DOWN

Maps to the `openconfig-isis` YANG model.

Implemented for

- iosxr

Syslog message example

```
<190>12345: gw1.acy1 RP/0/RSP0/CPU0:Nov  1 11:11:24.927: isis[1006]: %ROUTING-ISIS-5-
↪ADJCHANGE : Adjacency to gw1.nyc1 (TenGigE1/2/0/8.92) (L2) Down, Interface state␣
↪down
```

Structured message example

```
{
    "error": "ISIS_NEIGHBOR_DOWN",
    "facility": 23,
    "host": "gw1.acy1",
    "ip": "127.0.0.1",
    "message_details": {
        "date": "Nov  1",
        "facility": 23,
        "host": "gw1.acy1",
        "message": "Adjacency to gw1.nyc1 (TenGigE1/2/0/8.92) (L2) Down, Interface␣
↪state down",
        "messageId": "12345",
        "milliseconds": ".927",
        "nodeId": "RP/0/RSP0/CPU0",
        "pri": "190",
        "processId": "1006",
        "processName": "isis",
        "severity": 6,
        "tag": "ROUTING-ISIS-5-ADJCHANGE",
        "time": "11:11:24",
        "timeZone": null
    },
    "os": "iosxr",
    "severity": 6,
    "timestamp": 1509534684,
    "yang_message": {
        "network-instances": {
            "network-instance": {
                "global": {
                    "protocols": {
```

```
                        "protocol": {
                            "isis": {
                                "interfaces": {
                                    "interface": {
                                        "TenGigE1/2/0/8.92": {
                                            "levels": {
                                                "level": {
                                                    "L2": {
                                                        "adjacencies": {
                                                            "adjacency": {
                                                                "gw1.nyc1": {
                                                                    "state": {
                                                                        "adjacency-state
↪": "DOWN",
                                                                        "adjacency-
↪state-change-reason-message": "Interface state down"
                                                                    }
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    },
  "yang_model": "openconfig-isis"
}
```


ISIS_NEIGHBOR_UP

Maps to the `openconfig-isis` YANG model.

Implemented for

- iosxr

Syslog message example

```
<190>12345: gw3.frc1 RP/0/RSP0/CPU0:Nov  1 01:17:24.927: isis[1006]: %ROUTING-ISIS-5-
↪ADJCHANGE : Adjacency to gw3.lax1 (TenGigE0/2/0/8.995) (L2) Up, New adjacency
```

Structured message example

```
{
  "error": "ISIS_NEIGHBOR_UP",
  "facility": 23,
  "host": "gw3.frc1",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Nov  1",
      "facility": 23,
      "host": "gw3.frc1",
      "message": "Adjacency to gw3.lax1 (TenGigE0/2/0/8.995) (L2) Up, New adjacency",
      "messageId": "12345",
      "milliseconds": ".927",
      "nodeId": "RP/0/RSP0/CPU0",
      "pri": "190",
      "processId": "1006",
      "processName": "isis",
      "severity": 6,
      "tag": "ROUTING-ISIS-5-ADJCHANGE",
      "time": "01:17:24",
      "timeZone": null
  },
  "os": "iosxr",
  "severity": 6,
  "timestamp": 1509499044,
  "yang_message": {
      "network-instances": {
          "network-instance": {
              "global": {
                  "protocols": {
                      "protocol": {
                          "isis": {
                              "interfaces": {
                                  "interface": {
                                      "TenGigE0/2/0/8.995": {
                                          "levels": {
                                              "level": {
                                                  "L2": {
                                                      "adjacencies": {
                                                          "adjacency": {
                                                              "gw3.lax1": {
                                                                  "state": {
                                                                      "adjacency-state
↪": "UP",
                                                                      "adjacency-
↪state-change-reason-message": "New adjacency"
                                                                  }
                                                              }
                                                          }
                                                      }
                                                  }
                                              }
                                          }
                                      }
                                  }
                              }
                          }
```

```
                        }
                    }
                }
            }
        }
    },
    "yang_model": "openconfig-isis"
}
```


NTP_SERVER_UNREACHABLE

This message is sent when the synchronization is lost with an NTP server. According to the `openconfig-system` YANG model, the distinction between NTP peers and servers is made via the `association-type` field from the `config` container.

Maps to the `openconfig-system` YANG model.

Implemented for

- iosxr
- junos

Syslog message example

```
<99>2647599: device3 RP/0/RSP0/CPU0:Aug 21 09:39:14.747 UTC: ntpd[262]: %IP-IP_NTP-5-
↪SYNC_LOSS : Synchronization lost : 172.17.17.1 : The association was removed
```

Structured message example


```
{
    "error": "NTP_SERVER_UNREACHABLE",
    "facility": 12,
    "host": "device3",
    "ip": "127.0.0.1",
    "message_details": {
        "date": "Aug 21",
        "facility": 12,
        "host": "device3",
        "message": "Synchronization lost : 172.17.17.1 : The association was removed",
        "messageId": "2647599",
        "milliseconds": ".747",
        "nodeId": "RP/0/RSP0/CPU0",
        "pri": "99",
        "processId": "262",
        "processName": "ntpd",
        "severity": 3,
        "tag": "IP-IP_NTP-5-SYNC_LOSS",
        "time": "09:39:14",
        "timeZone": "UTC"
    },
    "os": "iosxr",
```

```
  "severity": 3,
  "timestamp": 1503308354,
  "yang_message": {
      "system": {
          "ntp": {
              "servers": {
                  "server": {
                      "172.17.17.1": {
                          "state": {
                              "stratum": 16
                          }
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-system"
}
```


OSPF_NEIGHBOR_DOWN

This error tag corresponds to syslog messages notifying that the configured ospf neighbor has changed state from Full

Maps to the `openconfig-ospf` YANG model.

Implemented for

- junos

Syslog message example

```
<29>Jun 21 14:03:12  vmx01 rpd[2902]: RPD_OSPF_NBRDOWN: OSPF neighbor 1.1.1.1 (realm␣
↪ospf-v2 ge-0/0/0.0 area 0.0.0.0) state changed from Full to Down due to␣
↪InActiveTimer (event reason: BFD session timed out and neighbor was declared dead)
```

Structured message example


```
{
  "error": "OSPF_NEIGHBOR_DOWN",
  "facility": 3,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jun 21",
      "facility": 3,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "OSPF neighbor 1.1.1.1 (realm ospf-v2 ge-0/0/0.0 area 0.0.0.0) state␣
↪changed from Full to Down due to InActiveTimer (event reason: BFD session timed out␣
↪and neighbor was declared dead)",
      "pri": "29",
```

```
      "processId": "2902",
      "processName": "rpd",
      "severity": 5,
      "tag": "RPD_OSPF_NBRDOWN",
      "time": "14:03:12"
  },
  "os": "junos",
  "severity": 5,
  "timestamp": 1498053792,
  "yang_message": {
      "network-instances": {
          "network-instance": {
              "global": {
                  "protocols": {
                      "protocol": {
                          "ospf": {
                              "ospfv2": {
                                  "areas": {
                                      "area": {
                                          "0.0.0.0": {
                                              "interfaces": {
                                                  "interface": {
                                                      "ge-0/0/0.0": {
                                                          "neighbors": {
                                                              "neighbor": {
                                                                  "1.1.1.1": {
                                                                      "state": {
                                                                          "adjacency-
↪state": "DOWN",
                                                                          "adjacency-
↪state-change-reason": "INACTIVE_TIMER",
                                                                          "adjacency-
↪state-change-reason-message": "BFD session timed out and neighbor was declared dead"
                                                                      }
                                                                  }
                                                              }
                                                          }
                                                      }
                                                  }
                                              }
                                          }
                                      }
                                  }
                              }
                          }
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-ospf"
}
```


OSPF_NEIGHBOR_UP

This error tag corresponds to syslog messages notifying that the configured ospf neighbor has changed to a higher state

Maps to the `openconfig-ospf` YANG model.

Implemented for

- junos

Syslog message example

```
<29>Jun 21 14:03:12  vmx01 rpd[2902]: RPD_OSPF_NBRUP: OSPF neighbor 1.1.1.1 (realm␣
↪ospf-v2 ge-0/0/0.0 area 0.0.0.0) state changed from Exchange to Full due to␣
↪ExchangeDone (event reason: exchange done)
```

Structured message example


```
{
  "error": "OSPF_NEIGHBOR_UP",
  "facility": 3,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jun 21",
      "facility": 3,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "OSPF neighbor 1.1.1.1 (realm ospf-v2 ge-0/0/0.0 area 0.0.0.0) state␣
↪changed from Exchange to Full due to ExchangeDone (event reason: exchange done)",
      "pri": "29",
      "processId": "2902",
      "processName": "rpd",
      "severity": 5,
      "tag": "RPD_OSPF_NBRUP",
      "time": "14:03:12"
  },
  "os": "junos",
  "severity": 5,
  "timestamp": 1498053792,
  "yang_message": {
      "network-instances": {
          "network-instance": {
              "global": {
                  "protocols": {
                      "protocol": {
                          "ospf": {
                              "ospfv2": {
                                  "areas": {
                                      "area": {
                                          "0.0.0.0": {
                                              "interfaces": {
                                                  "interface": {
                                                      "ge-0/0/0.0": {
                                                          "neighbors": {
                                                              "neighbor": {
                                                                  "1.1.1.1": {
```

```
                                                                                                    "state": {
                                                                                                        "adjacency-
↪state": "FULL",
                                                                                                        "adjacency-
↪state-change-reason": "EXCHANGE_DONE",
                                                                                                        "adjacency-
↪state-change-reason-message": "exchange done"
                                                                                                    }
                                                                                                }
                                                                                            }
                                                                                        }
                                                                                    }
                                                                                }
                                                                            }
                                                                        }
                                                                    }
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
      },
  "yang_model": "openconfig-ospf"
}
```

RAW

This error tag is sent when napalm-logs was able to identify the operating system, but there was no tag matching the syslog message. Therefore, the output object will contain the syslog message parts, without further processing. By default, these messages are not published; they need to be explicitly enabled using the *send_raw: False* option for the publisher.

Note: These messages are not recommended for production use. They can be used as temporary helpers, at most. The right approach is appending a new message matcher inside the corresponding device profile. See *Device Profiles*.

Note: The syslog message parts under the `message_details` key are device-specific, as designed inside the profiler.

Example:

```
{
  "message_details": {
    "processId": null,
    "hostPrefix": null,
    "pri": "37",
    "processName": "sshd",
    "host": "vmx1",
    "tag": "SSHD_LOGIN_FAILED",
    "time": "10:32:03",
    "date": "Jul 10",
```

```
    "message": "Login failed for user 'root' from host '61.177.172.56'"
  },
  "ip": "172.17.17.1",
  "host": "vmx1",
  "timestamp": 1499682723,
  "os": "junos",
  "model_name": "raw",
  "error": "RAW",
  "facility": 4,
  "severity": 5
}
```


SYSTEM_ALARM

This error tag corresponds to syslog messages notifying that there has been a change in status for an alarm. There are multiple entries for this error. The reason being that the exact component name can be contained in the reason section, so has to be extracted via a specific regex.

Maps to the `ietf-hardware` YANG model.

Implemented for

- junos

Syslog message example

```
<28>Jul  8 23:04:13  vmx01 alarmd[2449]: Alarm set: Pwr supply color=YELLOW,␣
↪class=CHASSIS, reason=PEM 1 Fan Failed
```

Structured message example


```
{
  "error": "SYSTEM_ALARM",
  "facility": 3,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul  8",
      "facility": 3,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "Pwr supply color=YELLOW, class=CHASSIS, reason=PEM 1 Fan Failed",
      "pri": "28",
      "processId": "2449",
      "processName": "alarmd",
      "severity": 4,
      "tag": "Alarm set",
      "time": "23:04:13"
  },
  "os": "junos",
  "severity": 4,
```

```
  "timestamp": 1499555053,
  "yang_message": {
      "hardware-state": {
          "component": {
              "supply": {
                  "class": "CHASSIS",
                  "name": "supply",
                  "state": {
                      "alarm-reason": "PEM 1 Fan Failed",
                      "alarm-state": 4
                  }
              }
          }
      }
  },
  "yang_model": "ietf-hardware"
}
```

UNKNOWN

This error tag is sent when napalm-logs was unable to identify the operating system. By default, these messages are not published; they need to be explicitly enabled using the *send_unknown: False* option for the publisher.

Note: These messages are not recommended for production use. They can be used as temporary helpers, at most. The right approach is writing a new device profile matching the syslog message and generating the structured messages as required. See *Device Profiles*.

Example:

```
{
  "message_details": {
        "message": "<28>Jul 10 10:32:00 vmx1 inetd[2397]: /usr/sbin/sshd[89736]:␣
↪exited, status 255\n"
  },
  "timestamp": 1501685287,
  "ip": "127.0.0.1",
  "host": "unknown",
  "error": "UNKNOWN",
  "os": "unknown",
  "model_name": "unknown"
}
```

USER_ENTER_CONFIG_MODE

There is no YANG model available yet to map this class of messages. Please check the *Structured message example* section to see the structure.

Implemented for

- junos

Syslog message example

```
<189>Jul 20 21:44:00 vmx01 mgd[7729]: UI_DBASE_LOGIN_EVENT: User 'luke' entering␣
↪configuration mode
```

Structured message example


```
{
  "error": "USER_ENTER_CONFIG_MODE",
  "facility": 23,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 20",
      "facility": 23,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "User 'luke' entering configuration mode",
      "pri": "189",
      "processId": "7729",
      "processName": "mgd",
      "severity": 5,
      "tag": "UI_DBASE_LOGIN_EVENT",
      "time": "21:44:00"
  },
  "os": "junos",
  "severity": 5,
  "timestamp": 1500587040,
  "yang_message": {
      "users": {
          "user": {
              "luke": {
                  "action": {
                      "enter_config_mode": true
                  }
              }
          }
      }
  },
  "yang_model": "NO_MODEL"
}
```


USER_LOGIN

Match messages AUTHPRIV-6-SYSTEM_MSG from NX-OS.

Message example:

```
sw01.bjm01: 2017 Jul 26 14:42:46 UTC: %AUTHPRIV-6-SYSTEM_MSG: pam_unix(dcos_
↪sshd:session): session opened for user luke by (uid=0) - dcos_sshd[12977]  # noqa
```

Output example:

```
{
  "users": {
    "user": {
      "luke": {
        "action": {
          "login": true
        },
        "uid": 0
      }
    }
  }
}
```

There is no YANG model available yet to map this class of messages. Please check the *Structured message example* section to see the structure.

Implemented for

- nxos

Syslog message example

```
<190>sw01.pdx01: 2017 Jul 28 14:42:46 UTC: %AUTHPRIV-6-SYSTEM_MSG: pam_unix(dcos_
↪sshd:session): session opened for user luke by (uid=0) - dcos_sshd[12977]
```

Structured message example

```
{
  "error": "USER_LOGIN",
  "facility": 23,
  "host": "sw01.pdx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "2017 Jul 28",
      "facility": 23,
      "host": "sw01.pdx01",
      "message": "pam_unix(dcos_sshd:session): session opened for user luke by␣
↪(uid=0) - dcos_sshd[12977]",
      "pri": "190",
      "severity": 6,
      "tag": "AUTHPRIV-6-SYSTEM_MSG",
      "time": "14:42:46",
      "timeZone": "UTC"
  },
  "os": "nxos",
  "severity": 6,
  "timestamp": 1501252966,
  "yang_message": {
      "users": {
          "user": {
              "luke": {
                  "action": {
```

```
                        "login": true
                    },
                    "uid": 0
                }
            }
        }
    },
    "yang_model": "NO_MODEL"
}
```

3.5.6 Client Authentication

With the event-driven automation in mind, napalm-logs has been designed to be safe and securely publish the outgoing messages. As these messages may trigger automatic configurationc changes, or simply notifications, we must ensure their authenticity. For these reasons, napalm-logs encrypts and signs the outgoing messages.

Although highly discouraged, the user has the possibility to disable the security at their own risk.

Whether the security is enabled or disabled, the messages published are binary serialized using MessagePack.

The clients that connect to the publisher interface (see *Publisher*), have to retrieve the encryption and the signing key from the napalm-logs daemon. In the core architecture of napalm-logs, when the security is not turned off, another separate process is started, which listens to connections and exchanges the keys with the client. The exchange is realised over a secure SSL socket, using the certificate and the key configured when starting the daemon (see *certificate* and *keyfile*). The authentication subsystem listens on a socket, whose configuration details can be set using the *auth-address* and *auth-port* options (either from the CLI, or in the configuration file).

The client, before being able to decrypt the messages received from the napalm-logs publisher, must receive the keys from the authenticator sub-system.

In order to ease the authentication process on the client side, we have included a couple of helpers, making the key exchange and decryption easy:

```
#!/usr/bin/env python

import zmq  # when using the ZeroMQ publisher
import napalm_logs.utils

server_address = '127.0.0.1'  # IP
server_port = 49017           # Port for the napalm-logs publisher interface
auth_address = '127.0.0.1'    # IP
auth_port = 49018             # Port for the authentication interface

certificate = '/var/cache/napalm-logs.crt' # This is the server crt generated earlier

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect('tcp://{address}:{port}'.format(address=server_address,
                                               port=server_port))
socket.setsockopt(zmq.SUBSCRIBE, '')  # subscribe to the napalm-logs publisher

auth = napalm_logs.utils.ClientAuth(certificate,
                                    address=auth_address,
                                    port=auth_port)  # authenticate to napalm-logs

while True:
    raw_object = socket.recv()  # receive the encrypted object
```

```
    decrypted = auth.decrypt(raw_object)  # check the siganture, decrypt and␣
↪deserialize
    print(decrypted)
```

When the security is disabled, the clients no longer need to authenticate and receive the keys, however they need to bear in mind to deserialize the messages. We have also included a helper for that: `napalm_logs.utils.unserialize`, see the example below:

```
#!/usr/bin/env python

import zmq  # when using the ZeroMQ publisher
import napalm_logs.utils

server_address = '127.0.0.1'  # IP
server_port = 49017           # Port for the napalm-logs publisher interface

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect('tcp://{address}:{port}'.format(address=server_address,
                                               port=server_port))
socket.setsockopt(zmq.SUBSCRIBE, '')  # subscribe to the napalm-logs publisher

while True:
    raw_object = socket.recv()  # binary object
    print(napalm_logs.utils.unserialize(raw_object))  # deserialize
```

3.5.7 Listener

The Listener subsystem is a pluggable interface for inbound unstructured syslog messages. The messages can be received directly from the network devices, via UDP or TCP, or from other third parties, such as brokers, e.g. ZeroMQ, Kafka, etc., depending on the architecture of the network. The default listener is UDP.

From the command line, the Listener can be selected using the `--listener` option, e.g.:

```
$ napalm-logs --listener tcp
```

From the configuration file, the Listener can be specified using the `listener` option, eventually with several options. The options depend on the nature of the Listener.

Example: listener configuration using the default configuration

```
listener: tcp
```

Example: listener configuration using custom options

```
listener:
  tcp:
    buffer_size: 2048
    max_clients: 100
```

Note: The IP Address / port for the Listener be specified using the *address* and *port: 514* configuration options.

Multiple listeners

New in version 0.4.0.

It is possible to start multiple listeners, each with its separate set of configuration options, however this feature is available only from the configuration file, e.g.:

```
listener:
  - tcp:
      address: 1.2.3.4
      port: 1234
      buffer_size: 2048
      max_clients: 100
  - udp:
      address: 5.6.7.8
      port: 5678
```

Available listeners and their options

UDP

Receive the unstructured syslog messages over UDP.

Available options:

buffer_size: 1024

The socket buffer size, in bytes.

Example:

```
listener:
  udp:
    buffer_size: 2048
```

TCP

Receive the unstructured syslog messages over TCP.

Available options:

buffer_size: 1024

The socket buffer size, in bytes.

Example:

```
listener:
  tcp:
    buffer_size: 2048
```

`socket_timeout`: 60

The socket timeout, in seconds.

Example:

```
listener:
  tcp:
    socket_timeout: 5
```

`max_clients`: 5

The maximum number of parallel connections to accept.

Example:

```
listener:
  tcp:
    max_clients: 100
```

Kafka

Receive unstructured syslog messages from Apache Kafka.

Available options:

`bootstrap_servers`

`host[:port]` string (or list of `host[:port]` strings) that the consumer should contact to bootstrap initial cluster metadata. This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request.

Example:

```
listener:
  kafka:
    bootstrap_servers:
      - kk1.brokers.example.org
      - kk1.brokers.example.org:1234
      - 192.168.0.1
      - 192.168.0.2:5678
```

```
listener:
  kafka:
    bootstrap_servers: kk1.brokers.example.org:1234
```

`group_id`: `napalm-logs`

The bootstrap servers group ID name.

Example:

```
listener:
  kafka:
    group_id: napalm-logs-servers
```

topic: syslog.net

The topic to subscribe to and receive messages from.

Example:

```
listener:
  kafka:
    topic: napalm-logs-in
```

ZeroMQ

New in version 0.3.0.

Receive unstructured syslog messages via ZeroMQ.

While this listener can be used without any extensive knowledge, we recommend reading the ZeroMQ guide for advanced tunning, especially when the messages are transported over networks with misbehaving firewalls.

Available options:

hwm

Set the high water mark for inbound messages. This option will configure the ZeroMQ option `ZMQ_RCVHWM`. This option controls the message queue size. Read this document for more details.

Example:

```
listener:
  zmq:
    hwm: 0
```

keepalive: 1

Override `SO_KEEPALIVE` socket option. By default, the client will try to maintain the connection alive.

Example:

```
listener:
  zmq:
    keepalive: 1
```

keepalive_idle: 300

Override `TCP_KEEPALIVE` socket option (where supported by OS). The value is specified in miliseconds.

Example:

```
listener:
  zmq:
    keepalive_idle: 500
```

`keepalive_interval`: -1

Override `TCP_KEEPINTVL` socket option(where supported by OS). The value is specified in miliseconds.

Example:

```
listener:
  zmq:
    keepalive_interval: 300
```

`timeout`

Maximum wait time (in miliseconds) to receive a message. By default does not time out, and the listener will block waiting for a new message to arrive.

Example:

```
listener:
  zmq:
    timeout: 5000
```

`protocol`: `tcp`

The protocol to be used for the ZeroMQ listener. Can choose between: `tcp`, `ipc`, and `pgm`.

Example:

```
listener:
  zmq:
    protocol: ipc
```

`socket_type`: `PULL`

The nature of the socket to recevie the messages. Although the user can choose from a variety of types, `PULL` and `SUB` fit the best into napalm-logs.

Example:

```
listener:
  zmq:
    socket_type: SUB
```

3.5.8 Publisher

The Publisher subsystem is a pluggable interface for outbound messages, structured following the OpenConfig / IETF YANG models. The messages can be published over a variety of services – see *Available publishers and their options*. From the command line, the Publisher module can be selected using the `--publisher` option, e.g.:

```
$ napalm-logs --publisher kafka
```

From the configuration file, the Publisher can be specified using the `publisher` option, eventually with several options. The options depend on the nature of the Publisher.

Example: publisher configuration using the default configuration

```
publisher: zmq
```

Example: publisher configuration using custom options

```
publisher:
  kafka:
    topic: napalm-logs-out
```

Note: The IP Address / port for the Publisher be specified using the *publish-address: 0.0.0.0* and *publish-port: 49017* configuration options.

Multiple publishers

New in version 0.4.0.

It is possible to export the structured napalm-logs structured documents into multiple systems, over multiple channels, each with its separate configuration options. This feature is available only from the configuration file, e.g.:

```
publisher:
  - zmq:
      address: 1.2.3.4
      port: 5678
  - kafka:
      topic: napalm-logs-out
  - http:
      address: https://example.com/webhook
```

Available publishers and their options

CLI

This publisher is for debugging use only and does not have additional configuration options. It can be used from the CLI and the structured messages are printed in clear on the command line, e.g.:

```
$ sudo napalm-logs --publisher cli
```

HTTP

New in version 0.3.0.

Publish objects by invoking a HTTP endpoint.

This Publisher module can use several backends (currently just two: Tornado and Requests). If no explicity backend is specified, using the *backend* option, Tornado has the higher precedence due to its speed, as it allows asynchronous requests.

Note: The ref:*configuration-options-address* must have contain the `http://` or `https://` schema. The address can however be specified more explicitly under the publisher configuration options, using the ref:*publisher-http-address* field.

Configuration examples:

- From the command line

```
$ napalm-logs --publisher http --address https://example.com/hook
```

- Basic YAML configuration

```
publisher: http
```

- YAML configuration with more options

```
publisher:
  http:
    address: 'https://example.com/hook'
    method: POST
    headers:
      Authorization: OAuth 89a229ce1a8dbcf9f
    backend: tornado
```

Available options

`address`

Specifies the endpoint to invoke when a new event is published. The value must contain the `http://` or `https://` schema.

Example:

```
publisher:
  http:
    address: 'https://example.com/hook'
```

`backend`

The name of the toolset to use as backend to execute the HTTP requests. Can choose between:

- `tornado`
- `requests`

When this option is not specifically configured, the publisher will try to use the library found to be installed on the machine, Tornado having the highest precedence.

Example:

```
publisher:
  http:
    backend: requests
```

headers

A dictionary (hash / mapping) of the headers.

Example:

```
publisher:
  http:
    headers:
      Content-Type: text/json
      Pragma: no-cache
      Cache-Control: no-cache
```

max_clients: 10

The maximum number of parallel clients.

Example:

```
publisher:
  http:
    max_clients: 20
```

method: POST

HTTP method to use. Choose from: `GET`, `POST`, `PUT`, `HEAD` (the others probably don't make sense, however they are allowed). For more details see this document.

Example:

```
publisher:
  http:
    method: GET
```

params

A set of parameters (key-value) to be sent together with the request.

Example:

```
publisher:
  http:
    params: key1=val1&key2=val2
```

password

The password if needed to authenticate the HTTP request.

Example:

```
publisher:
  http:
    password: example
```

`username`

The username if needed to authenticate the HTTP request.

Example:

```
publisher:
  http:
    username: example
```

`verify_ssl`: `true`

By default, SSL certificates will be verified. However, for testing or debugging purposes, SSL verification can be turned off. It is higly discouraged to disable thio option in production environments.

Example:

```
publisher:
  http:
    verify_ssl: false
```

Kafka

Submit structured messages to Apache Kafka.

```
$ sudo napalm-logs --publisher kafka
```

Available options:

`bootstrap_servers`

`host[:port]` string (or list of `host[:port]` strings) that the consumer should contact to bootstrap initial cluster metadata. This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request.

Example:

```
publisher:
  kafka:
    bootstrap_servers:
      - kk1.brokers.example.org
      - kk1.brokers.example.org:1234
      - 192.168.0.1
      - 192.168.0.2:5678
```

`topic`: `napalm-logs`

The Kafka topic to use when publishing messages.

Example:

```
publisher:
  kafka:
    topic: napalm-logs-out
```

Log

Forward objects to an external logging server.

```
$ sudo napalm-logs --publisher log
```

ZeroMQ

Publish objects over ZeroMQ where multiple clients can subscribe.

```
$ sudo napalm-logs --publisher zmq
```

Globally available options

Additionally, the user can configure the following options, available to all publishers:

`disable_security: False`

The message encryption can be disabled per publisher as well. Similar to the main *disable-security* configuration option, is it recommended **not to disable security**, though this can be needed in certain particular cases.

Configuration example:

```
publisher:
  - cli:
      disable_security: true
  - zmq: {}
```

`error_whitelist: []`

New in version 0.4.0.

Publish only the error messages included in this list. The whitelist/blacklist logic is implemented in such a way that if anything is added in this list, *only* these message types will be published and nothing else.

Default: `None` (empty list)

Configuration example:

```
publisher:
  - kafka:
      error_whitelist:
        - UNKNOWN
        - RAW
  - zmq:
      error_whitelist:
        - BGP_MD5_INCORRECT
        - BGP_NEIGHBOR_STATE_CHANGED
```

error_blacklist: ['RAW', 'UNKNOWN']

New in version 0.4.0.

Filter out the error types publisher. The error messages included in this list will not be published.

Default: `RAW, UNKNOWN` (both `RAW` and `UNKNOWN` message types will not be published by default).

Configuration example:

```
publisher:
  - kafka:
      error_blacklist:
        - UNKNOWN
        - RAW
        - USER_ENTER_CONFIG_MODE
  - zmq:
      error_blacklist:
        - UNKNOWN
```

only_raw: False

New in version 0.4.0.

When this option is enabled, the publisher will publish *only* the syslog messages that could not be parsed.

Example:

```
publisher:
  - zmq:
      address: 1.2.3.4
      port: 1234
  - zmq:
      address: 5.6.7.8
      port: 5678
      only_raw: true
```

Note: This option is a shortcut to the *error_whitelist: []* configuration option introduced in 0.4.0 (codename Crowbar), by adding the `RAW` message to the whitelist message types, i.e.,

```
publisher:
  - zmq:
      address: 1.2.3.4
      port: 1234
  - zmq:
      address: 5.6.7.8
      port: 5678
      error_whitelist:
        - RAW
```

`only_unknown`: `False`

New in version 0.4.0.

When this option is configured, napalm-logs will publish *only* the structured documents that are marked as `UNKNWON` (i.e., napalm-logs was unable to parse the message and determine the operating system).

Example:

```
publisher:
  kafka:
    only_unknown: true
```

Note: This option is a shortcut to the *error_whitelist: []* option introduced in 0.4.0 (codename Crowbar), by adding the `UNKNOW` message type to the whitelist, i.e.,

```
publisher:
  kafka:
    error_whitelist:
      - UNKNOWN
```

`send_raw`: `False`

If this option is set, all processed syslog messages, even ones that have not matched a configured error, will be published over the specified transport. This can be used to forward to log server for storage.

Example:

```
publisher:
  zmq:
    send_raw: true
```

Note: This option is just a shortcut to the *error_blacklist: ['RAW', 'UNKNOWN']* configuration option introduced in 0.4.0 (codename Crowbar), by removing the `RAW` error type from the blacklisted message types, i.e.,

```
publisher:
  zmq:
    error_blacklist:
      - UNKNOWN
```

`send_unknown`: `False`

If this option is set, all processed syslog messages, even ones that have not matched a certain operating system, will be published over the specified transport. This can be used to forward to log server for storage.

Example:

```
publisher:
  kafka:
    send_unknown: true
```

Note: This option is just a shortcut to the *error_blacklist: ['RAW', 'UNKNOWN']* option introduced in 0.4.0 (codename Crowbar), by removing the UNKNOWN message from the blacklist, i.e.,

```
publisher:
  kafka:
    error_blacklist:
      - RAW
```

serializer: msgpack

New in version 0.4.0.

The serializer to be used when publishing the structure napalm-logs document.

Default: *MessagePack*.

You can specify a separate serialize per publisher, e.g.:

```
publisher:
  - kafka:
      serializer: json
  - cli:
      serializer: pprint
```

3.5.9 Serializer

New in version 0.4.0.

The Serializer subsystem is a pluggable interface used just before a structured napalm-logs document is sent to the *Publisher* interface.

The default Serializer used is *MessagePack*.

From the command line, the Serializer can be selected using the `--serializer` (or `-s`) option, e.g.:

```
$ napalm-logs -s yaml
$ napalm-logs --serializer pprint
```

From the configuration file, the Serializer can be specified using the `serializer` option.

Configuration file example:

```
serializer: json
```

Multiple Publishers

It is possible to select a separate serializer per Publisher, specifying the name using the *serializer: msgpack* configuration option.

Available serializers

MessagePack

This is the default Serializer used by napalm-logs.

> MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves.

Source: MessagePack.

Given the following napalm-logs document (as JSON):

```
{
  "yang_message": {
        "bgp": {
          "neighbors": {
                "neighbor": {
                  "192.168.140.254": {
                        "state": {
                          "session_state": "CONNECT"
                        }
                  }
                }
          }
        }
  },
  "message_details": {
        "processId": null,
        "severity": 4,
        "facility": 0,
        "hostPrefix": null,
        "pri": "4",
        "processName": "kernel",
        "host": "vmx01",
        "tag": "tcp_auth_ok",
        "time": "21:23:00",
        "date": "Jul 20",
        "message": "Packet from 192.168.140.254:61664 wrong MD5 digest"
  },
  "timestamp": 1500585780,
  "facility": 0,
  "ip": "127.0.0.1",
  "host": "vmx01",
  "yang_model": "openconfig-bgp",
  "error": "BGP_MD5_INCORRECT",
  "os": "junos",
  "severity": 4
}
```

The document will be binary serialized as:


```
\x8a\xacyang_message\x81\xa3bgp\x81\xa9neighbors\x81\xa8neighbor\x81\xaf192.168.140.
↪254\x81\xa5state\x81\xadsession_state\xa7CONNECT\xafmessage_
↪details\x8b\xa9processId\xc0\xa8severity\x04\xa8facility\x00\xaahostPrefix\xc0\xa3pri\xa14\xabproce
↪auth_ok\xa4time\xa821:23:00\xa4date\xa6Jul 20\xa7message\xd92Packet from 192.168.
↪140.254:61664 wrong MD5 digest\xa8facility\x00\xa2ip\xa9127.0.0.1\xa5error\xb1BGP_
↪MD5_INCORRECT\xa4host\xa5vmx01\xaayang_model\xaeopenconfig-
↪bgp\xa9timestamp\xceYq\x1f4\xa2os\xa5junos\xa8severity\x04
```

JSON

The structured messages can be JSON serialized.

Given the following napalm-logs document (represented as a Python object):

```
{'error': 'BGP_MD5_INCORRECT',
 'facility': 0,
 'host': 'vmx01',
 'ip': '127.0.0.1',
 'message_details': {'date': 'Jul 20',
                                         'facility': 0,
                                         'host': 'vmx01',
                                         'hostPrefix': None,
                                         'message': 'Packet from 192.168.140.
↪254:61664 wrong MD5 digest',
                                         'pri': '4',
                                         'processId': None,
                                         'processName': 'kernel',
                                         'severity': 4,
                                         'tag': 'tcp_auth_ok',
                                         'time': '21:23:00'},
 'os': 'junos',
 'severity': 4,
 'timestamp': 1500585780,
 'yang_message': {'bgp': {'neighbors': {'neighbor': {'192.168.140.254': {'state': {
↪'session_state': 'CONNECT'}}}}}},
 'yang_model': 'openconfig-bgp'}
```

The document will be JSON serialized as:

```
{
  "yang_message": {
        "bgp": {
          "neighbors": {
                "neighbor": {
                  "192.168.140.254": {
                        "state": {
                          "session_state": "CONNECT"
                        }
                  }
                }
          }
        }
  },
  "message_details": {
        "processId": null,
        "severity": 4,
        "facility": 0,
        "hostPrefix": null,
        "pri": "4",
        "processName": "kernel",
        "host": "vmx01",
        "tag": "tcp_auth_ok",
        "time": "21:23:00",
```

```
        "date": "Jul 20",
        "message": "Packet from 192.168.140.254:61664 wrong MD5 digest"
  },
  "timestamp": 1500585780,
  "facility": 0,
  "ip": "127.0.0.1",
  "host": "vmx01",
  "yang_model": "openconfig-bgp",
  "error": "BGP_MD5_INCORRECT",
  "os": "junos",
  "severity": 4
}
```

YAML

The structured messages can be YAML serialized. This can be used for a variety of cases, including CLI usage for a human readable display, but also for other Publisher interfaces.

Given the following napalm-logs document (as JSON):

```
{
  "yang_message": {
        "bgp": {
          "neighbors": {
                "neighbor": {
                  "192.168.140.254": {
                        "state": {
                          "session_state": "CONNECT"
                        }
                  }
                }
          }
        }
  },
  "message_details": {
        "processId": null,
        "severity": 4,
        "facility": 0,
        "hostPrefix": null,
        "pri": "4",
        "processName": "kernel",
        "host": "vmx01",
        "tag": "tcp_auth_ok",
        "time": "21:23:00",
        "date": "Jul 20",
        "message": "Packet from 192.168.140.254:61664 wrong MD5 digest"
  },
  "timestamp": 1500585780,
  "facility": 0,
  "ip": "127.0.0.1",
  "host": "vmx01",
  "yang_model": "openconfig-bgp",
  "error": "BGP_MD5_INCORRECT",
  "os": "junos",
  "severity": 4
}
```

The document will be YAML serialized as:


```
error: BGP_MD5_INCORRECT
facility: 0
host: vmx01
ip: 127.0.0.1
message_details:
  date: Jul 20
  facility: 0
  host: vmx01
  hostPrefix: null
  message: Packet from 192.168.140.254:61664
        wrong MD5 digest
  pri: 4
  processId: null
  processName: kernel
  severity: 4
  tag: tcp_auth_ok
  time: 21:23:00
os: junos
severity: 4
timestamp: 1500585780
yang_message:
  bgp:
        neighbors:
          neighbor:
                192.168.140.254:
                  state:
                        session_state: CONNECT
yang_model: openconfig-bgp
```


Pretty Print

PrettyPrint string representation of the Python object. This Serializer is mainly recommended to be used for CLI debugging.

For example, given the following napalm-logs document (as JSON):


```
{
  "yang_message": {
        "bgp": {
          "neighbors": {
                "neighbor": {
                  "192.168.140.254": {
                        "state": {
                          "session_state": "CONNECT"
                        }
                  }
                }
          }
        }
  },
  "message_details": {
        "processId": null,
        "severity": 4,
        "facility": 0,
        "hostPrefix": null,
```

```
        "pri": "4",
        "processName": "kernel",
        "host": "vmx01",
        "tag": "tcp_auth_ok",
        "time": "21:23:00",
        "date": "Jul 20",
        "message": "Packet from 192.168.140.254:61664 wrong MD5 digest"
  },
  "timestamp": 1500585780,
  "facility": 0,
  "ip": "127.0.0.1",
  "host": "vmx01",
  "yang_model": "openconfig-bgp",
  "error": "BGP_MD5_INCORRECT",
  "os": "junos",
  "severity": 4
}
```

The document will be serialized as:

```
{'error': 'BGP_MD5_INCORRECT',
 'facility': 0,
 'host': 'vmx01',
 'ip': '127.0.0.1',
 'message_details': {'date': 'Jul 20',
                                          'facility': 0,
                                          'host': 'vmx01',
                                          'hostPrefix': None,
                                          'message': 'Packet from 192.168.140.
↪254:61664 wrong MD5 digest',
                                          'pri': '4',
                                          'processId': None,
                                          'processName': 'kernel',
                                          'severity': 4,
                                          'tag': 'tcp_auth_ok',
                                          'time': '21:23:00'},
 'os': 'junos',
 'severity': 4,
 'timestamp': 1500585780,
 'yang_message': {'bgp': {'neighbors': {'neighbor': {'192.168.140.254': {'state': {
↪'session_state': 'CONNECT'}}}}}},
 'yang_model': 'openconfig-bgp'}
```

String

Simply a string representation of the Python object. This Serializer can mainly be used for CLI debugging.

For example, given the following napalm-logs document (as JSON):

```
{
  "yang_message": {
        "bgp": {
          "neighbors": {
                "neighbor": {
                  "192.168.140.254": {
                        "state": {
```

```
                            "session_state": "CONNECT"
                          }
                    }
                  }
            }
          }
  },
  "message_details": {
        "processId": null,
        "severity": 4,
        "facility": 0,
        "hostPrefix": null,
        "pri": "4",
        "processName": "kernel",
        "host": "vmx01",
        "tag": "tcp_auth_ok",
        "time": "21:23:00",
        "date": "Jul 20",
        "message": "Packet from 192.168.140.254:61664 wrong MD5 digest"
  },
  "timestamp": 1500585780,
  "facility": 0,
  "ip": "127.0.0.1",
  "host": "vmx01",
  "yang_model": "openconfig-bgp",
  "error": "BGP_MD5_INCORRECT",
  "os": "junos",
  "severity": 4
}
```


The document will be serialized as:


```
{u'yang_message': {u'bgp': {u'neighbors': {u'neighbor': {u'192.168.140.254': {u'state
↪': {u'session_state': u'CONNECT'}}}}}}, u'message_details': {u'processId': None, u
↪'severity': 4, u'facility': 0, u'hostPrefix': None, u'pri': u'4', u'processName': u
↪'kernel', u'host': u'vmx01', u'tag': u'tcp_auth_ok', u'time': u'21:23:00', u'date':␣
↪u'Jul 20', u'message': u'Packet from 192.168.140.254:61664 wrong MD5 digest'}, u
↪'facility': 0, u'ip': u'127.0.0.1', u'error': u'BGP_MD5_INCORRECT', u'host': u'vmx01
↪', u'yang_model': u'openconfig-bgp', u'timestamp': 1500585780, u'os': u'junos', u
↪'severity': 4}
```


3.5.10 Logger

Deprecated since version 0.4.0.

Warning: The Logger interface has been deprecated beginning with release 0.4.0. Please use the *Publisher* interface instead, using the *only_raw: False* or *send_raw: False* Publisher configuration options. For example, if you used the following configuration for the Logger:

```
logger:
  kafka:
    send_raw: true
```

The configuration must be updated to:

Using `only_raw` is recommended to ensure that the Publisher will be used only for this exact purpose. However, the user can decide what is the most suitable for their use case.

The logger subsystem uses the modules from the publisher pluggable subsystem to send partially parsed syslog messages. The configuration options are the same as for the publisher referenced – see the *Available publishers and their options*. It can be used together with the publisher system in such a way the publisher externalizes the fully processed objects and the clients can subscribe and collect them, while the logger submits the partially parsed messages. This is ideal for logging these unprocessed messages, hence the *logger* name.

This subsystem is by default disabled and it cannot be configured from the command line, but only from the configuration file. Besides the publisher name to be specified, it also requires to configure at least one set one of the options below:

`send_raw`

If this option is set, all processed syslog messages, even ones that have not matched a configured error, will be output via the specified transport. This can be used to forward to log server for storage.

Example:

```
logger:
  kafka:
    send_raw: true
```

`send_unknown`

If this option is set, all processed syslog messages, even ones that have not matched a certain operating system, will be output via the specified transport. This can be used to forward to log server for storage.

Example:

```
logger:
  zmq:
    send_unknown: true
```

3.5.11 The format of the syslog messages

While the structure of the syslog messages should not be very much different than the base IEFT syslog protocol, each platform has its own format which does not necessarily commit to the standards.

As in opposite to the standard structure, the most common format of the syslog messages has two components:

- *HEADER* (including *PRI*)
- *MSG*

In general their format varies between platforms, the structure being explained in the following documents, individually:

Junos

In general, the structure of the syslog messages generated by Junos has the following format:

```
<PRI><datetime> <hostname> <process-name>[<process-id>]:  <syslog-tag>:  <MSG>
```

Where:

- `datetime`: The time when the message was generated in the format: `MMM dd hh:mm:ss`.
- `hostname`: The device that generated the message.
- `process-name`: The name of the process that generated the mesage.
- `process-id` The PID of the process that generated the message.
- `syslog-tag`: The Junos tag of the syslog message. To see all the possible tags, execute `help syslog ?`.

Examples:

```
<25>Jun 21 14:03:12 vmx01 eswd[2902]: ESWD_BPDU_BLOCK_ERROR_DISABLED: ge-0/0/
17.0: bpdu-block disabled port
```

```
<87>Jul 5 05:52:44 vmx01 rpd[1848]: bgp_read_message:2764: NOTIFICATION
received from 1.2.3.4 (External AS 1234): code 6 (Cease) subcode 5
(Connection Rejected)
```

PRI

Junos defines the following facilities, based on the standard PRI:

Numerical code	Standard keyword	Junos facility name	Description
0	kern	LOG_KERN	Actions performed or errors encountered by the Junos kernel
1	user	LOG_USER	Actions performed or errors encountered by user-space processes
3	daemon	LOG_DAEMON	Actions performed or errors encountered by system processes
4	auth	LOG_AUTH	Authentication and authorization attempts
5	syslog	LOG_SYSLOG	Actions performed or errors encountered by the Junos system logging utility
7	news	LOG_NEWS	Network news subsystem
10	authpriv	LOG_AUTHPRIV	Authentication and authorization attempts that can be viewed by superusers only
11	ftp	LOG_FTP	Actions performed or errors encountered by the FTP process
12	ntp	LOG_NTP	Actions performed or errors encountered by the Network Time Protocol (NTP)
15	cron	LOG_CRON	Actions performed or errors encountered by the cron process

To see the messages that are under a specific facility, Junos allows you to check that using the following command: `help syslog facility <junos facility name>`, e.g., `help syslog facility LOG_USER`.

Cisco IOS-XR

In general, the structure of the syslog messages generated by IOS-XR has the following format:

```
<PRI><messageid>: <hostname> <linecard>:<datetime>: <process-name>[<process-id>]:
%<facility-name>-<severity>-<tag>: <MSG>
```

Where:

- `messageid`: The ID number of the message.
- `hostname`: The device that generated the message. To ensure that the hostname is included, follow the instructions from *Cisco IOS-XR*.
- `linecard`: The linecard slot.
- `datetime`: The time when the message was generated in the format: `MMM dd hh:mm:ss.fff` or `MMM dd hh:mm:ss.fff ZZZ`.
- `process-name`: The name of the process that generated the mesage.
- `process-id` The PID of the process that generated the message.
- `facility-name`: The name of the Facility.
- `severify`: The value of the Severity.
- `tag`: The syslog message tag.

Examples:

```
<149>2647599: vmx01 RP/0/RSP1/CPU0:Mar 28 15:08:30.941 UTC: bgp[1051]:
%ROUTING-BGP-5-MAXPFX : No. of IPv4 Unicast prefixes received from 1.2.3.4
has reached 94106, max 125000
```

```
<187>94307: gw2.acy1 LC/0/2/CPU0:Jul 7 20:16:14.834 : ifmgr[214]:
%PKT_INFRA-LINK-3-UPDOWN : Interface TenGigE0/2/0/4, changed state to Down
```

Arista EOS

In general, the structure of the syslog messages generated by EOS has the following format:

```
<PRI><datetime> <hostname> <process-name>: %<facility-name>-<severity>-<tag>:
<MSG>
```

Where:

- `hostname`: The device that generated the message. To ensure that the hostname is included, follow the instructions from *Arista EOS*.
- `datetime`: The time when the message was generated in the format: `MMM dd hh:mm:ss`.
- `process-name`: The name of the process that generated the mesage.
- `facility-name`: The name of the Facility.
- `severify`: The value of the Severity.
- `tag`: The syslog message tag.

Examples:

``<149>Apr 16 11:04:17 edge01 Rib: %BGP-3-NOTIFICATION: received from neighbor 194.53.172.97 (AS 2611) 6/1 (Cease/maximum number of prefixes reached) 0 bytes ``

Cisco NX-OS

In general, the structure of the syslog messages generated by IOS-XR has the following format:

`<PRI><hostname>: <datetime>: %<facility-name>-<severity>-<tag>: <MSG>`

Where:

- `hostname`: The device that generated the message.
- `datetime`: The time when the message was generated in the format: `MMM dd hh:mm:ss.fff ZZZ`.
- `facility-name`: The name of the Facility.
- `severify`: The value of the Severity.
- `tag`: The syslog message tag.

Examples:

``<190>sw01.pdx01: 2017 Jul 28 14:42:46 UTC: %AUTHPRIV-6-SYSTEM_MSG: pam_unix(dcos_sshd:session): session opened for user luke by (uid=0) - dcos_sshd[12977] ``

PRI

The Priority value is calculated by first multiplying the Facility number by 8 and then adding the numerical value of the Severity. For example, a kernel message (Facility=0) with a Severity of Emergency (Severity=0) would have a Priority value of 0.

In addition to the standard PRI classification, each platform defines additional values for Facility which may differ from a platform to another.

The Severity however usually respects the standard:

Numerical code	Severity level	Description
0	emergency	System panic or other condition that causes the router to stop functioning
1	alert	Conditions that require immediate correction, such as a corrupted system database
2	critical	Critical conditions, such as hard errors
3	error	Error conditions that generally have less serious consequences than errors in the emergency, alert, and critical levels
4	warning	Conditions that warrant monitoring
5	notice	Conditions that are not errors but might warrant special handling
6	info	Events or nonerror conditions of interest
7	debug	Software debugging messages (these appear only if a technical support representative has instructed you to configute this severity level

HEADER

The syslog messages from network devices do not respect the standard HEADER.

MSG

The MSG part contains the text of the message itself.

3.5.12 Development

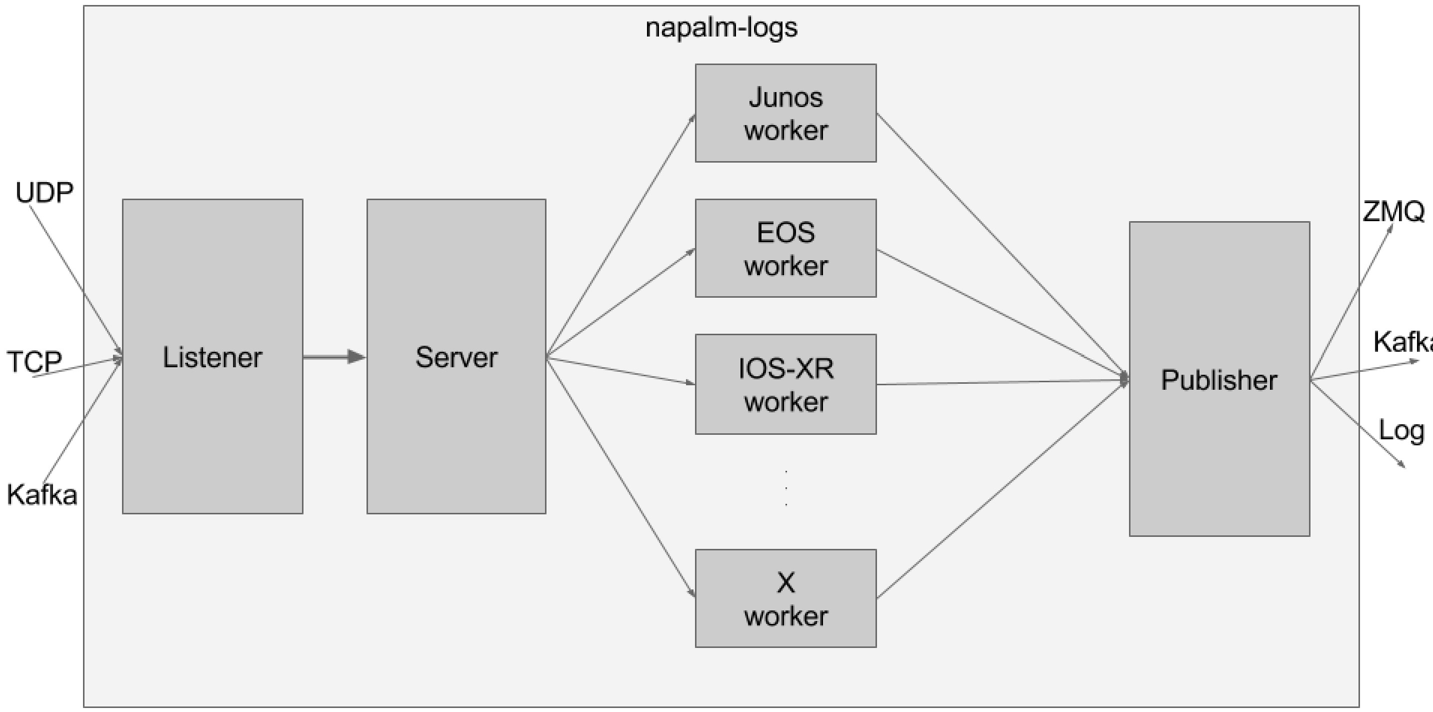
Here we you will find out how to add new functionality to `napalm-logs`.

Architecture

Besides speed, there were a couple of considerations we had in mind when we designed napalm-logs:

- Size
- Security
- Flexibility
- Reliability

The core achitecture can be represented in the diagram below; for simplicity, we will analyse the security in the *Authenticator* paragraph:



The napalm-logs prgram starts a couple of processes to handle and manipulate the syslog messages. We called them:

- *Listener*
- *Server*
- *Device (one per platform)*
- *Publisher*

The processes communicate between them using ZeroMQ IPC.

Listener

The Listener has the role to receive the syslog snippets and queue them (`PUSH`) into an IPC socket to the Server. This has two advantages: we ensure we queue the messages immediately as received - no time wasted triaging (this is very important when the messages are received over UDP, for example – see the *UDP* listener). The other gain by doing so is that we don't loose any messages: not even when the napalm-logs process crashes or is intentionally stopped: after restart, the Server will continue dequeuing messages from that buffer.

The Listener is a pluggable interface, check *Listener* for more details.

The communication between the Listener and the Server is a straight PUSH-PULL socket.

Server

The Server is the process that deals with the triage: using the *Device Profiles*, it identifies the platform it comes from. Using this information, the messages will be queued to the corresponding worker (see next section). The pattern in this case is a Ventiallator Sink, more specifically implemented using ROUTER and DEALER sockets, where the Server is the ROUTER, and each Platform Worker is a DEALER.

The messages at this point are partially parsed, and they can be published using one of the available *Publisher*, through the *Logger* interface. When unable to identify the platform, the message is by default discarded. However, the user can activate the messages to be published using the *send_unknown: False* option, the format being *UNKNOWN*. Note that the Logger interface has a similar option, *send_unknown*.

Device

There is one device worker started per platform. Each worker receives the partially processed messages from the Server, then extracts the data and maps it to the OpenConfig or IETF YANG model, as configured in the *Device Profiles*. When a message does not have a corresponding profile mapping, it is discarded. To receive these messages, the user can choose to publish them using the *send_raw: False* option.

The messages are then sent to the Publisher IPC socket using `PUSH`.

You can avoid unwanted workers using the *device_blacklist* and *device_whitelist* options.

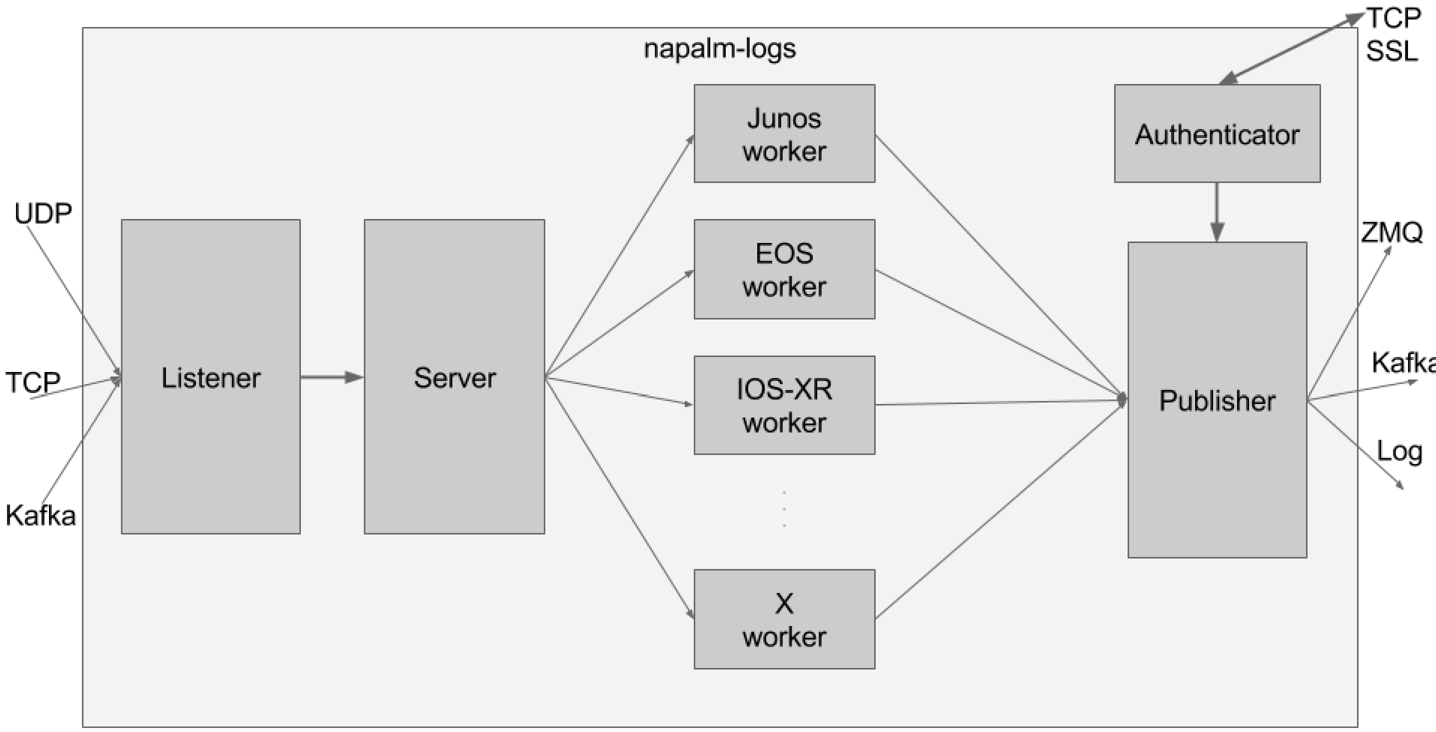
Publisher

The Publisher process retrieves the messages from the IPC socket using `PULL` operations, then forwards them over the *Publisher* interfaces. When the messages encryption is not turned off (see *disable-security*), the Publisher also has the role of encrypting and signing before publishing. Regardless of the security being disabled or not, the messages are binary serialised using MessagePack.

The Publisher is another pluggable interface, check *Publisher* for more further information.

Authenticator

By default, napalm-logs starts an additional process, the Authenticator. When security is explicitly disabled using the *disable-security* option, this process is not started.



The Authenticator generates a private and a signature key, which are used by the Publisher to encrypt and sign the binary serialised messages.

The clients receive these keys through an exchange via a TCP socket; this socket is SSL secured using the *certificate* and the *keyfile* provided by the user. Each client connection is handled in a separate thread, and the Authenticator keeps this connection alive for further notifications.

Read more about the *Client Authentication*.

Pluggable Modules

`napalm-logs` is designed to be pluggable, so new methods for both input and output and can be added easily. This is to allow for the widest compatibility possible.

Adding a New Module

If you need to use a different method to pass in your syslog messages or to get the processed messages, and it is not yet defined, you can write your own.

These are the basic steps required, and are the same for all of the pluggable sections.

Create a new module in the appropriate directory, name it the same as the protocol it will be using.

Copy the general format from an existing module.

All options for your module can be specified in the general config file, these will be passed to your module as `kwargs`.

The module will be initialised, then started by calling the `start()` function.

If a signal is sent to the parent process, it will send a `SIGTERM` to your module, therefore this should be caught and the module should exit cleanly.

This can be done by including the following in your `start()`:

```
signal.signal(signal.SIGTERM, self._exit_gracefully)
self.__up = True
# Code before the loop
while self.__up:
    # Code to execute for each object
```

Then adding the following function:

```
def _exit_gracefully(self, signum, _):
    log.debug('Caught signal in <process name> process')
    self.stop()
```

And also having a `stop()` which closes everything cleanly.

It is a good idea to look at some of the other modules to get an idea of how to structure yours.

Once written you should update `__init__` in the appropriate directory to include your newly created class, and add this class to the `dict` of all selectable classes.

If your module has dependencies then you should add a check to make sure the dependency is present, and call that function before adding your class to the `dict` of all selectable classes.

If you would like to have any default values for your module you can add these to `napalm_logs/config/__init__.py` under the appropriate `*_opts` dictionary.

Device Profiles

Most network equipment vendors use different syslog message format to each other, some even use a different format for each of their devices. For `napalm-logs` to be able to take a syslog message from a device and output it in a vendor-agnostic way, it needs to know the format of that device's messages.

Each network operating system has a set of profiles, defined under a directory with the name of the platform, by default defined under `napalm_logs/config`. For example, the profiles for `eos` are defined under `napalm_logs/config/eos/`, for `junos` under `napalm_logs/config/junos/` and so on.

Directory tree structure example:

```
napalm_logs/config/
├── __init__.py
├── eos
│   └── init.yml
├── iosxr
│   └── __init__.py
├── junos
│   └── init.yml
```

```
└── nxos
    └── init.yml
```

The user can select to extend the capabilities of the public library, by defining profiles under a directory, specifying the path using the *extension-config-path* option.

Custom directory tree example:

```
/pat/to/custom/config/
├── eos
│   └── bgp_3_notification.py
├── junos
    └── init.yml
    └── UI_DBASE_LOGIN_EVENT.py
    └── SNMP_TRAP_LINK_DOWN.py
```

Each syslog message can be divided into two logical sections:

- the identification section, which provides enough information to identify the operating system that generated the message, together with other details, such as datetime, hostname, PRI, process daemon, PID, etc. In napalm-logs, this section will be referenced as *prefix*.
- the actual message section, which is the part of the syslog message which contains the useful information. In napalm-logs, this section will be referenced as *message*.

Example: given the message `Mar 30 12:45:19 re0.edge01.bjm01 rpd[15852]: BGP_PREFIX_THRESH_EXCEEDED 1.2.3.4 (External AS 15169): Configured maximum prefix-limit threshold(160) exceeded for inet-unicast nlri: 181 (instance master)`:

- `Mar 30 12:45:19 re0.edge01.bjm01 rpd[15852]: BGP_PREFIX_THRESH_EXCEEDED` is the *prefix* section.
- `1.2.3.4 (External AS 15169): Configured maximum prefix-limit threshold(160) exceeded for inet-unicast nlri: 181 (instance master)` is the *message* section.

Both sections are platform-specific, and the *prefix* part can be used to idenfiy the operating system that generated a certain syslog message. The identification is done via *prefix matchers* (*prefix parsers*). Similarly, the extraction of the information from the message section is done via *message parsers*.

Please note that some platforms do not respect a single prefix pattern, but a variety, this is why we need a couple of *prefix matchers*.

YAML Profiles

Each config file has two distinct sections, one to identify the OS that the message originated from (called `prefixes`), and one to identify each log message that `napalm-logs` should convert (called `messages`).

`prefixes`

This section defines what we have defined above as *prefix matches*, or *prefix parsers*, for the OS in question.

Here is the config for `junos`:

```yaml
prefixes:
  - time_format: "%b %d %H:%M:%S"
    values:
    date: (\w+\s+\d+)
    time: (\d\d:\d\d:\d\d)
    hostPrefix: (re\d.)?
    host: ([^ ]+)
    processName: /?(\w+)
    processId: \[?(\d+)?\]?
    tag: (\w+)
    line: '{date} {time} {hostPrefix}{host} {processName}{processId}: {tag}: '
```

Note: Prefix parsers are usually defined as `__init__.yml`, `init.yml` or `index.yml`.

What does each option mean?

`line`

This represents the format of the part of the log message that present most of the time. Each section of the message that can change should be replaced by a variable. If a variable isn't always present then you should add it to the line but make that variable optional (covered in the `values` section).

Any white space in `line` will match any number of contiguous white space, therefore if it is possible for there to be either one white space or two white spaces, you should only add one white space to `line`.

`values`

This is used to specify the regex pattern for each of the variables specified in `line`. All variables in `line` should have an entry under `values`, even if you have no use for them.

Each of these variables will be output in a message dict after processing.

`messages`

Here is where all log messages that should be matched are specified.

Note: Message parsers are usually defined under a YAML file having the name of the error ID they produce. However, this is not absolutely mandatory.

Here is an example message:

```yaml
messages:
  - error: INTERFACE_DOWN
    tag: SNMP_TRAP_LINK_DOWN
    values:
      snmpID: (\d+)
      adminStatusString|uppercase: (\w+)
      adminStatusValue: (\d)
      operStatusString|uppercase: (\w+)
      operStatusValue: (\d)
      interface: ([\w\-\/]+)
```

```
    line: 'ifIndex {snmpID}, ifAdminStatus {adminStatusString}({adminStatusValue}),␣
↪ifOperStatus {operStatusString}({operStatusValue}), ifName {interface}'
    model: openconfig_interfaces
    mapping:
      variables:
      interfaces//interface//{interface}//state//admin_status: adminStatusString
      interfaces//interface//{interface}//state//oper_status: operStatusString
      static: {}
```

What does each option mean?

error

This is the vendor agnostic ID for the error message, the `error` for each message should be unique. Currently we are using the `junos` definitions where possible, this is likely to change.

tag

This is the unique ID from the device itself.

This field is used when identifying if the log message is related to the configured error. Some devices use the same name for different types of logs, therefore this does not need to be unique.

If you look at the config for `prefix` above, you will see the variable `tag` in `line`, this is the same `tag` as configured here and matched on.

match_on: tag

This field name the field that try to match on. Defaults to `tag`.

line

This is the same as `line` above.

values

This is the same as `values` above, other than the fact they can be used in `mapping` (this will be covered under `mapping`). You can manipulate these values using replace functions found in napalm_logs.utils.Replace i.e `adminStatusString|uppercase`.

model

This is the YANG model to use to output the log message. You can find all models and their structure here.

mapping

This shows where in the OpenConfig model each of the variables in the message should be placed. There are two options, `variables` and `static`. `variables` should be used when the value being set is taken from the message, and `static` should be used when the value is manually set.

Pure Python profiles

Writing YAML profiles is flexible and fast, but this model comes with many logical limitations. For this reason, the developer can equally write pure Python `prefixes` or `messages` parsers. They can be defined under the same directory as the YAML descriptors, and they will be loaded dynamically.

Note: The user is allowed to use any combination of YAML and pure Python parsers to match the messages and defined the prefixes.

Similarly to the YAML profilers, the Python profiles have two logical sections: `prefixes` that provide the operating system identification and `messages` that extract the information from the raw syslog messages and maps to an object having the YANG hierarchy. Both are free-form Python modules, with a single constraint that will be explained below.

prefixes

A pure Python module that provides the prefix configuration, in order to identify the operating system generating the message.

A module providing the prefix needs to define a function called `extract` that takes a single argument, `msg` which is the raw syslog message received from the network device. The function has to return a dictionary with the parts extracted from the syslog message, without any further processing. The following keys are mandatory:

- `host`: the network device hostname, as provided in the syslog message

prefix section. - `tag`: which is the unique identification tag of the syslog message, e.g. in the message `Mar 30 12:45:19 re0.edge01.bjm01 rpd[15852]: BGP_PREFIX_THRESH_EXCEEDED 1.2.3.4 (External AS 15169): Configured maximum prefix-limit threshold(160) exceeded for inet-unicast nlri: 181 (instance master),` the tag is `BGP_PREFIX_THRESH_EXCEEDED`. Other tag examples: `bgp_read_message`, `ROUTING-BGP-5-MAXPFX` or even `Alarm set`. - `message`: is the message that what we have defied earlier as *the message section*, e.g. `User 'dummy' entering configuration mode.`

Note: Prefix parsers are usually defined as `__init__.py`, `init.py` or `index.py`.

The following example is a Python prefix parser for NX-OS:

```
import re
from collections import OrderedDict

import napalm_logs.utils

_RGX_PARTS = [
    ('pri', r'(\d+)'),
    ('host', r'([^ ]+)'),
    ('date', r'(\d+ \w+ +\d+)'),
    ('time', r'(\d\d:\d\d:\d\d)'),
```

```
    ('timeZone', r'(\w\w\w)'),
    ('tag', r'([\w\d-]+)'),
    ('message', r'(.*)')
]
_RGX_PARTS = OrderedDict(_RGX_PARTS)

_RGX = '\<{0[pri]}\>{0[host]}: {0[date]} {0[time]} {0[timeZone]}: %{0[tag]}:
↪{0[message]}'.format(_RGX_PARTS)


def extract(msg):
    return napalm_logs.utils.extract(_RGX, msg, _RGX_PARTS)
```

The example above matches messages from NX-OS looking like: `<190>sw01.bjm01: 2017 Jul 26 14:42:46 UTC: %SOME-TAG: this is a very useful syslog message`, and extracts the following details:

- `pri`: 190
- `host`: sw01.bjm01
- `tag`: SOME-TAG
- `date`: 2017 Jul 26
- `time`: 14:42:46
- `timeZone`: UTC
- `message`: this is a very useful syslog message

These details are returned by the `extract` function, which returns a dictionary such as:

```
{
  'pri': '190',
  'host': 'sw01.bjm01',
  'tag': 'SOME-TAG',
  'time': '14:42:46',
  'date': '2017 Jul 26',
  'timeZone': 'UTC',
  'message': 'this is a very useful syslog message'
}
```

Except `tag`, `host` and `message`, all the other fields can be optional, and **they are platform-specific** (or even message-type-specific, in some very sad cases). However, there are some particular cases when the other fields can provide interesting information, eventually to be used to match messages using the `match_on` option.

`messages`

Writing a message parser can be equally simple and flexible, the rules to consider being:

- Define a function called `emit` that generates the syslog message.
- A dunder called `__yang_model__` that specifies the YANG model.
- A variable names `__tag__` that specifies the tag name, that is used to match when comparing the value of the `tag` field extracted from the message prefix and determine what parser should process the syslog message. However, this variable is optional – when not defined, it will use the filename as tag.

- A variable called `__error__` that defines the name of the global error. Each structured message published by napalm-logs has a certain error tag, that is unique and cross-platform. This variable is also optional – when not defined, the error ID will be the file name.

Note: Message parsers are usually defined under a Python file having the name of the error ID they produce. However, this is not absolutely mandatory.

Useful functions

At times, the developer may find very useful several functions, in order to acomplish recurrent tasks:

- `napalm_logs.utils.extract`: Extracts the fields from a unstructured text, given a field-regex mapping. Please check the previous paragraph for an usage example.
- `napalm_logs.utils.setval`: Set a value under the dictionary hierarchy identified under the key. The key `'foo//bar//baz'` will configure the value under the dictionary hierarchy `{'foo': {'bar': {'baz': {}}}}`. Example:

```
>>> napalm_logs.utils.setval('foo//bar//baz', 'value')
{'foo': {'bar': {'baz': 'value'}}}
```

- `napalm_logs.utils.traverse`: Traverse a dict or list using a slash delimiter target string. The target `'foo//bar//0'` will return `data['foo']['bar'][0]` if this value exists, otherwise will return empty dict. Return `None` when not found. This can be used to verify if a certain key exists under dictionary hierarchy.

3.5.13 Release Notes

Latest Release

- release-0.4.0

Previous Releases

- release-0.3.0
- release-0.2.0
- release-0.1.0