opEvents input sources

opEvents can process information from a variety of sources, some of which can be extended to suit non-standard deployments. This document briefly documents how to configure opEvents' input sources.

Input Configuration

The inputs opEvents is supposed to handle are specified in the `opevents` section of `conf/opCommon.nmis`, primarily in subsection `opevents_logs`. opEvents primarily handles event information sourced by consuming and collating log files from sources like NMIS, Tivoli or general syslogs.

Here is an example configuration fragment:

```
'opevents_logs' => {
 # parsertype => list of logfiles (or dirs for nmis_json_dir)
 # natively supported: tivoli_log, cisco_syslog, nmis_traplog,
 # nmis_eventlog, nmis_slavelog and nmis_json_dir
 'cisco_syslog' => [ '<nmis_logs>/cisco.log', "/some/other/log.file" ],
 'tivoli_log' => [ '<nmis_logs>/tivoli.log' ],
 'nmis_traplog' => [ '<nmis_logs>/trap.log' ],
 'nmis_slavelog' => [ '<nmis_logs>/slave_event.log' ],  ### DEPRECATED - now uses below parser
 'nmis_pollerlog' => [ '<nmis_logs>/poller_event.log' ],
 'nmis_eventlog' => [ '<nmis_logs>/event.log' ],
 # attention: json logs in this directory are REMOVED after consumption
 # 'nmis_json_dir' => [ '<nmis_logs>/json' ],
},
```

The natively understood formats are:

Format Name (Parser Type)	Description
nmis_eventlog	An event log file created by NMIS
nmis_slavelog	An NMIS poller event log file
nmis_traplog	An NMIS trap log file
nmis_json_dir	A directory of NMIS event logs in JSON format
cisco_syslog	A Syslog log file containing logs created by Cisco devices
tivoli_log	A Tivoli log file

To enable a particular log file or format, you need to add an entry for the log file in question to the list of files for the appropriate log format; check the `cisco_syslog` entry in the example above for the syntax. The tokens `<nmis_something>` in the example work like centrally-defined shortcuts or macros; they are replaced by the actual locations given in the `directories` section at the beginning of `conf/opCommon.nmis`.

opEvents handles non-existent log files gracefully, but the log formats need to match the actual content. All log files are reopened on demand (e.g. when log rotation renames a file), and checked at least once every `opeventsd_update_rate` seconds. The order of log file specifications is not relevant.

Please note that the selection of the built-in parsers is tied to the value of the format name; all custom parsers that you might define must have their own, unique format names which must not clash with any of the built-in parser types.

Black and Whitelisting

opEvents ships with ready-made black and whitelist rules to reduce voluminous inputs down to the relevant details, but these can be adjusted at need. These lists are active if the settings `black_list_enabled` or `white_list_enabled` are set to `"true"`, respectively.

The black list contains a set of filtering rules which remove matching log entries from opEvents' input stream. The white list rules can be used to ensure that matching input entries are processed; if the white list is enabled, then *only* events matching the white list will be processed (but raw logging is still performed for forensics purposes). Enabling both black and white list options simultaneously is not useful.

Both black and white lists are configured in `conf/EventListRules.nmis`, in sections like this example:

```
'blackList' => {
 '10' => 'NTP Core \(INFO\)',
 '20' => 'OLD-CISCO-TS-MIB::tslineSesType\.6\.1=tcp',
 '30' => 'CISCO-SYSLOG-MIB::clogMessageGenerated',
 },
'whiteList' => {
 '1' => 'TIVOLI\|\w+\|ams',
 '10' => 'SYS-[0123]-\w+',
 '20' => 'LINEPROTO',
 '30' => 'OSPF-\d-ADJCHG',
...
```

The format is straight-forward: the numeric key controls order of rule application, and the right side is a regular expression that the log entries are matched against.

Normalisation and Enrichment

For the natively-supported log formats (except `nmis_json_dir`) only the actual parsing is hard-coded; the act of subsequent further extraction and collection of relevant details is configurable - but of course opEvents ships with a substantial set of default normalisation rules. Event normalisation consists of associating a log entry with a node, extracting details, determining whether the event is stateful or stateless, followed by optional additional enrichment from external sources.

Normalisation for snmptraps and logs are controlled by the configuration file `conf/EventParserRules.nmis` . The next section Generic Extensible Parser documents how this works.

There are also some built-in parsers for NMIS logs `EventNmisRules.nmis`, and even tivoli via `EventTivoliRules.nmis` which have a similar format. Discussed in the last section.

Further enrichment can be performed using policy actions (using the `tag.tagname()` action), enrichment statements in correlation rules or from external databases.

There is also the option to directly create Events using the Rest API which uses a similar json formats to the file format `nmis_json_dir` which is not subject to normalisation; instead the contents of these are expected to be normalised already.

Generic Extensible Parser

The default method for Normalisation is with the Generic Extensible Parser `conf/EventParserRules.nmis` you extend current parser entries or you ca define your own generic parser rules to integrate just about any text-based log information into opEvents. Your event is expected to contain all required event properties, the minimum for it to be accepted is a date and host entry but for the event to be usable in the Gui it also needs an "event" entry at a minimum.

The generic parser is activated for different log files in the configuration option `opevents_parser_rules`, in `conf/opCommon.nmis` , there is one entry for each log file which defines which 'parser' entry is to be used. The rules are defined in `conf/EventParserRules.nmis`. Here is an excerpt from the generic parser rules example that opEvents ships with: In this case the parser entry (used to associate it with certain log files) is called 'cisco_alternate'

```
'cisco_alternate' => {
 1 => {
         "IF" => qr/%/, # no cisco log if no % present
         "THEN" => {
                   # match date/time, host and details
                  10 => {
                         IF => qr/^(\S+\s+\d+\s+[\d:]+)\s+(\S+)[^%]+%(.+)$/,
                         THEN => "capture(date,host,details)",
                  },
                  # some units have Local instead of hms
                 11 => {
                         IF => qr/^(\S+\s+\d+)\s+Local\s+(\S+)[^%]+%(.+)$/,
                         THEN => "capture(date,host,details)",
                 },
                 # match event name, could have done that in one of the regexp above
                 20 => {
                         IF => qr/%(\w+\-\d-\w+):/,
                         THEN => "capture(event) AND capture(syslog)", # save this in two places
                 },
                  '23' => {
                         IF => qr/%BGP-5-ADJCHANGE: neighbor (\d+\.\d+\.\d+\.\d+) Down/,
                         THEN => 'capture(element) AND set.event(BGP Neighbor Down) AND set.state(down) AND set.
priority(4) AND set.stateful(BGP Neighbor)',
                  },
...
```

The format is straight-forward: the top key allocates a new log format type (here `cisco_alternate`) which you would use in `opevents_logs` for your log files. This parser type or format name must be distinct and not clash with any built-in parsers (e.g. creating a parser type "nmis_eventlog" won't work).

Under that key there are any number of (nested) capture rules, which control what to match in an input, and how to copy material to the newly created event. These rules use a format very similar to the Event Actions and Escalation policies: `IF` defines a regular expression that the log entry has to match, `THEN` declares what to do in that case, and a successful rule with optional `BREAK` statement skips the rules on the same nesting level.

The `THEN` expression consists of a nested sub-policy or of an action statement.

Before opEvents 2.2 the action statement must be an single string containing an AND-separated list of directives; from opEvents 2.2 onwards it can also be an explicit list of directives (which is faster and more flexible; see the `EventParserrules.nmis` that ships with opEvents for a Best-current-practice example).

In both cases the action statement must contain one or more of the supported directives:

- `set.`*`propertyname`*`(`*`value`*`)` sets the named property to the static value.
  No quoting of the value is required or supported.
  The character ")" cannot be part of the value before opEvents 2.2; In 2.2 and above it may only be present if you use the explicit list format for your action statement.
- `capture(`*`propname1,propname2`*`,...)` saves the respective captures from the regex in the named properties. The captures are assigned in their order in the regular expression; if you want grouping but not capturing, use `(?:....)` in your regex. Note that you cannot use multiple capture statements in one THEN.
- opEvents version 2.0 introduces the new action `ignore`. This aborts all parsing of this input line altogether and no event is created for it. Normally the generic parser is expected to extract suitable information for an event from *every* single input line, which might not work well if your log data is coming from multiple sources or can't be suitably prefiltered.
- In opEvents version 2.2 we've added the directives `resolve.fwd(`*`propname`*`) and resolve.rev(`*`propname`*`)`.
  The `resolve.fwd()` directive expects the property to be a DNS name and queries the DNS for an IP address associated with the name; the `resolve.rev()` directive interprets the property as an IP address and looks for a host name for it. If the resolution is successful, the property value is replaced by the DNS data; otherwise the property is left as-is.
  e.g. to resolve a BGP Peer address which is stored in the element name, add an entry to DNS or /etc/hosts and include resolve.rev(element) as a parser directive.
- opEvents 2.2 also adds the new directive `plugin(`*`PluginName`*`)`, which invokes an external parser plugin for further enrichment or modification of the event.
  This functionality is described in more detail in the next section.

Rules are applied in ascending order, defined by their numeric key, and nesting is fully supported.
Note that the numeric key may contain fractional numbers (e.g. "14.8"), which makes it very easy to insert new rules between existing ones.

Your event is has to include a Host and Date entry to be accepted. For it to be usable in the GUI it also at a minimum needs an "event" property. We recommend it includes further details per this page, event properties.

opEvents 2.0.6 and newer ships with complete generic parser rules for parsing Cisco syslogs (log format type "`cisco_alternate`") and SNMP trap logs (log format type "`traplog`"), plus other syslog, nxlog parsers for various vendors such as Huawei, Juniper, Microsoft, these can be extended and new entries can be contributed via code@opmantek.com .

Parser Plugins

For situations where external input must be incorporated into events at the time of parsing, opEvents 2.2 and newer support user-defined parser plugins. The directory `install/parser_plugins` contains an example plugin called `TestPlugin.pm` and a `README` file with documentation.

Requirements

All plugins must be valid Perl code. The files must be named `Something.pm` and reside in the directory `conf/parser-plugins`. Each plugin must have a proper '`package Something;`' namespace declaration that matches the file name, and this package namespace must not clash with any existing opEvents or NMIS components.

It's recommended that the plugin have a version declaration right after the package namespace declaration, e.g.. '`our $VERSION = "1.2.3";`'
The '`1;`' line at the end of the file is required and must not be omitted.

The plugin may load extra Perl modules with 'use', but it must not use any package-level global variables. All its variables and any objects that it might create must have local scope. The plugin must not use Exporter to export anything from its namespace.

Plugin Interface

A plugin must offer a function called `parse_enrich` or it will be ignored by opEvents.

When triggered by a `plugin(SomeName)` parser directive, the `parse_enrich` function of that plugin will be called with two arguments, line and event (in that order).

- Line is the complete log entry/line that is being processed and cannot be modified.
- Event is a hash reference and contains the preliminary *live* data structure of the event as parsed so far.
  Event properties can be changed, added and deleted by the plugin function by modifying this live event hash.

The parse_enrich function must return one of the following:

- 1: to indicate that it succeeded,
- 0 or undef: to indicate that event parsing should be aborted for this line and no event should be created (like the parser directive `ignore`),
- or any other string value as an error message (which will be logged).

Any changes made to the event properties are ignored unless the function completes successfully and returns 1. This also applies to a crashing `parse_enrich` function, or if is terminated because it ran over time.

Plugin Configuration

The configuration option `opevents_plugin_max_runtime` (default: 5 seconds) sets the maximum execution time for a single `parse_enrich` call. If a plugin's function runs longer than that it is terminated and an error message is logged; any changes that it may have made to the live event datastructure are ignored in this case.

Plugin Activation

A generic extensible parser rule can invoke one or more parser plugins using the action directive `plugin(PluginName)`.

When such a plugin call directive is encountered for the first time, the opEvents daemon loads (and caches) all available plugins that meet the requirements. To reread modified plugins, opeventsd must be restarted.

If a plugin named 'PluginName' is available, its `parse_enrich` function is executed and if successful, the modifications made by the function replace the event properties. Parsing then continues normally.

Command-line Event Creation

To provide a simple interface for external programs, opEvents also can create an event "on the fly" with event details from command-line arguments or a JSON file.

To create an event on the fly, you have to call `opeventsd.pl` with the argument `act=create-event`, which causes it to use all further key=value pairs in the arguments to construct an event, like this example:

```
opeventsd.pl act=create-event event=testevent node="somenode" details="this is just a test event"
action_required=1 action_checked=0 priority=4
```

Your event is expected to contain all required event properties and no further normalisation is performed. The option action_required should be set to 1 so that opEvents will process the event with Action Policies, or 0 to have opEvents not process with action policies.

Alternatively you can save your desired event's properties in a file in JSON format, and use `act=create-json` to instruct opeventsd to create an event from it:

```
opeventsd.pl act=create-json path=./myevent_in_format.json
```

Built-in Parsers

```
'rules' => {
 '1' => {
        event => 'Interface Down',
        regex => qr/LINEPROTO-5-UPDOWN:.+down/,
        stateful => 'Interface',
        priority => 1,
 },
 ...
 '10' => {
         regex => qr/Interface (\w+[\d\/\.]+)/,
         name => 'element',
 },
```

The key component is the `rules` section, which controls what details are extracted from a log entry and how they are saved a sevent properties. There are a few ways of augmenting the event with information:

- if both `regex` and `name` directives are present and if the regex matches and captures something from the log entry, then a named property (with name from the `name` directive) will be created, with the value being the captured content.
- The regex matching is performed on most of log input, but different across the various parsers:
  For parser types other than `nmis_eventlog` and `nmis_slavelog` the regex is applied to the event `details` property, which at this point holds most of the input log entry (usually everything except node and timestamp)
  For the event log parsers, the match is applied to:

    a. only to the event property named by the directive `variable` if that directive is present (e.g. `'variable' => 'node'`),
    b. or to the whole, unsplit input log entry.

- Parser types **except** `nmis_eventlog` and `nmis_slavelog`: if a rule block contains a `regex` directive which matches, then any key-value entries for `event`, `priority`, `state` or `stateful` in that rule block will be copied to the event as static properties.

(You might also encounter the deprecated legacy format of using directives `name` and `value` to set just one property to a fixed value.)

In the example above, rule 1 will be active if a "line protocol down" log entry is detected, and in that case it'll add properties "priority", "event", and "stateful", all with static values. Rule 10 will be active if the log entry contains "Interface <something>", and it'll copy over the matched <something> as the value of the property named "event".

All normalisation rules are checked in sequence of their numeric key, and all the ones whose `regex` directive matches will contribute to the new event's properties. Normalisation and enrichment then continues using information from NMIS; events are associated with the relevant nodes, stateful deduplication is performed etc.