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:

```
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)
```

Will produce the following object:

```
{
  "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",
    "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 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
}
```


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 *Instalation*.

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 Instalation

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
```

3.5.2 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.

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/
```

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
```

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/
```

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

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

Default: `udp`.

CLI usage example:

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

Configuration file example:

```
listener: kafka
```

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

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

This can be assigned using `-p`

The port to use to listen for all incoming syslog messages.

Default: `514`.

CLI usage example:

```
$ napalm-logs -p 1024
$ napalm-logs --port 1024
```

Configuration file example:

```
port: 1024
```

publish-address

The IP address to use to output the processed message.

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

The port to use to output the processes message.

Default: `49017`.

CLI usage example:

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

Configuration file example:

```
publish_port: 2048
```

transport

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

Default: `zmq` (ZeroMQ).

CLI usage example:

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

Configuration file example:

```
transport: kafka
```

Or:

```
publisher: 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
```

3.5.3 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:


```
{
  "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_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

- junos
- iosxr

Syslog message example

```
<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)
```

Structured message example


```
{
  "error": "BGP_PREFIX_THRESH_EXCEEDED",
  "facility": 18,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jun 21",
      "facility": 18,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "192.168.140.254 (External AS 4230): Configured maximum prefix-limit␣
↪threshold(140) exceeded for inet4-unicast nlri: 141 (instance master)",
```

```
        "pri": "149",
        "processId": "2902",
        "processName": "rpd",
        "severity": 5,
        "tag": "BGP_PREFIX_THRESH_EXCEEDED",
        "time": "14:03:12"
    },
    "os": "junos",
    "severity": 5,
    "timestamp": 1498053792,
    "yang_message": {
        "bgp": {
            "neighbors": {
                "neighbor": {
                    "192.168.140.254": {
                        "afi_safis": {
                            "afi_safi": {
                                "inet4": {
                                    "ipv4_unicast": {
                                        "prefix_limit": {
                                            "state": {
                                                "max_prefixes": 140
                                            }
                                        }
                                    },
                                    "state": {
                                        "prefixes": {
                                            "received": 141
                                        }
                                    }
                                }
                            }
                        },
                        "state": {
                            "peer_as": "4230"
                        }
                    }
                }
            }
        }
    },
    "yang_model": "openconfig-bgp"
}
```


INTERFACE_DOWN

Maps to the `openconfig-interfaces` YANG model.

Implemented for

- junos

Syslog message example

```
<28>Jul 20 21:45:59 vmx01 mib2d[2424]: SNMP_TRAP_LINK_DOWN: ifIndex 502,␣
↪ifAdminStatus down(2), ifOperStatus down(2), ifName xe-0/0/0
```

Structured message example


```
{
  "error": "INTERFACE_DOWN",
  "facility": 3,
  "host": "vmx01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 20",
      "facility": 3,
      "host": "vmx01",
      "hostPrefix": null,
      "message": "ifIndex 502, ifAdminStatus down(2), ifOperStatus down(2), ifName xe-
↪0/0/0",
      "pri": "28",
      "processId": "2424",
      "processName": "mib2d",
      "severity": 4,
      "tag": "SNMP_TRAP_LINK_DOWN",
      "time": "21:45:59"
  },
  "os": "junos",
  "severity": 4,
  "timestamp": 1500587159,
  "yang_message": {
      "interfaces": {
          "interface": {
              "xe-0/0/0": {
                  "state": {
                      "admin_status": "DOWN",
                      "oper_status": "DOWN"
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-interfaces"
}
```


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

- junos
- iosxr

Syslog message example

```
<99>Jul 13 22:53:14  re0.edge01.bjm01 xntpd[16015]: NTP Server 1.2.3.4 is Unreachable
```

Structured message example


```
{
  "error": "NTP_SERVER_UNREACHABLE",
  "facility": 12,
  "host": "edge01.bjm01",
  "ip": "127.0.0.1",
  "message_details": {
      "date": "Jul 13",
      "facility": 12,
      "host": "edge01.bjm01",
      "hostPrefix": "re0.",
      "message": "NTP Server 1.2.3.4 is Unreachable",
      "pri": "99",
      "processId": "16015",
      "severity": 3,
      "tag": "xntpd",
      "time": "22:53:14"
  },
  "os": "junos",
  "severity": 3,
  "timestamp": 1499986394,
  "yang_message": {
      "system": {
          "ntp": {
              "servers": {
                  "server": {
                      "1.2.3.4": {
                          "state": {
                              "association-type": "SERVER",
                              "stratum": 16
                          }
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-system"
}
```

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* 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* 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"
}
```


3.5.4 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
```

3.5.5 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* configuration options.

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
```

3.5.6 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* and *publish-port* configuration options.

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
```

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
```

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

send_raw

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
```

send_unknown

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
```

3.5.7 Logger

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
```

3.5.8 Development

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

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`:

```
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:

```
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 function

At times, the developer may find very useful several function, 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.