Heka Documentation

Release 0.6.1

Mozilla

August 27, 2014

Contents

Heka is an open source stream processing software system developed by Mozilla. Heka is a "Swiss Army Knife" type tool for data processing, useful for a wide variety of different tasks, such as:

- Loading and parsing log files from a file system.
- Accepting statsd type metrics data for aggregation and forwarding to upstream time series data stores such as graphite or InfluxDB.
- Launching external processes to gather operational data from the local system.
- Performing real time analysis, graphing, and anomaly detection on any data flowing through the Heka pipeline.
- Shipping data from one location to another via the use of an external transport (such as AMQP) or directly (via TCP).
- Delivering processed data to one or more persistent data stores.

Heka is a heavily plugin based system. There are five different types of Heka plugins:

- *Inputs*

  Input plugins acquire data from the outside world and inject it into the Heka pipeline. They can do this by reading files from a file system, actively making network connections to acquire data from remote servers, listening on a network socket for external actors to push data in, launching processes on the local system to gather arbitrary data, or any other mechanism. They must be written in Go.

- *Decoders*

  Decoder plugins convert data that comes in through the Input plugins to Heka's internal Message data structure. Typically decoders are responsible for any parsing, deserializing, or extracting of structure from unstructured data that needs to happen. They can be written entirely in Go, or the core logic can be written in sandboxed Lua code.

- *Filters*

  Filter plugins are Heka's processing engines. They are configured to receive messages matching certain specific characteristics (using Heka's *Message Matcher Syntax*) and are able to perform arbitrary monitoring, aggregation, and/or processing of the data. Filters are also able to generate new messages that can be reinjected into the Heka pipeline, such as summary messages containing aggregate data, notification messages in cases where suspicious anomalies are detected, or circular buffer data messages that will show up as real time graphs in Heka's dashboard. Filters can be written entirely in Go, or the core logic can be written in sandboxed Lua code. It is also possible to configure Heka to allow Lua filters to be dynamically injected into a running Heka instance with needing to reconfigure or restart the Heka process, nor even to have shell access to the server on which Heka is running.

- *Encoders*

  Encoder plugins are the inverse of Decoders. They generate arbitrary byte streams using data extracted from Heka Message structs. Encoders are embedded within Output plugins; Encoders handle the serialization, Outputs handle the details of interacting with the outside world. They can be written entirely in Go, or the core logic can be written in sandboxed Lua code.

- *Outputs*

  Output plugins send data that has been serialized by an Encoder to some external destination. They handle all of the details of interacting with the network, filesystem, or any other outside resource. They are, like Filters, configured using Heka's *Message Matcher Syntax* so they will only receive and deliver messages matching certain characteristics. They must be written in Go.

Information about developing plugins in Go can be found in the *Extending Heka* section. Details about using Lua sandboxes for Decoder, Filter, and Encoder plugins can be found in the *Sandbox* section.

CHAPTER 1

hekad

The core of the Heka system is the *hekad* daemon. A single *hekad* process can be configured with any number of plugins, simultaneously performing a variety of data gathering, processing, and shipping tasks. Details on how to configure a *hekad* daemon are in the *Configuring hekad* section.

CHAPTER 2

hekad Command Line Options

-version Output the version number, then exit.

-config ***config_path*** Specify the configuration file or directory to use; the default is /etc/hekad.toml. If *config_path* resolves to a directory, all files in that directory must be valid TOML files. (See hekad.config(5).)

Contents:

2.1 Installing

2.1.1 Binaries

hekad releases are available on the Github project releases page. Binaries are available for Linux and OSX, with packages for Debian and RPM based distributions.

2.1.2 From Source

hekad requires a Go work environment to be setup for the binary to be built; this task is automated by the build process. The build script will override the Go environment for the shell window it is executed in. This creates an isolated environment that is intended specifically for building and developing Heka. The build script should be be run every time a new shell is opened for Heka development to ensure the correct dependencies are found and being used. To create a working *hekad* binary for your platform you'll need to install some prerequisites. Many of these are standard on modern Unix distributions and all are available for installation on Windows systems.

Prerequisites (all systems):

- CMake 2.8.7 or greater http://www.cmake.org/cmake/resources/software.html
- Git http://git-scm.com/download
- Go 1.3 or greater http://code.google.com/p/go/downloads/list
- Mercurial http://mercurial.selenic.com/downloads/
- Protobuf 2.3 or greater (optional - only needed if message.proto is modified) http://code.google.com/p/protobuf/downloads/list
- Sphinx (optional - used to generate the documentation) http://sphinx-doc.org/

Prerequisites (Unix):

- make

- gcc
- patch
- dpkg (optional)
- rpmbuild (optional)
- packagemaker (optional)

Prerequisites (Windows):

- MinGW http://sourceforge.net/projects/tdm-gcc/

Build Instructions

1. Check out the *heka* repository:

```
git clone https://github.com/mozilla-services/heka
```

2. Run *build* in the heka directory

```
cd heka
source build.sh # Unix (or `. build.sh`; must be sourced to properly setup the environment)
build.bat  # Windows
```

You will now have a *hekad* binary in the *build/heka/bin* directory.

3. (Optional) Run the tests to ensure a functioning *hekad*.

```
ctest             # All, see note
# Or use the makefile target
make test         # Unix
mingw32-make test # Windows
```

Note: In addition to the standard test build target, ctest can be called directly providing much greater control over the tests being run and the generated output (see ctest –help). i.e., 'ctest -R pi' will only run the pipeline unit test.

Clean Targets

- clean-heka - Use this target any time you change branches or pull from the Heka repository, it will ensure the Go workspace is in sync with the repository tree.
- clean - You will never want to use this target (it is autogenerated by cmake), it will cause all external dependencies to be re-fetched and re-built. The best way to 'clean-all' is to delete the build directory and re-run the build.(sh|bat) script.

Build Options

There are two build customization options that can be specified during the cmake generation process.

- INCLUDE_MOZSVC (bool) Include the Mozilla services plugins (default Unix: true, Windows: false).
- BENCHMARK (bool) Enable the benchmark tests (default false)

For example: to enable the benchmark tests in addition to the standard unit tests type 'cmake -DBENCHMARK=true ..' in the build directory.

2.1.3 Building *hekad* with External Plugins

It is possible to extend *hekad* by writing input, decoder, filter, or output plugins in Go (see *Extending Heka*). Because Go only supports static linking of Go code, your plugins must be included with and registered into Heka at compile time. The build process supports this through the use of an optional cmake file *{heka root}/cmake/plugin_loader.cmake*. A cmake function has been provided *add_external_plugin* taking the repository type (git, svn, or hg), repository URL, the repository tag to fetch, and an optional list of sub-packages to be initialized.

```
add_external_plugin(git https://github.com/mozilla-services/heka-mozsvc-plugins 6fe574dbd32a21f5
add_external_plugin(git https://github.com/example/path <tag> util filepath)
add_external_plugin(git https://github.com/bellycard/heka-sns-input :local)
# The ':local' tag is a special case, it copies {heka root}/externals/{plugin_name} into the Go
# work environment every time `make` is run. When local development is complete, and the source
# is checked in, the value can simply be changed to the correct tag to make it 'live'.
# i.e. {heka root}/externals/heka-sns-input -> {heka root}/build/heka/src/github.com/bellycard/h
```

The preceeding entry clones the *heka-mozsvc-plugins* git repository into the Go work environment, checks out SHA 6fe574dbd32a21f5d5583608a9d2339925edd2a7, and imports the package into *hekad* when *make* is run. By adding an init() function in your package you can make calls into *pipeline.RegisterPlugin* to register your plugins with Heka's configuration system.

2.1.4 Creating Packages

Installing packages on a system is generally the easiest way to deploy *hekad*. These packages can be easily created after following the above *From Source* directions:

1. Run *cpack* to build the appropriate package(s) for the current system:

```
cpack                # All
# Or use the makefile target
make package         # Unix (no deb, see below)
make deb             # Unix (if dpkg is available see below)
mingw32-make package # Windows
```

The packages will be created in the build directory.

Note: You will need *rpmbuild* installed to build the rpms.

See also:

Setting up an rpm-build environment

Note: For file name convention reasons, deb packages won't be created by running *cpack* or *make package*, even on a Unix machine w/ dpkg installed. Instead, running *source build.sh* on such a machine will generate a Makefile with a separate 'deb' target, so you can run *make deb* to generate the appropriate deb package.

2.2 Configuring hekad

A hekad configuration file specifies what inputs, decoders, filters, encoders, and outputs will be loaded. The configuration file is in TOML format. TOML looks very similar to INI configuration formats, but with slightly more rich data structures and nesting support.

If hekad's config file is specified to be a directory, all contained files with a filename ending in ".toml" will be loaded and merged into a single config. Files that don't end with ".toml" will be ignored. Merging will happen in alphabetical order, settings specified later in the merge sequence will win conflicts.

The config file is broken into sections, with each section representing a single instance of a plugin. The section name specifies the name of the plugin, and the "type" parameter specifies the plugin type; this must match one of the types registered via the *pipeline.RegisterPlugin* function. For example, the following section describes a plugin named "tcp:5565", an instance of Heka's plugin type "TcpInput":

```
[tcp:5565]
type = "TcpInput"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
address = ":5565"
```

If you choose a plugin name that also happens to be a plugin type name, then you can omit the "type" parameter from the section and the specified name will be used as the type. Thus, the following section describes a plugin named "TcpInput", also of type "TcpInput":

```
[TcpInput]
address = ":5566"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
```

Note that it's fine to have more than one instance of the same plugin type, as long as their configurations don't interfere with each other.

Any values other than "type" in a section, such as "address" in the above examples, will be passed through to the plugin for internal configuration (see *Plugin Configuration*).

If a plugin fails to load during startup, hekad will exit at startup. When hekad is running, if a plugin should fail (due to connection loss, inability to write a file, etc.) then hekad will either shut down or restart the plugin if the plugin supports restarting. When a plugin is restarting, hekad will likely stop accepting messages until the plugin resumes operation (this applies only to filters/output plugins).

Plugins specify that they support restarting by implementing the Restarting interface (see *restarting_plugins*). Plugins supporting Restarting can have *their restarting behavior configured*.

An internal diagnostic runner runs every 30 seconds to sweep the packs used for messages so that possible bugs in heka plugins can be reported and pinned down to a likely plugin(s) that failed to properly recycle the pack.

2.2.1 Global configuration options

You can optionally declare a *[hekad]* section in your configuration file to configure some global options for the heka daemon.

Config:

- **cpuprof (string *output_file*):** Turn on CPU profiling of hekad; output is logged to the *output_file*.
- **max_message_loops (uint):** The maximum number of times a message can be re-injected into the system. This is used to prevent infinite message loops from filter to filter; the default is 4.
- **max_process_inject (uint):** The maximum number of messages that a sandbox filter's ProcessMessage function can inject in a single call; the default is 1.
- **max_process_duration (uint64):** The maximum number of nanoseconds that a sandbox filter's ProcessMessage function can consume in a single call before being terminated; the default is 100000.
- **max_timer_inject (uint):** The maximum number of messages that a sandbox filter's TimerEvent function can inject in a single call; the default is 10.

- **max_pack_idle (string):** A time duration string (e.x. "2s", "2m", "2h") indicating how long a message pack can be 'idle' before its considered leaked by heka. If too many packs leak from a bug in a filter or output then heka will eventually halt. This setting indicates when that is considered to have occurred.
- **maxprocs (int):** Enable multi-core usage; the default is 1 core. More cores will generally increase message throughput. Best performance is usually attained by setting this to 2 x (number of cores). This assumes each core is hyper-threaded.
- **memprof (string *output_file*):** Enable memory profiling; output is logged to the *output_file*.
- **poolsize (int):** Specify the pool size of maximum messages that can exist; default is 100 which is usually sufficient and of optimal performance.
- **plugin_chansize (int):** Specify the buffer size for the input channel for the various Heka plugins. Defaults to 50, which is usually sufficient and of optimal performance.
- **base_dir (string):** Base working directory Heka will use for persistent storage through process and server restarts. The hekad process must have read and write access to this directory. Defaults to */var/cache/hekad* (or *c:varcachehekad* on Windows).
- **share_dir (string):** Root path of Heka's "share directory", where Heka will expect to find certain resources it needs to consume. The hekad process should have read- only access to this directory. Defaults to */usr/share/heka* (or *c:\usr\share\heka* on Windows).

New in version 0.6.

- **sample_denominator (int):** Specifies the denominator of the sample rate Heka will use when computing the time required to perform certain operations, such as for the ProtobufDecoder to decode a message, or the router to compare a message against a message matcher. Defaults to 1000, i.e. duration will be calculated for one message out of 1000.

New in version 0.6.

- **pid_file (string):** Optionally specify the location of a pidfile where the process id of the running hekad process will be written. The hekad process must have read and write access to the parent directory (which is not automatically created). On a successful exit the pidfile will be removed. If the path already exists the contained pid will be checked for a running process. If one is found, the current process will exit with an error.

2.2.2 Example hekad.toml file

```
[hekad]
maxprocs = 4

# Heka dashboard for internal metrics and time series graphs
[Dashboard]
type = "DashboardOutput"
address = ":4352"
ticker_interval = 15

# Email alerting for anomaly detection
[Alert]
type = "SmtpOutput"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert'"
send_from = "acme-alert@example.com"
send_to = ["admin@example.com"]
auth = "Plain"
user = "smtp-user"
password = "smtp-pass"
```

```
host = "mail.example.com:25"
encoder = "AlertEncoder"

# User friendly formatting of alert messages
[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"

# Nginx access log reader
[AcmeWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedNginxDecoder"

# Nginx access 'combined' log parser
[CombinedNginxDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [CombinedNginxDecoder.config]
    user_agent_transform = true
    user_agent_conditional = true
    type = "combined"
    log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$htt

# Collection and visualization of the HTTP status codes
[AcmeHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'AcmeWebserver'"

    # rate of change anomaly detection on column 1 (HTTP 200)
    [AcmeHTTPStatus.config]
    anomaly_config = 'roc("HTTP Status", 1, 15, 0, 1.5, true, false)'
```

2.2.3 Configuring Restarting Behavior

Plugins that support being restarted have a set of options that govern how the restart is handled. If preferred, the plugin can be configured to not restart at which point hekad will exit, or it could be restarted only 100 times, or restart attempts can proceed forever.

Adding the restarting configuration is done by adding a config section to the plugins' config called *retries*. A small amount of jitter will be added to the delay between restart attempts.

Config:

- **max_jitter (string):** The longest jitter duration to add to the delay between restarts. Jitter up to 500ms by default is added to every delay to ensure more even restart attempts over time.
- **max_delay (string):** The longest delay between attempts to restart the plugin. Defaults to 30s (30 seconds).
- **delay (string):** The starting delay between restart attempts. This value will be the initial starting delay for the exponential back-off, and capped to be no larger than the *max_delay*. Defaults to 250ms.
- **max_retries (int):** Maximum amount of times to attempt restarting the plugin before giving up and shutting

down hekad. Use 0 for no retry attempt, and -1 to continue trying forever (note that this will cause hekad to halt possibly forever if the plugin cannot be restarted).

Example (UdpInput does not actually support nor need restarting, illustrative purposes only):

```
[UdpInput]
address = "127.0.0.1:4880"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

[UdpInput.retries]
max_delay = 30s
delay = 250ms
max_retries = 5
```

2.3 Inputs

2.3.1 AMQPInput

Connects to a remote AMQP broker (RabbitMQ) and retrieves messages from the specified queue. As AMQP is dynamically programmable, the broker topology needs to be specified in the plugin configuration.

Config:

- **URL (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **Exchange (string):** AMQP exchange name
- **ExchangeType (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **ExchangeDurability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **ExchangeAutoDelete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.
- **RoutingKey (string):** The message routing key used to bind the queue to the exchange. Defaults to empty string.
- **PrefetchCount (int):** How many messages to fetch at once before message acks are sent. See RabbitMQ performance measurements for help in tuning this number. Defaults to 2.
- **Queue (string):** Name of the queue to consume from, an empty string will have the broker generate a name for the queue. Defaults to empty string.
- **QueueDurability (bool):** Whether the queue is durable or not. Defaults to non-durable.
- **QueueExclusive (bool):** Whether the queue is exclusive (only one consumer allowed) or not. Defaults to non-exclusive.
- **QueueAutoDelete (bool):** Whether the queue is deleted when the last consumer un-subscribes. Defaults to auto-delete.
- **QueueTTL (int):** Allows ability to specify TTL in milliseconds on Queue declaration for expiring messages. Defaults to undefined/infinite.
- **Decoder (string):** Decoder name used to transform a raw message body into a structured hekad message. Must be a decoder appropriate for the messages that come in from the exchange. If accepting messages that have been generated by an AMQPOutput in another Heka process then this should be a *ProtobufDecoder* instance.

New in version 0.6.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *AMQPS* URI scheme. See *Configuring TLS*.

Since many of these parameters have sane defaults, a minimal configuration to consume serialized messages would look like:

```
[AMQPInput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
```

Or you might use a PayloadRegexDecoder to parse OSX syslog messages with the following:

```
[AMQPInput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
decoder = "logparser"

[logparser]
type = "MultiDecoder"
subs = ["logline", "leftovers"]

[logline]
type = "PayloadRegexDecoder"
MatchRegex = '\w+ \d+ \d+:\d+:\d+ \S+ (?P<Reporter>[^\[]+)\[(?P<Pid>\d+)](?P<Sandbox>[^:]+)?: (?P Rem

    [logline.MessageFields]
    Type = "amqplogline"
    Hostname = "myhost"
    Reporter = "%Reporter%"
    Remaining = "%Remaining%"
    Logger = "%Logger%"
    Payload = "%Remaining%"

[leftovers]
type = "PayloadRegexDecoder"
MatchRegex = '.*'

    [leftovers.MessageFields]
    Type = "drop"
    Payload = ""
```

2.3.2 HttpInput

HttpInput plugins intermittently poll remote HTTP URLs for data and populate message objects based on the results of the HTTP interactions. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time HTTP request is completed.
- **Type: *heka.httpinput.data* or *heka.httpinput.error* depending on whether or** not the request completed. (Note that a response returned with an HTTP error code is still considered complete and will generate type *heka.httpinput.data*.)
- Hostname: Hostname of the machine on which Heka is running.

- Payload: Entire contents of the HTTP response body.
- **Severity: HTTP response 200 uses *success_severity* config value, all other** results use *error_severity* config value.
- Logger: Fetched URL.
- Fields["Status"] (string): HTTP status string value (e.g. "200 OK").
- Fields["StatusCode"] (int): HTTP status code integer value.
- Fields["ResponseSize"] (int): Value of HTTP Content-Length header.
- **Fields["ResponseTime"] (float64): Clock time elapsed for HTTP request, in** seconds.
- **Fields["Protocol"] (string): HTTP protocol used for the request (e.g.** "HTTP/1.0")

The *Fields* values above will only be populated in the event of a completed HTTP request. Also, it is possible to specify a decoder to further process the results of the HTTP response before injecting the message into the router.

Config:

- **url (string):** A HTTP URL which this plugin will regularly poll for data. This option cannot be used with the urls option. No default URL is specified.
- **urls (array):** New in version 0.5.

  An array of HTTP URLs which this plugin will regularly poll for data. This option cannot be used with the url option. No default URLs are specified.
- **method (string):** New in version 0.5.

  The HTTP method to use for the request. Defaults to "GET".
- **headers (subsection):** New in version 0.5.

  Subsection defining headers for the request. By default the User-Agent header is set to "Heka"
- **body (string):** New in version 0.5.

  The request body (e.g. for an HTTP POST request). No default body is specified.
- **username (string):** New in version 0.5.

  The username for HTTP Basic Authentication. No default username is specified.
- **password (string):** New in version 0.5.

  The password for HTTP Basic Authentication. No default password is specified.
- **ticker_interval (uint):** Time interval (in seconds) between attempts to poll for new data. Defaults to 10.
- **success_severity (uint):** New in version 0.5.

  Severity level of successful HTTP request. Defaults to 6 (information).
- **error_severity (uint):** New in version 0.5.

  Severity level of errors, unreachable connections, and non-200 responses of successful HTTP requests. Defaults to 1 (alert).
- **decoder (string):** The name of the decoder used to further transform the response body text into a structured hekad message. No default decoder is specified.

Example:

```
[HttpInput]
url = "http://localhost:9876/"
ticker_interval = 5
success_severity = 6
error_severity = 1
decoder = "MyCustomJsonDecoder"
    [HttpInput.headers]
    user-agent = "MyCustomUserAgent"
```

2.3.3 HttpListenInput

New in version 0.5.

HttpListenInput plugins start a webserver listening on the specified address and port. If no decoder is specified data in the request body will be populated as the message payload. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time HTTP request is handled.
- Type: *heka.httpdata.request*
- Hostname: The remote network address of requester.
- Payload: Entire contents of the HTTP response body.
- Severity: 6
- Logger: HttpListenInput
- Fields["UserAgent"] (string): Request User-Agent header (e.g. “GitHub Hookshot dd0772a”).
- Fields["ContentType"] (string): Request Content-Type header (e.g. “application/x-www-form-urlencoded”).
- **Fields["Protocol"] (string): HTTP protocol used for the request (e.g.** “HTTP/1.0”)

Config:

- **address (string):** An IP address:port on which this plugin will expose a HTTP server. Defaults to “127.0.0.1:8325”.
- **decoder (string):** The name of the decoder used to further transform the request body text into a structured hekad message. No default decoder is specified.

Example:

```
[HttpListenInput]
address = "0.0.0.0:8325"
```

2.3.4 Logstreamer Input

New in version 0.5.

Tails a single log file, a sequential single log source, or multiple log sources of either a single logstream or multiple logstreams.

See also:

Complete documentation with examples

Config:

- **hostname (string):** The hostname to use for the messages, by default this will be the machine's qualified hostname. This can be set explicitly to ensure it's the correct name in the event the machine has multiple interfaces/hostnames.
- **oldest_duration (string):** A time duration string (e.x. "2s", "2m", "2h"). Logfiles with a last modified time older than `oldest_duration` ago will not be included for parsing.
- **journal_directory (string):** The directory to store the journal files in for tracking the location that has been read to thus far. By default this is stored under heka's base directory.
- **log_directory (string):** The root directory to scan files from. This scan is recursive so it should be suitably restricted to the most specific directory this selection of logfiles will be matched under. The log_directory path will be prepended to the file_match.
- **rescan_interval (int):** During logfile rotation, or if the logfile is not originally present on the system, this interval is how often the existence of the logfile will be checked for. The default of 5 seconds is usually fine. This interval is in milliseconds.
- **file_match (string):** Regular expression used to match files located under the `log_directory`. This regular expression has `$` added to the end automatically if not already present, and `log_directory` as the prefix. WARNING: file_match should typically be delimited with single quotes, indicating use of a raw string, rather than double quotes, which require all backslashes to be escaped. For example, *'access\.log'* will work as expected, but *"access\.log"* will not, you would need *"access\\.log"* to achieve the same result.
- **priority (list of strings):** When using sequential logstreams, the priority is how to sort the logfiles in order from oldest to newest.
- **differentiator (list of strings):** When using multiple logstreams, the differentiator is a set of strings that will be used in the naming of the logger, and portions that match a captured group from the `file_match` will have their matched value substituted in.
- **translation (hash map of hash maps of ints):** A set of translation mappings for matched groupings to the ints to use for sorting purposes.
- **decoder (string):** A *ProtobufDecoder* instance must be specified for the message.proto parser. Use of a decoder is optional for token and regexp parsers; if no decoder is specified the parsed data is available in the Heka message payload.
- **parser_type (string):**
  - token - splits the log on a byte delimiter (default).
  - regexp - splits the log on a regexp delimiter.
  - message.proto - splits the log on protobuf message boundaries
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the log line depending on the delimiter_location configuration. Note: when a start delimiter is used the last line in the file will not be processed (since the next record defines its end) until the log is rolled.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of a log line.
  - end - the regexp delimiter occurs at the end of the log line (default).

2.3.5 ProcessInput

Executes one or more external programs on an interval, creating messages from the output. Supports a chain of commands, where stdout from each process will be piped into the stdin for the next process in the chain. In the event the program returns a non-zero exit code, ProcessInput will stop, logging the exit error.

Config:

- **command (map[uint]cmd_config):** The command is a structure that contains the full path to the binary, command line arguments, optional enviroment variables and an optional working directory (see below). ProcessInput expects the commands to be indexed by integers starting with 0, where 0 is the first process in the chain.
- **ticker_interval (uint):** The number of seconds to wait between each run of *command*. Defaults to 15. A ticker_interval of 0 indicates that the command is run only once, useful for long running processes.
- **stdout (bool):** If true, for each run of the process chain a message will be generated with the last command in the chain's stdout as the payload. Defaults to true.
- **stderr (bool):** If true, for each run of the process chain a message will be generated with the last command in the chain's stderr as the payload. Defaults to false.
- **decoder (string):** Name of the decoder instance to send messages to. If omitted messages will be injected directly into Heka's message router.
- **parser_type (string):**
  - token - splits the log on a byte delimiter (default).
  - regexp - splits the log on a regexp delimiter.
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the log line depending on the delimiter_location configuration. Note: when a start delimiter is used the last line in the file will not be processed (since the next record defines its end) until the log is rolled.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of a log line.
  - end - the regexp delimiter occurs at the end of the log line (default).
- **timeout (uint):** Timeout in seconds before any one of the commands in the chain is terminated.
- **trim (bool) :** Trim a single trailing newline character if one exists. Default is true.

cmd_config structure:

- **bin (string):** The full path to the binary that will be executed.
- **args ([]string):** Command line arguments to pass into the executable.
- **environment ([]string):** Used to set environment variables before *command* is run. Default is nil, which uses the heka process's environment.
- **directory (string):** Used to set the working directory of *Bin* Default is "", which uses the heka process's working directory.

Example:

```
[DemoProcessInput]
type = "ProcessInput"
ticker_interval = 2
parser_type = "token"
```

```
delimiter = " "
stdout = true
stderr = false
trim = true

    [DemoProcessInput.command.0]
    bin = "/bin/cat"
    args = ["../testsupport/process_input_pipes_test.txt"]

    [DemoProcessInput.command.1]
    bin = "/usr/bin/grep"
    args = ["ignore"]
```

2.3.6 ProcessDirectoryInput

New in version 0.5.

The ProcessDirectoryInput periodically scans a filesystem directory looking for ProcessInput configuration files. The ProcessDirectoryInput will maintain a pool of running ProcessInputs based on the contents of this directory, refreshing the set of running inputs as needed with every rescan. This allows Heka administrators to manage a set of data collection processes for a running hekad server without restarting the server.

Each ProcessDirectoryInput has a *process_dir* configuration setting, which is the root folder of the tree where scheduled jobs are defined. It should contain exactly one nested level of subfolders, named with ASCII numeric characters indicating the interval, in seconds, between each process run. These numeric folders must contain TOML files which specify the details regarding which processes to run.

For example, a process_dir might look like this:

```
-/usr/share/heka/processes/
 |-5
   |- check_myserver_running.toml
 |-61
   |- cat_proc_mounts.toml
   |- get_running_processes.toml
 |-302
   |- some_custom_query.toml
```

This indicates one process to be run every five seconds, two processes to be run every 61 seconds, and one process to be run every 302 seconds.

Note that ProcessDirectory will ignore any files that are not nested one level deep, are not in a folder named for an integer 0 or greater, and do not end with '.toml'. Each file which meets these criteria, such as those shown in the example above, should contain the TOML configuration for exactly one *ProcessInput*, matching that of a standalone ProcessInput with the following restrictions:

- The section name *must* be *ProcessInput*. Any TOML sections named anything other than ProcessInput will be ignored.
- Any specified *ticker_interval* value will be *ignored*. The ticker interval value to use will be parsed from the directory path.

If the specified process fails to run or the ProcessInput config fails for any other reason, ProcessDirectoryInput will log an error message and continue.

Config:

- **ticker_interval (int, optional):** Amount of time, in seconds, between scans of the process_dir. Defaults to 300 (i.e. 5 minutes).

- **process_dir (string, optional):** This is the root folder of the tree where the scheduled jobs are defined. Absolute paths will be honored, relative paths will be computed relative to Heka's globally specified share_dir. Defaults to "processes" (i.e. "$share_dir/processes").

Example:

```
[ProcessDirectoryInput]
process_dir = "/etc/hekad/processes.d"
ticker_interval = 120
```

2.3.7 StatAccumInput

Provides an implementation of the *StatAccumulator* interface which other plugins can use to submit *Stat* objects for aggregation and roll-up. Accumulates these stats and then periodically emits a "stat metric" type message containing aggregated information about the stats received since the last generated message.

Config:

- **emit_in_payload (bool):** Specifies whether or not the aggregated stat information should be emitted in the payload of the generated messages, in the format accepted by the carbon portion of the graphite graphing software. Defaults to true.
- **emit_in_fields (bool):** Specifies whether or not the aggregated stat information should be emitted in the message fields of the generated messages. Defaults to false. *NOTE*: At least one of 'emit_in_payload' or 'emit_in_fields' *must* be true or it will be considered a configuration error and the input won't start.
- **percent_threshold (int):** Percent threshold to use for computing "upper_N%" type stat values. Defaults to 90.
- **ticker_interval (uint):** Time interval (in seconds) between generated output messages. Defaults to 10.
- **message_type (string):** String value to use for the *Type* value of the emitted stat messages. Defaults to "heka.statmetric".
- **legacy_namespaces (bool):** If set to true, then use the older format for namespacing counter stats, with rates recorded under *stats.<counter_name>* and absolute count recorded under *stats_counts.<counter_name>*. See statsd metric namespacing. Defaults to false.
- **global_prefix (string):** Global prefix to use for sending stats to graphite. Defaults to "stats".
- **counter_prefix (string):** Secondary prefix to use for namespacing counter metrics. Has no impact unless *legacy_namespaces* is set to false. Defaults to "counters".
- **timer_prefix (string):** Secondary prefix to use for namespacing timer metrics. Defaults to "timers".
- **gauge_prefix (string):** Secondary prefix to use for namespacing gauge metrics. Defaults to "gauges".
- **statsd_prefix (string):** Prefix to use for the statsd *numStats* metric. Defaults to "statsd".
- **delete_idle_stats (bool):** Don't emit values for inactive stats instead of sending 0 or in the case of gauges, sending the previous value. Defaults to false.

2.3.8 StatsdInput

Listens for statsd protocol *counter*, *timer*, or *gauge* messages on a UDP port, and generates *Stat* objects that are handed to a *StatAccumulator* for aggregation and processing.

Config:

- **address (string):** An IP address:port on which this plugin will expose a statsd server. Defaults to "127.0.0.1:8125".

- **stat_accum_name (string):** Name of a StatAccumInput instance that this StatsdInput will use as its StatAccumulator for submitting received stat values. Defaults to "StatAccumInput".

Example:

```
[StatsdInput]
address = ":8125"
stat_accum_input = "custom_stat_accumulator"
```

2.3.9 TcpInput

Listens on a specific TCP address and port for messages. If the message is signed it is verified against the signer name and specified key version. If the signature is not valid the message is discarded otherwise the signer name is added to the pipeline pack and can be use to accept messages using the message_signer configuration option.

Config:

- **address (string):** An IP address:port on which this plugin will listen.
- **signer:** Optional TOML subsection. Section name consists of a signer name, underscore, and numeric version of the key.
  - **hmac_key (string):** The hash key used to sign the message.

New in version 0.4.

- **decoder (string):** A *ProtobufDecoder* instance must be specified for the message.proto parser. Use of a decoder is optional for token and regexp parsers; if no decoder is specified the raw input data is available in the Heka message payload.
- **parser_type (string):**
  - token - splits the stream on a byte delimiter.
  - regexp - splits the stream on a regexp delimiter.
  - message.proto - splits the stream on protobuf message boundaries.
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the message depending on the delimiter_location configuration.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of the message.
  - end - the regexp delimiter occurs at the end of the message (default).

New in version 0.5.

- **use_tls (bool):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.
- **tls (TlsConfig):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.
- **net (string, optional, default: "tcp")** Network value must be one of: "tcp", "tcp4", "tcp6", "unix" or "unixpacket".

Example:

```
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

[TcpInput.signer.ops_0]
hmac_key = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
[TcpInput.signer.ops_1]
hmac_key = "xdd908lfcgikauexdi8elogusridaxoalf"

[TcpInput.signer.dev_1]
hmac_key = "haeoufyaiofeugdsnzaogpi.ua,dp.804u"
```

2.3.10 UdpInput

Listens on a specific UDP address and port for messages. If the message is signed it is verified against the signer name and specified key version. If the signature is not valid the message is discarded otherwise the signer name is added to the pipeline pack and can be use to accept messages using the message_signer configuration option.

Note: The UDP payload is not restricted to a single message; since the stream parser is being used multiple messages can be sent in a single payload.

Config:

- **address (string):** An IP address:port or Unix datagram socket file path on which this plugin will listen.
- **signer:** Optional TOML subsection. Section name consists of a signer name, underscore, and numeric version of the key.
  - **hmac_key (string):** The hash key used to sign the message.

New in version 0.4.

- **decoder (string):** A *ProtobufDecoder* instance must be specified for the message.proto parser. Use of a decoder is optional for token and regexp parsers; if no decoder is specified the raw input data is available in the Heka message payload.
- **parser_type (string):**
  - token - splits the stream on a byte delimiter.
  - regexp - splits the stream on a regexp delimiter.
  - message.proto - splits the stream on protobuf message boundaries.
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the message depending on the delimiter_location configuration.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of the message.
  - end - the regexp delimiter occurs at the end of the message (default).

New in version 0.5.

- **net (string, optional, default: "udp")** Network value must be one of: "udp", "udp4", "udp6", or "unixgram".

Example:

```
[UdpInput]
address = "127.0.0.1:4880"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

[UdpInput.signer.ops_0]
hmac_key = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
[UdpInput.signer.ops_1]
hmac_key = "xdd908lfcgikauexdi8elogusridaxoalf"

[UdpInput.signer.dev_1]
hmac_key = "haeoufyaiofeugdsnzaogpi.ua,dp.804u"
```

2.4 Decoders

2.4.1 Apache Access Log Decoder

New in version 0.6.

Parses the Apache access logs based on the Apache 'LogFormat' configuration directive. The Apache format specifiers are mapped onto the Nginx variable names where applicable e.g. %a -> remote_addr. This allows generic web filters and outputs to work with any HTTP server input.

Config:

- **log_format (string)** The 'LogFormat' configuration directive from the apache2.conf. %t variables are converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://httpd.apache.org/docs/2.4/mod/mod_log_config.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/apache"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/apache_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
```

```
log_format = '%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# common log format
# log_format = '%h %l %u %t \"%r\" %>s %O'

# vhost_combined log format
# log_format = '%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# referer log format
# log_format = '%{Referer}i -> %U'
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

New in version 0.6.

2.4.2 GeoIpDecoder

Decoder plugin that generates GeoIP data based on the IP address of a specified field. It uses the Go project: https://github.com/abh/geoip as a wrapper around MaxMind's geoip-api-c library. This decoder assumes you have downloaded and installed the geoip-api-c library from MaxMind's website. Currently, only the GeoLiteCity database is supported, which you must also download and install yourself into a location to be referenced by the db_file config option. By default the database file is opened using "GEOIP_MEMORY_CACHE" mode. This setting is hard-coded into the wrapper's geoip.go file. You will need to manually override that code if you want to specify one of the other modes listed here:

Note: If you are using this with the ES output you will likely need to specify the raw_bytes_field option for the target_field specified. This is required to preserve the formatting of the JSON object.

Config:

- **db_file:** The location of the GeoLiteCity.dat database. Defaults to "/var/cache/hekad/GeoLiteCity.dat"
- **source_ip_field:** The name of the field containing the IP address you want to derive the location for.
- **target_field:** The name of the new field created by the decoder. The decoder will output a JSON object with the following elements:
  - latitute: string,
  - longitude: string,
  - **location: [float64, float64],**
    - GeoJSON format intended for use as a geo_point for ES output. Useful when using Kibana's Bettermap panel
  - coordinates: [string, string],
  - countrycode: string,
  - countrycode3: string,
  - region: string,
  - city: string,
  - postalcode: string,
  - areacode: int,
  - charset: int,
  - continentalcode: string

```
[apache_geoip_decoder]
type = "GeoIpDecoder"
db_file="/etc/geoip/GeoLiteCity.dat"
source_ip_field="remote_host"
target_field="geoip"
```

2.4.3 MultiDecoder

This decoder plugin allows you to specify an ordered list of delegate decoders. The MultiDecoder will pass the PipelinePack to be decoded to each of the delegate decoders in turn until decode succeeds. In the case of failure to decode, MultiDecoder will return an error and recycle the message.

Config:

- **subs ([]string):** An ordered list of subdecoders to which the MultiDecoder will delegate. Each item in the list should specify another decoder configuration section by section name. Must contain at least one entry.
- **log_sub_errors (bool):** If true, the DecoderRunner will log the errors returned whenever a delegate decoder fails to decode a message. Defaults to false.
- **cascade_strategy (string):** Specifies behavior the MultiDecoder should exhibit with regard to cascading through the listed decoders. Supports only two valid values: "first-wins" and "all". With "first-wins", each decoder will be tried in turn until there is a successful decoding, after which decoding will be stopped. With "all", all listed decoders will be applied whether or not they succeed. In each case, decoding will only be considered to have failed if *none* of the sub-decoders succeed.

Here is a slightly contrived example where we have protocol buffer encoded messages coming in over a TCP connection, with each message containin a single nginx log line. Our MultiDecoder will run each message through two decoders, the first to deserialize the protocol buffer and the second to parse the log text:

```
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "shipped-nginx-decoder"

[shipped-nginx-decoder]
type = "MultiDecoder"
subs = ['ProtobufDecoder', 'nginx-access-decoder']
cascade_strategy = "all"
log_sub_errors = true

[ProtobufDecoder]

[nginx-access-decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [nginx-access-decoder.config]
    type = "combined"
    user_agent_transform = true
    log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$htt
```

2.4.4 Nginx Access Log Decoder

New in version 0.5.

Parses the Nginx access logs based on the Nginx 'log_format' configuration directive.

Config:

- **log_format (string)** The 'log_format' configuration directive from the nginx.conf. $time_local or $time_iso8601 variable is converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://nginx.org/en/docs/http/ngx_http_log_module.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
```

```
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

2.4.5 MySQL Slow Query Log Decoder

New in version 0.6.

Parses and transforms the MySQL slow query logs. Use mariadb_slow_query.lua to parse the MariaDB variant of the MySQL slow query logs.

Config:

- **truncate_sql (int, optional, default nil)** Truncates the SQL payload to the specified number of bytes (not UTF-8 aware) and appends "...". If the value is nil no truncation is performed. A negative value will truncate the specified number of bytes from the end.

Example Heka Configuration

```
[Sync-1_5-SlowQuery]
type = "LogstreamerInput"
log_directory = "/var/log/mysql"
```

```
file_match = 'mysql-slow\.log'
parser_type = "regexp"
delimiter = "\n(# User@Host:)"
delimiter_location = "start"
decoder = "MySqlSlowQueryDecoder"

[MySqlSlowQueryDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/mysql_slow_query.lua"

    [MySqlSlowQueryDecoder.config]
    truncate_sql = 64
```

Example Heka Message

Timestamp 2014-05-07 15:51:28 -0700 PDT

Type mysql.slow-query

Hostname 127.0.0.1

Pid 0

UUID 5324dd93-47df-485b-a88e-429f0fcd57d6

Logger Sync-1_5-SlowQuery

Payload /* [queryName=FIND_ITEMS] */ SELECT bso.userid, bso.collection, ...

EnvVersion

Severity 7

Fields

name:"Rows_examined" value_type:DOUBLE value_double:16458
name:"Query_time" value_type:DOUBLE representation:"s" value_double:7.24966
name:"Rows_sent" value_type:DOUBLE value_double:5001
name:"Lock_time" value_type:DOUBLE representation:"s" value_double:0.047038

2.4.6 Nginx Error Log Decoder

New in version 0.6.

Parses the Nginx error logs based on the Nginx hard coded internal format.

Config:

- **tz (string, optional, defaults to UTC)** The conversion actually happens on the Go side since there isn't good TZ support here.

Example Heka Configuration

```
[TestWebserverError]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'error\.log'
decoder = "NginxErrorDecoder"

[NginxErrorDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_error.lua"
```

```
[NginxErrorDecoder.config]
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type nginx.error

Hostname trink-x230

Pid 16842

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserverError

Payload using inherited sockets from "6;"

EnvVersion

Severity 5

Fields

name:"tid" value_type:DOUBLE value_double:0
name:"connection" value_type:DOUBLE value_double:8878

2.4.7 PayloadRegexDecoder

Decoder plugin that accepts messages of a specified form and generates new outgoing messages from extracted data, effectively transforming one message format into another.

Note: The Go regular expression tester is an invaluable tool for constructing and debugging regular expressions to be used for parsing your input data.

Config:

- **match_regex:** Regular expression that must match for the decoder to process the message.
- **severity_map:** Subsection defining severity strings and the numerical value they should be translated to. hekad uses numerical severity codes, so a severity of *WARNING* can be translated to *3* by settings in this section. See *Heka Message*.
- **message_fields:** Subsection defining message fields to populate and the interpolated values that should be used. Valid interpolated values are any captured in a regex in the message_matcher, and any other field that exists in the message. In the event that a captured name overlaps with a message field, the captured name's value will be used. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. ResponseSize|B = "%ResponseSize%" will create Fields[ResponseSize] representing the number of bytes. Adding a representation string to a standard message header name will cause it to be added as a user defined field i.e., Payload|json will create Fields[Payload] with a json representation (see *Field Variables*).

  Interpolated values should be surrounded with % signs, for example:

  ```
  [my_decoder.message_fields]
  Type = "%Type%Decoded"
  ```

  This will result in the new message's Type being set to the old messages Type with *Decoded* appended.

- **timestamp_layout (string):** A formatting string instructing hekad how to turn a time string into the actual time representation used internally. Example timestamp layouts can be seen in Go's time documentation.
- **timestamp_location (string):** Time zone in which the timestamps in the text are presumed to be in. Should be a location name corresponding to a file in the IANA Time Zone database (e.g. "America/Los_Angeles"), as parsed by Go's *time.LoadLocation()* function (see http://golang.org/pkg/time/#LoadLocation). Defaults to "UTC". Not required if valid time zone info is embedded in every parsed timestamp, since those can be parsed as specified in the *timestamp_layout*.
- **log_errors (bool):** New in version 0.5.

  If set to false, payloads that can not be matched against the regex will not be logged as errors. Defaults to true.

Example (Parsing Apache Combined Log Format):

```
[apache_transform_decoder]
type = "PayloadRegexDecoder"
match_regex = '^(?P<RemoteIP>\S+) \S+ \S+ \[(?P<Timestamp>[^\]]+)\] "(?P<Method>[A-Z]+) (?P<Url>[^\s
timestamp_layout = "02/Jan/2006:15:04:05 -0700"

# severities in this case would work only if a (?P<Severity>...) matching
# group was present in the regex, and the log file contained this information.
[apache_transform_decoder.severity_map]
DEBUG = 7
INFO = 6
WARNING = 4

[apache_transform_decoder.message_fields]
Type = "ApacheLogfile"
Logger = "apache"
Url|uri = "%Url%"
Method = "%Method%"
Status = "%Status%"
RequestSize|B = "%RequestSize%"
Referer = "%Referer%"
Browser = "%Browser%"
```

2.4.8 PayloadXmlDecoder

This decoder plugin accepts XML blobs in the message payload and allows you to map parts of the XML into Field attributes of the pipeline pack message using XPath syntax using the xmlpath library.

Config:

- **xpath_map:** A subsection defining a capture name that maps to an XPath expression. Each expression can fetch a single value, if the expression does not resolve to a valid node in the XML blob, the capture group will be assigned an empty string value.
- **severity_map:** Subsection defining severity strings and the numerical value they should be translated to. hekad uses numerical severity codes, so a severity of *WARNING* can be translated to *3* by settings in this section. See *Heka Message*.
- **message_fields:** Subsection defining message fields to populate and the interpolated values that should be used. Valid interpolated values are any captured in an XPath in the message_matcher, and any other field that exists in the message. In the event that a captured name overlaps with a message field, the captured name's value will be used. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. ResponseSize|B = "%ResponseSize%" will create Fields[ResponseSize] representing the number of bytes. Adding a representation string to a standard message header name will cause it to be

added as a user defined field i.e., Payload|json will create Fields[Payload] with a json representation (see *Field Variables*).

Interpolated values should be surrounded with % signs, for example:

```
[my_decoder.message_fields]
Type = "%Type%Decoded"
```

This will result in the new message's Type being set to the old messages Type with *Decoded* appended.

- **timestamp_layout (string):** A formatting string instructing hekad how to turn a time string into the actual time representation used internally. Example timestamp layouts can be seen in Go's time documentation. The default layout is ISO8601 - the same as Javascript.
- **timestamp_location (string):** Time zone in which the timestamps in the text are presumed to be in. Should be a location name corresponding to a file in the IANA Time Zone database (e.g. "America/Los_Angeles"), as parsed by Go's *time.LoadLocation()* function (see http://golang.org/pkg/time/#LoadLocation). Defaults to "UTC". Not required if valid time zone info is embedded in every parsed timestamp, since those can be parsed as specified in the *timestamp_layout*.

Example:

```
[myxml_decoder]
type = "PayloadXmlDecoder"

[myxml_decoder.xpath_map]
Count = "/some/path/count"
Name = "/some/path/name"
Pid = "//pid"
Timestamp = "//timestamp"
Severity = "//severity"

[myxml_decoder.severity_map]
DEBUG = 7
INFO = 6
WARNING = 4

[myxml_decoder.message_fields]
Pid = "%Pid%"
StatCount = "%Count%"
StatName =  "%Name%"
Timestamp = "%Timestamp%"
```

PayloadXmlDecoder's xpath_map config subsection supports XPath as implemented by the xmlpath library.

- All axes are supported ("child", "following-sibling", etc)
- All abbreviated forms are supported (".", "//", etc)
- All node types except for namespace are supported
- Predicates are restricted to [N], [path], and [path=literal] forms
- Only a single predicate is supported per path step
- Richer expressions and namespaces are not supported

2.4.9 ProtobufDecoder

The ProtobufDecoder is used for Heka message objects that have been serialized into protocol buffers format. This is the format that Heka uses to communicate with other Heka instances, so one will always be included in your Heka configuration whether specified or not. The ProtobufDecoder has no configuration options.

The hekad protocol buffers message schema in defined in the *message.proto* file in the *message* package.

Example:

```
[ProtobufDecoder]
```

See also:

Protocol Buffers - Google's data interchange format

2.4.10 Rsyslog Decoder

New in version 0.5.

Parses the rsyslog output using the string based configuration template.

Config:

- **template (string)** The 'template' configuration string from rsyslog.conf. http://rsyslog-5-8-6-doc.neocities.org/rsyslog_conf_templates.html
- **tz (string, optional, defaults to UTC)** If your rsyslog timestamp field in the template does not carry zone offset information, you may set an offset to be applied to your events here. Typically this would be used with the "Traditional" rsyslog formats.

  Parsing is done by Go, supports values of "UTC", "Local", or a location name corresponding to a file in the IANA Time Zone database, e.g. "America/New_York".

Example Heka Configuration

```
[RsyslogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/rsyslog.lua"

[RsyslogDecoder.config]
type = "RSYSLOG_TraditionalFileFormat"
template = '%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n'
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-02-10 12:58:58 -0800 PST

Type RSYSLOG_TraditionalFileFormat

Hostname trink-x230

Pid 0

UUID e0eef205-0b64-41e8-a307-5772b05e16c1

Logger RsyslogInput

Payload "imklog 5.8.6, log source = /proc/kmsg started."

EnvVersion

Severity 7

Fields

name:"programname" value_string:"kernel"]

2.4.11 SandboxDecoder

The SandboxDecoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka. See *Sandbox*. Config:

- *config_common_sandbox_parameters*

Example

```
[sql_decoder]
type = "SandboxDecoder"
filename = "sql_decoder.lua"
```

2.4.12 ScribbleDecoder

New in version 0.5.

The ScribbleDecoder is a trivial decoder that makes it possible to set one or more static field values on every decoded message. It is often used in conjunction with another decoder (i.e. in a MultiDecoder w/ cascade_strategy set to "all") to, for example, set the message type of every message to a specific custom value after the messages have been decoded from Protocol Buffers format. Note that this only supports setting the exact same value on every message, if any dynamic computation is required to determine what the value should be, or whether it should be applied to a specific message, a *SandboxDecoder* using the provided *write_message* API call should be used instead.

Config:

- **message_fields:** Subsection defining message fields to populate. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. *host|ipv4 = "192.168.55.55"* will create Fields[Host] containing an IPv4 address. Adding a representation string to a standard message header name will cause it to be added as a user defined field, i.e. Payload|json will create Fields[Payload] with a json representation (see *Field Variables*). Does not support Timestamp or Uuid.

Example (in MultiDecoder context)

```
[mytypedecoder]
type = "MultiDecoder"
subs = ["ProtobufDecoder", "mytype"]
cascade_strategy = "all"
log_sub_errors = true

[ProtobufDecoder]

[mytype]
type = "ScribbleDecoder"

    [mytype.message_fields]
    Type = "MyType"
```

2.4.13 StatsToFieldsDecoder

New in version 0.4.

The StatsToFieldsDecoder will parse time series statistics data in the graphite message format and encode the data into the message fields, in the same format produced by a *StatAccumInput* plugin with the *emit_in_fields* value set to true. This is useful if you have externally generated graphite string data flowing through Heka that you'd like to process without having to roll your own string parsing code.

This decoder has no configuration options. It simply expects to be passed messages with statsd string data in the payload. Incorrect or malformed content will cause a decoding error, dropping the message.

The fields format only contains a single "timestamp" field, so any payloads containing multiple timestamps will end up generating a separate message for each timestamp. Extra messages will be a copy of the original message except a) the payload will be empty and b) the unique timestamp and related stats will be the only message fields.

Example:

```
[StatsToFieldsDecoder]
```

2.5 Filters

2.5.1 Common Filter Parameters

There are some configuration options that are universally available to all Heka filter plugins. These will be consumed by Heka itself when Heka initializes the plugin and do not need to be handled by the plugin-specific initialization code.

- **message_matcher (string, optional):** Boolean expression, when evaluated to true passes the message to the filter for processing. Defaults to matching nothing. See: *Message Matcher Syntax*
- **message_signer (string, optional):** The name of the message signer. If specified only messages with this signer are passed to the filter for processing.
- **ticker_interval (uint, optional):** Frequency (in seconds) that a timer event will be sent to the filter. Defaults to not sending timer events.

2.5.2 Circular Buffer Delta Aggregator

New in version 0.5.

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). The purpose is to recreate the view at a larger scope in each level of the aggregation i.e., host view -> datacenter view -> service level view.

Config:

- **enable_delta (bool, optional, default false)** Specifies whether or not this aggregator should generate cbuf deltas.
- **anomaly_config(string) - (see *sandbox_anomaly_module*)** A list of anomaly detection specifications. If not specified no anomaly detection/alerting will be performed.

Example Heka Configuration

```
[TelemetryServerMetricsAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsAggregator.config]
enable_delta = false
anomaly_config = 'roc("Request Statistics", 1, 15, 0, 1.5, true, false)'
```

2.5.3 CBuf Delta Aggregator By Hostname

New in version 0.5.

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). Each column from the source circular buffer will become its own graph. i.e., 'Error Count' will become a graph with each host being represented in a column.

Config:

- **max_hosts (uint)** Pre-allocates the number of host columns in the graph(s). If the number of active hosts exceed this value, the plugin will terminate.
- **rows (uint)** The number of rows to keep from the original circular buffer. Storing all the data from all the hosts is not practical since you will most likely run into memory and output size restrictions (adjust the view down as necessary).
- **host_expiration (uint, optional, default 120 seconds)** The amount of time a host has to be inactive before it can be replaced by a new host.

Example Heka Configuration

```
[TelemetryServerMetricsHostAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_host_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsHostAggregator.config]
max_hosts = 5
rows = 60
host_expiration = 120
```

2.5.4 CounterFilter

Once per ticker interval a CounterFilter will generate a message of type *heka .counter-output*. The payload will contain text indicating the number of messages that matched the filter's *message_matcher* value during that interval (i.e. it counts the messages the plugin received). Every ten intervals an extra message (also of type *heka.counter-output*) goes out, containing an aggregate count and average per second throughput of messages received.

Config:

- **ticker_interval (int, optional):** Interval between generated counter messages, in seconds. Defaults to 5.

Example:

```
[CounterFilter]
message_matcher = "Type != 'heka.counter-output'"
```

2.5.5 Frequent Items

New in version 0.5.

Calculates the most frequent items in a data stream.

Config:

- **message_variable (string)** The message variable name containing the items to be counted.

- **max_items (uint, optional, default 1000)** The maximum size of the sample set (higher will produce a more accurate list).
- **min_output_weight (uint, optional, default 100)** Used to reduce the long tail output by only outputting the higher frequency items.
- **reset_days (uint, optional, default 1)** Resets the list after the specified number of days (on the UTC day boundary). A value of 0 will never reset the list.

Example Heka Configuration

```
[FxaAuthServerFrequentIP]
type = "SandboxFilter"
filename = "lua_filters/frequent_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerFrequentIP.config]
message_variable = "Fields[remote_addr]"
max_items = 10000
min_output_weight = 100
reset_days = 1
```

2.5.6 Heka Memory Statistics

New in version 0.6.

Graphs the Heka memory statistics using the heka.memstat message generated by pipeline/report.go.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *sandbox_anomaly_module*)

Example Heka Configuration

```
[HekaMemstat]
type = "SandboxFilter"
filename = "lua_filters/heka_memstat.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'heka.memstat'"
```

2.5.7 Heka Message Schema

New in version 0.5.

Generates documentation for each unique message in a data stream. The output is a hierarchy of Logger, Type, EnvVersion, and a list of associated message field attributes including their counts (number in the brackets). This plugin is meant for data discovery/exploration and should not be left running on a production system.

Config:

<none>

Example Heka Configuration

```
[SyncMessageSchema]
type = "SandboxFilter"
filename = "lua_filters/heka_message_schema.lua"
ticker_interval = 60
preserve_data = false
message_matcher = "Logger =~ /^Sync/"
```

Example Output

```
Sync-1_5-Webserver [54600]
    slf [54600]
        -no version- [54600]
            upstream_response_time (mismatch)
            http_user_agent (string)
            body_bytes_sent (number)
            remote_addr (string)
            request (string)
            upstream_status (mismatch)
            status (number)
            request_time (number)
            request_length (number)
Sync-1_5-SlowQuery [37]
    mysql.slow-query [37]
        -no version- [37]
            Query_time (number)
            Rows_examined (number)
            Rows_sent (number)
            Lock_time (number)
```

2.5.8 HTTP Status Graph

New in version 0.5.

Graphs HTTP status codes using the numeric Fields[status] variable collected from web server access logs.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *sandbox_anomaly_module*)

Example Heka Configuration

```
[FxaAuthServerHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
```

```
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerHTTPStatus.config]
sec_per_row = 60
rows = 1440
anomaly_config = 'roc("HTTP Status", 1, 15, 0, 1.5, true, false)'
```

2.5.9 MySQL Slow Query

New in version 0.6.

Graphs MySQL slow query data produced by the *MySQL Slow Query Log Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *sandbox_anomaly_module*)

Example Heka Configuration

```
[Sync-1_5-SlowQueries]
type = "SandboxFilter"
message_matcher = "Logger == 'Sync-1_5-SlowQuery'"
ticker_interval = 60
filename = "lua_filters/mysql_slow_query.lua"

    [Sync-1_5-SlowQueries.config]
    anomaly_config = 'mww_nonparametric("Statistics", 5, 15, 10, 0.8)'
```

2.5.10 StatFilter

Filter plugin that accepts messages of a specfied form and uses extracted message data to generate statsd-style numerical metrics in the form of *Stat* objects that can be consumed by a *StatAccumulator*.

Config:

- **Metric:** Subsection defining a single metric to be generated:
  - **type (string):** Metric type, supports "Counter", "Timer", "Gauge".
  - **name (string):** Metric name, must be unique.
  - **value (string):** Expression representing the (possibly dynamic) value that the *StatFilter* should emit for each received message.
- **stat_accum_name (string):** Name of a StatAccumInput instance that this StatFilter will use as its StatAccumulator for submitting generate stat values. Defaults to "StatAccumInput".

Example (Assuming you had TransformFilter inserting messages as above):

```
[StatAccumInput]
ticker_interval = 5

[StatsdInput]
address = "127.0.0.1:29301"
```

```
[Hits]
type = "StatFilter"
message_matcher = 'Type == "ApacheLogfile"'

[Hits.Metric.bandwidth]
type = "Counter"
name = "httpd.bytes.%Hostname%"
value = "%Bytes%"

[Hits.Metric.method_counts]
type = "Counter"
name = "httpd.hits.%Method%.%Hostname%"
value = "1"
```

Note: StatFilter requires an available StatAccumInput to be running.

2.5.11 SandboxFilter

The sandbox filter provides an isolated execution environment for data analysis. Any output generated by the sandbox is injected into the payload of a new message for further processing or to be output.

Config:

- *Common Filter Parameters*
- *config_common_sandbox_parameters*

Example:

```
[hekabench_counter]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench'"
ticker_interval = 1
filename = "counter.lua"
preserve_data = true
profile = false

    [hekabench_counter.config]
    rows = 1440
    sec_per_row = 60
```

2.5.12 SandboxManagerFilter

The SandboxManagerFilter provides dynamic control (start/stop) of sandbox filters in a secure manner without stopping the Heka daemon. Commands are sent to a SandboxManagerFilter using a signed Heka message. The intent is to have one manager per access control group each with their own message signing key. Users in each group can submit a signed control message to manage any filters running under the associated manager. A signed message is not an enforced requirement but it is highly recommended in order to restrict access to this functionality.

SandboxManagerFilter Settings

- *Common Filter Parameters*

- **working_directory (string):** The directory where the filter configurations, code, and states are preserved. The directory can be unique or shared between sandbox managers since the filter names are unique per manager. Defaults to a directory in ${BASE_DIR}/sbxmgrs with a name generated from the plugin name.
- **module_directory (string):** The directory where 'require' will attempt to load the external Lua modules from. Defaults to ${SHARE_DIR}/lua_modules.
- **max_filters (uint):** The maximum number of filters this manager can run.

New in version 0.5.

- **memory_limit (uint):** The number of bytes managed sandboxes are allowed to consume before being terminated (max 8MiB, default max).
- **instruction_limit (uint):** The number of instructions managed sandboxes are allowed the execute during the process_message/timer_event functions before being terminated (max 1M, default max).
- **output_limit (uint):** The number of bytes managed sandbox output buffers can hold before before being terminated (max 63KiB, default max). Anything less than 64B is set to 64B.

Example

```
[OpsSandboxManager]
type = "SandboxManagerFilter"
message_signer = "ops"
# message_matcher = "Type == 'heka.control.sandbox'" # automatic default setting
max_filters = 100
```

2.5.13 Unique Items

New in version 0.6.

Counts the number of unique items per day e.g. active daily users by uid.

Config:

- **message_variable (string, required)** The Heka message variable containing the item to be counted.
- **title (string, optional, default "Estimated Unique Daily *message_variable*")** The graph title for the cbuf output.
- **enable_delta (bool, optional, default false)** Specifies whether or not this plugin should generate cbuf deltas. Deltas should be enabled when sharding is used; see: *Circular Buffer Delta Aggregator*.

Example Heka Configuration

```
[FxaActiveDailyUsers]
type = "SandboxFilter"
filename = "lua_filters/unique_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'FxaAuth' && Type == 'request.summary' && Fields[path] == '/v1/certifica

    [FxaActiveDailyUsers.config]
    message_variable = "Fields[uid]"
    title = "Estimated Active Daily Users"
```

New in version 0.6.

2.6 Encoders

2.6.1 Alert Encoder

Produces more human readable alert messages.

Config:

<none>

Example Heka Configuration

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "((Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert') || Type == 'he
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"

[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

Example Output

Timestamp 2014-05-14T14:20:18Z

Hostname ip-10-226-204-51

Plugin FxaBrowserIdHTTPStatus

Alert HTTP Status - algorithm: roc col: 1 msg: detected anomaly, standard deviation exceeds 1.5

2.6.2 ESJsonEncoder

This encoder serializes a Heka message into a clean JSON format, preceded by a separate JSON structure containing information required for ElasticSearch BulkAPI indexing. The JSON serialization is done by hand, without the use of Go's stdlib JSON marshalling. This is so serialization can succeed even if the message contains invalid UTF-8 characters, which will be encoded as U+FFFD.

Config:

- **index (string):** Name of the ES index into which the messages will be inserted. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', a field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-%{Logger}-data' would add the records to an ES index called 'some.example.com-processname-data'. Defaults to 'heka-%{2006.01.02}'.
- **type_name (string):** Name of ES record type to create. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-stat' would create an ES record with a type of 'some.example.com-stat'. Defaults to 'message'.
- **fields ([]string):** The 'fields' parameter specifies that only specific message data should be indexed into ElasticSearch. Available fields to choose are "Uuid", "Timestamp", "Type", "Logger", "Severity", "Payload",

"EnvVersion", "Pid", "Hostname", and "Fields" (where "Fields" causes the inclusion of any and all dynamically specified message fields. Defaults to including all of the supported message fields.

- **timestamp (string):** Format to use for timestamps in generated ES documents. Defaults to "2006-01-02T15:04:05.000Z".
- **es_index_from_timestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.
- **id (string):** Allows you to optionally specify the document id for ES to use. Useful for overwriting existing ES documents. If the value specified is placed within %{}, it will be interpolated to its Field value. Default is allow ES to auto-generate the id.
- **raw_bytes_fields ([]string):** This specifies a set of fields which will be passed through to the encoded JSON output without any processing or escaping. This is useful for fields which contain embedded JSON objects to prevent the embedded JSON from being escaped as normal strings. Only supports dynamically specified message fields.

Example

```
[ESJsonEncoder]
index = "%{Type}-%{2006.01.02}"
es_index_from_timestamp = true
type_name = "%{Type}"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESJsonEncoder"
flush_interval = 50
```

2.6.3 ESLogstashV0Encoder

This encoder serializes a Heka message into a JSON format, preceded by a separate JSON structure containing information required for ElasticSearch BulkAPI indexing. The message JSON structure uses the original (i.e. "v0") schema popularized by Logstash. Using this schema can aid integration with existing Logstash deployments. This schema also plays nicely with the default Logstash dashboard provided by Kibana.

The JSON serialization is done by hand, without using Go's stdlib JSON marshalling. This is so serialization can succeed even if the message contains invalid UTF-8 characters, which will be encoded as U+FFFD.

Config:

- **index (string):** Name of the ES index into which the messages will be inserted. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', a field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-%{Logger}-data' would add the records to an ES index called 'some.example.com-processname-data'. Defaults to 'logstash-%{2006.01.02}'.
- **type_name (string):** Name of ES record type to create. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-stat' would create an ES record with a type of 'some.example.com-stat'. Defaults to 'message'.
- **fields ([]string):** The 'fields' parameter specifies that only specific message data should be indexed into ElasticSearch. Available fields to choose are "Uuid", "Timestamp", "Type", "Logger", "Severity", "Payload", "EnvVersion", "Pid", "Hostname", and "Fields" (where "Fields" causes the inclusion of any and all dynamically specified message fields. Defaults to including all of the supported message fields.

- **es_index_from_timestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.
- **id (string):** Allows you to optionally specify the document id for ES to use. Useful for overwriting existing ES documents. If the value specified is placed within %{}, it will be interpolated to its Field value. Default is allow ES to auto-generate the id.
- **raw_bytes_fields ([]string):** This specifies a set of fields which will be passed through to the encoded JSON output without any processing or escaping. This is useful for fields which contain embedded JSON objects to prevent the embedded JSON from being escaped as normal strings. Only supports dynamically specified message fields.

Example

```
[ESLogstashV0Encoder]
es_index_from_timestamp = true
type_name = "%{Type}"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESLogstashV0Encoder"
flush_interval = 50
```

2.6.4 ESPayloadEncoder

Prepends ElasticSearch BulkAPI index JSON to a message payload.

Config:

- **index (string, optional, default "heka-%{%Y.%m.%d}")** String to use as the *_index* key's value in the generated JSON. Supports field interpolation as described below.
- **type_name (string, optional, default "message")** String to use as the *_type* key's value in the generated JSON. Supports field interpolation as described below.
- **id (string, optional)** String to use as the *_id* key's value in the generated JSON. Supports field interpolation as described below.
- **es_index_from_timestamp (boolean, optional)** If true, then any time interpolation (often used to generate the ElasticSeach index) will use the timestamp from the processed message rather than the system time.

Field interpolation:

> Data from the current message can be interpolated into any of the string arguments listed above. A *%{}* enclosed field name will be replaced by the field value from the current message. Supported default field names are "Type", "Hostname", "Pid", "UUID", "Logger", "EnvVersion", and "Severity". Any other values will be checked against the defined dynamic message fields. If no field matches, then a C strftime (on non-Windows platforms) or C89 strftime (on Windows) time substitution will be attempted.

Example Heka Configuration

```
[es_payload]
type = "SandboxEncoder"
filename = "lua_encoders/es_payload.lua"
    [es_payload.config]
    es_index_from_timestamp = true
    index = "%{Logger}-%{%Y.%m.%d}"
    type_name = "%{Type}-%{Hostname}"

[ElasticSearchOutput]
```

```
message_matcher = "Type == 'mytype'"
encoder = "es_payload"
```

Example Output

{"index":{"_index":"mylogger-2014.06.05","_type":"mytype-host.domain.com"}} {"json":"data","extracted":"from","message":"payl

2.6.5 PayloadEncoder

The PayloadEncoder simply extracts the payload from the provided Heka message and converts it into a byte stream for delivery to an external resource. Config:

- **append_newlines (bool, optional):** Specifies whether or not a newline character (i.e. *n*) will be appended to the captured message payload before serialization. Defaults to true.
- **prefix_ts (bool, optional):** Specifies whether a timestamp will be prepended to the captured message payload before serialization. Defaults to false.
- **ts_from_message (bool, optional):** If true, the prepended timestamp will be extracted from the message that is being processed. If false, the prepended timestamp will be generated by the system clock at the time of message processing. Defaults to true. This setting has no impact if *prefix_ts* is set to false.
- **ts_format (string, optional):** Specifies the format that should be used for prepended timestamps, using Go's standard time format specification strings. Defaults to *[2006/Jan/02:15:04:05 -0700]*. If the specified format string does not end with a space character, then a space will be inserted between the formatted timestamp and the payload.

Example

```
[PayloadEncoder]
append_newlines = false
prefix_ts = true
ts_format = "2006/01/02 3:04:05PM MST"
```

2.6.6 ProtobufEncoder

The ProtobufEncoder is used to serialize Heka message objects back into Heka's standard protocol buffers format. This is the format that Heka uses to communicate with other Heka instances, so one will always be included in your Heka configuration using the default "ProtobufEncoder" name whether specified or not.

The hekad protocol buffers message schema is defined in the *message.proto* file in the *message* package.

Config:

<none>

Example:

```
[ProtobufEncoder]
```

See also:

Protocol Buffers - Google's data interchange format

2.6.7 SandboxEncoder

The SandboxEncoder provides an isolated execution environment for converting messages into binary data without the need to recompile Heka. See *Sandbox*. Config:

- *config_common_sandbox_parameters*

Example

```
[custom_json_encoder]
type = "SandboxEncoder"
filename = "path/to/custom_json_encoder.lua"

    [custom_json_encoder.config]
    msg_fields = ["field1", "field2"]
```

2.7 Outputs

2.7.1 Common Output Parameters

There are some configuration options that are universally available to all Heka output plugins. These will be consumed by Heka itself when Heka initializes the plugin and do not need to be handled by the plugin-specific initialization code.

- **message_matcher (string, optional):** Boolean expression, when evaluated to true passes the message to the filter for processing. Defaults to matching nothing. See: *Message Matcher Syntax*
- **message_signer (string, optional):** The name of the message signer. If specified only messages with this signer are passed to the filter for processing.
- **ticker_interval (uint, optional):** Frequency (in seconds) that a timer event will be sent to the filter. Defaults to not sending timer events.
- **encoder (string, optional):** New in version 0.6.

  Encoder to be used by the output. This should refer to the name of an encoder plugin section that is specified elsewhere in the TOML configuration. Messages can be encoded using the specified encoder by calling the OutputRunner's *Encode()* method.
- **use_framing (bool, optional):** New in version 0.6.

  Specifies whether or not Heka's *Stream Framing* should be applied to the binary data returned from the OutputRunner's *Encode()* method.

2.7.2 AMQPOutput

Connects to a remote AMQP broker (RabbitMQ) and sends messages to the specified queue. The message is serialized if specified, otherwise only the raw payload of the message will be sent. As AMQP is dynamically programmable, the broker topology needs to be specified.

Config:

- **URL (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **Exchange (string):** AMQP exchange name
- **ExchangeType (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **ExchangeDurability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **ExchangeAutoDelete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.

- **RoutingKey (string):** The message routing key used to bind the queue to the exchange. Defaults to empty string.
- **Persistent (bool):** Whether published messages should be marked as persistent or transient. Defaults to non-persistent.

New in version 0.6.

- **ContentType (string):** MIME content type of the payload used in the AMQP header. Defaults to "application/hekad".
- **encoder (string, optional)** Specifies which of the registered encoders should be used for converting Heka messages to binary data that is sent out over the AMQP connection. Defaults to the always available "ProtobufEncoder".
- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true.

New in version 0.6.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *AMQPS* URI scheme. See *Configuring TLS*.

Example (that sends log lines from the logger):

```
[AMQPOutput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
message_matcher = 'Logger == "TestWebserver"'
```

2.7.3 CarbonOutput

CarbonOutput plugins parse the "stat metric" messages generated by a StatAccumulator and write the extracted counter, timer, and gauge data out to a graphite compatible carbon daemon. Output is written over a TCP or UDP socket using the plaintext protocol.

Config:

- **address (string):** An IP address:port on which this plugin will write to. (default: "localhost:2003")

New in version 0.5.

- **protocol (string):** "tcp" or "udp" (default: "tcp")
- **tcp_keep_alive (bool)** if set, keep the TCP connection open and reuse it until a failure; then retry (default: false)

Example:

```
[CarbonOutput]
message_matcher = "Type == 'heka.statmetric'"
address = "localhost:2003"
protocol = "udp"
```

2.7.4 DashboardOutput

Specialized output plugin that listens for certain Heka reporting message types and generates JSON data which is made available via HTTP for use in web based dashboards and health reports.

Config:

- **ticker_interval (uint):** Specifies how often, in seconds, the dashboard files should be updated. Defaults to 5.
- **message_matcher (string):** Defaults to *"Type == 'heka.all-report' || Type == 'heka.sandbox-output' || Type == 'heka.sandbox-terminated"'*. Not recommended to change this unless you know what you're doing.
- **address (string):** An IP address:port on which we will serve output via HTTP. Defaults to "0.0.0.0:4352".
- **working_directory (string):** File system directory into which the plugin will write data files and from which it will serve HTTP. The Heka process must have read / write access to this directory. Relative paths will be evaluated relative to the Heka base directory. Defaults to *$(BASE_DIR)/dashboard*.
- **static_directory (string):** File system directory where the Heka dashboard source code can be found. The Heka process must have read access to this directory. Relative paths will be evaluated relative to the Heka base directory. Defaults to *${SHARE_DIR}/dasher*.

Example:

```
[DashboardOutput]
ticker_interval = 30
```

2.7.5 ElasticSearchOutput

Output plugin that uses HTTP or UDP to insert records into an ElasticSearch database. Note that it is up to the specified encoder to both serialize the message into a JSON structure *and* to prepend that with the appropriate ElasticSearch BulkAPI indexing JSON. Usually this output is used in conjunction with an ElasticSearch-specific encoder plugin, such as *ESJsonEncoder*, *ESLogstashV0Encoder*, or *ESPayloadEncoder*.

Config:

- **flush_interval (int):** Interval at which accumulated messages should be bulk indexed into ElasticSearch, in milliseconds. Defaults to 1000 (i.e. one second).
- **flush_count (int):** Number of messages that, if processed, will trigger them to be bulk indexed into ElasticSearch. Defaults to 10.
- **server (string):** ElasticSearch server URL. Supports http://, https:// and udp:// urls. Defaults to "http://localhost:9200".
- **http_timeout (int):** Time in milliseconds to wait for a response for each http post to ES. This may drop data as there is currently no retry. Default is 0 (no timeout).

Example:

```
[ElasticSearchOutput]
message_matcher = "Type == 'sync.log'"
server = "http://es-server:9200"
flush_interval = 5000
flush_count = 10
encoder = "ESJsonEncoder"
```

2.7.6 FileOutput

Writes message data out to a file system.

Config:

- **path (string):** Full path to the output file.
- **perm (string, optional):** File permission for writing. A string of the octal digit representation. Defaults to "644".

- **folder_perm (string, optional):** Permissions to apply to directories created for FileOutput's parent directory if it doesn't exist. Must be a string representation of an octal integer. Defaults to "700".
- **flush_interval (uint32, optional):** Interval at which accumulated file data should be written to disk, in milliseconds (default 1000, i.e. 1 second). Set to 0 to disable.
- **flush_count (uint32, optional):** Number of messages to accumulate until file data should be written to disk (default 1, minimum 1).
- **flush_operator (string, optional):** Operator describing how the two parameters "flush_interval" and "flush_count" are combined. Allowed values are "AND" or "OR" (default is "AND").

New in version 0.6.

- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true if a ProtobufEncoder is used, false otherwise.

Example:

```
[counter_file]
type = "FileOutput"
message_matcher = "Type == 'heka.counter-output'"
path = "/var/log/heka/counter-output.log"
prefix_ts = true
perm = "666"
flush_count = 100
flush_operator = "OR"
encoder = "PayloadEncoder"
```

New in version 0.6.

2.7.7 HttpOutput

A very simple output plugin that uses HTTP GET, POST, or PUT requests to deliver data to an HTTP endpoint. When using POST or PUT request methods the encoded output will be uploaded as the request body. When using GET the encoded output will be ignored.

This output doesn't support any request batching; each received message will generate an HTTP request. Batching can be achieved by use of a filter plugin that accumulates message data, periodically emitting a single message containing the batched, encoded HTTP request data in the payload. An HttpOutput can then be configured to capture these batch messages, using a *PayloadEncoder* to extract the message payload.

For now the HttpOutput only supports statically defined request parameters (URL, headers, auth, etc.). Future iterations will provide a mechanism for dynamically specifying these values on a per-message basis.

Config:

- **address (string):** URL address of HTTP server to which requests should be sent. Must begin with "http://" or "https://".
- **method (string, optional):** HTTP request method to use, must be one of GET, POST, or PUT. Defaults to POST.
- **username (string, optional):** If specified, HTTP Basic Auth will be used with the provided user name.
- **password (string, optional):** If specified, HTTP Basic Auth will be used with the provided password.
- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing request by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.

- **tls (subsection, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if an "https://" address is used. See *Configuring TLS*.

Example:

```
[PayloadEncoder]

[influxdb]
message_matcher = "Type == 'influx.formatted'"
address = "http://influxdb.example.com:8086/db/stats/series"
encoder = "PayloadEncoder"
username = "MyUserName"
password = "MyPassword"
```

2.7.8 LogOutput

Logs messages to stdout using Go's *log* package.

Config:

<none>

Example:

```
[counter_output]
type = "LogOutput"
message_matcher = "Type == 'heka.counter-output'"
encoder = "PayloadEncoder"
```

2.7.9 NagiosOutput

Specialized output plugin that listens for Nagios external command message types and delivers passive service check results to Nagios using either HTTP requests made to the Nagios cmd.cgi API or the use of the *send_ncsa* binary. The message payload must consist of a state followed by a colon and then the message e.g., "OK:Service is functioning properly". The valid states are: OK|WARNING|CRITICAL|UNKNOWN. Nagios must be configured with a service name that matches the Heka plugin instance name and the hostname where the plugin is running.

Config:

- **url (string, optional):** An HTTP URL to the Nagios cmd.cgi. Defaults to http://localhost/nagios/cgi-bin/cmd.cgi.
- **username (string, optional):** Username used to authenticate with the Nagios web interface. Defaults to empty string.
- **password (string, optional):** Password used to authenticate with the Nagios web interface. Defaults to empty string.
- **response_header_timeout (uint, optional):** Specifies the amount of time, in seconds, to wait for a server's response headers after fully writing the request. Defaults to 2.
- **nagios_service_description (string, optional):** Must match Nagios service's service_description attribute. Defaults to the name of the output.
- **nagios_host (string, optional):** Must match the hostname of the server in nagios. Defaults to the Hostname attribute of the message.
- **send_nsca_bin (string, optional):** New in version 0.5.

Use send_nsca program, as provided, rather than sending HTTP requests. Not supplying this value means HTTP will be used, and any other send_nsca_* settings will be ignored.

- **send_nsca_args ([]string, optional):** New in version 0.5.

  Arguments to use with send_nsca, usually at least the nagios hostname, e.g. *[”-H”, “nagios.somehost.com”]*. Defaults to an empty list.

- **send_nsca_timeout (int, optional):** New in version 0.5.

  Timeout for the send_nsca command, in seconds. Defaults to 5.

- **use_tls (bool, optional):** New in version 0.5.

  Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

- **tls (TlsConfig, optional):** New in version 0.5.

  A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

Example configuration to output alerts from SandboxFilter plugins:

```
[NagiosOutput]
url = "http://localhost/nagios/cgi-bin/cmd.cgi"
username = "nagiosadmin"
password = "nagiospw"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'nagios-external-command'
```

Example Lua code to generate a Nagios alert:

```
inject_payload("nagios-external-command", "PROCESS_SERVICE_CHECK_RESULT", "OK:Alerts are working!")
```

2.7.10 SmtpOutput

New in version 0.5.

Outputs a Heka message in an email. The message subject is the plugin name and the message content is controlled by the payload_only setting. The primary purpose is for email alert notifications e.g., PagerDuty.

Config:

- **send_from (string)** The email address of the sender. (default: “heka@localhost.localdomain”)
- **send_to (array of strings)** An array of email addresses where the output will be sent to.
- **subject (string)** Custom subject line of email. (default: “Heka [SmtpOutput]”)
- **host (string)** SMTP host to send the email to (default: “127.0.0.1:25”)
- **auth (string)** SMTP authentication type: “none”, “Plain”, “CRAMMD5” (default: “none”)
- **user (string, optional)** SMTP user name
- **password (string, optional)** SMTP user password

Example:

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "((Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert') || Type == 'he
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
```

```
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"
```

2.7.11 TcpOutput

Output plugin that serializes messages into the Heka protocol format and delivers them to a listening TCP connection. Can be used to deliver messages from a local running Heka agent to a remote Heka instance set up as an aggregator and/or router.

Config:

- **address (string):** An IP address:port to which we will send our output data.
- **use_tls (bool, optional):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

New in version 0.5.

- **tls (TlsConfig, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.
- **ticker_interval (uint, optional):** Specifies how often, in seconds, the output queue files are rolled. Defaults to 300.

New in version 0.6.

- **local_address (string, optional):** A local IP address to use as the source address for outgoing traffic to this destination. Cannot currently be combined with TLS connections.
- **encoder (string, optional):** Specifies which of the registered encoders should be used for converting Heka messages to binary data that is sent out over the TCP connection. Defaults to the always available "ProtobufEncoder".
- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true if a ProtobufEncoder is used, false otherwise.

Example:

```
[aggregator_output]
type = "TcpOutput"
address = "heka-aggregator.mydomain.com:55"
local_address = "127.0.0.1"
message_matcher = "Type != 'logfile' && Type != 'heka.counter-output' && Type != 'heka.all-report'"
```

2.7.12 WhisperOutput

WhisperOutput plugins parse the "statmetric" messages generated by a StatAccumulator and write the extracted counter, timer, and gauge data out to a graphite compatible whisper database file tree structure.

Config:

- **base_path (string):** Path to the base directory where the whisper file tree will be written. Absolute paths will be honored, relative paths will be calculated relative to the Heka base directory. Defaults to "whisper" (i.e. "$(BASE_DIR)/whisper").

- **default_agg_method (int):** Default aggregation method to use for each whisper output file. Supports the following values:
  0. Unknown aggregation method.
  1. Aggregate using averaging. (default)
  2. Aggregate using summation.
  3. Aggregate using last received value.
  4. Aggregate using maximum value.
  5. Aggregate using minimum value.
- **default_archive_info ([][]int):** Default specification for new whisper db archives. Should be a sequence of 3-tuples, where each tuple describes a time interval's storage policy: [<offset> <# of secs per datapoint> <# of datapoints>] (see whisper docs for more info). Defaults to:

  ```
  [ [0, 60, 1440], [0, 900, 8], [0, 3600, 168], [0, 43200, 1456]]
  ```

  The above defines four archive sections. The first uses 60 seconds for each of 1440 data points, which equals one day of retention. The second uses 15 minutes for each of 8 data points, for two hours of retention. The third uses one hour for each of 168 data points, or 7 days of retention. Finally, the fourth uses 12 hours for each of 1456 data points, representing two years of data.
- **folder_perm (string):** Permission mask to be applied to folders created in the whisper database file tree. Must be a string representation of an octal integer. Defaults to "700".

Example:

```
[WhisperOutput]
message_matcher = "Type == 'heka.statmetric'"
default_agg_method = 3
default_archive_info = [ [0, 30, 1440], [0, 900, 192], [0, 3600, 168], [0, 43200, 1456] ]
folder_perm = "755"
```

2.8 Monitoring Internal State

Heka can emit metrics about it's internal state to either an outgoing Heka message (and, through the DashboardOutput, to a web dashboard) or to stdout. Sending SIGUSR1 to hekad on a UNIX will send a plain text report tostdout. On Windows, you will need to send signal 10 to the hekad process using Powershell.

Sample text output

```
========[heka.all-report]========
inputRecycleChan:
    InChanCapacity: 100
    InChanLength: 99
injectRecycleChan:
    InChanCapacity: 100
    InChanLength: 98
Router:
    InChanCapacity: 50
    InChanLength: 0
    ProcessMessageCount: 26
ProtobufDecoder-0:
    InChanCapacity: 50
    InChanLength: 0
ProtobufDecoder-1:
```

```
    InChanCapacity: 50
    InChanLength: 0
ProtobufDecoder-2:
    InChanCapacity: 50
    InChanLength: 0
ProtobufDecoder-3:
    InChanCapacity: 50
    InChanLength: 0
DecoderPool-ProtobufDecoder:
    InChanCapacity: 4
    InChanLength: 4
OpsSandboxManager:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 0
    ProcessMessageCount: 0
hekabench_counter:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 445
    ProcessMessageCount: 0
    InjectMessageCount: 0
    Memory: 20644
    MaxMemory: 20644
    MaxInstructions: 18
    MaxOutput: 0
    ProcessMessageAvgDuration: 0
    TimerEventAvgDuration: 78532
LogOutput:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 406
DashboardOutput:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 336
========
```

To enable the HTTP interface, you will need to enable the dashboard output plugin, see *DashboardOutput*.

2.9 Extending Heka

The core of the Heka engine is written in the Go programming language. Heka supports five different types of plugins (inputs, decoders, filters, encoders, and outputs), which are also written in Go. This document will try to provide enough information for developers to extend Heka by implementing their own custom plugins. It assumes a small amount of familiarity with Go, although any reasonably experienced programmer will probably be able to follow along with no trouble.

NOTE: Heka also supports the use of security sandboxed Lua code for implementing the core logic of decoder, filter, and encoder plugins. This document only covers the development of Go plugins. You can learn more about sandboxed plugins in the *Sandbox* section.

2.9.1 Definitions

You should be familiar with the *glossary* terminology before proceeding.

2.9.2 Overview

Each Heka plugin type performs a specific task: inputs receive input from the outside world and inject the data into the Heka pipeline, decoders turn binary data into Message objects that Heka can process, filters perform arbitrary processing of Heka message data, encoders serialize Heka messages into arbitrary byte streams, and outputs send data from Heka back to the outside world. Each specific plugin has some custom behaviour, but it also shares behaviour w/ every other plugin of that type. A UDPInput and a TCPInput listen on the network differently, and a LogstreamerInput (reading files off the file system) doesn't listen on the network at all, but all of these inputs need to interact w/ the Heka system to access data structures, gain access to decoders to which we pass our incoming data, respond to shutdown and other system events, etc.

To support this all Heka plugins except encoders actually consist of two parts: the plugin itself, and an accompanying "plugin runner". Inputs have an InputRunner, decoders have a DecoderRunner, filters have a FilterRunner, and Outputs have an OutputRunner. The plugin itself contains the plugin-specific behaviour, and is provided by the plugin developer. The plugin runner contains the shared (by type) behaviour, and is provided by Heka. When Heka starts a plugin, it a) creates and configures a plugin instance of the appropriate type, b) creates a plugin runner instance of the appropriate type (passing in the plugin), and c) calls the Start method of the plugin runner. Most plugin runners (all except decoders) then call the plugin's Run method, passing themselves and an additional PluginHelper object in as arguments so the plugin code can use their exposed APIs to interact w/ the Heka system.

For inputs, filters, and outputs, there's a 1:1 correspondence between sections specified in the config file and running plugin instances. This is not the case for decoders and encoders, however. Decoder and encoder sections register possible configurations, but actual decoder and encoder instances aren't created until they are used by input or output plugins.

2.9.3 Plugin Configuration

Heka uses a slightly modified version of TOML as its configuration file format (see: *Configuring hekad*), and provides a simple mechanism through which plugins can integrate with the configuration loading system to initialize themselves from settings in hekad's config file.

The minimal shared interface that a Heka plugin must implement in order to use the config system is (unsurprisingly) *Plugin*, defined in pipeline_runner.go:

```
type Plugin interface {
    Init(config interface{}) error
}
```

During Heka initialization an instance of every plugin listed in the configuration file will be created. The TOML configuration for each plugin will be parsed and the resulting configuration object will be passed in to the above specified *Init* method. The argument is of type *interface{}*. By default the underlying type will be **pipeline.PluginConfig*, a map object that provides config data as key/value pairs. There is also a way for plugins to specify a custom struct to be used instead of the generic *PluginConfig* type (see *Custom Plugin Config Structs*). In either case, the config object will be already loaded with values read in from the TOML file, which your plugin code can then use to initialize itself. The input, filter, and output plugins will then be started so they can begin processing messages. The decoder and encoder instances will be thrown away, with new ones created as needed when requested by input (for decoder) or output (for encoder) plugins.

As an example, imagine we're writing a filter that will deliver messages to a specific output plugin, but only if they come from a list of approved hosts. Both 'hosts' and 'output' would be required in the plugin's config section. Here's one version of what the plugin definition and *Init* method might look like:

```
type HostFilter struct {
    hosts  map[string]bool
    output string
}

// Extract hosts value from config and store it on the plugin instance.
func (f *HostFilter) Init(config interface{}) error {
    var (
        hostsConf  interface{}
        hosts      []interface{}
        host       string
        outputConf interface{}
        ok         bool
    )
    conf := config.(pipeline.PluginConfig)
    if hostsConf, ok = conf["hosts"]; !ok {
        return errors.New("No 'hosts' setting specified.")
    }
    if hosts, ok = hostsConf.([]interface{}); !ok {
        return errors.New("'hosts' setting not a sequence.")
    }
    if outputConf, ok = conf["output"]; !ok {
        return errors.New("No 'output' setting specified.")
    }
    if f.output, ok = outputConf.(string); !ok {
        return errors.New("'output' setting not a string value.")
    }
    f.hosts = make(map[string]bool)
    for _, h := range hosts {
        if host, ok = h.(string); !ok {
            return errors.New("Non-string host value.")
        }
        f.hosts[host] = true
    }
    return nil
}
```

(Note that this is a bit of a contrived example. In practice, you would generally route messages to specific outputs using the *Message Matcher Syntax*.)

2.9.4 Restarting Plugins

In the event that your plugin fails to initialize properly at startup, hekad will exit. However, once hekad is running, if a plugin should fail (perhaps because a network connection dropped, a file became unavailable, etc), then hekad will shutdown. This shutdown can be avoided if your plugin supports being restarted.

To add restart support to your plugin, the *Restarting* interface defined in the config.go file:

```
type Restarting interface {
    CleanupForRestart()
}
```

A plugin that implements this interface will not trigger shutdown should it fail while hekad is running. The *Cleanup-ForRestart* method will be called when the plugins' main run method exits, a single time. Then the runner will repeatedly call the plugins Init method until it initializes successfully. It will then resume running it unless it exits again at which point the restart process will begin anew.

2.9.5 Custom Plugin Config Structs

In simple cases it might be fine to get plugin configuration data as a generic map of keys and values, but if there are more than a couple of config settings then checking for, extracting, and validating the values quickly becomes a lot of work. Heka plugins can instead specify a schema struct for their configuration data, into which the TOML configuration will be decoded.

Plugins that wish to provide a custom configuration struct should implement the *HasConfigStruct* interface defined in the config.go file:

```
type HasConfigStruct interface {
    ConfigStruct() interface{}
}
```

Any plugin that implements this method should return a struct that can act as the schema for the plugin configuration. Heka's config loader will then try to decode the plugin's TOML config into this struct. Note that this also gives you a way to specify default config values; you just populate your config struct as desired before returning it from the *ConfigStruct* method.

Let's say we wanted to write a *UdpOutput* that delivered messages to a UDP listener somewhere, defaulting to using my.example.com:44444 as the destination. The initialization code might look as follows:

```
// This is our plugin struct.
type UdpOutput struct {
    conn net.Conn
}

// This is our plugin's config struct
type UdpOutputConfig struct {
    Address string
}

// Provides pipeline.HasConfigStruct interface.
func (o *UdpOutput) ConfigStruct() interface{} {
    return &UdpOutputConfig{"my.example.com:44444"}
}

// Initialize UDP connection
func (o *UdpOutput) Init(config interface{}) (err error) {
    conf := config.(*UdpOutputConfig) // assert we have the right config type
    var udpAddr *net.UDPAddr
    if udpAddr, err = net.ResolveUDPAddr("udp", conf.Address); err != nil {
        return fmt.Errorf("can't resolve %s: %s", conf.Address,
            err.Error())
    }
    if o.conn, err = net.DialUDP("udp", nil, udpAddr); err != nil {
        return fmt.Errorf("error dialing %s: %s", conf.Address,
            err.Error())
    }
    return
}
```

In addition to specifying configuration options that are specific to your plugin, it is also possible to use the config struct to specify default values for the *ticker_interval* and *message_matcher* values that are available to all Filter and Output plugins. If a config struct contains a uint attribute called *TickerInterval*, that will be used as a default ticker interval value (in seconds) if none is supplied in the TOML. Similarly, if a config struct contains a string attribute called *MessageMatcher*, that will be used as the default message routing rule if none is specified in the configuration file.

There is an optional configuration interface called WantsName. It provides a a plug-in access to its configured name before the runner has started. The Sandbox filter plug-in uses the name to locate/load any preserved state before being run:

```
type WantsName interface {
    SetName(name string)
}
```

2.9.6 Inputs

Input plugins are responsible for acquiring data from the outside world and injecting this data into the Heka pipeline. An input might be passively listening for incoming network data or actively scanning external sources (either on the local machine or over a network). The input plugin interface is:

```
type Input interface {
    Run(ir InputRunner, h PluginHelper) (err error)
    Stop()
}
```

The *Run* method is called when Heka starts and, if all is functioning as intended, should not return until Heka is shut down. If a condition arises such that the input can not perform its intended activity it should return with an appropriate error, otherwise it should continue to run until a shutdown event is triggered by Heka calling the input's *Stop* method, at which time any clean-up should be done and a clean shutdown should be indicated by returning a *nil* error.

Inside the *Run* method, an input has three primary responsibilities:

1. Acquire information from the outside world
2. Use acquired information to populate *PipelinePack* objects that can be processed by Heka.
3. Pass the populated *PipelinePack* objects on to the appropriate next stage in the Heka pipeline (either to a decoder plugin so raw input data can be converted to a *Message* object, or by injecting them directly into the Heka message router if the *Message* object is already populated.)

The details of the first step are clearly entirely defined by the plugin's intended input mechanism(s). Plugins can (and should!) spin up goroutines as needed to perform tasks such as listening on a network connection, making requests to external data sources, scanning machine resources and operational characteristics, reading files from a file system, etc.

For the second step, before you can populate a *PipelinePack* object you have to actually *have* one. You can get empty packs from a channel provided to you by the *InputRunner*. You get the channel itself by calling *ir.InChan()* and then pull a pack from the channel whenever you need one.

Often, populating a *PipelinePack* is as simple as storing the raw data that was retrieved from the outside world in the pack's *MsgBytes* attribute. For efficiency's sake, it's best to write directly into the already allocated memory rather than overwriting the attribute with a *[]byte* slice pointing to a new array. Overwriting the array is likely to lead to a lot of garbage collector churn.

The third step involves the input plugin deciding where next to pass the *PipelinePack* and then doing so. Once the *MsgBytes* attribute has been set the pack will typically be passed on to a decoder plugin, which will convert the raw bytes into a *Message* object, also an attribute of the *PipelinePack*. An input can gain access to the decoders that are available by calling *PluginHelper.DecoderRunner*, which can be used to access decoders by the name they have been registered as in the config. Each call to *PluginHelper.DecoderRunner* will spin up a new decoder in its own goroutine. It's perfectly fine for an input to ask for multiple decoders; for instance the TcpInput creates one for each separate TCP connection. All decoders will be closed when Heka shuts down, but if a decoder will not longer be used (e.g. when a TCP connection is closed in the TcpInput example mentioned above) it's a good idea to call *PluginHelper.StopDecoderRunner* to shut it down or else it will continue to consume system resources throughout the life of the Heka process.

It is up to the input to decide which decoder should be used. Once the decoder has been determined and fetched from the *PluginHelper* the input can call *DecoderRunner.InChan()* to fetch a DecoderRunner's input channel upon which the *PipelinePack* can be placed.

Sometimes the input itself might wish to decode the data, rather than delegating that job to a separate decoder. In this case the input can directly populate the *pack.Message* and set the *pack.Decoded* value as *true*, as a decoder would do. Decoded messages are then injected into Heka's routing system by calling *InputRunner.Inject(pack)*. The message will then be delivered to the appropriate filter and output plugins.

One final important detail: if for any reason your input plugin should pull a *PipelinePack* off of the input channel and *not* end up passing it on to another step in the pipeline (i.e. to a decoder or to the router), you *must* call *PipelinePack.Recycle()* to free the pack up to be used again. Failure to do so will cause the *PipelinePack* pool to be depleted and will cause Heka to freeze.

2.9.7 Decoders

Decoder plugins are responsible for converting raw bytes containing message data into actual *Message* struct objects that the Heka pipeline can process. As with inputs, the *Decoder* interface is quite simple:

```
type Decoder interface {
    Decode(pack *PipelinePack) (packs []*PipelinePack, err error)
}
```

There are two optional Decoder interfaces. The first provides the Decoder access to its DecoderRunner object when it is started:

```
type WantsDecoderRunner interface {
    SetDecoderRunner(dr DecoderRunner)
}
```

The second provides a notification to the Decoder when the DecoderRunner is exiting:

```
type WantsDecoderRunnerShutdown interface {
    Shutdown()
}
```

A decoder's *Decode* method should extract the raw message data from *pack.MsgBytes* and attempt to deserialize this and use the contained information to populate the Message struct pointed to by the *pack.Message* attribute. Again, to minimize GC churn, take care to reuse the already allocated memory rather than creating new objects and overwriting the existing ones.

If the message bytes are decoded successfully then *Decode* should return a slice of PipelinePack pointers and a nil error value. The first item in the returned slice (i.e. *packs[0]*) should be the pack that was passed in to the method. If the decoding process produces more than one output pack, additonal packs can be appended to the slice.

If decoding fails for any reason, then *Decode* should return a nil value for the PipelinePack slice, causing the message to be dropped with no further processing. Returning an appropriate error value will cause Heka to log an error message about the decoding failure.

2.9.8 Filters

Filter plugins are the message processing engine of the Heka system. They are used to examine and process message contents, and trigger events based on those contents in real time as messages are flowing through the Heka system.

The filter plugin interface is just a single method:

```
type Filter interface {
    Run(r FilterRunner, h PluginHelper) (err error)
}
```

Like input plugins, filters have a *Run* method which accepts a runner and a helper, and which should not return until shutdown unless there's an error condition. And like input plugins, filters should call *runner.InChan()* to gain access to the plugin's input channel.

The similarities end there, however. A filter's input channel provides pointers to *PipelinePack* objects, defined in pipeline_runner.go

The *Pack* contains a fully decoded *Message* object from which the filter can extract any desired information.

Upon processing a message, a filter plugin can perform any of three tasks:

1. Pass the original message through unchanged to one or more specific alternative Heka filter or output plugins.
2. Generate one or more *new* messages, which can be passed to either a specific set of Heka plugins, or which can be handed back to the router to be checked against all registered plugins' *message_matcher* rules.
3. Nothing (e.g. when performing counting / aggregation / roll-ups).

To pass a message through unchanged, a filter can call *PluginHelper.Filter()* or *PluginHelper.Output()* to access a filter or output plugin, and then call that plugin's *Deliver()* method, passing in the *PipelinePack*.

To generate new messages, your filter must call *PluginHelper.PipelinePack(msgLoopCount int)*. The *msgloopCount* value to be passed in should be obtained from the *MsgLoopCount* value on the *PipelinePack* that you're already holding, or possibly zero if the new message is being triggered by a timed ticker instead of an incoming message. The *PipelinePack* method will either return a pack ready for you to populate or *nil* if the loop count is greater than the configured maximum value, as a safeguard against inadvertently creating infinite message loops.

Once a *PipelinePack* has been obtained, a filter plugin can populate its *Message* object. The pack can then be passed along to a specific plugin (or plugins) as above. Alternatively, the pack can be injected into the Heka message router queue, where it will be checked against all plugin message matchers, by passing it to the *FilterRunner.Inject(pack *PipelinePack)* method. Note that, again as a precaution against message looping, a plugin will not be allowed to inject a message which would get a positive response from that plugin's own matcher.

Sometimes a filter will take a specific action triggered by a single incoming message. There are many cases, however, when a filter is merely collecting or aggregating data from the incoming messages, and instead will be sending out reports on the data that has been collected at specific intervals. Heka has built-in support for this use case. Any filter (or output) plugin can include a *ticker_interval* config setting (in seconds, integers only), which will automatically be extracted by Heka when the configuration is loaded. Then from within your plugin code you can call *FilterRunner.Ticker()* and you will get a channel (type *<-chan time.Time*) that will send a tick at the specified interval. Your plugin code can listen on the ticker channel and take action as needed.

Observant readers might have noticed that, unlike the *Input* interface, filters don't need to implement a *Stop* method. Instead, Heka will communicate a shutdown event to filter plugins by closing the input channel from which the filter is receiving the *PipelinePack* objects. When this channel is closed, a filter should perform any necessary clean-up and then return from the *Run* method with a *nil* value to indicate a clean exit.

Finally, there is one very important point that all authors of filter plugins should keep in mind: if you are *not* passing your received *PipelinePack* object on to another filter or output plugin for further processing, then you *must* call *PipelinePack.Recycle()* to tell Heka that you are through with the pack. Failure to do so will cause Heka to not free up the packs for reuse, exhausting the supply and eventually causing the entire pipeline to freeze.

2.9.9 Encoders

Encoder plugins are the inverse of decoders. They convert *Message* structs into raw bytes that can be delivered to the outside world. Some encoders will serialize an entire *Message* struct, such as the *ProtobufEncoder* which uses Heka's native protocol buffers format. Other encoders extract data from the message and insert it into a different format such as plain text or JSON.

The *Encoder* interface consists of one method:

```
Encode(pack *PipelinePack) (output []byte, err error)
```

This method accepts a PiplelinePack containing a populated message object and returns a byte slice containing the data that should be sent out, or an error if serialization fails for some reason.

Unlike the other plugin types, encoders don't have a PluginRunner, nor do they run in their own goroutines. Outputs invoke encoders directly, by calling the Encode method exposed on the OutputRunner. This has the same signature as the Encoder interface's Encode method, to which it will will delegate. If *use_framing* is set to true in the output's configuration, however, the OutputRunner will prepend Heka's *Stream Framing* to the generated binary data.

Outputs can also directly access their encoder instance by calling OutputRunner.Encoder(). Encoders themselves don't handle the stream framing, however, so it is recommended that outputs use the OutputRunner method instead.

Even though encoders don't run in their own goroutines, it is possible that they might need to perform some clean up at shutdown time. If this is so, the encoder can implement the *NeedsStopping* interface:

```
Stop()
```

And the *Stop* method will be called during the shutdown sequence.

2.9.10 Outputs

Finally we come to the output plugins, which are responsible for receiving Heka messages and using them to generate interactions with the outside world. The *Output* interface is nearly identical to the *Filter* interface:

```
type Output interface {
    Run(or OutputRunner, h PluginHelper) (err error)
}
```

In fact, there are many ways in which filter and output plugins are similar. Like filters, outputs should call the *InChan* method on the provided runner to get an input channel, which will feed *PipelinePack* objects. Like filters, outputs should listen on this channel until it is closed, at which time they should perform any necessary clean-up and then return. And, like filters, any output plugin with a *ticker_interval* value in the configuration will use that value to create a ticker channel that can be accessed using the runner's *Ticker* method. And, finally, outputs should also be sure to call *PipelinePack.Recycle()* when they finish w/ a pack so that Heka knows the pack is freed up for reuse.

The primary way that outputs differ from filters, of course, is that outputs need to serialize (or extract data from) the messages they receive and then send that data to an external destination. The serialization (or data extraction) should typically be performed by the output's specified encoder plugin. The OutputRunner exposes the following methods to assist with this:

```
Encode(pack *PipelinePack) (output []byte, err error)
UsesFraming() bool
Encoder() (encoder Encoder)
```

The Encode method will use the specified encoder to convert the pack's message to binary data, then if *use_framing* was set to true in the output's configuration it will prepend Heka's *Stream Framing*. The UsesFraming method will tell you whether or not *use_framing* was set to true. Finally, the Encoder method will return the actual encoder that was registered. This is useful to check to make sure that an encoder was actually registered, but generally you will want to use OutputRunner.Encode and not Encoder.Encode, since the latter will not honor the output's *use_framing* specification.

2.9.11 Registering Your Plugin

The last step you have to take after implementing your plugin is to register it with *hekad* so it can actually be configured and used. You do this by calling the *pipeline* package's *RegisterPlugin* function:

```
func RegisterPlugin(name string, factory func() interface{})
```

The *name* value should be a unique identifier for your plugin, and it should end in one of "Input", "Decoder", "Filter", or "Output", depending on the plugin type.

The *factory* value should be a function that returns an instance of your plugin, usually a pointer to a struct, where the pointer type implements the *Plugin* interface and the interface appropriate to its type (i.e. *Input*, *Decoder*, *Filter*, or *Output*).

This sounds more complicated than it is. Here are some examples from Heka itself:

```
RegisterPlugin("UdpInput", func() interface{} {return new(UdpInput)})
RegisterPlugin("TcpInput", func() interface{} {return new(TcpInput)})
RegisterPlugin("ProtobufDecoder", func() interface{} {return new(ProtobufDecoder)})
RegisterPlugin("CounterFilter", func() interface{} {return new(CounterFilter)})
RegisterPlugin("StatFilter", func() interface{} {return new(StatFilter)})
RegisterPlugin("LogOutput", func() interface{} {return new(LogOutput)})
RegisterPlugin("FileOutput", func() interface{} {return new(FileOutput)})
```

It is recommended that *RegisterPlugin* calls be put in your Go package's init() function so that you can simply import your package when building *hekad* and the package's plugins will be registered and available for use in your Heka config file. This is made a bit easier if you use *plugin_loader.cmake*, see *Building hekad with External Plugins*.

2.10 Heka Message

2.10.1 Message Variables

- uuid (required, []byte) - 16 byte array containing a type 4 UUID.
- timestamp (required, int64) - Number of nanoseconds since the UNIX epoch.
- type (optional, string) - Type of message i.e. "WebLog".
- logger (optional, string) - Data source i.e. "Apache", "TCPInput", "/var/log/test.log".
- severity (optional, int32) - Syslog severity level.
- payload (optional, string) - Textual data i.e. log line, filename.
- env_version (optional, string) - Unused, legacy envelope version.
- pid (optional, int32) - Process ID that generated the message.
- hostname (optional, string) - Hostname that generated the message.
- fields (optional, Field) - Array of Field structures.

2.10.2 Field Variables

- name (required, string) - Name of the field (key).
- **value_type (optional, int32) - Type of the value stored in this field.**
    - STRING = 0 (default)
    - BYTES = 1
    - INTEGER = 2
    - DOUBLE = 3
    - BOOL = 4
- representation (optional, string) - Freeform metadata string where you can describe what the data in this field represents. This information might provide cues to assist with processing, labeling, or rendering of the data performed by downstream plugins or UI elements. Examples of common usage follow:
    - **Numeric value representation - In most cases it is the unit.**
        * count - It is a standard practice to use 'count' for raw values with no units.
        * KiB
        * mm
    - **String value representation - Ideally it should reference a formal specification but you are free to create you own voca**
        * date-time RFC 3339, section 5.6
        * email RFC 5322, section 3.4.1
        * hostname RFC 1034, section 3.1
        * ipv4 RFC 2673, section 3.2
        * ipv6 RFC 2373, section 2.2
        * uri RFC 3986
    - **How the representation is/can be used**
        * data parsing and validation
        * unit conversion i.e., B to KiB
        * presentation i.e., graph labels, HTML links
- value_* (optional, value_type) - Array of values, only one type will be active at a time.

2.10.3 Stream Framing

Heka has some custom framing that can be used to delimit records when generating a stream of binary data. The entire structure encapsulating a single message consists of a one byte record separator, one byte representing the header length, a protobuf encoded message header, a one byte unit separator, and the binary record content (usually a protobuf encoded Heka message). This message structure is indicated in this diagram:

Record Separator (byte=0x1E)	Header Length (byte)	Header (protocol buffer)	Unit Separator (byte=0x1F)	Message

The header schema is as follows:

- message_length (required, uint32) - length in bytes of the serialized message data
- hmac_hash_function (optional, int32) - enum indicating the hash function used to sign the message, 0 for MD5, 1, for SHA1
- hmac_signer (optional, string) - string token identifying HMAC signer
- hmac_key_version (optional, uint32) - version number of the provided HMAC key
- hmac (optional, []byte) - binary representation of provided HMAC key

Clients interested in decoding a Heka stream will need to read the header length byte to determine the length of the header, extract the encoded header data and decode this into a Header structure using an appropriate protobuf library. From this they can then extract the length of the encoded message data, which can then be extracted from the data stream and processed and/or decoded as needed.

2.11 Message Matcher Syntax

Message matching is done by the *hekad* router to choose an appropriate filter(s) to run. Every filter that matches will get a copy of the message.

2.11.1 Examples

- Type == “test” && Severity == 6
- (Severity == 7 || Payload == “Test Payload”) && Type == “test”
- Fields[foo] != “bar”
- Fields[foo][1][0] == ‘alternate’
- Fields[MyBool] == TRUE
- TRUE
- Fields[created] =~ /%TIMESTAMP%/

2.11.2 Relational Operators

- == equals
- != not equals
- > greater than
- >= greater than equals

- **<** less than
- **<=** less than equals
- **=~** regular expression match
- **!~** regular expression negated match

2.11.3 Logical Operators

- Parentheses are used for grouping expressions
- **&&** and (higher precedence)
- || or

2.11.4 Boolean

- **TRUE**
- **FALSE**

2.11.5 Message Variables

- All message variables must be on the left hand side of the relational comparison
- **String**
  - **Uuid**
  - **Type**
  - **Logger**
  - **Payload**
  - **EnvVersion**
  - **Hostname**
- **Numeric**
  - **Timestamp**
  - **Severity**
  - **Pid**
- **Fields**
  - **Fields[_field_name_]** (shorthand for Field[_field_name_][0][0])
  - **Fields[_field_name_][_field_index_]** (shorthand for Field[_field_name_][_field_index_][0])
  - **Fields[_field_name_][_field_index_][_array_index_]**
  - If a field type is mis-match for the relational comparison, false will be returned i.e. Fields[foo] == 6 where ‘foo’ is a string

2.11.6 Quoted String

- single or double quoted strings are allowed
- must be placed on the right side of a relational comparison i.e. Type == 'test'

2.11.7 Regular Expression String

- enclosed by forward slashes
- must be placed on the right side of the relational comparison i.e. Type =~ /test/
- capture groups will be ignored

Regular Expression Helpers

Commonly used complex regular expressions are provide as template variables in the form of %TEMPLATE%.

i.e., Fields[created] =~ /%TIMESTAMP%/

Available templates - TIMESTAMP - matches most common date/time string formats

See also:

Regular Expression re2 syntax

2.12 Sandbox

Sandboxes are Heka plugins that are implemented in a sandboxed scripting language. They provide a dynamic and isolated execution environment for data parsing, transformation, and analysis. They allow real time access to data in production without jeopardizing the integrity or performance of the monitoring infrastructure and do not require Heka to be recompiled. This broadens the audience that the data can be exposed to and facilitates new uses of the data (i.e. debugging, monitoring, dynamic provisioning, SLA analysis, intrusion detection, ad-hoc reporting, etc.)

2.12.1 Features

- **dynamic loading**
  - SandboxFilters can be started/stopped on a self-service basis while Heka is running
  - SandboxDecoder can only be started/stopped on a Heka restart but no recompilation is required to add new functionality.
- small - memory requirements are about 16 KiB for a basic sandbox
- fast - microsecond execution times
- stateful - ability to resume where it left off after a restart/reboot
- isolated - failures are contained and malfunctioning sandboxes are terminated

2.12.2 Lua Sandbox

The *Lua* sandbox provides full access to the Lua language in a sandboxed environment under *hekad* that enforces configurable restrictions.

See also:

Lua Reference Manual

API

Functions that must be exposed from the Lua sandbox

int process_message() Called by Heka when a message is available to the sandbox. The instruction_limit configuration parameter is applied to this function call.

Arguments none

Return

- < 0 for non-fatal failure (increments ProcessMessageFailures)
- 0 for success
- > 0 for fatal error (terminates the sandbox)

timer_event(ns) Called by Heka when the ticker_interval expires. The instruction_limit configuration parameter is applied to this function call. This function is only required in SandboxFilters (SandboxDecoders do not support timer events).

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch

Return none

Core functions that are exposed to the Lua sandbox

See: https://github.com/mozilla-services/lua_sandbox/blob/master/docs/sandbox_api.md

require(libraryName)

add_to_payload(arg1, arg2, ...argN) Appends the arguments to the payload buffer for incremental construction of the final payload output (inject_payload finalizes the buffer and sends the message to Heka). This function is simply a rename of the generic sandbox *output* function to improve the readability of the plugin code.

Arguments

- arg (number, string, bool, nil, circular_buffer)

Return none

Heka specific functions that are exposed to the Lua sandbox

read_config(variableName) Provides access to the sandbox configuration variables.

Arguments

- variableName (string)

Return number, string, bool, nil depending on the type of variable requested

read_message(variableName, fieldIndex, arrayIndex) Provides access to the Heka message data.

Arguments

- **variableName (string)**
  - raw (accesses the raw MsgBytes in the PipelinePack)
  - Uuid
  - Type
  - Logger
  - Payload
  - EnvVersion
  - Hostname
  - Timestamp
  - Severity
  - Pid
  - Fields[_name_]
- **fieldIndex (unsigned) only used in combination with the Fields variableName**
  - use to retrieve a specific instance of a repeated field _name_
- **arrayIndex (unsigned) only used in combination with the Fields variableName**
  - use to retrieve a specific element out of a field containing an array

Return number, string, bool, nil depending on the type of variable requested

write_message(variableName, value, representation, fieldIndex, arrayIndex) New in version 0.5.

Decoders only. Mutates specified field value on the message that is being deocded.

Arguments

- **variableName (string)**
  - Uuid (accepts raw bytes or RFC4122 string representation)
  - Type (string)
  - Logger (string)
  - Payload (string)
  - EnvVersion (string)
  - Hostname (string)
  - **Timestamp (accepts Unix ns-since-epoch number or a handful of** parseable string representations.)
  - Severity (number or int-parseable string)
  - Pid (number or int-parseable string)
  - Fields[_name_] (field type determined by value type: bool, number, or string)
- **value (bool, number or string)**
  - value to which field should be set

- **representation (string) only used in combination with the Fields variableName**
  - representation tag to set
- **fieldIndex (unsigned) only used in combination with the Fields variableName**
  - use to set a specfic instance of a repeated field _name_
- **arrayIndex (unsigned) only used in combination with the Fields variableName**
  - use to set a specific element of a field containing an array

Return none

read_next_field() Iterates through the message fields returning the field contents or nil when the end is reached.

Arguments none

Return value_type, name, value, representation, count (number of items in the field array)

inject_payload(payload_type, payload_name, arg3, ..., argN)

Creates a new Heka message using the contents of the payload buffer (pre-populated with *add_to_payload*) combined with any additional payload_args passed to this function. The output buffer is cleared after the injection. The payload_type and payload_name arguments are two pieces of optional metadata. If specified, they will be included as fields in the injected message e.g., Fields[payload_type] == 'csv', Fields[payload_name] == 'Android Usage Statistics'. The number of messages that may be injected by the process_message or timer_event functions are globally controlled by the hekad *global configuration options*; if these values are exceeded the sandbox will be terminated.

Arguments

- payload_type (**optional, default "txt"** string) Describes the content type of the injected payload data.
- payload_name (**optional, default ""** string) Names the content to aid in downstream filtering.
- arg3 (**optional) Same type restrictions as add_to_payload.

... - argN

Return none

inject_message(message_table) Creates a new Heka protocol buffer message using the contents of the specified Lua table (overwriting whatever is in the output buffer). Notes about message fields:

- Timestamp is automatically generated if one is not provided. Nanosecond since the UNIX epoch is the only valid format.
- UUID is automatically generated, anything provided by the user is ignored.
- Hostname and Logger are automatically set by the SandboxFilter and cannot be overridden.
- Type is prepended with "heka.sandbox." by the SandboxFilter to avoid data confusion/mis-representation.
- **Fields can be represented in multiple forms and support the following primitive types: string, double, bool. These co**
  - name=value i.e., foo="bar"; foo=1; foo=true
  - name={array} i.e., foo={"b", "a", "r"}
  - **name={object} i.e. foo={value=1, representation="s"}; foo={value={1010, 2200, 1567}, representation="ms"}**
    - value (required) may be a single value or an array of values
    - representation (optional) metadata for display and unit management

Arguments

- message_table A table with the proper message structure.

Return none

Notes

- injection limits are enforced as described above

Sample Lua Message Structure

```
{
Uuid        = "data",                 -- always ignored
Logger      = "nginx",                -- ignored in the SandboxFilter
Hostname    = "bogus.mozilla.com",    -- ignored in the SandboxFilter

Timestamp   = 1e9,
Type        = "TEST",                 -- will become "heka.sandbox.TEST" in the SandboxFilter
Paplod      = "Test Payload",
EnvVersion  = "0.8",
Pid         = 1234,
Severity    = 6,
Fields      = {
            http_status     = 200,
            request_size    = {value=1413, representation="B"}
            }
}
```

2.12.3 Lua Sandbox Tutorial

How to create a simple sandbox filter

1. Implement the required Heka interface in Lua

```
function process_message ()
    return 0
end

function timer_event(ns)
end
```

2. Add the business logic (count the number of ‘demo’ events per minute)

```
require "string"

total = 0 -- preserved between restarts since it is in global scope
local count = 0 -- local scope so this will not be preserved

function process_message()
    total= total + 1
    count = count + 1
    return 0
end

function timer_event(ns)
    count = 0
    inject_payload("txt", "",
                   string.format("%d messages in the last minute; total=%d", count, total))
end
```

3. Setup the configuration

```
[demo_counter]
type = "SandboxFilter"
message_matcher = "Type == 'demo'"
ticker_interval = 60
filename = "counter.lua"
preserve_data = true
```

4. Extending the business logic (count the number of 'demo' events per minute per device)

```
require "string"

device_counters = {}

function process_message()
    local device_name = read_message("Fields[DeviceName]")
    if device_name == nil then
        device_name = "_unknown_"
    end

    local dc = device_counters[device_name]
    if dc == nil then
        dc = {count = 1, total = 1}
        device_counters[device_name] = dc
    else
        dc.count = dc.count + 1
        dc.total = dc.total + 1
    end
    return 0
end

function timer_event(ns)
    add_to_payload("#device_name\tcount\ttotal\n")
    for k, v in pairs(device_counters) do
        add_to_payload(string.format("%s\t%d\t%d\n", k, v.count, v.total))
        v.count = 0
    end
    inject_payload()
end
```

2.12.4 SandboxManagerFilter

The SandboxManagerFilter provides dynamic control (start/stop) of sandbox filters in a secure manner without stopping the Heka daemon. Commands are sent to a SandboxManagerFilter using a signed Heka message. The intent is to have one manager per access control group each with their own message signing key. Users in each group can submit a signed control message to manage any filters running under the associated manager. A signed message is not an enforced requirement but it is highly recommended in order to restrict access to this functionality.

SandboxManagerFilter Settings

- *Common Filter Parameters*

- **working_directory (string):** The directory where the filter configurations, code, and states are preserved. The directory can be unique or shared between sandbox managers since the filter names are unique per manager. Defaults to a directory in ${BASE_DIR}/sbxmgrs with a name generated from the plugin name.
- **module_directory (string):** The directory where 'require' will attempt to load the external Lua modules from. Defaults to ${SHARE_DIR}/lua_modules.
- **max_filters (uint):** The maximum number of filters this manager can run.

New in version 0.5.

- **memory_limit (uint):** The number of bytes managed sandboxes are allowed to consume before being terminated (max 8MiB, default max).
- **instruction_limit (uint):** The number of instructions managed sandboxes are allowed the execute during the process_message/timer_event functions before being terminated (max 1M, default max).
- **output_limit (uint):** The number of bytes managed sandbox output buffers can hold before before being terminated (max 63KiB, default max). Anything less than 64B is set to 64B.

Example

```
[OpsSandboxManager]
type = "SandboxManagerFilter"
message_signer = "ops"
# message_matcher = "Type == 'heka.control.sandbox'" # automatic default setting
max_filters = 100
```

Control Message

The sandbox manager control message is a regular Heka message with the following variables set to the specified values.

Starting a SandboxFilter

- Type: "heka.control.sandbox"
- Payload: *sandbox code*
- Fields[action]: "load"
- Fields[config]: the TOML configuration for the *SandboxFilter*

Stopping a SandboxFilter

- Type: "heka.control.sandbox"
- Fields[action]: "unload"
- Fields[name]: The SandboxFilter name specified in the configuration

heka-sbmgr

Heka Sbmgr is a tool for managing (starting/stopping) sandbox filters by generating the control messages defined above.

Command Line Options

heka-sbmgr [`-config` *config_file*] [`-action` *load|unload*] [`-filtername` *specified on unload*] [`-script` *sandbox script filename*] [`-scriptconfig` *sandbox script configuration filename*]

Configuration Variables

- ip_address (string): IP address of the Heka server.
- use_tls (bool): Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.
- **signer (object): Signer information for the encoder.**
  - name (string): The name of the signer.
  - hmac_hash (string): md5 or sha1
  - hmac_key (string): The key the message will be signed with.
  - version (int): The version number of the hmac_key.
- tls (TlsConfig): A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

Example

```
ip_address       = "127.0.0.1:5565"
use_tls          = true
[signer]
    name         = "test"
    hmac_hash    = "md5"
    hmac_key     = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
    version      = 0

[tls]
    cert_file = "heka.crt"
    key_file = "heka.key"
    client_auth = "NoClientCert"
    prefer_server_ciphers = true
    min_version = "TLS11"
```

heka-sbmgrload

Heka Sbmgrload is a test tool for starting/stopping a large number of sandboxes. The script and configuration are built into the tool and the filters will be named: CounterSandbox**N** where **N** is the instance number.

Command Line Options

heka-sbmgrload [`-config` *config_file*] [`-action` *load|unload*] [`-num` *number of sandbox instances*]

Configuration Variables (same as heka-sbmgr)

2.12.5 Tutorial - How to use the dynamic sandboxes

SandboxManager/Heka Setup

1. Configure the SandboxManagerFilter.

The SandboxManagerFilters are defined in the hekad configuration file and are created when hekad starts. The manager provides a location/namespace for SandboxFilters to run and controls access to this space via a signed Heka message. By associating a message_signer with the manager we can restrict who can load and unload the associated filters. Lets start by configuring a SandboxManager for a specific set of users; platform developers. Choose a unique filter name [PlatformDevs] and a signer name "PlatformDevs", in this case we will use the same name for each.

```
[PlatformDevs]
type = "SandboxManagerFilter"
message_signer = "PlatformDevs"
working_directory = "/var/heka/sandbox"
max_filters = 100
```

2. Configure the input that will receive the SandboxManager control messages.

For this setup we will extend the current TCP input to handle our signed messages. The signer section consists of the signer name followed by an underscore and the key version number (the reason for this notation is to simply flatten the signer configuration structure into a single map). Multiple key versions are allowed to be active at the same time facilitating the rollout of new keys.

```
[TCP:5565]
type = "TcpInput"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
address = ":5565"
    [TCP:5565.signer.PlatformDevs_0]
    hmac_key = "Old Platform devs signing key"
    [TCP:5565.signer.PlatformDevs_1]
    hmac_key = "Platform devs signing key"
```

3. Configure the sandbox manager utility (sbmgr). The signer information must exactly match the values in the input configuration above otherwise the messages will be discarded. Save the file as PlatformDevs.toml.

```
ip_address       = ":5565"
[signer]
    name         = "PlatformDevs"
    hmac_hash    = "md5"
    hmac_key     = "Platform devs signing key"
    version      = 1
```

SandboxFilter Setup

1. Create a SandboxFilter script and save it as "example.lua". See *lua_tutorials* for more detail.

```
require "circular_buffer"

data = circular_buffer.new(1440, 1, 60) -- message count per minute
local COUNT = data:set_header(1, "Messages", "count")
function process_message ()
    local ts = read_message("Timestamp")
    data:add(ts, COUNT, 1)
    return 0
end

function timer_event(ns)
    inject_payload("cbuf", "", data)
end
```

2. Create the SandboxFilter configuration and save it as "example.toml".

The only difference between a static and dynamic SandboxFilter configuration is the filename. In the dynamic configuration it can be left blank or left out entirely. The manager will assign the filter a unique system wide name, in this case, "PlatformDevs-Example".

```
[Example]
type = "SandboxFilter"
message_matcher = "Type == 'Widget'"
ticker_interval = 60
filename = ""
preserve_data = false
```

3. Load the filter using sbmgr.

```
sbmgr -action=load -config=PlatformDevs.toml -script=example.lua -scriptconfig=example.toml
```

If you are running the *DashboardOutput* the following links are available:

- Information about the running filters: http://localhost:4352/heka_report.html.
- Graphical Output (after 1 minute in this case): http://localhost:4352 /PlatformDevs-Example.html

Otherwise

- Information about the terminated filters: http://localhost:4352/heka_sandbox_termination.html.

Note: A running filter cannot be 'reloaded' it must be unloaded and loaded again. During the unload/load process some data can be missed and gaps will be created. In the future we hope to remedy this but for now it is a limitation of the dynamic sandbox.

4. Unload the filter using sbmgr.

```
sbmgr -action=unload -config=PlatformDevs.toml -filtername=Example
```

2.12.6 SandboxDecoder

The SandboxDecoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka. See *Sandbox*. Config:

- *config_common_sandbox_parameters*

Example

```
[sql_decoder]
type = "SandboxDecoder"
filename = "sql_decoder.lua"
```

2.12.7 Available Sandbox Decoders

Apache Access Log Decoder

Parses the Apache access logs based on the Apache 'LogFormat' configuration directive. The Apache format specifiers are mapped onto the Nginx variable names where applicable e.g. %a -> remote_addr. This allows generic web filters and outputs to work with any HTTP server input.

Config:

- **log_format (string)** The 'LogFormat' configuration directive from the apache2.conf. %t variables are converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://httpd.apache.org/docs/2.4/mod/mod_log_config.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value

- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/apache"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/apache_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# common log format
# log_format = '%h %l %u %t \"%r\" %>s %O'

# vhost_combined log format
# log_format = '%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# referer log format
# log_format = '%{Referer}i -> %U'
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:”remote_user” value_string:”-“
name:”http_x_forwarded_for” value_string:”-“
name:”http_referer” value_string:”-“

name:”body_bytes_sent” value_type:DOUBLE representation:”B” value_double:82
name:”remote_addr” value_string:”62.195.113.219” representation:”ipv4”
name:”status” value_type:DOUBLE value_double:200
name:”request” value_string:”GET /v1/recovery_email/status HTTP/1.1”
name:”user_agent_os” value_string:”FirefoxOS”
name:”user_agent_browser” value_string:”Firefox”
name:”user_agent_version” value_type:DOUBLE value_double:29

MySQL Slow Query Log Decoder

Parses and transforms the MySQL slow query logs. Use mariadb_slow_query.lua to parse the MariaDB variant of the MySQL slow query logs.

Config:

- **truncate_sql (int, optional, default nil)** Truncates the SQL payload to the specified number of bytes (not UTF-8 aware) and appends ”...”. If the value is nil no truncation is performed. A negative value will truncate the specified number of bytes from the end.

Example Heka Configuration

```
[Sync-1_5-SlowQuery]
type = "LogstreamerInput"
log_directory = "/var/log/mysql"
file_match = 'mysql-slow\.log'
parser_type = "regexp"
delimiter = "\n(# User@Host:)"
delimiter_location = "start"
decoder = "MySqlSlowQueryDecoder"

[MySqlSlowQueryDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/mysql_slow_query.lua"

    [MySqlSlowQueryDecoder.config]
    truncate_sql = 64
```

Example Heka Message

Timestamp 2014-05-07 15:51:28 -0700 PDT

Type mysql.slow-query

Hostname 127.0.0.1

Pid 0

UUID 5324dd93-47df-485b-a88e-429f0fcd57d6

Logger Sync-1_5-SlowQuery

Payload /* [queryName=FIND_ITEMS] */ SELECT bso.userid, bso.collection, ...

EnvVersion

Severity 7

Fields

name:”Rows_examined” value_type:DOUBLE value_double:16458
name:”Query_time” value_type:DOUBLE representation:”s” value_double:7.24966

name:"Rows_sent" value_type:DOUBLE value_double:5001
name:"Lock_time" value_type:DOUBLE representation:"s" value_double:0.047038

Nginx Access Log Decoder

Parses the Nginx access logs based on the Nginx 'log_format' configuration directive.

Config:

- **log_format (string)** The 'log_format' configuration directive from the nginx.conf. $time_local or $time_iso8601 variable is converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://nginx.org/en/docs/http/ngx_http_log_module.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

Nginx Error Log Decoder

Parses the Nginx error logs based on the Nginx hard coded internal format.

Config:

- **tz (string, optional, defaults to UTC)** The conversion actually happens on the Go side since there isn't good TZ support here.

Example Heka Configuration

```
[TestWebserverError]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'error\.log'
decoder = "NginxErrorDecoder"

[NginxErrorDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_error.lua"

[NginxErrorDecoder.config]
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type nginx.error

Hostname trink-x230

Pid 16842

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserverError

Payload using inherited sockets from "6;"

EnvVersion

Severity 5

Fields

name:"tid" value_type:DOUBLE value_double:0
name:"connection" value_type:DOUBLE value_double:8878

Rsyslog Decoder

Parses the rsyslog output using the string based configuration template.

Config:

- **template (string)** The 'template' configuration string from rsyslog.conf. http://rsyslog-5-8-6-doc.neocities.org/rsyslog_conf_templates.html
- **tz (string, optional, defaults to UTC)** If your rsyslog timestamp field in the template does not carry zone offset information, you may set an offset to be applied to your events here. Typically this would be used with the "Traditional" rsyslog formats.

  Parsing is done by Go, supports values of "UTC", "Local", or a location name corresponding to a file in the IANA Time Zone database, e.g. "America/New_York".

Example Heka Configuration

```
[RsyslogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/rsyslog.lua"

[RsyslogDecoder.config]
type = "RSYSLOG_TraditionalFileFormat"
template = '%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n'
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-02-10 12:58:58 -0800 PST

Type RSYSLOG_TraditionalFileFormat

Hostname trink-x230

Pid 0

UUID e0eef205-0b64-41e8-a307-5772b05e16c1

Logger RsyslogInput

Payload "imklog 5.8.6, log source = /proc/kmsg started."

EnvVersion

Severity 7

Fields

name:"programname" value_string:"kernel"]

2.12.8 Available Sandbox Modules

Alert Module

API

Stores the last alert time in the global *_LAST_ALERT* so alert throttling will persist between restarts.

queue(ns, msg) Queue an alert message to be sent.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.
- msg (string) alert payload.

Return

- true if the message is queued, false if it would be throttled.

send(ns, msg) Send an alert message.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.
- msg (string) alert payload.

Return

- true if the message is sent, false if it is throttled.

send_queue(ns) Sends all queued alert message as a single message.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.

Return

- true if the queued messages are sent, false if they are throttled.

set_throttle(ns_duration) Sets the minimum duration between alert event outputs.

Arguments

- ns_duration (int64) minimum duration in nanoseconds between alerts.

Return

- none

throttled(ns) Test to see if sending an alert at this time would be throttled.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.

Return

- true if a message would be throttled, false if it would be sent.

Note: Use a zero timestamp to override message throttling.

Annotation Module

API

add(name, ns, col, stext, text) Create an annotation in the global *_ANNOTATIONS* table.

Arguments

- name (string) circular buffer payload name.
- ns (int64) current time in nanoseconds since the UNIX epoch.
- col (uint) circular buffer column to annotate.
- stext (string) short text to display on the graph.

- text (string) long text to display in the rollover.

Return

- none

create(ns, col, stext, text) Helper function to create an annotation table but not add it to the global list of annotations.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.
- col (uint) circular buffer column to annotate.
- stext (string) short text to display on the graph.
- text (string) long text to display in the rollover.

Return

- annotation table

concat(name, annotations) Concatenates an array of annotation tables to the specified key in the global _ANNOTATIONS table.

Arguments

- name (string) circular buffer payload name.
- annotations (array) annotation tables.

Return

- none

prune(name, ns)

Arguments

- name (string) circular buffer payload name.
- ns (int64) current time in nanoseconds since the UNIX epoch.

Return

- The json encoded list of annotations.

remove(name) Entirely remove the payload name from the global *_ANNOTATIONS* table.

Arguments

- name (string) circular buffer payload name.

Return

- none

set_prune(name, ns_duration)

Arguments

- name (string) circular buffer payload name.
- ns_duration (int64) time in nanoseconds the annotation should remain in the list.

Return

- none

Anomaly Detection Module

API

parse_config(anomaly_config) Parses the anomaly_config into a Lua table. If the configuration is invalid an error is thrown.

Arguments

- anomaly_config (string or nil)

The configuration can specify any number of algorithm function calls (space delimited if desired, but they will also work back to back). If the payload name contains a double quote it should be escaped as two double quotes in a row.

Rate of change test

roc("payload name", col, win, hwin, sd, loss_of_data, start_of_data)

- **col (uint)** The circular buffer column to perform the analysis on.
- **win (uint)** The number of intervals in an analysis window.
- **hwin (uint)** The number of intervals in the historical analysis window (0 uses the full history). Must be greater than or equal to 'win'.
- **sd (double)** The standard deviation threshold to trigger the anomaly.
- **loss_of_data (bool)** Alert if data stops.
- **start_of_data (bool)** Alert if data starts.

e.g. roc("Output1", 1, 15, 0, 2, true, false)

Mann-Whitney-Wilcoxon test

mww("payload name", col, win, nwin, pvalue, trend)

- **col (uint)** The circular buffer column to perform the analysis on.
- **win (uint)** The number of intervals in an analysis window (should be at least 20).
- **nwin (uint)** The number of analysis windows to compare.
- **pvalue (double)** The pvalue threshold to trigger the prediction.
- **trend (string)** (decreasing|increasing|any)

e.g. mww("Output1", 2, 60, 10, 0.0001, decreasing)

mww_nonparametric("payload name", col, win, nwin, pstat)

- **col (uint)** The circular buffer column to perform the analysis on.
- **win (uint)** The number of intervals in an analysis window.
- **nwin (uint)** The number of analysis windows to compare.
- **pstat (double)** Value between 0 and 1. Anything above 0.5 is an increasing trend anything below 0.5 is a decreasing trend.

e.g. mww_nonparametric("Output1", 2, 15, 10, 0.55)

Return

- configuration table if parsing was successful or nil, if nil was passed in.

detect(ns, name, cbuf, anomaly_config) Detects anomalies in the circular buffer data returning any error messages for alert generation and array of annotations for the graph.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch. It used to advance the circular buffer if necessary (i.e., if no data is being received). The anomaly detection is always performed on the newest data (ignoring the current interval since it is incomplete).
- name (string) circular buffer payload name
- cbuf (userdata) circular buffer
- anomaly_config (table) returned from the parse() method

Return

- string if an anomaly was detected, otherwise nil.
- array of annotation tables

ElasticSearch Module

API

bulkapi_index_json(index, type_name, id, ns)

Returns a simple JSON 'index' structure satisfying the ElasticSearch BulkAPI

Arguments

- **index (string or nil)** String to use as the *_index* key's value in the generated JSON, or nil to omit the key. Supports field interpolation as described below.
- **type_name (string or nil)** String to use as the *_type* key's value in the generated JSON, or nil to omit the key. Supports field interpolation as described below.
- **id (string or nil)** String to use as the *_id* key' value in the generated JSON, or nil to omit the key. Supports field interpolation as described below.
- **ns (number or nil)** Nanosecond timestamp to use for any strftime field interpolation into the above fields. Current system time will be used if nil.

Field interpolation

Data from the current message can be interpolated into any of the string arguments listed above. A *%{}* enclosed field name will be replaced by the field value from the current message. Supported default field names are "Type", "Hostname", "Pid", "UUID", "Logger", "EnvVersion", and "Severity". Any other values will be checked against the defined dynamic message fields. If no field matches, then a C strftime (on non-Windows platforms) or C89 strftime (on Windows) time substitution will be attempted, using the nanosecond timestamp (if provided) or the system clock (if not).

Return

- JSON string suitable for use as ElasticSearch BulkAPI index directive.

2.12.9 Lua Parsing Expression Grammars (LPeg)

Best practices (using Lpeg in the sandbox)

1. Read the LPeg reference
2. **There are no plans to include the 're' module so embrace the SNOBOL tradition. Why?**
    - Consistency and readability of a single syntax.
    - Promotes more modular grammars.
    - Is easier to comment.
3. Do not use parentheses around function calls that take a single string argument.

```
-- prefer
lpeg.P"Literal"

-- instead of
lpeg.P("Literal")
```

4. When writing sub-grammars with an ordered choice (+) place each choice on its own line; this make it easier to pick out the alternates. Also, if possible order them from most frequent to least frequent use.

```
local date_month = lpeg.P"0" * lpeg.R"19"
                 + "1" * lpeg.R"02"

-- The exception: when grouping alternates together in a higher level grammar.

local log_grammar = (rfc3339 + iso8601) * log_severity * log_message
```

5. Use the locale patterns when matching standard character classes.

```
-- prefer
lpeg.digit

-- instead of
lpeg.R"09".
```

6. If a literal occurs within an expression avoid wrapping it in a function.

```
-- prefer
lpeg.digit * "Test"

-- instead of
lpeg.digit * lpeg.P"Test"
```

7. When creating a parser from an RFC standard mirror the ABNF grammar that is provided.
8. If creating a grammar that would also be useful to others, please consider contributing it back to the project, thanks.
9. Use the grammar tester http://lpeg.trink.com.

2.12.10 SandboxFilter

The sandbox filter provides an isolated execution environment for data analysis. Any output generated by the sandbox is injected into the payload of a new message for further processing or to be output.

Config:

- *Common Filter Parameters*
- *config_common_sandbox_parameters*

Example:

```
[hekabench_counter]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench'"
ticker_interval = 1
filename = "counter.lua"
preserve_data = true
profile = false

    [hekabench_counter.config]
    rows = 1440
    sec_per_row = 60
```

2.12.11 Available Sandbox Filters

Circular Buffer Delta Aggregator

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). The purpose is to recreate the view at a larger scope in each level of the aggregation i.e., host view -> datacenter view -> service level view.

Config:

- **enable_delta (bool, optional, default false)** Specifies whether or not this aggregator should generate cbuf deltas.
- **anomaly_config(string) - (see *sandbox_anomaly_module*)** A list of anomaly detection specifications. If not specified no anomaly detection/alerting will be performed.

Example Heka Configuration

```
[TelemetryServerMetricsAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsAggregator.config]
enable_delta = false
anomaly_config = 'roc("Request Statistics", 1, 15, 0, 1.5, true, false)'
```

Circular Buffer Delta Aggregator (by hostname)

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). Each column from the source circular buffer will become its own graph. i.e., 'Error Count' will become a graph with each host being represented in a column.

Config:

- **max_hosts (uint)** Pre-allocates the number of host columns in the graph(s). If the number of active hosts exceed this value, the plugin will terminate.

- **rows (uint)** The number of rows to keep from the original circular buffer. Storing all the data from all the hosts is not practical since you will most likely run into memory and output size restrictions (adjust the view down as necessary).
- **host_expiration (uint, optional, default 120 seconds)** The amount of time a host has to be inactive before it can be replaced by a new host.

Example Heka Configuration

```
[TelemetryServerMetricsHostAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_host_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsHostAggregator.config]
max_hosts = 5
rows = 60
host_expiration = 120
```

Frequent Items

Calculates the most frequent items in a data stream.

Config:

- **message_variable (string)** The message variable name containing the items to be counted.
- **max_items (uint, optional, default 1000)** The maximum size of the sample set (higher will produce a more accurate list).
- **min_output_weight (uint, optional, default 100)** Used to reduce the long tail output by only outputting the higher frequency items.
- **reset_days (uint, optional, default 1)** Resets the list after the specified number of days (on the UTC day boundary). A value of 0 will never reset the list.

Example Heka Configuration

```
[FxaAuthServerFrequentIP]
type = "SandboxFilter"
filename = "lua_filters/frequent_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerFrequentIP.config]
message_variable = "Fields[remote_addr]"
max_items = 10000
min_output_weight = 100
reset_days = 1
```

Heka Memory Statistics (self monitoring)

Graphs the Heka memory statistics using the heka.memstat message generated by pipeline/report.go.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *sandbox_anomaly_module*)

Example Heka Configuration

```
[HekaMemstat]
type = "SandboxFilter"
filename = "lua_filters/heka_memstat.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'heka.memstat'"
```

Heka Message Schema (Message Documentation)

Generates documentation for each unique message in a data stream. The output is a hierarchy of Logger, Type, EnvVersion, and a list of associated message field attributes including their counts (number in the brackets). This plugin is meant for data discovery/exploration and should not be left running on a production system.

Config:

<none>

Example Heka Configuration

```
[SyncMessageSchema]
type = "SandboxFilter"
filename = "lua_filters/heka_message_schema.lua"
ticker_interval = 60
preserve_data = false
message_matcher = "Logger =~ /^Sync/"
```

Example Output

```
Sync-1_5-Webserver [54600]
    slf [54600]
        -no version- [54600]
            upstream_response_time (mismatch)
            http_user_agent (string)
            body_bytes_sent (number)
            remote_addr (string)
            request (string)
            upstream_status (mismatch)
            status (number)
            request_time (number)
            request_length (number)
Sync-1_5-SlowQuery [37]
    mysql.slow-query [37]
        -no version- [37]
            Query_time (number)
```

Rows_examined (number)
Rows_sent (number)
Lock_time (number)

HTTP Status Graph

Graphs HTTP status codes using the numeric Fields[status] variable collected from web server access logs.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *sandbox_anomaly_module*)

Example Heka Configuration

```
[FxaAuthServerHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerHTTPStatus.config]
sec_per_row = 60
rows = 1440
anomaly_config = 'roc("HTTP Status", 1, 15, 0, 1.5, true, false)'
```

Unique Items

Counts the number of unique items per day e.g. active daily users by uid.

Config:

- **message_variable (string, required)** The Heka message variable containing the item to be counted.
- **title (string, optional, default "Estimated Unique Daily *message_variable*")** The graph title for the cbuf output.
- **enable_delta (bool, optional, default false)** Specifies whether or not this plugin should generate cbuf deltas. Deltas should be enabled when sharding is used; see: *Circular Buffer Delta Aggregator*.

Example Heka Configuration

```
[FxaActiveDailyUsers]
type = "SandboxFilter"
filename = "lua_filters/unique_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'FxaAuth' && Type == 'request.summary' && Fields[path] == '/v1/certifica

    [FxaActiveDailyUsers.config]
    message_variable = "Fields[uid]"
    title = "Estimated Active Daily Users"
```

2.12.12 SandboxEncoder

The SandboxEncoder provides an isolated execution environment for converting messages into binary data without the need to recompile Heka. See *Sandbox*. Config:

- *config_common_sandbox_parameters*

Example

```
[custom_json_encoder]
type = "SandboxEncoder"
filename = "path/to/custom_json_encoder.lua"

    [custom_json_encoder.config]
    msg_fields = ["field1", "field2"]
```

2.12.13 Available Sandbox Encoders

Alert Encoder

Produces more human readable alert messages.

Config:

<none>

Example Heka Configuration

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "((Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert') || Type == 'he
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"

[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

Example Output

Timestamp 2014-05-14T14:20:18Z

Hostname ip-10-226-204-51

Plugin FxaBrowserIdHTTPStatus

Alert HTTP Status - algorithm: roc col: 1 msg: detected anomaly, standard deviation exceeds 1.5

ESPayloadEncoder

Prepends ElasticSearch BulkAPI index JSON to a message payload.

Config:

- **index (string, optional, default "heka-%{%Y.%m.%d}")** String to use as the *_index* key's value in the generated JSON. Supports field interpolation as described below.
- **type_name (string, optional, default "message")** String to use as the *_type* key's value in the generated JSON. Supports field interpolation as described below.
- **id (string, optional)** String to use as the *_id* key's value in the generated JSON. Supports field interpolation as described below.
- **es_index_from_timestamp (boolean, optional)** If true, then any time interpolation (often used to generate the ElasticSeach index) will use the timestamp from the processed message rather than the system time.

Field interpolation:

> Data from the current message can be interpolated into any of the string arguments listed above. A *%{}* enclosed field name will be replaced by the field value from the current message. Supported default field names are "Type", "Hostname", "Pid", "UUID", "Logger", "EnvVersion", and "Severity". Any other values will be checked against the defined dynamic message fields. If no field matches, then a C strftime (on non-Windows platforms) or C89 strftime (on Windows) time substitution will be attempted.

Example Heka Configuration

```
[es_payload]
type = "SandboxEncoder"
filename = "lua_encoders/es_payload.lua"
    [es_payload.config]
    es_index_from_timestamp = true
    index = "%{Logger}-%{%Y.%m.%d}"
    type_name = "%{Type}-%{Hostname}"

[ElasticSearchOutput]
message_matcher = "Type == 'mytype'"
encoder = "es_payload"
```

Example Output

{"index":{"_index":"mylogger-2014.06.05","_type":"mytype-host.domain.com"}} {"json":"data","extracted":"from","message":"payl

2.12.14 Sandbox Development

Decoders

Since decoders cannot be dynamically loaded and they stop Heka processing on fatal errors they must be developed outside of a production enviroment. Most Lua decoders are LPeg based as it is the best way to parse and transform data within the sandbox. The other alternatives are the built-in Lua pattern matcher or the JSON parser with a manual transformation.

1. Procure some sample data to be used as test input.

   ```
   timestamp=time_t key1=data1 key2=data2
   ```

2. Configure a simple LogstreamerInput to deliver the data to your decoder.

   ```
   [LogstreamerInput]
   log_directory = "."
   file_match = 'data\.log'
   decoder = "SandboxDecoder"
   ```

3. Configure your test decoder.

```
[SandboxDecoder]
filename = "decoder.lua"
```

4. Configure the DasboardOutput for visibility into the decoder (performance, memory usage, messages processed/failed, etc.)

```
[DashboardOutput]
address = "127.0.0.1:4352"
ticker_interval = 10
working_directory = "dashboard"
static_directory = "/usr/share/heka/dasher"
```

5. Configure a LogOutput to display the generated messages.

```
[LogOutput]
message_matcher = "TRUE"
```

6. **Build the decoder.** The decoder will receive a message from an input plugin. The input may have set some additional message headers but the 'Payload' header contains the data for the decoder. The decoder can access the payload using read_message("Payload"). The payload can be used to construct an entirely new message, multiple messages or modify any part of the existing message (see inject_message, write_message in the *lua* API). Message headers not modified by the decoder are left intact and in the case of multiple message injections the initial message header values are duplicated for each message.

   (a) **LPeg grammar.** Incrementally build and test your grammar using http://lpeg.trink.com.

   (b) **Lua pattern matcher.** Test match expressions using http://www.lua.org/cgi-bin/demo.

   (c) **JSON parser.** For data transformation use the LPeg/Lua matcher links above. Something like simple field remapping i.e. *msg.Hostname = json.host* can be verified in the LogOutput.

7. Run Heka with the test configuration.

8. Inspect/verify the messages written by LogOutput.

Filters

Since filters can be dynamically loaded it is recommended you develop them in production with live data.

1. Read *sandbox_manager_tutorial*

OR

1. If you are developing the filter in conjunction with the decoder you can add it to the test configuration.

```
[SandboxFilter]
filename = "filter.lua"
```

2. Debugging

   (a) Watch for a dashboard sandbox termination report. The termination message provides the line number and cause of the failure. These are usually straight forward to correct and commonly caused by a syntax error in the script or invalid assumptions about the data (e.g. *cnt = cnt + read_message("Fields[counter]")* will fail if the counter field doesn't exist or is non-numeric due to a error in the data).

   (b) No termination report and the output does not match expectations. These are usually a little harder to debug.

      i. Check the Heka dasboard to make sure the router is sending messages to the plugin. If not, verify your message_matcher configuration.

ii. Visually review the the plugin for errors. Are the message field names correct, was the result of the cjson.decode tested, are the output variables actually being assigned to and output/injected, etc.

iii. Add a debug output message with the pertinent information.

```
require "string"
require "table"
local dbg = {}

-- table.insert(dbg, string.format("Entering function x arg1: %s", arg1))
-- table.insert(dbg, "Exiting function x")

inject_payload("txt", "debug", table.concat(dbg, "\n"))
```

i. LAST RESORT: Move the filter out of production, turn on preservation, run the tests, stop Heka, and review the entire preserved state of the filter.

2.12.15 Lua Sandbox Cookbooks

- **Decoders**
  - *json_payload_transform*
- **Presentation**
  - *graph_annotation*

2.13 Testing Heka

2.13.1 heka-flood

heka-flood is a Heka load test tool; it is capable of generating a large number of messages to exercise Heka using different protocols, message types, and error conditions.

Command Line Options

- -config=”flood.toml”: Path to heka-flood config file
- -test=”default”: Name of config file defined test to run

Example:

```
heka-flood -config="/etc/flood.toml" -test="my_test_name"
```

Configuration Variables

- test (object): Name of the test section (toml key) in the configuration file.
- ip_address (string): IP address of the Heka server.
- sender (string): tcp or udp
- pprof_file (string): The name of the file to save the profiling data to.

- encoder (string): protobuf or json
- num_messages (int): The number of messages to be sent, 0 for infinite.
- corrupt_percentage (float): The percentage of messages that will be randomly corrupted.
- signed_percentage (float): The percentage of message that will signed.
- variable_size_messages (bool): True, if a random selection of variable size messages are to be sent. False, if a single fixed message will be sent.
- **signer (object): Signer information for the encoder.**
  - name (string): The name of the signer.
  - hmac_hash (string): md5 or sha1
  - hmac_key (string): The key the message will be signed with.
  - version (int): The version number of the hmac_key.
- ascii_only (bool): True, if generated message payloads should only contain ASCII characters. False, if message payloads should contain arbitrary binary data. Defaults to false.

New in version 0.5.

- use_tls (bool): Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.
- tls (TlsConfig): A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

Example

```
[default]
ip_address             = "127.0.0.1:5565"
sender                 = "tcp"
pprof_file             = ""
encoder                = "protobuf"
num_messages           = 0
corrupt_percentage     = 0.0001
signed_percentage      = 0.00011
variable_size_messages = true
[default.signer]
    name               = "test"
    hmac_hash          = "md5"
    hmac_key           = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
    version             = 0
```

2.13.2 heka-inject

New in version 0.5.

heka-inject is a Heka client allowing for the injecting of arbitrary messages into the Heka pipeline. It is capable of generating a message of specified message variables with values. It allows for quickly testing plugins. Inject requires TcpInput with Protobufs encoder availability.

Command Line Options

- -heka: Heka instance to connect
- -hostname: message hostname

- -logger: message logger
- -payload: message payload
- -pid: message pid
- -severity: message severity
- -type: message type

Example:

```
heka-inject -payload="Test message with high severity." -severity=1
```

2.13.3 heka-cat

New in version 0.5.

A command-line utility for counting, viewing, filtering, and extracting Heka protobuf logs.

Command Line Options

- -format=”txt”: output format [txt|json|heka|count]
- -match=”TRUE”: message_matcher filter expression
- -offset=0: starting offset for the input file in bytes
- -output=””: output filename, defaults to stdout
- -tail=false: don’t exit on EOF
- *input filename*

Example:

```
heka-cat -format=count -match="Fields[status] == 404" test.log
```

Output:

```
Input:test.log  Offset:0  Match:Fields[status] == 404  Format:count  Tail:false  Output:
Processed: 1002646, matched: 15660 messages
```

2.14 Configuring TLS

Many input and output plugins that rely on TCP as the underlying transport for network communication also support the use of SSL/TLS encryption for their connections. Typically the TOML configuration for these plugins will support a boolean *use_tls* flag that specifies whether or not encryption should be used, and a *tls* sub-section that specifies the settings to be used for negotiating the TLS connections. If *use_tls* is not set to true, the *tls* section will be ignored.

Modeled after Go’s stdlib TLS configuration struct, the same configuration structure is used for both client and server connections, with some of the settings being applicable for a client’s configuration, some for a server’s, and some for both. In the description of the TLS configuration settings below, each setting is marked as appropriate to client, server, or both as appropriate.

2.14.1 TLS configuration settings

- **server_name (string, client):** Name of the server being requested. Included in the client handshake to support virtual hosting server environments.
- **cert_file (string, both):** Full filesystem path to the certificate file to be presented to the other side of the connection.
- **key_file (string, both):** Full filesystem path to the specified certificate's associated private key file.
- **client_auth (string, server):** Specifies the server's policy for TLS client authentication. Must be one of the following values:
  - NoClientCert
  - RequestClientCert
  - RequireAnyClientCert
  - VerifyClientCertIfGiven
  - RequireAndVerifyClientCert

  Defaults to "NoClientCert".
- **ciphers ([]string, both):** List of cipher suites supported for TLS connections. Earlier suites in the list have priority over those following. Must only contain values from the following selection:
  - RSA_WITH_RC4_128_SHA
  - RSA_WITH_3DES_EDE_CBC_SHA
  - RSA_WITH_AES_128_CBC_SHA
  - RSA_WITH_AES_256_CBC_SHA
  - ECDHE_ECDSA_WITH_RC4_128_SHA
  - ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  - ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  - ECDHE_RSA_WITH_RC4_128_SHA
  - ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  - ECDHE_RSA_WITH_AES_128_CBC_SHA
  - ECDHE_RSA_WITH_AES_256_CBC_SHA
  - ECDHE_RSA_WITH_AES_128_GCM_SHA256
  - ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

  If omitted, the implementation's default ordering will be used.
- **insecure_skip_verify (bool, client):** If true, TLS client connections will accept any certificate presented by the server and any host name in that certificate. This causes TLS to be susceptible to man-in-the-middle attacks and should only be used for testing. Defaults to false.
- **prefer_server_ciphers (bool, server):** If true, a server will always favor the server's specified cipher suite priority order over that requested by the client. Defaults to true.
- **session_tickets_disabled (bool, server):** If true, session resumption support as specified in RFC 5077 will be disabled.
- **session_ticket_key (string, server):** Used by the TLS server to provide session resumption per RFC 5077. If left empty, it will be filled with random data before the first server handshake.

- **min_version (string, both):** Specifies the mininum acceptable SSL/TLS version. Must be one of the following values:
  - SSL30
  - TLS10
  - TLS11
  - TLS12

  Defaults to SSL30.
- **max_version (string, both):** Specifies the maximum acceptable SSL/TLS version. Must be one of the following values:
  - SSL30
  - TLS10
  - TLS11
  - TLS12

  Defaults to TLS12.
- **client_cafile (string, server):** File for server to authenticate client TLS handshake. Any client certs recieved by server must be chained to a CA found in this PEM file.

  Has no effect when NoClientCert is set.
- **root_cafile (string, client):** File for client to authenticate server TLS handshake. Any server certs recieved by client must be must be chained to a CA found in this PEM file.

2.14.2 Sample TLS configuration

The following is a sample TcpInput configuration showing the use of TLS encryption.

```
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
use_tls = true

    [TcpInput.tls]
    cert_file = "/usr/share/heka/tls/cert.pem"
    key_file = "/usr/share/heka/tls/cert.key"
    client_auth = "RequireAndVerifyClientCert"
    prefer_server_ciphers = true
    min_version = "TLS11"
```

Indices and tables

- *search*
- *glossary*