Hortonworks Cybersecurity Package

Administration

(January 26, 2018)

docs.cloudera.com

Hortonworks Cybersecurity Package: Administration

Hortonworks Cybersecurity Package (HCP) is a modern data application based on Apache Metron, powered by Apache Hadoop, Apache Storm, and related technologies.

HCP provides a framework and tools to enable greater efficiency in Security Operation Centers (SOCs) along with better and faster threat detection in real-time at massive scale. It provides ingestion, parsing and normalization of fully enriched, contextualized data, threat intelligence feeds, triage and machine learning based detection. It also provides end user near real-time dashboards.

Based on a strong foundation in the Hortonworks Data Platform (HDP) and Hortonworks DataFlow (HDF) stacks, HCP provides an integrated advanced platform for security analytics.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to Contact Us directly to discuss your specific needs.

Table of Contents

List of Figures

List of Tables

1. HCP Information Roadmap

This roadmap contains additional information on Hortonworks Cybersecurity Package (HCP) and Apache Metron.

Table 1.1. HCP Additional Information

Information type	Resources
Overview	• Apache Metron Website (Source: Apache wiki)
Installing	• Ambari Install Guide (Source: Hortonworks) • Command Line Install Guide (Source: Hortonworks) • Ambari Upgrade Guide (Source: Hortonworks) • Command Line Upgrade Guide (Source: Hortonworks)
Administering	• Apache Metron Documentation (Source: Apache wiki)
Developing	• Community Resources (Source: Apache wiki)
Reference	• About Metron (Source: Apache wiki)
Resources for contributors	• How to Contribute (Source: Apache wiki)
Hortonworks Community Connection	• Hortonworks Community Connection for Metron (Source: Hortonworks)

2. Introduction to Hortonworks CyberSecurity Suite

This guide is intended for Platform Engineers responsible for installing, configuring, and maintaining Hortonworks CyberSecurity Package (HCP) powered by Apache Metron. This guide is divided into three major sections:

- Configuring and Customizing [6]
- Monitor and Management [85]
- Concepts [101]

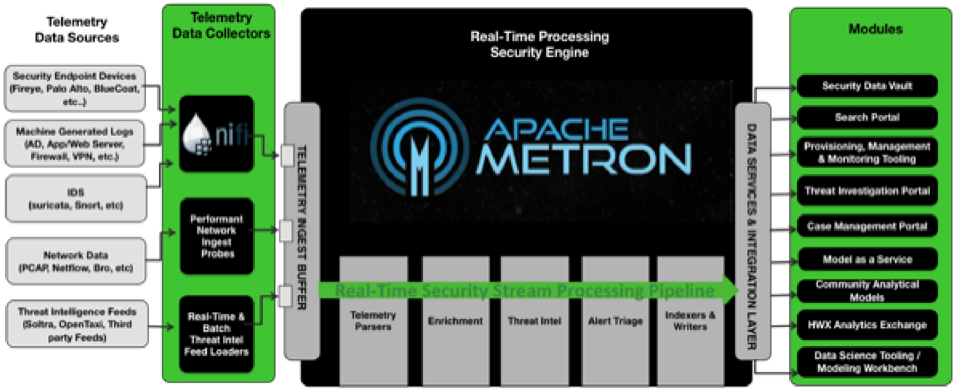
2.1. HCP Architecture

Hortonworks CyberSecurity Package (HCP) is a cybersecurity platform. It consists of the following components:

- Real-time Processing Security Engine [3]
- Telemetry Data Collectors [3]
- Data Services and Integration Layer [3]

Each of these components is described in the following sections.

Figure 2.1. HCP Architecture


Telemetry Data Sources
Security Endpoint Devices (Fireye, Palo Alto, BlueCoat, etc..)
Machine Generated Logs (AD, App/Web Server, Firewall, VPN, etc.)
IDS (suricata, Snort, etc)
Network Data (PCAP, Netflow, Bro, etc)
Threat Intelligence Feeds (Soltra, OpenTaxii, Third party Feeds)
Telemetry Data Collectors
nifi
Performant Network Ingest Probes
Real-Time & Batch Threat Intel Feed Loaders
TELEMETRY INGEST BUFFER
Real-Time Processing Security Engine
APACHE METRON
Real-Time Security Stream Processing Pipeline
Telemetry Parsers
Enrichment
Threat Intel
Alert Triage
Indexers & Writers
DATA SERVICES & INTEGRATION LAYER
Modules
Security Data Vault
Search Portal
Provisioning, Management & Monitoring Tooling
Threat Investigation Portal
Case Management Portal
Model as a Service
Community Analytical Models
HWX Analytics Exchange
Data Science Tooling / Modeling Workbench

The core of the HCP architecture is the Apache Metron real-time processing security engine. The data flow for HCP is performed in real-time and contains the following steps:

1. Information from telemetry data sources is ingested into Kafka topics. (Kafka is the telemetry event buffer.) A Kafka topic is created for every telemetry data source. This information is the raw telemetry data consisting of host logs, firewall logs, emails, and network data.
2. Once the information is ingested into Kafka topics, the data is parsed into a normalized JSON structure that Metron can read.

3. The information is then enriched with asset, geo, threat intelligence information, etc.

4. The information is then indexed, stored, and any resulting alerts are sent to the Metron dashboard, the Alerts user interface, as well as telemetry.

2.1.1. Real-time Processing Security Engine

The core of the HCP architecture is the Apache Metron real-time processing security engine. This component provides the ingest buffer to capture the raw events, and, in real time, parses the raw events, enriches the events with relevant contextual information, enriches the events with threat intelligence, and applies available models (such as triaging threats via the Stellar language), then writes the events to a searchable index as well as HDFS for after-the-fact analytics.

2.1.2. Telemetry Data Collectors

Telemetry data collectors push or stream the data source events into Apache Metron. HCP works with NiFi to push the majority of data sources into Apache Metron. For high-volume network data, HCP provides a performant network ingest probe. And for threat intelligence feeds, HCP supports a set of both streaming and batch loaders that enables you to push third-party intelligence feeds into Apache Metron.

2.1.3. Data Services and Integration Layer

This set of HCP modules provides different features for different SOC personas. HCP provides the following three modules for the integration layer:

Security data vault
Stores the data in HDFS.

Search portal
The Metron dashboard.

Provisioning, management, and monitoring tool
HCP provides a Management module that expedites provisioning and managing sensors. Other provisioning, management, and monitoring functions are supported through Ambari.

2.2. Understanding HCP Terminology

This section defines the key terminology associated with cybersecurity, Hadoop, and HCP:

Alert
Alerts provide information about currently security issues, vulnerabilities, and exploits.

Apache Kafka
Apache Kafka is a fast, scalable, durable, fault-tolerant publish-subscribe messaging system, that can be used for stream processing, messaging, website activity tracking, metrics collection and monitoring, log aggregation, and event sourcing.

Apache Storm
Apache Storm enables data-driven, automated activity by providing a real-time, scalable, fault-tolerant, highly available, distributed solution for streaming data.

Apache ZooKeeper	Apache ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.
Cybersecurity	The protection of information systems from theft or damage to the hardware, software, and to the information on them, as well as from disruption or misdirection of the services they provide.
Data management	A set of data management utilities aimed at getting data into HBase in a format which will allow data flowing through metron to be enriched with the results. Contains integrations with threat intelligence feeds exposed via TAXII as well as simple flat file structures.
Enrichment data source	A data source containing additional information about telemetry ingested by HCP.
Enrichment bolt	The storm bolt that applies the enrichment to the telemetry.
Enrichment data loader	A streaming or a batch loader that stages the data from the enrichment source into HCP so that telemetry can be enriched in real-time with the information from the enrichment source
Forensic Investigator	Collects evidence on breach and attack incidents and prepares legal responses to breaches.
Model as a Service	A Yarn application which can deploy machine learning and statistical models onto the cluster along with the associated Stellar functions to be able to call out to them in a scalable manner.
Parser	A storm bolt that transforms telemetry from its native format to a JSON that Metron is able to understand.
Profiler	A feature extraction mechanism that can generate a profile describing the behavior of an entity. An entity might be a server, user, subnet or application. Once a profile has been generated defining what normal behavior looks-like, models can be built that identify anomalous behavior.
Security Data Scientist	Works with security data, performing data munging, visualization, plotting, exploration, feature engineering, model creation. Evaluates and monitors the correctness and currency of existing models
Security Operations Center (SOC)	A centralized unit that deals with cybersecurity issues for an organization by monitoring, assessing, and defending against cybersecurity attacks.

Security Platform Engineer

Installs, configures, and maintains security tools. Performs capacity planning and upgrades. Establishes best practices and reference architecture with respect to provisioning, managing, and using the security tools. Maintains the probes to collect data, load enrichment data, and manage threat feeds.

SOC Analyst

Responsible for monitoring security information and event management (SIEM) tools; searching for and investigating breaches and malware, and reviewing alerts; escalating alerts when appropriate; and following security playbooks.

SOC Investigator

Responsible for investigating more complicated or escalated alerts and breaches, such as Advanced Persistent Threats (APT). Hunts for malware attacks. Removes or quarantines the malware, breach, or infected system.

Stellar

A custom data transformation language that is used throughout HCP from simple field transformation to expressing triage rules.

Telemetry data source

The source of telemetry data which can vary in level from low level (packet capture), intermediate level (deep packet analysis) or very high level (application logs).

Telemetry event

A single event in a stream of telemetry data. This can vary in level from low level (packet capture), intermediate level (deep packet analysis) or very high level (application logs).

3. Configuring and Customizing

One of the key design goals of Hortonworks Cybersecurity Package (HCP) powered by Apache Metron is that it should be easily extensible. HCP comes bundled with several telemetry data sources, enrichment topologies, and threat intelligence feeds. However, you might want to use HCP as a platform and build custom capabilities on top of it.

This chapter describes the following ways you can customize your HCP platform:

3.1. Adding a New Telemetry Data Source

This section describes how you add a new telemetry data source. Before HCP can process the information from a new telemetry data source, you must use one of the telemetry data collectors to ingest the information into the telemetry ingest buffer. Information moves from the data ingest buffer into the Metron real-time processing security engine, where it is parsed, enriched, triaged, and indexed. Finally, certain telemetry events can initiate alerts that can be assessed in the Metron dashboard.

To add a new telemetry data source, perform the following tasks:

1. Streaming Data into HCP [8]
2. Parsing a New Data Source to HCP [13]
3. Verifying That the Events Are Indexed [27]
4. For instructions on how to configure the Metron Dashboard to view the new data source telemetry events, see Hortonworks Cybersecurity User Guide.

The following sections provide steps for each task. You can perform these tasks by using the HCP Management module or CLI. Instructions are provided for both methods.

3.1.1. Prerequisites

Before you add a new telemetry device, you must perform the following actions:

- Install HDP and HDF, and then install HCP.

  For information about installing HCP, see the HCP Installation Guide.

- Ensure that the new sensor is installed and set up.

- Ensure that NiFi or another telemetry data collection tool can feed the telemetry data source events into a Kafka topic.

- Determine your requirements.

  For example, you might decide that you need to meet the following requirements:

  - Proxy events from the data source logs must be ingested in real-time.

  - Proxy logs must be parsed into a standardized JSON structure suitable for analysis by Metron.

  - In real-time, new data source proxy events must be enriched so that the domain names contain the IP information.

  - In real-time, the IP within the proxy event must be checked against for threat intelligence feeds.

  - If there is a threat intelligence hit, an alert must be raised.

  - The SOC analyst must be able to view new telemetry events and alerts from the new data source.

- Set HCP values

  When you install HCP, you will set up several hosts. You will need the locations of these hosts, along with port numbers, and the Metron version. These values are listed below.

  - KAFKA_HOST = The host where a Kafka broker is installed.

  - ZOOKEEPER_HOST = The host where a ZooKeeper server is installed.

  - PROBE_HOST = The host where your sensor, probes are installed. If don't have any sensors installed, pick the host where a Storm supervisor is running.

  - NIFI_HOST = Host where you will install NIFI.

  - HOST_WITH_ENRICHMENT_TAG = The host in your inventory hosts file that you put under the group "enrichment."

  - SEARCH_HOST = The host where you have Elastic or Solr running. This is the host in your inventory hosts file that you put under the group "search". Pick one of the search hosts.

- SEARCH_HOST_PORT = The port of the search host where indexing is configured. (For example, 9300).
- METRON_UI_HOST = The host where your Metron UI web application is running. This is the host in your inventory hosts file that you put under the group "web."
- METRON_VERSION = The release of the Metron binaries you are working with. (For example, HCP-1.4.0.0)

3.1.2. Streaming Data into HCP

The first step in adding a new data source telemetry is to stream all raw events from the telemetry data source into its own Kafka topic.

Note

Although HCP includes parsers for several data sources (for example, Bro, Snort, and YAF), you must still stream the raw data into HCP through a Kafka topic.

By default, the Snort parser is configured to use ZoneId.systemDefault() for the source `timeZone` for the incoming data and MM/dd/yy-HH:mm:ss.SSSSSS as the default `dateFormat`. Valid timezones are per Java's ZoneId.getAvailableZoneIds(). DateFormats should be valid per the options defined in https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html. Below is a sample configuration with the `dateFormat` and `timeZone` explicitly set in the parser config.

```
"parserConfig": {
"dateFormat" : "MM/dd/yy-HH:mm:ss.SSSSSS",
 "timeZone" : "America/New_York"
```

Note

When you install and configure Snort, you must configure Snort to include the year in the timestamp by modifying the `snort.conf` file as follows:

```
# Configure Snort to show year in timestamps
config show_year
```

This is important for the proper functioning of indexing and analytics.

Depending on the type of data you are streaming into HCP, you can use one of the following methods:

NiFi

This type of streaming method works for most types of data sources. For information on installing NiFi, see the NiFi documentation.

Important

NiFi cannot be installed on top of HDP, so you must install NiFi manually to use it with HCP.

Note

Ensure that the NiFi web application is using port 8089.

Performant network ingestion probes

This type of streaming method is ideal for streaming high volume packet data. See Setting up pcap to View Your Raw Data for more information.

Real-time and batch threat intelligence feed loaders

This type of streaming method is used for real-time and batch threat intelligence feed loaders. For more information see Using Threat Intelligence Feeds.

3.1.2.1. Creating a NiFi Flow to Stream Events to HCP

This section provides instructions to create a flow to capture events from the new data source and push them into HCP.

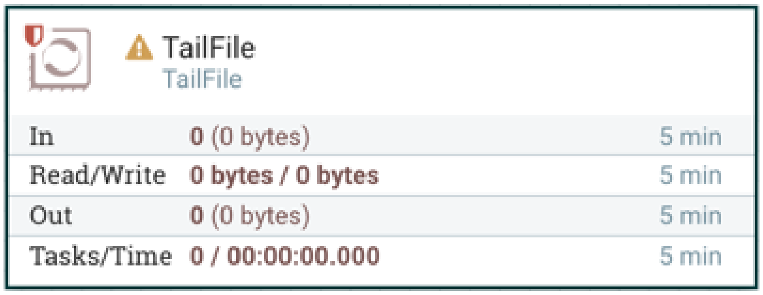
1. Drag the first icon on the toolbar (the processor icon) to your workspace.

   NiFi displays the Add Processor dialog box.

2. Select the TailFile type of processor and click **Add**.

   NiFi displays a new TailFile processor.

   Figure 3.1. New TailFile Processor

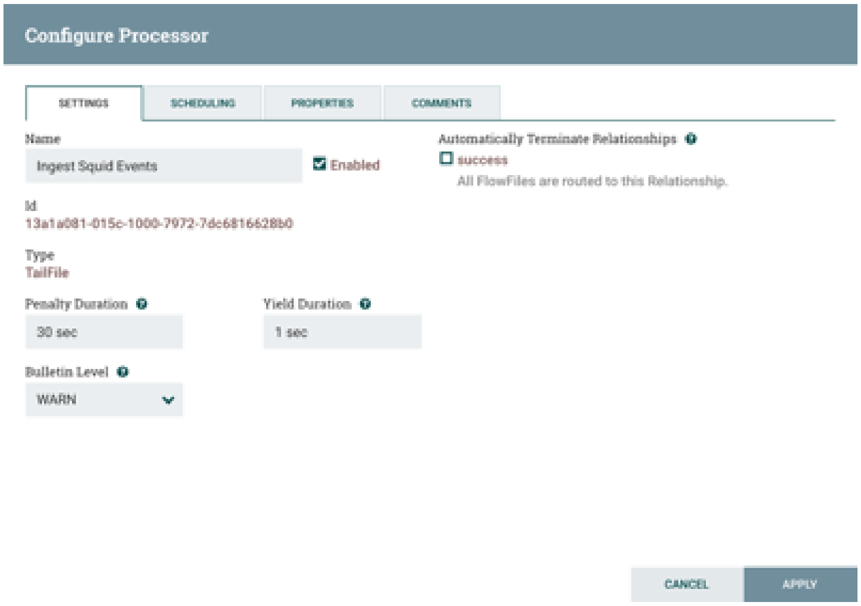
   TailFile
   TailFile

In	0 (0 bytes)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	0 (0 bytes)	5 min
Tasks/Time	0 / 00:00:00.000	5 min

3. Right-click the processor icon and select **Configure** to display the Configure Processor dialog box.

   - In the Settings tab, change the name to `Ingest $DATASOURCE Events`.

Figure 3.2. Configure Processor Dialog Box Settings Tab



Name: Ingest Squid Events

☑ Enabled

Automatically Terminate Relationships

□ success

All FlowFiles are routed to this Relationship.

Id: 13a1a081-015c-1000-7972-7dc6816628b0

Type: TailFile

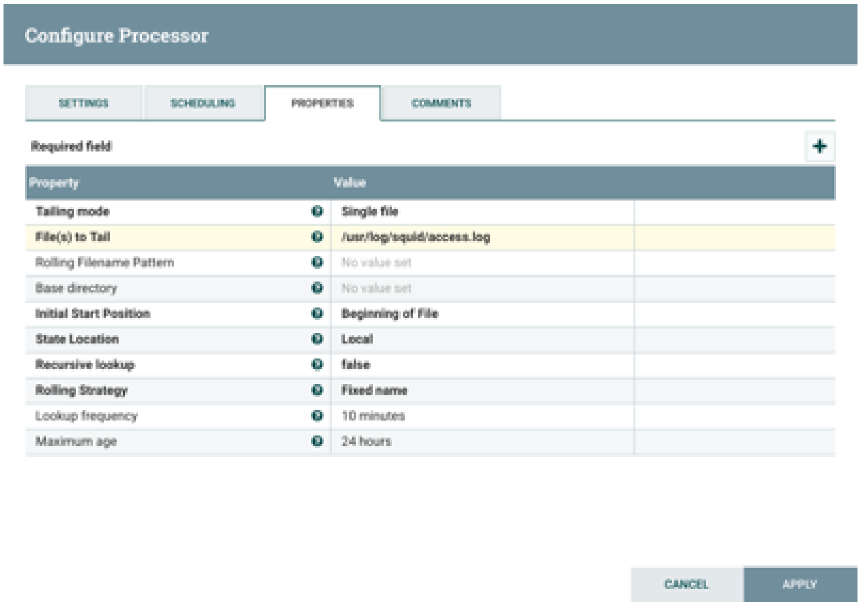
Penalty Duration: 30 sec

Yield Duration: 1 sec

Bulletin Level: WARN

- In the **Properties** tab, enter the path to the datasource file in the **Value** column for the **File(s) to Tail** property:

Figure 3.3. NiFi Configure Processor



Property	Value
Tailing mode	Single file
File(s) to Tail	/usr/log/squid/access.log
Rolling Filename Pattern	No value set
Base directory	No value set
Initial Start Position	Beginning of File
State Location	Local
Recursive lookup	false
Rolling Strategy	Fixed name
Lookup frequency	10 minutes
Maximum age	24 hours

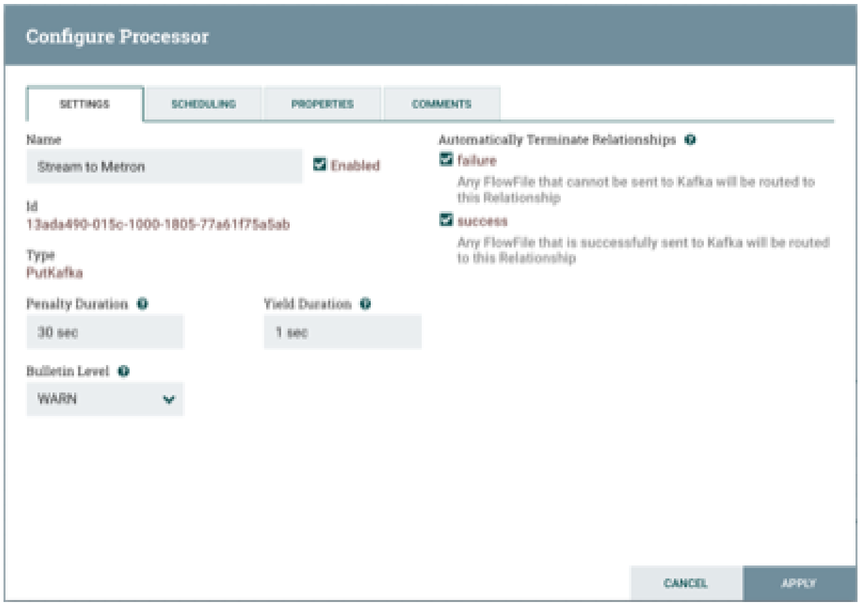
4. Add another processor by dragging the Processor icon to the main window.

5. Select the **PutKafka** type of processor and click **Add**.

6. Right-click the processor and select **Configure**.

7. In the Settings tab, change the name to `Stream to Metron` and then click the relationship check boxes for failure and success.

Figure 3.4. Configure Processor Settings Tab



Name
Stream to Metron ☑ Enabled

Id
13ada490-015c-1000-1805-77a61f75a5ab

Type
PutKafka

Penalty Duration: 30 sec

Yield Duration: 1 sec

Bulletin Level: WARN

Automatically Terminate Relationships

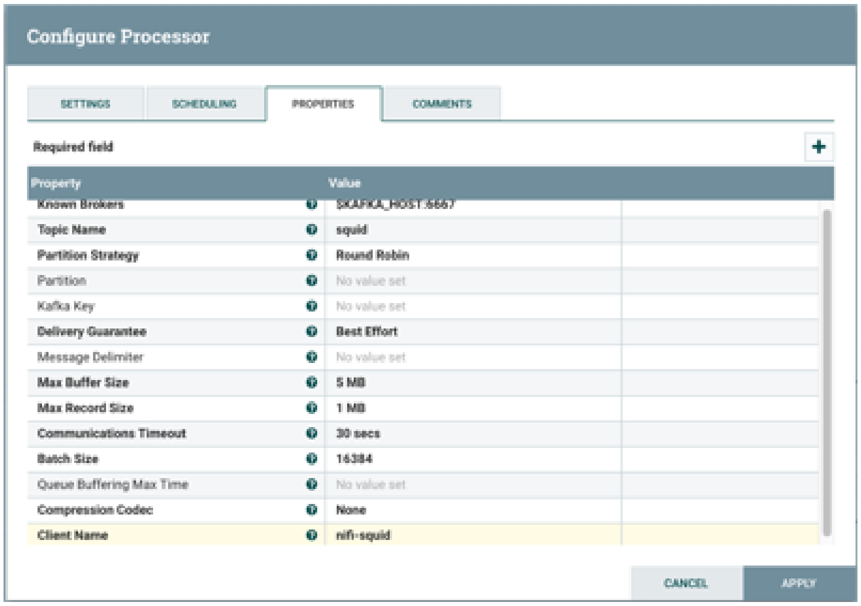
☑ failure
Any FlowFile that cannot be sent to Kafka will be routed to this Relationship

☑ success
Any FlowFile that is successfully sent to Kafka will be routed to this Relationship

8. In the Properties tab, set the following three properties:

   - Known Brokers: $KAFKA_HOST:6667
   - Topic Name: $DATAPROCESSOR
   - Client Name: nifi-$DATAPROCESSOR

Figure 3.5. Configure Processor Properties Tab



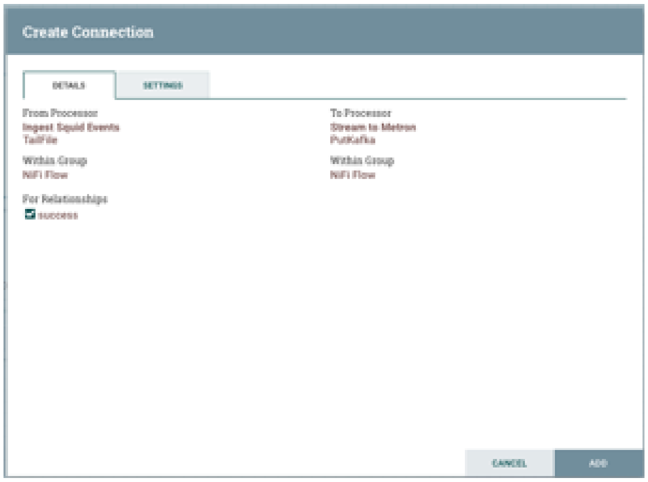
Required field

Property	Value
Known Brokers	$KAFKA_HOST:6667
Topic Name	squid
Partition Strategy	Round Robin
Partition	No value set
Kafka Key	No value set
Delivery Guarantee	Best Effort
Message Delimiter	No value set
Max Buffer Size	5 MB
Max Record Size	1 MB
Communications Timeout	30 secs
Batch Size	16384
Queue Buffering Max Time	No value set
Compression Codec	None
Client Name	nifi-squid

9. Create a connection by dragging the arrow from the Ingest $DATAPROCESSOR Events processor to the Stream to Metron processor.

   NiFi displays a Create Connection dialog box.

   Figure 3.6. nifi_create_connection.png

   From Processor
   Ingest Squid Events
   TailFile

   Within Group
   NiFi Flow

   For Relationships
   ☑ success

   To Processor
   Stream to Metron
   PutKafka

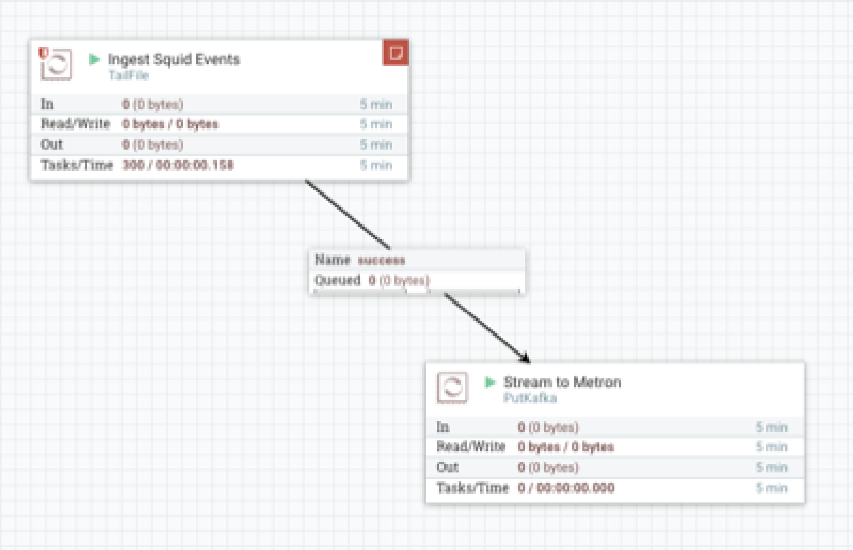
   Within Group
   NiFi Flow

10. Click **Add** to accept the default settings for the connection.

11. Press the Shift key and draw a box around both parsers to select the entire flow; then click the play button (green arrow).

    You should see all of the processor icons turn into green arrows.

    Figure 3.7. NiFi Dataflow

    Ingest Squid Events
    TailFile

    ```
    In          0 (0 bytes)               5 min
    Read/Write  0 bytes / 0 bytes         5 min
    Out         0 (0 bytes)               5 min
    Tasks/Time  300 / 00:00:00.158        5 min
    ```

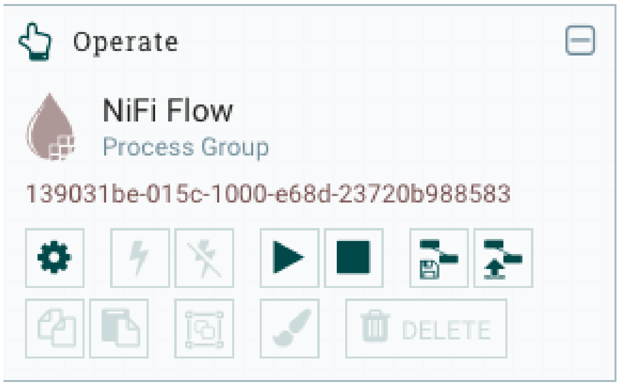
    Name success
    Queued 0 (0 bytes)

    Stream to Metron
    PutKafka

    ```
    In          0 (0 bytes)               5 min
    Read/Write  0 bytes / 0 bytes         5 min
    Out         0 (0 bytes)               5 min
    Tasks/Time  0 / 00:00:00.000          5 min
    ```

12. Click (Start button in the Operate panel.

Figure 3.8. Operate Panel



13. Generate some data using the new data processor client.

    You should see metrics on the processor of data being pushed into Metron.

14. Look at the Storm UI for the parser topology and you should see tuples coming in.

15. After about five minutes, you should see a new Elastic Search index called $DATAPROCESSOR_index* in the Elastic Admin UI.

For more information about creating a NiFi data flow, see the NiFi documentation.

3.1.3. Parsing a New Data Source to HCP

Parsers transform raw data (textual or raw bytes) into JSON messages suitable for downstream enrichment and indexing by HCP. There is one parser for each data source and the information is piped to the Enrichment/Threat Intelligence topology.

You can transform the field output in the JSON messages into information and formats to make the output more useful. For example, you can change the timestamp field output from GMT to your timezone.

You must make the following decisions before you parse a new data source:

- Type of parser you will use for your data source

  For more information about which parser to use, see Parsers [101].

  HCP supports two types of parsers: Java and general purpose:

  - General Purpose - HCP supports two general purpose parsers: Grok and CSV. These parsers are ideal for structured or semi structured logs that are well understood and telemetries with lower volumes of traffic.

  - A Java parser is appropriate for a telemetry type that is complex to parse, with high volumes of traffic.

- How you will parse the new data source

HCP enables you to parse a new data source and transform data fields using the HCP Management module or the command line interface. Both methods are described in the following sections:

  - Using the Management Module [16]
  - Using the CLI [22]

- What data you intend to search, sort, and aggregate when using the Alerts UI.

  String values are mapped by default with a "type": "text" mapping that does not work with the Alerts UI. In order to properly enable sorting and aggregate operations, you have two options:

  - Explicitly add a mapping for that property to an Elasticsearch template. You call also refer to Section 3.3.6.1, “Updating Elasticsearch Templates to Work with Elasticsearch 5.x” [47] for information about using Elasticsearch 5.x.
  - Add a global mapping to Elasticsearch that will automatically map that property to a type that is searchable/sortable/aggregatable for all indexes.

    For example, you can set a template to match all indexes that maps strings to text with fielddata enabled:

    ```
    # curl -XPUT 'http://${ES_HOST}:${ES_PORT}/_template/
    default_string_template' -d '
    {
        "template": "*",
        "mappings" : {
            "${your_type_here}": {
                "dynamic_templates": [
                {
                        "strings": {
                            "match_mapping_type": "string",
                            "mapping": {
                                "type": "text",
                                "fielddata": "true"
                            }
                        }
                }
    ```

    For information on the difference between types=text and type=keyword, see Section 3.1.3.1, “Type Mapping Changes” [14].

3.1.3.1. Type Mapping Changes

Type mappings in Elasticsearch 5.6.2 have changed from ES 2.x. This section provides an overview of the most significant changes.

The following is a list of the major changes in Elasticsearch 5.6.2:

- String fields replaced by text/keyword type
- Strings have new default mappings as follows:

  ```
  {
  ```

```
  "type": "text",
  "fields": {
    "keyword": {
      "type": "keyword",
      "ignore_above": 256
    }
  }
}
```

- There is no longer a `_timestamp` field that you can set "enabled" on.

  This field now causes an exception on templates. The Metron model has a timestamp field that is sufficient.

The semantics for string types have changed. In 2.x, index settings are either "analyzed" or "not_analyzed" which means "full text" and "keyword", respectively. Analyzed text means the indexer will split the text using a text analyzer, thus allowing you to search on substrings within the original text. "New York" is split and indexed as two buckets, "New" and "York", so you can search or query for aggregate counts for those terms independently and match against the individual terms "New" or "York." "Keyword" means that the original text will not be split/analyzed during indexing and instead treated as a whole unit. For example, "New" or "York" will not match in searches against the document containing "New York", but searching on "New York" as the full city name will match. In Elasticsearch 5.6 language, instead of using the "index" setting, you now set the "type" to either "text" for full text, or "keyword" for keywords.

Below is a table listing the changes to how String types are now handled.

sort, aggregate, or access values	Elasticsearch 2.x	Elasticsearch 5.x	Example

no	`"my_property" : {`
	`  "type": "string",`
	`  "index": "analyzed"`
	`}`
		`"my_property" : {`
		`  "type": "text"`
		`}`
		Additional defaults: "index": "true", "fielddata": "false"
			"New York" handled via in-mem search as "New" and "York" buckets. **No** aggregation or sort.

yes	`"my_property": {`
	`  "type": "string",`
	`  "index": "analyzed"`
	`}`
		`"my_property": {`
		`  "type": "text",`
		`  "fielddata": "true"`
		`}`
			"New York" handled via in-mem search as "New" and "York" buckets. **Can** aggregate and sort.

yes	`"my_property": {`
	`  "type": "string",`
	`  "index": "not_analyzed"`
	`}`
		`"my_property" : {`
		`  "type": "keyword"`
		`}`
			"New York" searchable as single value. **Can** aggregate and sort. A search for "New" or "York" will not match against the whole value.

yes	`"my_property": {`
	`  "type": "string",`
	`  "index": "analyzed"`
	`}`
		`"my_property": {`
		`  "type": "text",`
		`  "fields": {`
		`    "keyword": {`
		`      "type": "keyword",`
		`      "ignore_above": 256`
		`    }`
		`  }`
		`}`
			"New York" searchable as single value or as text document, can aggregate and sort on the sub term "keyword."

If you want to set default string behavior for all strings for a given index and type, you can do so with a mapping similar to the following (replace ${your_type_here} accordingly):

```
# curl -XPUT 'http://${ES_HOST}:${ES_PORT}/_template/default_string_template'
 -d '
{
    "template": "*",
    "mappings" : {
        "${your_type_here}": {
            "dynamic_templates": [
                {
                    "strings": {
                        "match_mapping_type": "string",
                        "mapping": {
                            "type": "text"
                            "fielddata": "true"
                        }
                    }
                }
            ]
        }
    }
}
```

By specifying the `template` property with value `*`, the template will apply to all indexes that have documents indexed of the specified type (${your_type_here}).

The following are other settings for types in ES:

- doc_values
  - On-disk data structure
  - Provides access for sorting, aggregation, and field values
  - Stores same values as _source, but in column-oriented fashion better for sorting and aggregating
  - Not supported on text fields
  - Enabled by default
- fielddata
  - In-memory data structure
  - Provides access for sorting, aggregation, and field values
  - Primarily for text fields
  - Disabled by default because the heap space required can be large

3.1.3.2. Using the Management Module

This section explains how to use the HCP Management module to parse a new data source and transform data fields.

Although HCP supports both Java and general purpose parsers, the following workflow uses the general purpose parser, Grok.

1. Determine the format of the new data source’s log entries, so you can parse them:

   a. Look at the different log files that can be created and determine which log file needs to be parsed:

   ```
   sudo su -
   cd /var/log/$NEW_DATASOURCE
   ls
   ```

   The file you want is typically the `access.log`, but your data source might use a different name.

   b. Generate entries for the log that needs to be parsed so that you can see the format of the entries.

   For example:

   ```
   timestamp | time elapsed | remotehost | code/status | bytes | method |
    URL rfc931 peerstatus/peerhost | type
   ```

2. Create a Kafka topic for the new data source:

   a. Log in to $KAFKA_HOST as root.

   b. Create a Kafka topic named the same as the new data source:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-topics.sh
   --zookeeper $ZOOKEEPER_HOST:2181 --create --topic $NEW_DATASOURCE
   --partitions 1 --replication-factor 1
   ```

   c. List all of the Kafka topics, to ensure that the new topic exists:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-topics.sh --zookeeper
    $ZOOKEEPER_HOST:2181 --list
   ```

3. Create a Grok statement file that defines the Grok expression for the log type you identified in Step 1.

   Refer to the Grok documentation for additional details.

4. Launch the HCP Management module:

   a. From the Ambari Dashboard panel, click Metron.

   b. Make sure you have the Summary tab selected.

   c. Select the Metron Management UI from the Summary list.

   The Metron Management UI tool should display in a separate browser tab.

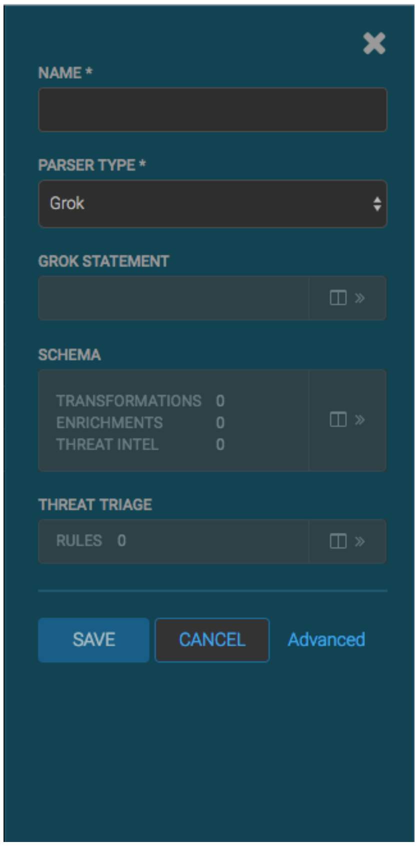
   Alternatively, you can launch the module from `$METRON_MANAGEMENT_UI_HOST:4200` in a browser.

5. Click **Sensors** on the left side of the window, under **Operations**.

6. Click + (the add button) in the lower right corner of the screen.

The Management module displays a panel used to create the new sensor.

Figure 3.9. New Sensor Panel



NAME *

PARSER TYPE * Grok

GROK STATEMENT

SCHEMA

TRANSFORMATIONS 0
ENRICHMENTS 0
THREAT INTEL 0

THREAT TRIAGE

RULES 0

7. In the **NAME** field, enter the name of the new sensor.

   If a Kafka topic already exists for the sensor name, the module displays a message similar to **Kafka Topic Exists. Emitting**. If no matching Kafka topic is found, the module displays **No Matching Kafka Topic**.

8. In the **Parser Type** field, choose the type of parser for the new sensor.

   If you chose a Grok parser type and no Kafka type is detected, the module prompts for a Grok Statement.

9. If no Kafka topic exists for your sensor, create a Kafka topic for the sensor.

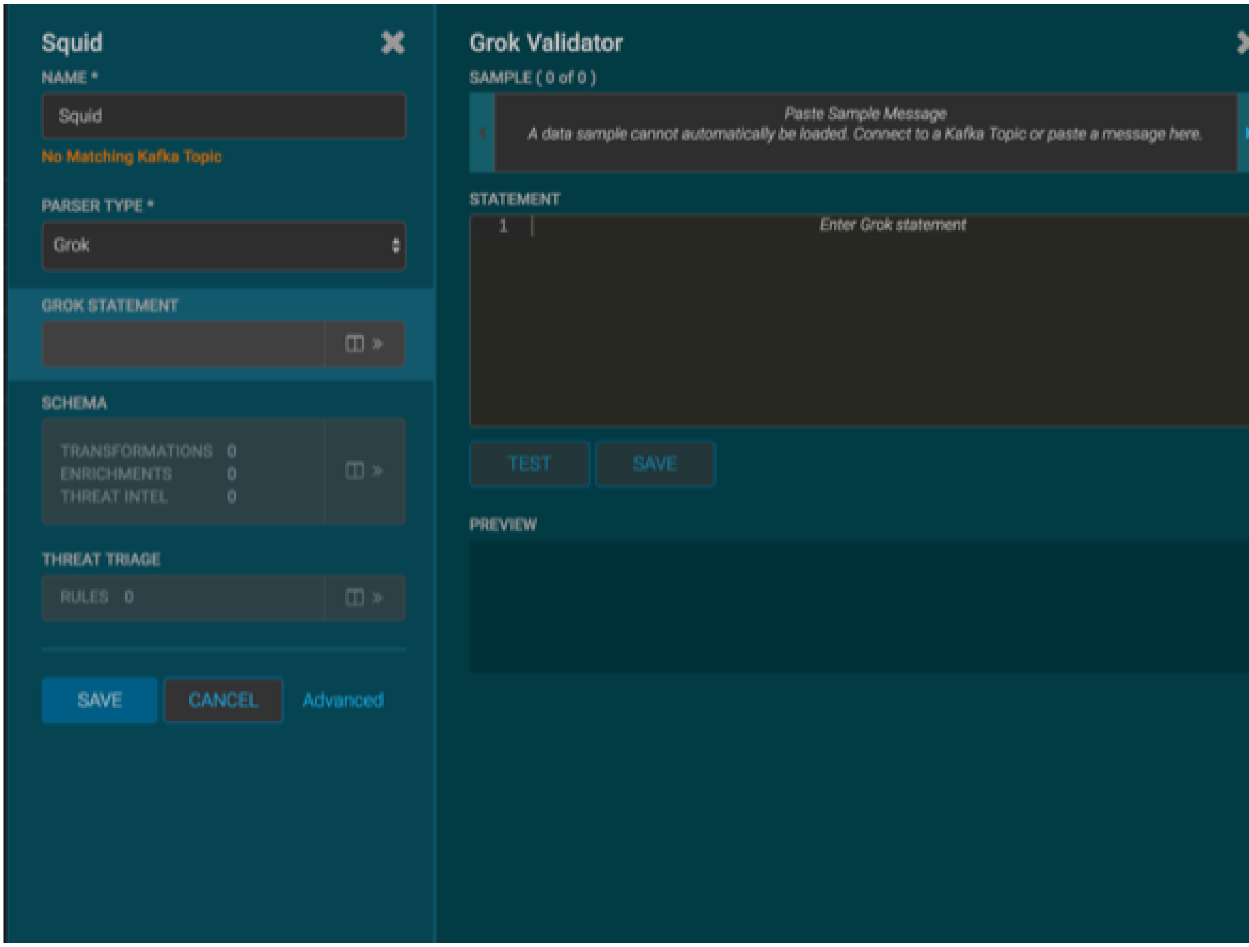
   a. In the Kafka Topic text box, click the arrow to display the Configure Kafka Topic dialog box

   b. Enter the partition and replication factor for the Kafka type associated with the new sensor, and then click Save.

10. Enter a Grok statement for the new parser:

a. In the Grok Statement box, click the ◫ » (expand window button) to display the Grok Validator panel.

Figure 3.10. Grok Validator Panel



Squid

NAME * Squid

No Matching Kafka Topic

PARSER TYPE * Grok

b. In the **SAMPLE** text field, enter a sample log entry for the data source.

c. In the **STATEMENT** text field, enter the Grok statement you created for the data source, and then click **TEST**.

The Management module will automatically complete partial words in your Grok statement as you enter them.

The validator displays the results of the test. If the validator finds an error, it displays the error information. If the validation succeeds, it displays the valid mapping in the **PREVIEW** field.

Note

You should perform the Grok validation using several different sensor log entries to ensure that the Grok statement is valid for all sensor logs. To display additional sensor log entries, click the forward or backward arrow icon on the side of the **SAMPLE** text box.

d. Click **SAVE** to save the Grok statement for the sensor.

11.Click **SAVE** to save the sensor information and add it to the list of Sensors.

This new data source processor topology ingests from the $Kafka topic and then parses the event with the HCP Grok framework using the Grok pattern. The result is a standard JSON Metron structure that then is added to the "enrichment" Kafka topic for further processing.

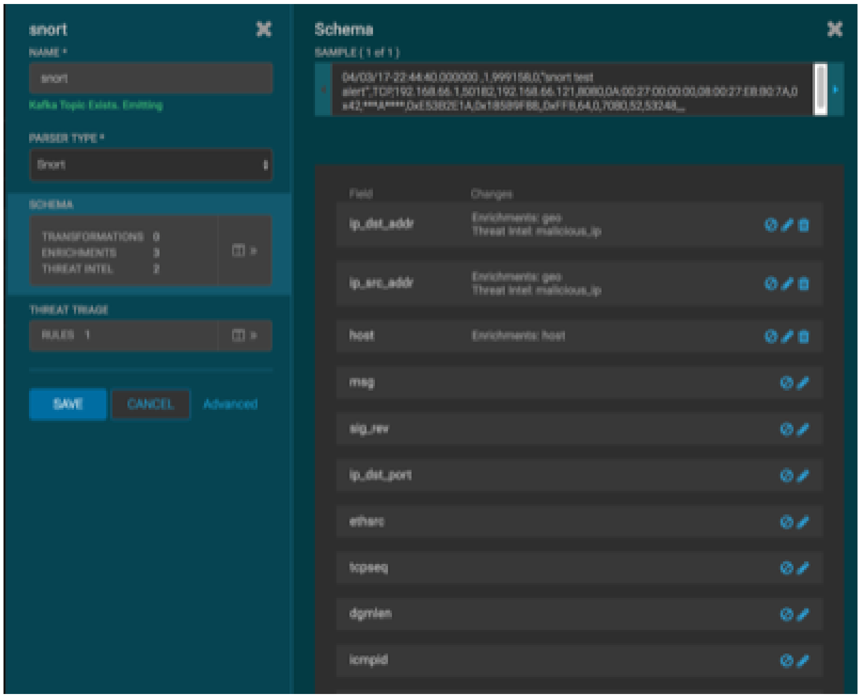
12.Add your transformation information:

Note

Your sensor must be running and producing data before you can add transformation information.

a. In the Schema box, click ◫ » (expand window button).

The Management module populates the panel with message, field, and value information.



snort

Kafka Topic Exists. Emitting

Snort

TRANSFORMATIONS 0
ENRICHMENTS 3
THREAT INTEL 2

RULES 1

```
04/03/17-22:44:40.000000 ,1,999158,0,"snort test
alert",TCP,192.168.66.1,50182,192.168.66.121,8080,0A:00:27:00:00:00,08:00:27:E8:B0:7A,0
x42,***A****,0xE5382E1A,0x18589F88,0xFFB,64,0,7080,52,53248,,
```

Field	Changes
ip_dst_addr	Enrichments: geo Threat Intel: malicious_ip
ip_src_addr	Enrichments: geo Threat Intel: malicious_ip
host	Enrichments: host
msg	
sig_rev	
ip_dst_port	
ethsrc	
tcpseq	
dgmlen	
icmpid	

The Sample field, at the top of the panel, displays a parsed version of a sample message from the sensor. The Management module will test your transformations against these parsed messages.

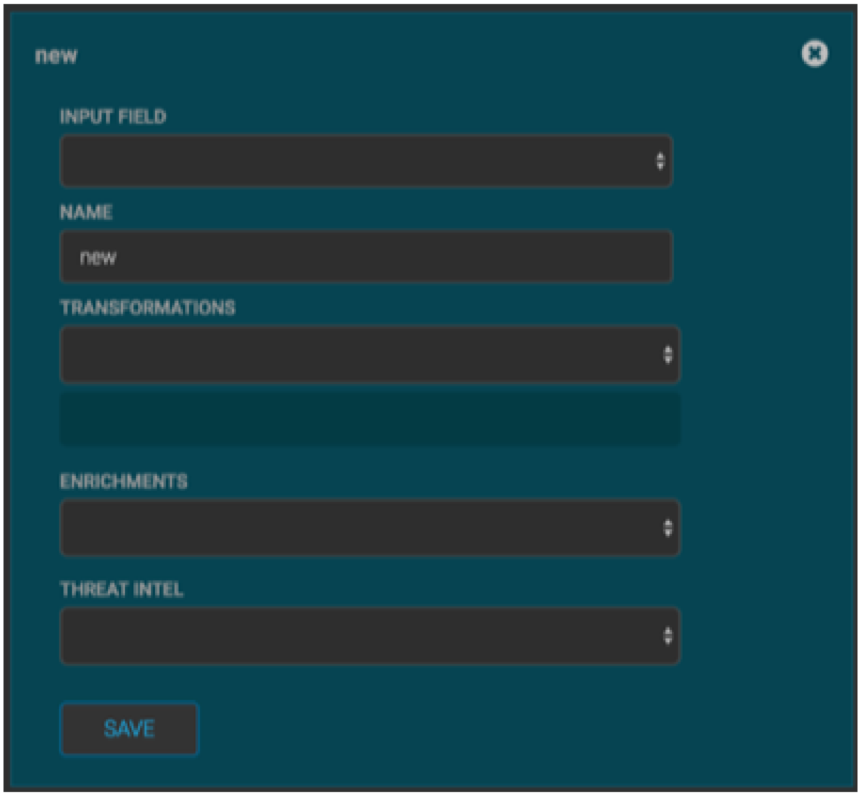
You can use the right and left arrow buttons in the Sample field to view the parsed version of each sample message available from the sensor.

You can apply transformations to an existing field or create a new field. Typically users choose to create and transform a new field, rather than transforming an existing field.

b. To add a new transformation, either click the ✎ next to a field or click the + (plus sign) at the bottom of the **Schema** panel.

The module displays a new dialog box for your transformations.

Figure 3.11. New Schema Information Panel



new

NAME: new

c. In the dialog box, choose the field you want to transform from the INPUT FIELD box, enter the name of the new field in the NAME field, and then choose a function with the appropriate parameters in the TRANSFORMATIONS box.

d. Click **SAVE** to save your additions.

   The Management module populates the Transforms field with the number of transformations applied to the sensor.

   If you change your mind and want to remove a transformation, click the "x" next to the field.

e. You can also suppress fields with the transformation feature by clicking ⊘ (suppress icon).

   This icon prevents the field from being displayed, but it does not remove the field entirely.

f. Click **SAVE** in the parser panel to save the transformation information.

3.1.3.3. Using the CLI

This section shows you how to use the Grok parser to parse a new data source using the CLI.

1. Determine the format of the new data source’s log entries, so that you can parse them:

   a. Use ssh to access the host for the new data source.

   b. Look at the different log files that can be created and determine which log file needs to be parsed. This is typically the access.log, but your data source might use a different name.

   ```
   sudo su -
   cd /var/log/$NEW_DATASOURCE
   ls
   ```

   c. Generate entries for the log that needs to be parsed so you can see the format of the entries.

   For example:

   ```
   timestamp | time elapsed | remotehost | code/status | bytes | method |
    URL rfc931 peerstatus/peerhost | type
   ```

2. Create a Kafka topic for the new data source:

   a. Log in to $KAFKA_HOST as root.

   b. Create a Kafka topic named the same as the new data source:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-topics.sh
   --zookeeper $ZOOKEEPER_HOST:2181 --create --topic $NEW_DATASOURCE
   --partitions 1 --replication-factor 1
   ```

   c. List all of the Kafka topics, to ensure that the new topic exists:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-topics.sh --zookeeper
    $ZOOKEEPER_HOST:2181 --list
   ```

3. Create a Grok statement.

   a. Define the Grok expression for the log type you identified in Step 1 by creating a Grok statement file.

      Refer to the Grok documentation for additional details.

   b. Validate the Grok pattern to make sure it is valid.

      You can use a tool such as Grok Constructor to validate your Grok pattern.

   c. Save the Grok pattern and load it into Hadoop Distributed File System (HDFS) in a named location:

      i. Create a local file for the new data source:

         ```
         touch /tmp/$DATASOURCE
         ```

      ii. Open $DATASOURCE and add the Grok pattern defined in Step 3b:

         ```
         vi /tmp/$DATASOURCE
         ```

      iii. Put the $DATASOURCE file into the HDFS directory where Metron stores its Grok parsers.

         Existing Grok parsers that ship with HCP are staged under `/apps/metron/patterns`:

         ```
         su - hdfs
         hadoop fs -rmr /apps/metron/patterns/$DATASOURCE
         hdfs dfs -put /tmp/$DATASOURCE /apps/metron/patterns/
         ```

4. Define a parser configuration for the Metron Parsing Topology.

   After the Grok pattern is staged in HDFS, you must define a parser configuration for the Metron Parsing Topology. The Metron Parsing Topology (also known as the Normalizing Topology) is designed to take a sensor input (in its native format) and turn it into a Metron JSON Object. For more information about the Metron parsing topology, see Parsers [101].

   a. ssh as root into host with HCP installed.

   b. Create a $DATASOURCE parser configuration file at `$METRON_HOME/config/zookeeper/parsers/$DATASOURCE.json`:

      For example:

      ```
      {
      "parserClassName": "org.apache.metron.parsers.GrokParser",
      "sensorTopic": "$DATASOURCE",
      "readMetadata" : true
      "mergeMetadata" : true
      "metron.metadata.topic : topic"
      "metron.metadata.customer_id : "my_customer_id"
      "filterClassName" : "STELLAR"
      ,"parserConfig" : {
      "filter.query" : "exists(field1)"
      ```

```
"parserConfig": {
   "grokPath": "/apps/metron/patterns/$DATASOURCE",
   "patternLabel": "$DATASOURCE_DELIMITED",
   "timestampField": "timestamp"
},
"fieldTransformations" : [
   {
     "transformation" : "STELLAR"
     ,"output" : [ "full_hostname", "domain_without_subdomains" ]
     ,"config" : {
                  "full_hostname" : "URL_TO_HOST(url)"
                  ,"domain_without_subdomains" :
 "DOMAIN_REMOVE_SUBDOMAINS(full_hostname)"
                  }
    }
   ]
}
```

Where:

parserClassName

The name of the parser's class that is in the jar file.

filterClassName

The filter to use. This may be a fully qualified classname of a Class that implements the `org.apache.metron.parsers.interfaces.MessageFilter<` interface. Message Filters are intended to allow the user to ignore a set of messages via custom logic. The existing implementations are:

- `STELLAR` : Allows you to apply a stellar statement which returns a boolean, which will pass every message for which the statement returns `true` . The Stellar statement that is to be applied is specified by the `filter.query` property in the `parserConfig`. Example Stellar Filter which includes messages which contain a `field1` field:

```
{
 "filterClassName" : "STELLAR"
,"parserConfig" : {
 "filter.query" : "exists(field1)"
 }
}
```

sensorTopic

The Kafka topic on which the telemetry is being streamed.

readMetadata

A Boolean indicating whether or not to read metadata and make it available to field transformations (`false` by default).

There are two types of metadata supported in HCP:

- Environmental metadata : Metadata about the system at large

	• For example, if you have multiple Kafka topics being processed by one parser, you might want to tag the messages with the Kafka topic. • Custom metadata: Custom metadata from an individual telemetry source that you might want to use within Metron.
mergeMetadata	Boolean indicating whether to merge metadata with the message or not (`false` by default). If this property is set to `True`, then every metadata field will become part of the messages and, consequently, also available for use in field transformations.
parserConfig	The configuration file.
grokPath	The path for the Grok statement.
patternLabel	The top-level pattern of the Grok file.
fieldTransformations	An array of complex objects representing the transformations to be done on the message generated from the parser before writing out to the Kafka topic. In this example, the Grok parser is designed to extract the URL, but the only information that you need is the domain (or even the domain without subdomains). To obtain this, you can use the Stellar Field Transformation (under the fieldTransformations element). The Stellar Field Transformation allows you to use the Stellar DSL (Domain Specific Language) to define extra transformations to be performed on the messages flowing through the topology. For more information on using the fieldTransformations element in the parser configuration, see Parsers [101].
spoutParallelism	The kafka spout parallelism (default to `1`). This can be overridden on the command line.
spoutNumTasks	The number of tasks for the spout (default to `1`). This can be overridden on the command line.
parserParallelism	The parser bolt parallelism (default to `1`). This can be overridden on the command line.
parserNumTasks	The number of tasks for the parser bolt (default to `1`). This can be overridden on the command line.

errorWriterParallelism	The error writer bolt parallelism (default to 1). This can be overridden on the command line.
errorWriterNumTasks	The number of tasks for the error writer bolt (default to 1). This can be overridden on the command line.
numWorkers	The number of workers to use in the topology (default is the storm default of 1).
numAckers	The number of acker executors to use in the topology (default is the storm default of 1).
spoutConfig	A map representing a custom spout config (this is a map). This can be overridden on the command line.
securityProtocol	The security protocol to use for reading from kafka (this is a string). This can be overridden on the command line and also specified in the spout config via the `security.protocol` key. If both are specified, then they are merged and the CLI will take precedence.
stormConfig	The storm config to use (this is a map). This can be overridden on the command line. If both are specified, they are merged with CLI properties taking precedence.

c. Use the following script to upload configurations to Apache ZooKeeper:

```
 $METRON_HOME/bin/zk_load_configs.sh --mode PUSH -i $METRON_HOME/config/
zookeeper -z $ZOOKEEPER_HOST:2181
```

Note

You might receive the following warning messages when you execute the previous command. You can safely ignore these warning messages.

```
log4j:WARN No appenders could be found for logger (org.apache.
curator.framework.imps.CuratorFrameworkImpl).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.
html#noconfig for more info.
```

5. Deploy the new parser topology to the cluster:

a. Log in to the host that has Metron installed as root user.

b. Deploy the new parser topology:

```
$METRON_HOME/bin/start_parser_topology.sh -k $KAFKA_HOST:6667 -z
 $ZOOKEEPER_HOST:2181 -s $DATASOURCE
```

c. Use the Apache Storm UI to ensure that the new topology is listed and that it has no errors.

This new data source processor topology ingests from $DATASOURCE Kafka topic that you created earlier and then parses the event with the HCP Grok framework using the Grok pattern defined earlier. The result of the parsing is a standard JSON Metron structure that is added to the enrichment Kafka topic for further processing.

3.1.4. Verifying That the Events Are Indexed

After you finish adding your new data source, you should verify that the data source events are indexed and the output matches any Stellar transformation functions you used.

By convention, the index where the new messages are indexed is called $DATASOURCE_index_[timestamp] and the document type is $DATASOURCE_doc.

From the Alerts UI, search the source:type filter for the $DATASOURCE messages. For more information about using the Alerts UI, see Triaging Alerts.

3.2. Enriching Telemetry Events

After the raw security telemetry events have been parsed and normalized, the next step is to enrich the data elements of the normalized event. Enrichments add external data from data stores (such as HBase). Examples of enrichments are GEO where an external IP address is enriched with GeoIP information (lat/long coordinates + City/State/Country) and HOST enrichment where an IP gets enriched with Host details (for example, IP corresponds to Host X which is part of a web server farm for an e-commerce application). This information makes the data more useful and relevant, which assists the SOC analyst and SOC investigator in researching alerts. Threat intelligence is another type of enrichment. For information about threat intelligence see Using Threat Intelligence.

HCP provides the following enrichment sources but you can add your own enrichment sources to suit your needs:

- Asset
- GeoIP
- User

Note

The telemetry data sources for which HCP includes parsers (for example, Bro, Snort, and YAF) already include enrichment topologies. These topologies will become effective when you start the data sources in HCP.

One of the features of the enrichment topology is that it groups messages together by the HBase key. An advantage of grouping messages together is that whenever you execute a Stellar function, you can add a caching layer, thus decreasing the need to do a call to HBase for every event.

Prior to enabling an enrichment capability within HCP, the enrichment store (which for HCP is primarily HBase) must be loaded with enrichment data. Enrichment data can either be bulk loaded from the local file system, HDFS, or be streamed into the enrichment store via the parser framework. The enrichment loader transforms the enrichment into a JSON format that is understandable to Metron. The loading framework has additional capabilities for aging data out of the enrichment stores based on time. Once the stores are loaded, an enrichment bolt that can interact with the enrichment store can be incorporated into the enrichment topology.

Each enrichment bolt can enrich a specific field/tag within a Metron message. When a bolt recognizes that it is able to enrich a field, it reaches into the enrichment store, pulls out the enrichment, and tags the message with the enrichment. The enrichment is then stored within the bolt's in-memory cache. HCP uses the underlying Storm routing capabilities to make sure that similar enrichment values are sent to the appropriate bolts that already have these values cached in-memory.

HCP provides the following enrichment sources but you can add your own enrichment sources to suit your needs:

- Asset
- GeoIP
- User

To configure an enrichment source, complete the following steps:

- Prerequisites [7]
- Bulk Loading Enrichment Information [28]
- Streaming Enrichment Information [38]

For more information about the Metron enrichment framework, see Enrichment Framework [108].

3.2.1. Bulk Loading Enrichment Information

Enrichment data can either be bulk loaded from HDFS or be streamed into enrichment store via pluggable loading framework. This section provides a description of the bulk loading sources supported by HCP and the steps to bulk load enrichment data.

- Bulk Loading Sources [28]
- Configuring an Extractor Configuration File [31]
- Configuring Element-to-Enrichment Mapping [33]
- Running the Enrichment Loader [34]
- Mapping Fields to HBase Enrichments [34]

3.2.1.1. Bulk Loading Sources

You can bulk load enrichment information from the following sources:

- Flat File Ingestion
- HDFS via MapReduce
- Taxii Loader

CSV File

The shell script `$METRON_HOME/bin/flatfile_loader.sh` will read data from local disk and load the enrichment or threat intel data into an HBase table.

One special thing to note here is that there is a special configuration parameter to the Extractor config that is only considered during this loader:

inputFormatHandler	This specifies how to consider the data. The two implementations are `BY_LINE` and `org.apache.metron.dataloads.extractor.inputformat.WholeFileFormat`

The default is `BY_LINE`, which makes sense for a list of CSVs where each line indicates a unit of information which can be imported. However, if you are importing a set of STIX documents, then you want each document to be considered as input to the Extractor.

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generate the help screen/ set of options
-e	–extractor_config	Yes	JSON document describing the extractor for this input data source
-t	–hbase_table	Yes	The HBase table to import into
-c	–hbase_cf	Yes	The HBase table column family to import into
-i	–input	Yes	The input data location on local disk. If this is a file, then that file will be loaded. If this is a directory, then the files will be loaded recursively under that directory.
-l	–log4j	No	The log4j properties file to load
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

HDFS via MapReduce

The shell script `$METRON_HOME/bin/flatfile_loader.sh` will kick off MapReduce job to load data stated in HDFS into an HBase table. The following is as example of the syntax:

```
$METRON_HOME/bin/flatfile_loader.sh -i /tmp/top-10k.csv -t enrichment -c t -
e ./extractor.json -m MR
```

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generate the help screen/ set of options
-e	–extractor_config	Yes	JSON document describing the extractor for this input data source
-t	–hbase_table	Yes	The HBase table to import into
-c	–hbase_cf	Yes	The HBase table column family to import into
-i	–input	Yes	The input data location on local disk. If this is a file, then that file will be loaded. If this is a directory, then the files will be loaded recursively under that directory.
-l	–log4j	No	The log4j properties file to load
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

Taxii Loader

The shell script `$METRON_HOME/bin/threatintel_taxii_load.sh` can be used to poll a Taxii server for STIX documents and ingest them into HBase.

It is quite common for this Taxii server to be an aggregation server such as Soltra Edge.

In addition to the Enrichment and Extractor configs described in the following sections, this loader requires a configuration file describing the connection information to the Taxii server. The following is an example of a configuration file:

```
{
   "endpoint" : "http://localhost:8282/taxii-discovery-service"
  ,"type" : "DISCOVER"
  ,"collection" : "guest.Abuse_ch"
  ,"table" : "threat_intel"
  ,"columnFamily" : "cf"
  ,"allowedIndicatorTypes" : [ "domainname:FQDN", "address:IPV_4_ADDR" ]
}
```

where:

endpoint	The URL of the endpoint.
type	`POLL` or `DISCOVER` depending on the endpoint.
collection	The Taxii collection to ingest.
table	The HBase table to import into.
columnFamily	The column family to import into.

allowedIndicatorTypes	An array of acceptable threat intelligence types (see the "Enrichment Type Name" column of the Stix table above for the possibilities).

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generate the help screen/ set of options
-e	–extractor_config	Yes	JSON document describing the extractor for this input data source
-c	–taxii_connection_config	Yes	The JSON config file to configure the connection
-p	–time_between_polls	No	The time between polling the Taxii server in milliseconds. (default: 1 hour)
-b	–begin_time	No	Start time to poll the Taxii server (all data from that point will be gathered in the first pull). The format for the date is yyyy-MM-dd HH:mm:ss
-l	–log4j	No	The Log4j properties to load
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

3.2.1.2. Configuring an Extractor Configuration File

The extractor configuration file is used to bulk load the enrichment store into HBase. Complete the following steps to configure the extractor configuration file:

1. Log in as root to the host on which Metron is installed.

2. Determine the schema of the enrichment source.

3. Create an extractor configuration file called `extractor_config_temp.json` and populate it with the enrichment source schema.

   For example:

```
{
 "config" : {
   "columns" : {
      "domain" : 0
      ,"owner" : 1
      ,"home_country" : 2
      ,"registrar": 3
      ,"domain_created_timestamp": 4
   }
   ,"indicator_column" : "domain"
   ,"type" : "whois"
   ,"separator" : ","
```

```
  }
  ,"extractor" : "CSV"
}
```

4. You can transform and filter the enrichment data as it is loaded into HBase by using Stellar extractor properties in the extractor configuration file. HCP supports the following Stellar extractor properties:

   `value_transform`

   Transforms fields defined in the `columns` mapping with Stellar transformations. New keys introduced in the transform are added to the key metadata. For example:

   ```
   "value_transform" : {
      "domain" : "DOMAIN_REMOVE_TLD(domain)"
   ```

   `value_filter`

   Allows additional filtering with Stellar predicates based on results from the value transformations. In the following example, records whose domain property is empty after removing the TLD are omitted.

   ```
   "value_filter" : "LENGTH(domain) > 0",
     "indicator_column" : "domain",
   ```

   `indicator_transform`

   Transforms the `indicator` column independent of the value transformations. You can refer to the original indicator value by using `indicator` as the variable name, as shown in the following example. In addition, if you prefer to piggyback your transformations, you can refer to the variable `domain`, which allows your indicator transforms to inherit transformations done to this value during the value transformations.

   ```
   "indicator_transform" : {
      "indicator" : "DOMAIN_REMOVE_TLD(indicator)"
   ```

   `indicator_filter`

   Allows additional filtering with Stellar predicates based on results from the value transformations. In the following example, records whose indicator value is empty after removing the TLD are omitted.

   ```
   "indicator_filter" : "LENGTH(indicator) > 0",
     "type" : "top_domains",
   ```

   If you include all of the supported Stellar extractor properties in the extractor configuration file, it will look similar to the following:

   ```
   {
     "config" : {
       "zk_quorum" : "$ZOOKEEPER_HOST:2181",
       "columns" : {
          "rank" : 0,
          "domain" : 1
       },
       "value_transform" : {
          "domain" : "DOMAIN_REMOVE_TLD(domain)"
       },
   ```

```
    "value_filter" : "LENGTH(domain) > 0",
    "indicator_column" : "domain",
    "indicator_transform" : {
       "indicator" : "DOMAIN_REMOVE_TLD(indicator)"
    },
    "indicator_filter" : "LENGTH(indicator) > 0",
    "type" : "top_domains",
    "separator" : ","
  },
  "extractor" : "CSV"
}
```

Running a file import with the above data and extractor configuration will result in the following two extracted data records:

Indicator	Type	Value
google	top_domains	{ "rank" : "1", "domain" : "google" }
yahoo	top_domains	{ "rank" : "2", "domain" : "yahoo" }

5. To access properties that reside in the global configuration file, provide a ZooKeeper quorum via the `zk_quorum` property. If the global configuration looks like `"global_property" : "metron-ftw"`, enter the following to expand the `value_transform`:

```
"value_transform" : {
    "domain" : "DOMAIN_REMOVE_TLD(domain)",
     "a-new-prop" : "global_property"
 },
```

The resulting value data will look like the following:

Indicator	Type	Value
google	top_domains	{ "rank" : "1", "domain" : "google", "a-new-prop" : "metron-ftw" }
yahoo	top_domains	{ "rank" : "2", "domain" : "yahoo", "a-new-prop" : "metron-ftw" }

6. Remove any non-ASCII invisible characters that might have been included when you cut and pasted:

```
iconv -c -f utf-8 -t ascii extractor_config_temp.json -o extractor_config.
json
```

Note

The extractor_config.json file is not stored anywhere by the loader. This file is used once by the bulk loader to parse the enrichment dataset. It is up the user to keep this configuration file for future use, if needed.

3.2.1.3. Configuring Element-to-Enrichment Mapping

Configure which element of a tuple should be enriched with which enrichment type.

This configuration is stored in ZooKeeper.

1. Log in as root user to the host that has Metron installed.

2. Cut and paste the following into a file called `enrichment_config_temp.json`, being sure to customize $ZOOKEEPER_HOST and $DATASOURCE to your specific values, where $DATASOURCE refers to the name of the datasource that is used to bulk load the enrichment:

```
{
     "zkQuorum" : "$ZOOKEEPER_HOST:2181"
    ,"sensorToFieldList" : {
          "$DATASOURCE" : {
             "type" : "ENRICHMENT"
            ,"fieldToEnrichmentTypes" : {
                 "domain_without_subdomains" : [ "whois" ]
              }
          }
    }
}
```

3. Remove any non-ASCII invisible characters that might have been included when you cut and pasted:

```
iconv -c -f utf-8 -t ascii enrichment_config_temp.json -o enrichment_config.
json
```

3.2.1.4. Running the Enrichment Loader

After the enrichment source and enrichment configuration are defined, you must run the loader to move the data from the enrichment source to the HCP enrichment store and store the enrichment configuration in ZooKeeper.

1. Use the loader to move the enrichment source to the enrichment store in ZooKeeper:

```
$METRON_HOME/bin/flatfile_loader.sh -n enrichment_config.json -i whois_ref.
csv -t enrichment -c t -e extractor_config.json
```

   HCP loads the enrichment data into Apache HBase and establishes a ZooKeeper mapping. The data is extracted using the extractor and configuration defined in the `extractor_config.json` file and populated into an HBase table called `enrichment`.

2. Verify that the logs were properly ingested into HBase:

```
hbase shell
scan 'enrichment'
```

3. Verify that the ZooKeeper enrichment tag was properly populated:

```
$METRON_HOME/bin/zk_load_configs.sh -m DUMP -z $ZOOKEEPER_HOST:2181
```

4. Generate some data by using a client for your particular data source to execute requests.

3.2.1.5. Mapping Fields to HBase Enrichments

Now that you have data flowing into the HBase table, you need to ensure that the enrichment topology can be used to enrich the data flowing past.

You can perform this step using either the HCP Management module or the CLI. Both of these methods are described in the following subsections.

3.2.1.5.1. Management Module Method

Now that you have parsed the data source, you can refine the parser output in three ways:

- Transformations
- Enrichments
- Threat Intel

Each of the parser outputs is added or modified in the **Schema** field. To modify any of the parser outputs, complete the following steps:

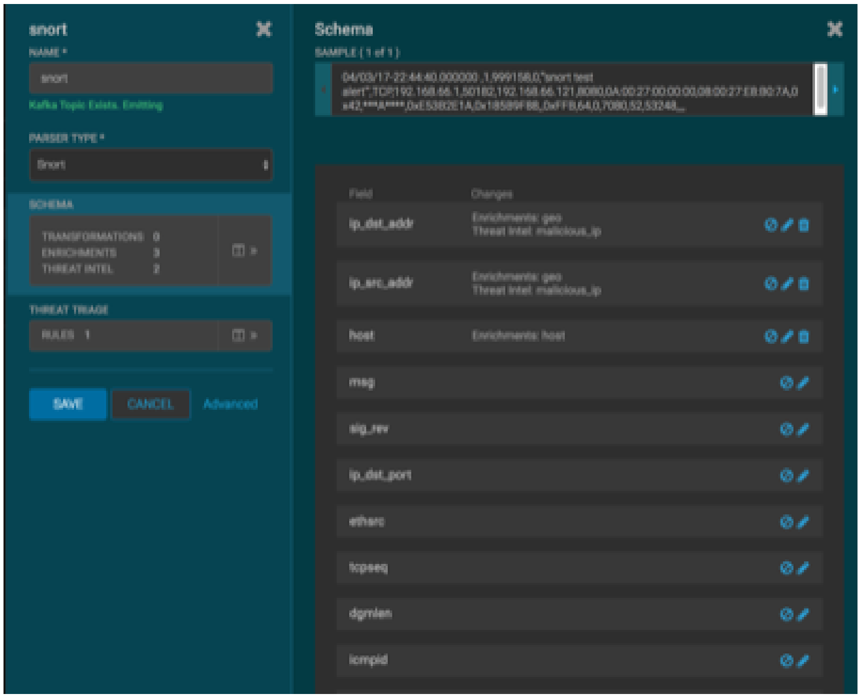
Note

To load sample data from your sensor, the sensor must be running and producing data.

1. Select the new sensor from the list of sensors on the main window.
2. Click the pencil icon in the list of tool icons ■ ⊘ ✎ 🗑 for the new sensor.

   The Management Module displays the sensor panel for the new sensor.
3. In the Schema box, click ◫ » (expand window button).

   The Management module displays a second panel and populates the panel with message, field, and value information.



snort

NAME *

snort

Kafka Topic Exists. Emitting

PARSER TYPE *

Snort

SCHEMA

TRANSFORMATIONS 0
ENRICHMENTS 3
THREAT INTEL 2

THREAT TRIAGE

RULES 1

Schema

SAMPLE (1 of 1)

04/03/17-22:44:40.000000 ,1,999158,0,"snort test alert",TCP,192.168.66.1,50182,192.168.66.121,8080,0A:00:27:00:00:00,08:00:27:E8:B0:7A,0x42,***A****,0xE5382E1A,0x18589F88,0xFFB,64,0,7080,52,53248,_

Field	Changes
ip_dst_addr	Enrichments: geo Threat Intel: malicious_ip
ip_src_addr	Enrichments: geo Threat Intel: malicious_ip
host	Enrichments: host
msg	
sig_rev	
ip_dst_port	
ethsrc	
tcpseq	
dgmlen	
icmpid	

The Sample field, at the top of the panel, displays a parsed version of a sample message from the sensor. The Management module will test your transformations against these parsed messages.

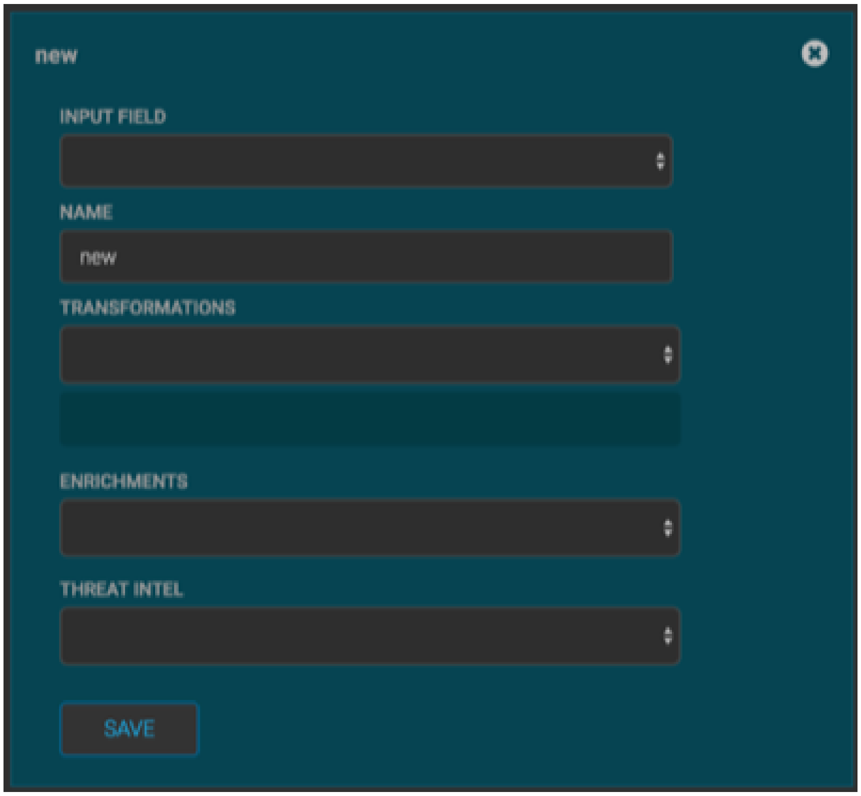
You can use the right and left arrow buttons in the Sample field to view the parsed version of each sample message available from the sensor.

4. You can apply transformations to an existing field or create a new field. Click the ✎ (edit icon) next to a field to apply transformations to that field. Or click + (plus sign) at the bottom of the Schema panel to create new fields.

   Typically users store transformations in a new field rather than overriding existing fields.

   For both options, the Management module expands the panel with a dialog box containing fields in which you can enter field information.

   Figure 3.12. New Schema Information Panel

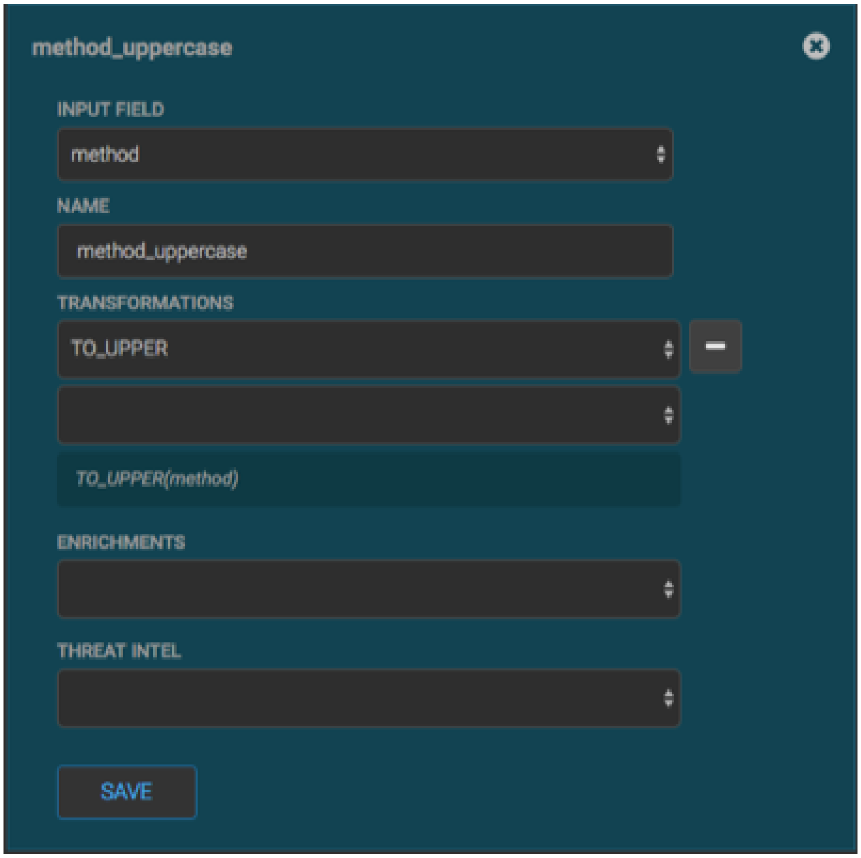
5. In the dialog box, enter the name of the new field in the **NAME** field, choose an input field from the **INPUT FIELD** box, and choose your transformation from the **TRANSFORMATIONS** field or enrichment from the **ENRICHMENTS** field.

For example, to create a new field showing the lower case version of the method field, do the following:

- Enter method-uppercase in the **NAME** field.
- Choose **`method`** from the **INPUT FIELD**.
- Choose **TO_UPPER** in the **TRANSFORMATIONS** field.

  Your new schema information panel should look like this:

Figure 3.13. Populated New Schema Information Panel



method_uppercase

INPUT FIELD: method

NAME: method_uppercase

TRANSFORMATIONS: TO_UPPER

TO_UPPER(method)

ENRICHMENTS

THREAT INTEL

6. Click **SAVE** to save your changes.
7. You can suppress fields from showing in the Index by clicking ⊘ (suppress icon).
8. Click **SAVE** to save the changed information.

   The Management module updates the Schema field with the number of changes applied to the sensor.

3.2.1.5.2. CLI Method

1. Edit the new data source enrichment configuration at `$METRON_HOME/config/zookeeper/enrichments/$DATASOURCE` to associate the `ip_src_addr` with the user enrichment.

   For example:

   ```
   {
     "index" : "squid",
     "batchSize" : 1,
     "enrichment" : {
       "fieldMap" : {
         "hbaseEnrichment" : [ "ip_src_addr" ]
       },
       "fieldToTypeMap" : {
         "ip_src_addr" : [ "whois" ]
       },
       "config" : { }
     },
     "threatIntel" : {
       "fieldMap" : { },
       "fieldToTypeMap" : { },
       "config" : { },
       "triageConfig" : {
         "riskLevelRules" : { },
         "aggregator" : "MAX",
         "aggregationConfig" : { }
       }
     },
     "configuration" : { }
   }
   ```

2. Push this configuration to ZooKeeper:

   ```
   $METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
    $METRON_HOME/zookeeper
   ```

After you have finished enriching the telemetry events, ensure that the enriched data is displaying on the Metron dashboard. For instructions on adding a new telemetry data source to the Metron Dashboard, see Adding a New Data Source.

3.2.2. Streaming Enrichment Information

Streaming enrichment information is useful when you need enrichment information in real time. Our example steps through how to associate IP addresses with user names for the Squid information. This type of information is most useful in real time as opposed to waiting for a bulk load of the enrichment information.

Streaming intelligence feeds are incorporated slightly differently than bulk loading. The enrichment information resides in its own parser topology instead of an extraction configuration file. The parser file defines the input structure and how that data can be used in enrichment. Streaming information goes to HBase rather than to Kafka so you need to configure the writer by defining both the writerClassName and Simple HBase Enrichment Writer (shew) parameters.

The following steps illustrate how to associate IP addresses from Squid with user names.

1. Define a parser topology in `$METRON_HOME/zookeeper/parsers/user.json` to handle the streaming data:

```
touch $METRON_HOME/config/zookeeper/parsers/user.json
```

2. Populate the file with the parser topology definition. For example:

```
{
 "parserClassName" : "org.apache.metron.parsers.csv.CSVParser"
 ,"writerClassName" : "org.apache.metron.enrichment.writer.
SimpleHbaseEnrichmentWriter"
 ,"sensorTopic":"user"
 ,"parserConfig":
 {
    "shew.table" : "enrichment"
   ,"shew.cf" : "t"
   ,"shew.keyColumns" : "ip"
   ,"shew.enrichmentType" : "user"
   ,"columns" : {
      "user" : 0
     ,"ip" : 1
                }
 }
}
```

where

parserClassName	The parser name.
writerClassName	The writer destination. For streaming parsers, the destination is `SimpleHbaseEnrichmentWriter`.
sensorTopic	Name of the sensor topic.
shew.table	The simple HBase enrichment writer (shew) table to which we want to write.
shew.cf	The simple HBase enrichment writer (shew) column family.
shew.keyColumns	The simple HBase enrichment writer (shew) key.
shew.enrichmentType	The simple HBase enrichment writer (shew) enrichment type.
columns	The CSV parser information. For our example, this information is the user name and IP address.

This file fully defines the input structure and how that data can be used in enrichment.

3. Push the configuration file to ZooKeeper:

   a. Create a Kafka topic:

```
/usr/hdp/current/kafka-broker/bin/kafka-topics.sh --create --zookeeper
 $ZOOKEEPER_HOST:2181 --replication-factor 1 --partitions 1 --topic user
```

When you create the Kafka topic, consider how much data will be flowing into this topic.

b. Push the configuration file to ZooKeeper.

```
$METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
 $METRON_HOME/zookeeper
```

4. Start the user parser topology by running the following:

```
$METRON_HOME/bin/start_parser_topology.sh -s user -z $ZOOKEEPER_HOST:2181 -k
 $KAKFA_HOST:6667
```

The parser topology listens for data streaming in and pushes the data to HBase. Now you have data flowing into the HBase table, but you need to ensure that the enrichment topology can be used to enrich the data flowing past.

5. Edit the new data source enrichment configuration at `$METRON_HOME/config/zookeeper/enrichments/squid` to associate the `ip_src_addr` with the user name for more user enrichment.

```
{
  "enrichment" : {
    "fieldMap" : {
      "hbaseEnrichment" : [ "ip_src_addr" ]
    },
    "fieldToTypeMap" : {
      "ip_src_addr" : [ "user" ]
    },
    "config" : { }
  },
  "threatIntel" : {
    "fieldMap" : { },
    "fieldToTypeMap" : { },
    "config" : { },
    "triageConfig" : {
      "riskLevelRules" : { },
      "aggregator" : "MAX",
      "aggregationConfig" : { }
    }
  },
  "configuration" : { }
}
```

6. Push the new data source enrichment configuration to ZooKeeper:

```
$METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
 $METRON_HOME/zookeeper
```

3.3. Configuring Indexing

The indexing topology is a topology dedicated to taking the data from a topology that has been enriched and storing the data in one or more supported indices. More specifically, the enriched data is ingested into Kafka, written in an indexing batch or bolt with a specified size, and sent to one or more specified indices. The configuration is intended to configure the indexing used for a given sensor type (for example, snort).

This section provides the following information:

3.3.1. Overview

Currently, HCP supports the following indices:

- Elasticsearch
- Solr
- HDFS under `/apps/metron/enrichment/indexed`

Depending on how you start the indexing topology, it can have HDFS and either Elasticsearch or SOLR writers running.

Just like the `Global Configuration` file, the `Indexing Configuration` file format is a JSON file stored in ZooKeeper and on disk at `$METRON_HOME/config/zookeeper/indexing`.

Errors during indexing are sent to a Kafka queue called `index_errors`.

Within the sensor-specific configuration, you can configure the individual writers. The parameters currently supported are:

- `index`: The name of the index to write to (defaulted to the name of the sensor).
- `batchSize`: The size of the batch that is written to the indices at once (defaulted to 1).
- `enabled`: Whether the index or writer is enabled (default `true`).

3.3.2. Default Configuration

If you do not configure the individual writers, the sensor-specific configuration will use the default values. You can choose to use this default configuration by either not creating the Indexing Configuration file or by entering the following in the file:

```
{
}
```

If a writer configuration is unspecified, then a warning is indicated in the Storm console. For example, `WARNING: Default and (likely) unoptimized writer config used for hdfs writer and sensor squid`. You can ignore this warning message if you intend to use the default configuration.

This default configuration uses the following configuration:

- elasticsearch writer
  - index name the same as the sensor
  - batch size of 1
  - enabled
- hdfs writer
  - index name the same as the sensor
  - batch size of 1
  - enabled

3.3.3. Specifying Index Parameters

You can specify the parameters for the writers rather than using the default values.

You can use either the Management Module or the CLI to specify writer parameters:

- Specifying Index Parameters using the Management Module [42]
- Specifying Index Parameters Using the CLI [43]

Note

Certain properties are managed by Ambari. You should only modify these properties in Ambari. If you modify these properties outside of Ambari, Ambari will overwrite them with the contents of the Global Configuration file whenever you restart a Metron topology.

For a list of the properties managed by Ambari, see Updating Properties [86].

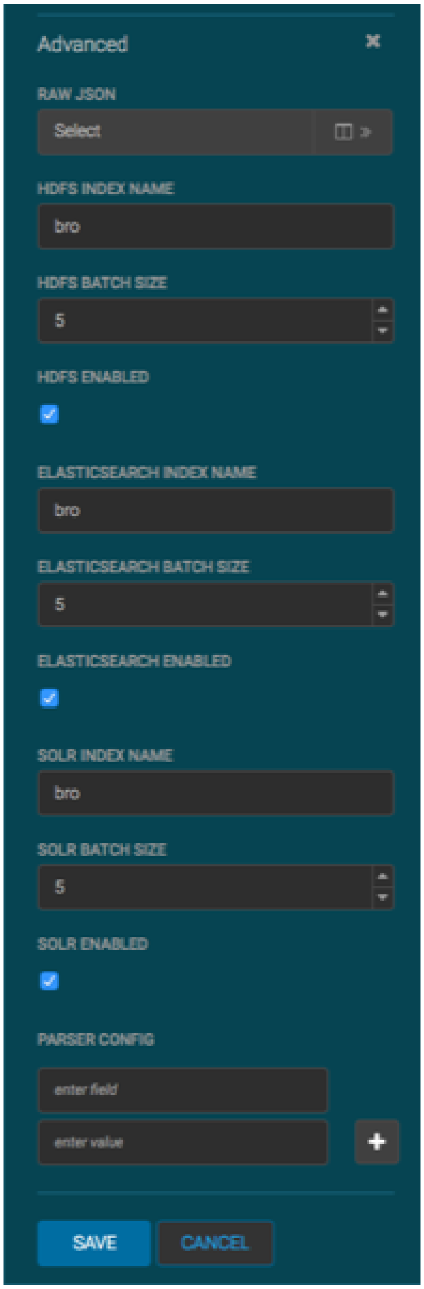
3.3.3.1. Specifying Index Parameters using the Management Module

1. Edit your sensor by clicking ✎ (the edit button) next your sensor in the Management Module.

2. Click the **Advanced** button next to **Save** and **Cancel**.

   The Management Module expands the panel to display the Advanced fields.

Figure 3.14. Management Module Advanced Panel



RAW JSON
Select

HDFS INDEX NAME
bro

HDFS BATCH SIZE
5

HDFS ENABLED
☑

ELASTICSEARCH INDEX NAME
bro

ELASTICSEARCH BATCH SIZE
5

ELASTICSEARCH ENABLED
☑

SOLR INDEX NAME
bro

SOLR BATCH SIZE
5

SOLR ENABLED
☑

PARSER CONFIG

3. Enter index configuration information for your sensor.
4. Click **Save** to save your changes and push your configuration to ZooKeeper.

3.3.3.2. Specifying Index Parameters Using the CLI

To specify the parameters for the writers rather than using the default values, you can use the following syntax in the Indexing Configuration file, located at `$METRON_HOME/config/zookeeper/indexing`.

Important

Any property that is managed by Ambari should only be modified via Ambari. Otherwise, when you restart a service, Ambari might overwrite your updates. For more information about properties managed by Ambari, see Updating Properties [86].

1. Create the Indexing Configuration file at `$METRON_HOME/config/zookeeper/indexing`.

   ```
   touch /$METRON_HOME/config/zookeeper/indexing/$sensor_name.json
   ```

2. Populate the `$sensor_name.json` file with index configuration information for each of your sensors, using syntax similar to the following:

   ```
   {
      "elasticsearch": {
         "index": "foo",
         "batchSize" : 100,
         "enabled" : true
       },
      "hdfs": {
         "index": "foo",
         "batchSize": 1,
         "enabled" : true
       },
       "alert": {
         "type": "nested"
   }
   ```

   This syntax specifies the following parameter values:

   - Elasticsearch writer or index
     - index name of "foo"
     - batch size of 100
     - enabled
   - HDFS writer or index
     - index name of "foo"
     - batch size of 1
     - enabled
   - alert

     This field must be set to `"type": "nested"`. If this field is not set, Elasticsearch can throw an error and the field will not be queryable.

3. Push the configuration to ZooKeeper:

   ```
    /usr/metron/$METRON_VERSION/bin/zk_load_configs.sh --mode PUSH -i /usr/
   metron/$METRON_VERSION/config/zookeeper -z $ZOOKEEPER_HOST:2181
   ```

3.3.4. Indexing (HDFS) Tuning

There are 48 partitions set for the indexing partition, per the enrichment exercise above.

These are the batch size settings for the Bro index.

```

cat ${METRON_HOME}/config/zookeeper/indexing/bro.json
{
"hdfs" : {
"index": "bro",
    "batchSize": 50,
    "enabled" : true
  }...
}
```

And here are the settings we used for the HDFS indexing topology:

General storm settings

```
topology.workers: 4
topology.acker.executors: 24
topology.max.spout.pending: 2000
```

Spout and Bolt Settings

```

hdfsSyncPolicy
    org.apache.storm.hdfs.bolt.sync.CountSyncPolicy
    constructor arg=100000
hdfsRotationPolicy
    bolt.hdfs.rotation.policy.units=DAYS
    bolt.hdfs.rotation.policy.count=1
kafkaSpout
    parallelism: 24
    session.timeout.ms=29999
    enable.auto.commit=false
    setPollTimeoutMs=200
    setMaxUncommittedOffsets=10000000
    setOffsetCommitPeriodMs=30000
hdfsIndexingBolt
    parallelism: 24
```

3.3.4.1. PCAP Tuning

PCAP is a specialized topology that is a Spout-only topology. Both Kafka topic consumption and HDFS writing is done within a spout to avoid the additional network hop required if using an additional bolt.

General Storm topology properties

```

topology.workers=16
topology.ackers.executors: 0


                                +__Spout and Bolt properties__
                                +
                                +kafkaSpout
                                +    parallelism: 128
                                +    poll.timeout.ms=100
                                +    offset.commit.period.ms=30000
                                +    session.timeout.ms=39000
                                +    max.uncommitted.offsets=200000000
                                +    max.poll.interval.ms=10
                                +    max.poll.records=200000
                                +    receive.buffer.bytes=431072
                                +    max.partition.fetch.bytes=10000000
                                +    enable.auto.commit=false
                                +    setMaxUncommittedOffsets=20000000
                                +    setOffsetCommitPeriodMs=30000
                                +
                                +writerConfig
                                +    withNumPackets=1265625
                                +    withMaxTimeMS=0
                                +    withReplicationFactor=1
                                +    withSyncEvery=80000
                                +    withHDFSConfig
                                +        io.file.buffer.size=1000000
                                +        dfs.blocksize=1073741824
                                +
                                +
```

3.3.5. Turning Off HDFS Writer

You can also turn off the HDFS index or writer using the following syntax in the `index.json` file.

Create or modify

```
{
   "elasticsearch": {
      "index": "foo",
      "enabled" : true
    },
   "hdfs": {
      "index": "foo",
      "batchSize": 100,
      "enabled" : false
    }
}
```

3.3.6. Support for HCP 1.4.1

Elasticsearch 5.x requires that all sensors templates have a nested alert field defined. Without this field, an error is thrown during all searches resulting in no alerts being found. This error is found in the REST service's logs:

```
QueryParsingException[[nested] failed to find nested object under path
 [alert]];
```

As a result, Elasticsearch 5.x requires changes to support HCP queries. See the following sections for these changes:

- Updating Elasticsearch Templates to Work with Elasticsearch 5.x [47]
- Updating Existing Indexes to Work with Elasticsearch 5.x [47]

3.3.6.1. Updating Elasticsearch Templates to Work with Elasticsearch 5.x

To update your existing Elasticsearch templates for each sensor so any new indexes have the appropriate field, perform the following steps:

1. Update the Elasticsearch template for each sensor, so any new indice will have the alert field.

    1. Retrieve the template:

        $SENSOR can contain wildcards, so if rollover has occurred, it's not necessary to do each index individually. The following example appends `index*` to get all indexes for the provided sensor.

        ```
        export ELASTICSEARCH="node1"
         export SENSOR="bro"
         curl -XGET "http://${ELASTICSEARCH}:9200/_template/${SENSOR}_index*?
        pretty=true" -o "${SENSOR}.template"

        ```

    2. Remove an extraneous JSON field so you can put it back later, and add the alert field

        ```
        sed -i '' '2d;$d' ./${SENSOR}.template
         sed -i '' '/"properties" : {/ a\
         "alert": { "type": "nested"},' ${SENSOR}.template
        ```

2. Verify your changes:

    ```
    python -m json.tool bro.template
    ```

3. Add the template back into Elasticsearch:

    ```
    curl -XPUT "http://${ELASTICSEARCH}:9200/_template/${SENSOR}_index" -d @
    ${SENSOR}.template
    ```

3.3.6.2. Updating Existing Indexes to Work with Elasticsearch 5.x

To update existing indexes to work with Elasticsearch 5.x, perform the following

1. Update Elasticsearch mappings with the new field for each sensor.

    ```
    curl -XPUT "http://${ELASTICSEARCH_HOST}:9200/${SENSOR}_index*/_mapping/
    ${SENSOR}_doc" -d '
     {
             "properties" : {
               "alert" : {
                 "type" : "nested"
               }
             }
     }
     '
     rm ${SENSOR}.template
    ```

3.3.7. Troubleshooting Indexing

If Ambari indicates that your indexing is stopped after you have started your indexing, this might be a problem with the Python requests module.

Check the Storm UI to to ensure that indexing has started for your topologies. If the Storm UI indicates that the topology indexing has started, you might need to install the latest version of of python-requests. Version 2.6.1 of python-requests fixes a bug introduced in version 2.5.2 that causes the system modules to break. See https://pypi.python.org/pypi/requestsfor more information.

3.4. Using Threat Intelligence Feeds

The threat intelligence topology takes a normalized JSON message and cross references it against threat intelligence, tags it with alerts if appropriate, runs the results against the scoring component of machine learning models where appropriate, and stores the telemetry in a data store. This section provides the following steps for using threat intelligence feeds:

- Prerequisites [48]
- Bulk Loading Enrichment Information [28]
- Creating a Streaming Threat Intel Feed Source [58]

Threat intelligence topologies perform the following tasks:

- Mark messages as threats based on data in external data stores
- Mark threat alerts with a numeric triage level based on a set of Stellar rules

3.4.1. Prerequisites

Perform the following tasks before configuring threat intelligence feeds:

1. Choose your threat intelligence sources.
2. **Recommended but not required:** Install a threat intelligence feed aggregator, such as SoltraEdge.

3.4.2. Bulk Loading Threat Intelligence Information

This section provides a description of bulk loading threat intelligence sources supported by HCP and the steps to bulk threat intelligence feeds.

- Bulk Loading Threat Intelligence Sources [48]
- Configuring an Extractor Configuration File [51]
- Configure Mapping for the Intelligence Feed [54]
- Running the Threat Intel Loader [54]
- Mapping Fields to HBase Threat Intel [55]

3.4.2.1. Bulk Loading Threat Intelligence Sources

You can bulk load threat intelligence information from the following sources:

- Flat File Ingestion (CVS)
- HDFS via MapReduce
- Taxii Loader

CSV File

The shell script `$METRON_HOME/bin/flatfile_loader.sh` will read data from local disk and load the enrichment or threat intel data into an HBase table.

One special thing to note here is that there is a special configuration parameter to the Extractor config that is only considered during this loader:

inputFormatHandler This specifies how to consider the data. The two implementations are `BY_LINE` and `org.apache.metron.dataloads.extractor.inputformat.WholeFile`

The default is `BY_LINE`, which makes sense for a list of CSVs where each line indicates a unit of information which can be imported. However, if you are importing a set of STIX documents, then you want each document to be considered as input to the Extractor.

Start the user parser topology by running the following:

```
$METRON_HOME/bin/start_parser_topology.sh -s user -z $ZOOKEEPER_HOST:2181 -k
 $KAKFA_HOST:6667
```

The parser topology listens for data streaming in and pushes the data to HBase. Now you have data flowing into the HBase table, but you need to ensure that the enrichment topology can be used to enrich the data flowing past.

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generate the help screen/ set of options
-e	–extractor_config	Yes	JSON document describing the extractor for this input data source
-t	–hbase_table	Yes	The HBase table to import into
-c	–hbase_cf	Yes	The HBase table column family to import into
-i	–input	Yes	The input data location on local disk. If this is a file, then that file will be loaded. If this is a directory, then the files will be loaded recursively under that directory.
-l	–log4j	No	The log4j properties file to load
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

HDFS via MapReduce

The shell script `$METRON_HOME/bin/flatfile_loader.sh` will kick off MapReduce job to load data stated in HDFS into an HBase table. The following is as example of the syntax:

```
$METRON_HOME/bin/flatfile_loader.sh -i /tmp/top-10k.csv -t enrichment -c t -
e ./extractor.json -m MR
```

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generate the help screen/ set of options
-e	–extractor_config	Yes	JSON document describing the extractor for this input data source
-t	–hbase_table	Yes	The HBase table to import into
-c	–hbase_cf	Yes	The HBase table column family to import into
-i	–input	Yes	The input data location on local disk. If this is a file, then that file will be loaded. If this is a directory, then the files will be loaded recursively under that directory.
-l	–log4j	No	The log4j properties file to load
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

Taxii Loader

The shell script `$METRON_HOME/bin/threatintel_taxii_load.sh` can be used to poll a Taxii server for STIX documents and ingest them into HBase. Taxii loader is a stand-alone Java application that never stops.

It is quite common for this Taxii server to be an aggregation server such as Soltra Edge.

In addition to the Enrichment and Extractor configs described in the following sections, this loader requires a configuration file describing the connection information to the Taxii server. The following is an example of a configuration file:

```
{
   "endpoint" : "http://localhost:8282/taxii-discovery-service"
  ,"type" : "DISCOVER"
  ,"collection" : "guest.Abuse_ch"
  ,"table" : "threat_intel"
  ,"columnFamily" : "cf"
  ,"allowedIndicatorTypes" : [ "domainname:FQDN", "address:IPV_4_ADDR" ]
}
```

where:

endpoint	The URL of the endpoint
type	POLL or DISCOVER depending on the endpoint.
collection	The Taxii collection to ingest
table	The HBase table to import into
columnFamily	The column family to import into
allowedIndicatorTypes	an array of acceptable threat intel types (see the "Enrichment Type Name" column of the Stix table above for the possibilities).

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generate the help screen/ set of options
-e	–extractor_config	Yes	JSON Document describing the extractor for this input data source
-c	–taxii_connection_config	Yes	The JSON config file to configure the connection
-p	–time_between_polls	No	The time between polling the Taxii server in milliseconds. (default: 1 hour)
-b	–begin_time	No	Start time to poll the Taxii server (all data from that point will be gathered in the first pull). The format for the date is yyyy-MM-dd HH:mm:ss
-l	–log4j	No	The Log4j Properties to load
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

3.4.2.2. Configuring an Extractor Configuration File

After you have a threat intelligence feed source, you must configure an extractor configuration file that describes the source.

1. Log in as root user to the host on which Metron is installed.
2. Create a file called `extractor_config_temp.json` and add the following content:

```
{
"config" : {
   "columns" : {
      "domain" : 0
      ,"source" : 1
   }
   ,"indicator_column" : "domain"
   ,"type" : "zeusList"
   ,"separator" : ","
  }
  ,"extractor" : "CSV"
}
```

3. You can transform and filter the enrichment data as it is loaded into HBase by using Stellar extractor properties in the extractor configuration file. HCP supports the following Stellar extractor properties:

   `value_transform`

   Transforms fields defined in the `columns` mapping with Stellar transformations. New keys introduced in the transform are added to the key metadata. For example:

   ```
   "value_transform" : {
      "domain" : "DOMAIN_REMOVE_TLD(domain)"
   ```

   `value_filter`

   Allows additional filtering with Stellar predicates based on results from the value transformations. In the following example, records whose domain property is empty after removing the TLD are omitted.

   ```
   "value_filter" : "LENGTH(domain) > 0",
     "indicator_column" : "domain",
   ```

   `indicator_transform`

   Transforms the `indicator` column independent of the value transformations. You can refer to the original indicator value by using `indicator` as the variable name, as shown in the following example. In addition, if you prefer to piggyback your transformations, you can refer to the variable `domain`, which allows your indicator transforms to inherit transformations done to this value during the value transformations.

   ```
   "indicator_transform" : {
      "indicator" : "DOMAIN_REMOVE_TLD(indicator)"
   ```

   `indicator_filter`

   Allows additional filtering with Stellar predicates based on results from the value transformations. In the following example, records whose indicator value is empty after removing the TLD are omitted.

   ```
   "indicator_filter" : "LENGTH(indicator) > 0",
     "type" : "top_domains",
   ```

   If you include all of the supported Stellar extractor properties in the extractor configuration file, it will look similar to the following:

   ```

   {
     "config" : {
       "zk_quorum" : "$ZOOKEEPER_HOST:2181",
       "columns" : {
          "rank" : 0,
          "domain" : 1
       },
       "value_transform" : {
   ```

```
      "domain" : "DOMAIN_REMOVE_TLD(domain)"
    },
    "value_filter" : "LENGTH(domain) > 0",
    "indicator_column" : "domain",
    "indicator_transform" : {
      "indicator" : "DOMAIN_REMOVE_TLD(indicator)"
    },
    "indicator_filter" : "LENGTH(indicator) > 0",
    "type" : "top_domains",
    "separator" : ","
  },
  "extractor" : "CSV"
}
```

Running a file import with the above data and extractor configuration will result in the following two extracted data records:

Indicator	Type	Value
google	top_domains	{ "rank" : "1", "domain" : "google" }
yahoo	top_domains	{ "rank" : "2", "domain" : "yahoo" }

4. To access properties that reside in the global configuration file, provide a ZooKeeper quorum via the `zk_quorum` property. If the global configuration looks like `"global_property" : "metron-ftw"`, enter the following to expand the `value_transform`:

```
"value_transform" : {
    "domain" : "DOMAIN_REMOVE_TLD(domain)",
     "a-new-prop" : "global_property"
 },
```

The resulting value data will look like the following:

Indicator	Type	Value
google	top_domains	{ "rank" : "1", "domain" : "google", "a-new-prop" : "metron-ftw" }
yahoo	top_domains	{ "rank" : "2", "domain" : "yahoo", "a-new-prop" : "metron-ftw" }

5. Remove any non-ASCII characters:

```
iconv -c -f utf-8 -t ascii extractor_config_temp.json -o extractor_config.
json
```

6. Configure the mapping for the element-to-threat intelligence feed.

This step configures which element of a tuple to cross-reference with which threat intelligence feed. This configuration is stored in ZooKeeper.

a. Log in as root user to the host that has Metron installed.

b. Cut and paste the following file into a file called `enrichment_config_temp.json`":

```
{
     "zkQuorum" : "$ZOOKEEPER_HOST:2181"
    ,"sensorToFieldList" : {
     "$DATASOURCE" : {
          "type" : "THREAT_INTEL"
         ,"fieldToEnrichmentTypes" : {
               "domain_without_subdomains" : [ "zeusList" ]
          }
     }
   }
}
```

c. Remove the non-ASCII characters:

```
iconv -c -f utf-8 -t ascii enrichment_config_temp.json -o
 enrichment_config.json
```

3.4.2.3. Configure Mapping for the Intelligence Feed

1. Configure the mapping for the element-to-threat intelligence feed.

   This step configures which element of a tuple to cross-reference with which threat intelligence feed. This configuration is stored in ZooKeeper.

   a. Log in as root user to the host on which Metron is installed.

   b. Cut and paste the following file into a file called `enrichment_config_temp.json`":

```
{
     "zkQuorum" : "$ZOOKEEPER_HOST:2181"
    ,"sensorToFieldList" : {
     "$DATASOURCE" : {
          "type" : "THREAT_INTEL"
         ,"fieldToEnrichmentTypes" : {
               "domain_without_subdomains" : [ "zeusList" ]
          }
     }
   }
}
```

   c. Remove the non-ASCII characters:

```
iconv -c -f utf-8 -t ascii enrichment_config_temp.json -o
 enrichment_config.json
```

3.4.2.4. Running the Threat Intel Loader

After you have defined the threat intelligence source, threat intelligence extractor, and threat intelligence mapping configuration, run the loader to move the data from the threat intelligence source to the Metron threat intelligence store and to store the enrichment configuration in ZooKeeper.

1. Log in to $HOST_WITH_ENRICHMENT_TAG as root.

2. Run the loader:

```
$METRON_HOME/bin/flatfile_loader.sh -n enrichment_config.json -i
 domainblocklist.csv -t threatintel -c t -e extractor_config.json
```

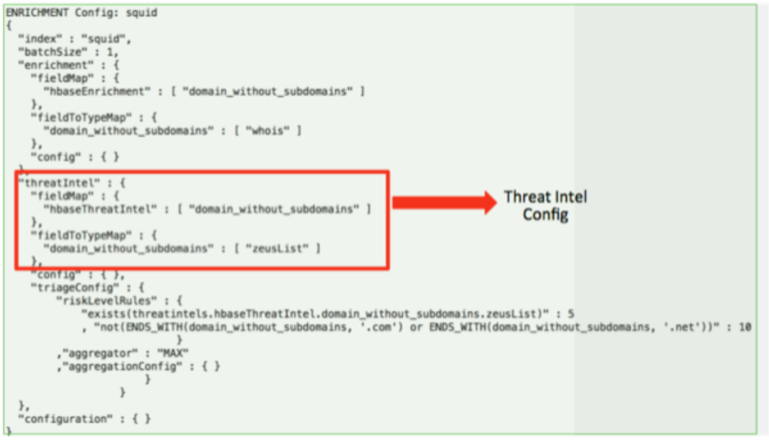
This command adds the threat intelligence data into HBase and establishes a ZooKeeper mapping. The data is extracted using the extractor and configuration defined in the `extractor_config.json` file and populated into an HBase table called `threatintel`.

3. Verify that the logs were properly ingested into HBase:

```
hbase shell
scan 'threatintel'
```

You should see a configuration for the sensor that looks something like the following:

Figure 3.15. Threat Intel Configuration



```
ENRICHMENT Config: squid
{
  "index" : "squid",
  "batchSize" : 1,
  "enrichment" : {
    "fieldMap" : {
      "hbaseEnrichment" : [ "domain_without_subdomains" ]
    },
    "fieldToTypeMap" : {
      "domain_without_subdomains" : [ "whois" ]
    },
    "config" : { }
  },
  "threatIntel" : {
    "fieldMap" : {
      "hbaseThreatIntel" : [ "domain_without_subdomains" ]
    },
    "fieldToTypeMap" : {
      "domain_without_subdomains" : [ "zeusList" ]
    },
    "config" : { },
    "triageConfig" : {
        "riskLevelRules" : {
            "exists(threatintels.hbaseThreatIntel.domain_without_subdomains.zeusList)" : 5
           , "not(ENDS_WITH(domain_without_subdomains, '.com') or ENDS_WITH(domain_without_subdomains, '.net'))" : 10
                        }
        ,"aggregator" : "MAX"
        ,"aggregationConfig" : { }
                    }
                }
  },
  "configuration" : { }
}
```

4. Generate some data to populate the Metron Dashboard.

3.4.2.5. Mapping Fields to HBase Threat Intel

Now that you have data flowing into the HBase table, you need to ensure that the threat intel topology can be used to enrich the data flowing past.

You can perform this step using either the HCP Management module or the CLI. Both of these methods are described in the following subsections.

3.4.2.5.1. Management Module Method

Defining the threat intel topology is very similar to defining the transformation and enrichment topology.

Each of the parser outputs is added or modified in the **Schema** field. To modify any of the parser outputs, complete the following steps:

Note

To load sample data from your sensor, the sensor must be running and producing data.

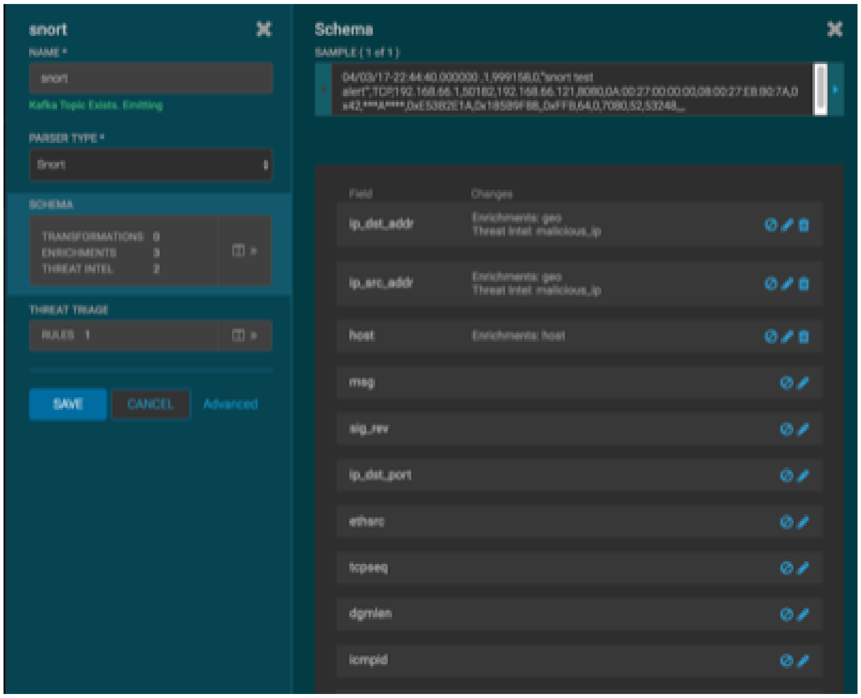
1. Select the new sensor from the list of sensors on the main window.

2. Click the pencil icon in the list of tool icons ■ ⊘ ✎ 🗑 for the new sensor.

   The Management module displays the sensor panel for the new sensor.

3. In the Schema box, click ◫ » (expand window button).

   The Management module displays a second panel and populates the panel with message, field, and value information.



snort

NAME *

snort

Kafka Topic Exists. Emitting

PARSER TYPE *

Snort

SCHEMA

TRANSFORMATIONS 0
ENRICHMENTS 3
THREAT INTEL 2

THREAT TRIAGE

RULES 1

SAVE CANCEL Advanced

Schema

SAMPLE (1 of 1)

04/03/17-22:44:40.000000 ,1,999158,0,"snort test alert",TCP,192.168.66.1,50182,192.168.66.121,8080,0A:00:27:00:00:00,08:00:27:E8:B0:7A,0x42,***A****,0xE5382E1A,0x18589F88,0xFFB,64,0,7080,52,53248,,,

Field	Changes
ip_dst_addr	Enrichments: geo Threat Intel: malicious_ip
ip_src_addr	Enrichments: geo Threat Intel: malicious_ip
host	Enrichments: host
msg	
sig_rev	
ip_dst_port	
ethsrc	
tcpseq	
dgmlen	
icmpid	

The Sample field, at the top of the panel, displays a parsed version of a sample message from the sensor. The Management module will test your threat intel against these parsed messages.

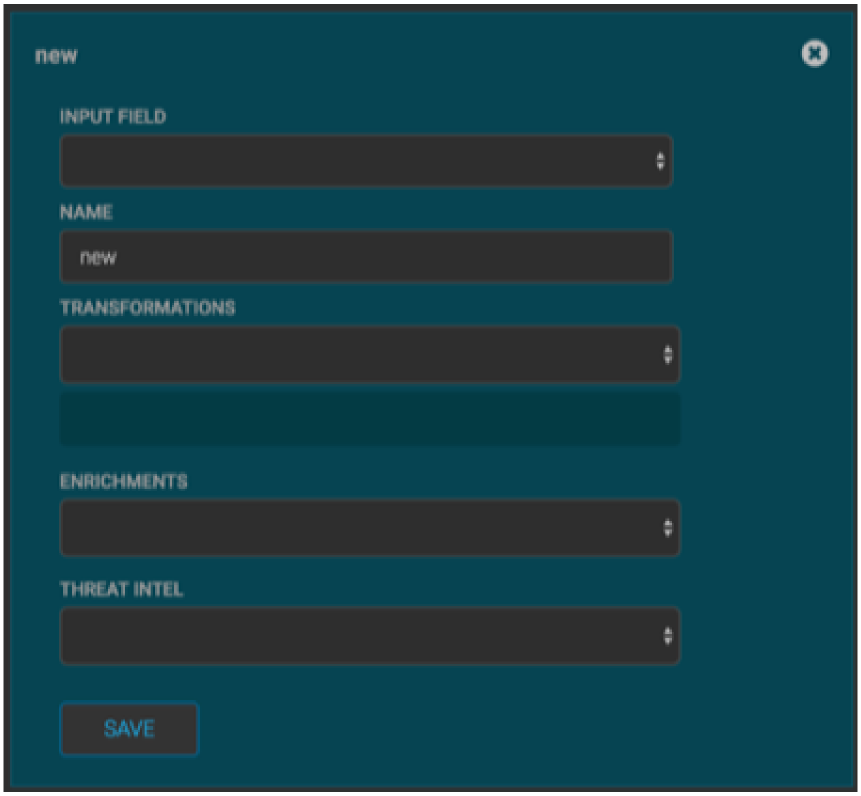
You can use the right and left arrow buttons in the Sample field to view the parsed version of each sample message available from the sensor.

4. You can apply threat intel to an existing field or create a new field. Click the ✎ (edit icon) next to a field to apply transformations to that field. Or click + (plus sign) at the bottom of the Schema panel to create new fields.

   Typically users choose to create and transform a new field, rather than transforming an existing field.

   For both options, the Management Module expands the panel with a dialog box containing fields in which you can enter field information.

Figure 3.16. New Schema Information Panel



5. In the dialog box, enter the name of the new field in the **NAME** field, choose an input field from the **INPUT FIELD** box, and choose your transformation from the **THREAT INTEL** field .

6. Click **SAVE** to save your changes.

7. You can suppress fields from the Index by clicking ⊘(suppress icon).

8. Click **SAVE** to save the changed information.

   The Management module updates the Schema field with the number of changes applied to the sensor.

3.4.2.5.2. CLI Method

1. Edit the new data source threat intel configuration at `$METRON_HOME/config/zookeeper/enrichments/$DATASOURCE` to associate the `ip_src_addr` with the user enrichment.

   For example:

```
{
  "index" : "squid",
  "batchSize" : 1,
  "enrichment" : {
    "fieldMap" : {
      "hbaseEnrichment" : [ "ip_src_addr" ]
    },
    "fieldToTypeMap" : {
      "ip_src_addr" : [ "whois" ]
    },
    "config" : { }
  },
  "threatIntel" : {
    "fieldMap" : { },
    "fieldToTypeMap" : { },
    "config" : { },
    "triageConfig" : {
      "riskLevelRules" : { },
      "aggregator" : "MAX",
      "aggregationConfig" : { }
    }
  },
  "configuration" : { }
}
```

2. Push this configuration to ZooKeeper:

```
$METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
 $METRON_HOME/zookeeper
```

After you have finished enriching the telemetry events, ensure that the enriched data is displaying on the Metron dashboard. For instructions on adding a new telemetry data source to the Metron Dashboard, see Adding a New Data Source.

3.4.3. Creating a Streaming Threat Intel Feed Source

Streaming intelligence feeds and are incorporated slightly differently than data from a flat CSV file. This section describes how to define a streaming source.

Because we are defining a streaming source, we need to define a parser topology to handle the streaming data. In order to do that, we will need to create a file in `$METRON_HOME/zookeeper/parsers/user.json`.

1. Define a parser topology to handle the streaming data:

```
touch $METRON_HOME/zookeeper/parsers/user.json
```

2. Populate the file the parser topology definition.

The following example assumes CSV data where the first field is a user and the second field is an ip address (for example, my_username,127.0.0.1).

```
{
 "parserClassName" : "org.apache.metron.parsers.csv.CSVParser"
 ,"writerClassName" : "org.apache.metron.enrichment.writer.
SimpleHbaseEnrichmentWriter"
 ,"sensorTopic":"user"
 ,"parserConfig":
 {
    "shew.table" : "threatintel"
   ,"shew.cf" : "t"
   ,"shew.keyColumns" : "ip"
   ,"shew.enrichmentType" : "user"
   ,"columns" : {
      "user" : 0
     ,"ip" : 1
                }
 }
}
```

where

parserClassName	The parser name.
writerClassName	The writer destination. For streaming parsers, the destination is `SimpleHbaseEnrichmentWriter`.
sensorTopic	Name of the sensor topic.
shew.table	The simple HBase enrichment writer (shew) table to which we want to write.
shew.cf	The simple HBase enrichment writer (shew) column family.
shew.keyColumns	The simple HBase enrichment writer (shew) key.
shew.enrichmentType	The simple HBase enrichment writer (shew) enrichment type.
columns	The CSV parser information. For our example, this information is the user name and IP address.

This file fully defines the input structure and how that data can be used in enrichment.

3. Push the configuration file to ZooKeeper:

   a. Create a Kafka topic:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-topics.sh --create --zookeeper
    $ZOOKEEPER_HOST:2181 --replication-factor 1 --partitions 1 --topic user
   ```

   When you create the Kafka topic, consider how much data will be flowing into this topic.

   b. Push the configuration file to ZooKeeper.

   ```
   $METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
    $METRON_HOME/zookeeper
   ```

Now that you are ingesting data into HBase, you need to use that enrichment data to enrich a separate datasource.

4. Set up an enrichment for `$METRON_HOME/config/zookeeper/enrichments/$DATASOURCE` to associate the ip_src_addr with the user enrichment that you streamed in from above.

   For example:

```
{
```

```
  "enrichment" : {
    "fieldMap" : {
      "hbaseEnrichment" : [ "ip_src_addr" ]
    },
    "fieldToTypeMap" : {
      "ip_src_addr" : [ "user" ]
    },
    "config" : { }
  },
  "threatIntel" : {
    "fieldMap" : { },
    "fieldToTypeMap" : { },
    "config" : { },
    "triageConfig" : {
      "riskLevelRules" : { },
      "aggregator" : "MAX",
      "aggregationConfig" : { }
    }
  },
  "configuration" : { }
}
```

5. Push this configuration to ZooKeeper:

```
$METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
 $METRON_HOME/zookeeper
```

3.5. Prioritizing Threat Intelligence

Not all threat intelligence indicators are equal. Some require immediate response, while others can be dealt with or investigated as time and availability permits. As a result you need to triage and rank threats by severity.

In HCP, you assign severity by associating possibly complex conditions with numeric scores. Then, for each message, you use a configurable aggregation function to evaluate the set of conditions and to aggregate the set of numbers for matching conditions This aggregated score is added to the message in the `threat.triage.level` field. For more information about Stellar and threat triage configurations, see Using Stellar to Set up Threat Triage Configurations [114].

This section details the steps to understand and create severity rules, configure them in ZooKeeper, and view the resulting alerts in the HCP Investigation module:

- Prerequisites [60]
- Performing Threat Triage Using the Management Module [61]
- Uploading the Threat Triage Configuration to ZooKeeper [65]
- Viewing Triaged or Scored Alerts [66]

3.5.1. Prerequisites

Before you can prioritize a threat intelligence enrichment, you must ensure that the enrichment is working properly

3.5.2. Performing Threat Triage Using the Management Module

To create a threat triage rule configuration, you must first define your rules. These rules identify the conditions in the data source data flow and associate alert scores with those conditions. Following are some examples:

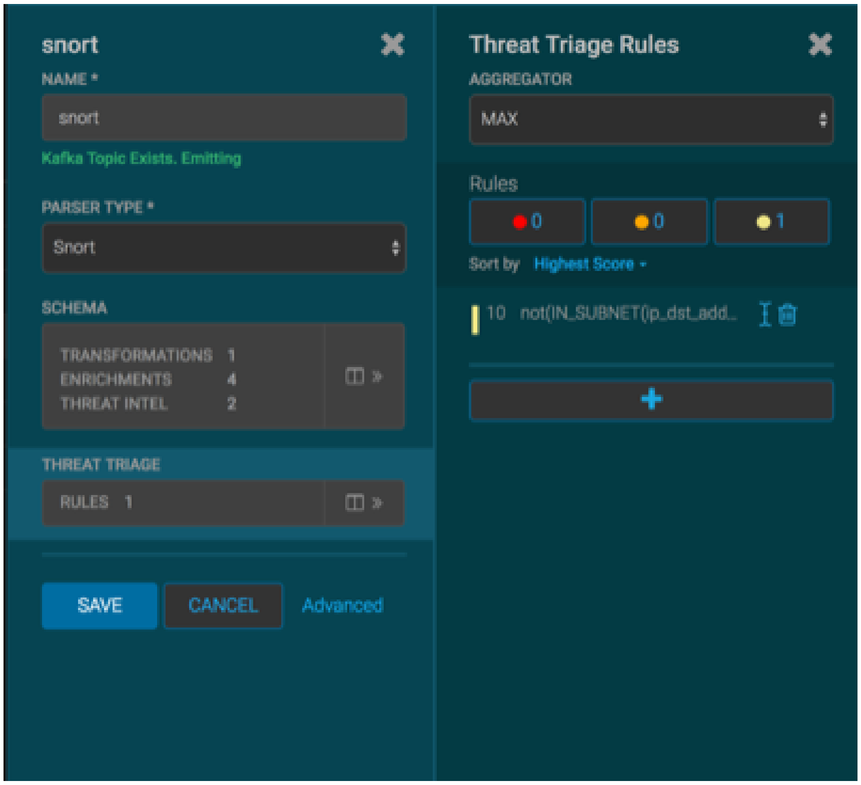
Rule 1	If a threat intelligence enrichment type is alerted, imagine that you want to receive an alert score of 5.
Rule 2	If the URL ends with neither .com nor .net, then imagine that you want to receive an alert score of 10.

To create these rules, complete the following steps:

1. On the sensor panel, in the Threat Triage field, click the ◫ » icon (expand window).

   The module displays the Threat Triage Rules panel.

   Figure 3.17. Threat Triage Rules Panel

   snort

   NAME * snort

   Kafka Topic Exists. Emitting

   PARSER TYPE * Snort

   SCHEMA

   TRANSFORMATIONS 1
   ENRICHMENTS 4
   THREAT INTEL 2

   THREAT TRIAGE

   RULES 1

   Threat Triage Rules

   AGGREGATOR MAX

   Rules

   ● 0 ● 0 ● 1

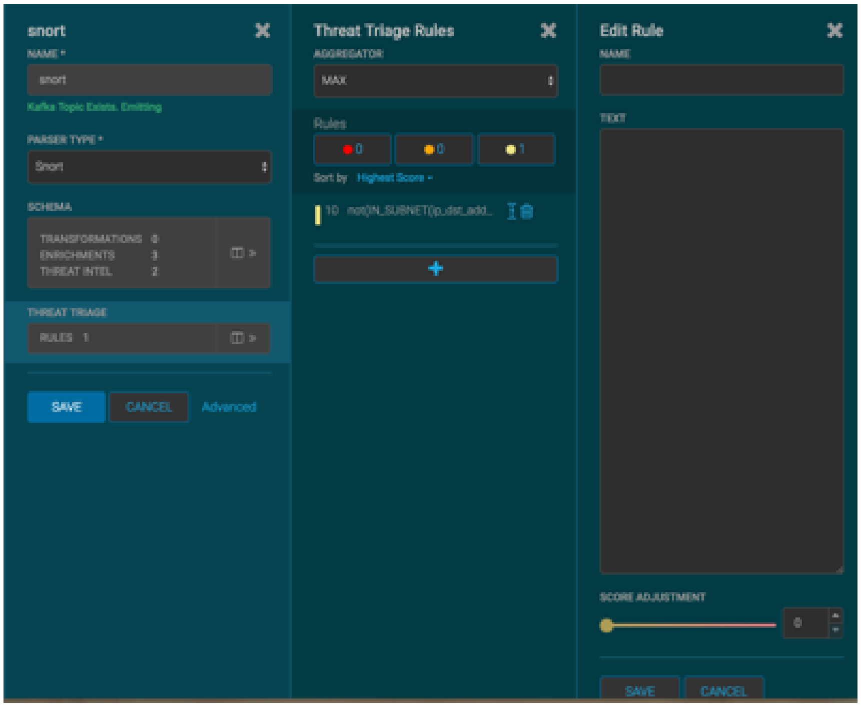
   Sort by Highest Score

   10 not(IN_SUBNET(ip_dst_add...

2. Click the + button to add a rule.

   The module displays the **Edit Rule** panel.

Figure 3.18. Edit Rule Panel



NAME * snort

Kafka Topic Exists. Emitting

PARSER TYPE * Snort

SCHEMA

TRANSFORMATIONS 0
ENRICHMENTS 3
THREAT INTEL 2

THREAT TRIAGE

RULES 1

AGGREGATOR MAX

Rules ● 0 ● 0 ● 1

Sort by Highest Score

10 not(IN_SUBNET(ip_dst_add...

NAME

TEXT

SCORE ADJUSTMENT 0

3. Assign a name to the new rule by entering the name in the NAME field.

4. In the Text field, enter the syntax for the new rule.

   For example:

   ```
   Exists(IsAlert)
   ```

5. Use the **SCORE ADJUSTMENT** slider to choose the threat score for the rule.

6. Click **SAVE** to save the new rule.

   The new rule is listed in the Threat Triage Rules panel.

7. Choose how you want to aggregate your rules by choosing a value from the Aggregator menu.

   You can choose between:

MAX	The maximum of all of the associated values for matching queries.
MIN	The minimum of all of the associated values for matching queries.
MEAN	the mean of all of the associated values for matching queries.
POSITIVE_MEAN	The mean of the positive associated values for the matching queries.

8. You can use the **Rules** section and the **Sort by** pull down menu below the **Rules** section to filter how threat triages display.

For example, to display only high levels alerts, click the box containing the red indicator. To sort the high level alerts from highest to lowest, choose **Highest Score** from the **Sort by** pull down menu.

9. Click **SAVE** on the Sensor panel to save your changes.

3.5.3. Performing Threat Triage Using the CLI

The perform threat triage using the CLI, you must complete the following steps:

- Creating the Threat Triage Rule Configuration [63]
- Uploading the Threat Triage Configuration to ZooKeeper [65]
- Viewing Triaged or Scored Alerts [66]

3.5.3.1. Creating the Threat Triage Rule Configuration

The goal of threat triage is to prioritize the alerts that pose the greatest threat and need urgent attention. To create a threat triage rule configuration, you must first define your rules. Each rule has a predicate to determine whether or not the rule applies. The threat score from each applied rule is aggregated into a single threat triage score that is used to prioritize high risk threats.

Following are some examples:

Rule 1 If a threat intelligence enrichment type zeusList is alerted, imagine that you want to receive an alert score of 5.

Rule 2 If the URL ends with neither .com nor .net, then imagine that you want to receive an alert score of 10.

Rule 3 For each message, the triage score is the maximum score across all conditions.

These example rules become the following example configuration:

```
“triageConfig” : {
   “riskLevelRules” : [
{
“name” : “zeusList is alerted"
“comment” : “Threat intelligence enrichment type zeusList is alerted."
“rule”: "exists(threatintels.hbaseThreatIntel.domain_without_subdomains.
zeusList)”
“score” : 5
}
{
“name” : “Does not end with .com or .net"
“comment” : “The URL ends with neither .com nor .net."
“rule”: “not(ENDS_WITH(domain_without_subdomains, ‘.com’) or
 ENDS_WITH(domain_without_subdomains, ‘.net’))“ : 10
“score” : 10
}
]
      ,“aggregator” : “MAX”
```

```
       ,”aggregationConfig” : { }
}
```

You can use the 'reason' field to generate a message explaining why a rule fired. One or more rules may fire when triaging a threat. Having detailed, contextual information about the environment when a rule fired can greatly assist actioning the alert. For example:

Rule 1 For hostname, the value exceeds threshold of value-threshold, receive an alert score of 10.

This example rule becomes the following example configuration:

```
“triageConfig” : {
   “riskLevelRules” : [
      {
      “name” : “Abnormal Value"
      “comment” : “The value has exceeded the threshold",
      "reason": "FORMAT('For '%s' the value '%d' exceeds threshold of '%d',
 hostname, value, value_threshold)"
      “rule”: "value > value_threshold”,
      “score” : 10
      }
   ],
   “aggregator” : “MAX”,
   ”aggregationConfig” : { }
}
```

If the value threshold is exceeded, Threat Triage will generate a message similar to the following:

```

"threat.triage.score": 10.0,
"threat.triage.rules.0.name": "Abnormal Value",
"threat.triage.rules.0.comment": "The value has exceeded the threshold",
"threat.triage.rules.0.score": 10.0,
"threat.triage.rules.0.reason": "For '10.0.0.1' the value '101' exceeds
 threshold of '42'"
```

where

riskLevelRules This is a list of rules (represented as Stellar expressions) associated with scores with optional names and comments.

name The name of the threat triage rule.

comment A comment describing the rule.

reason An optional Stellar expression that when executed results in a custom message describing why the rule fired.

rule The rule, represented as a Stellar statement.

score Associated threat triage score for the rule.

aggregator An aggregation function that takes all non-zero scores representing the matching queries from `riskLevelRules` and aggregates them into a single score.

You can choose between:

MAX	The maximum of all of the associated values for matching queries.
MIN	The minimum of all of the associated values for matching queries.
MEAN	the mean of all of the associated values for matching queries.
POSITIVE_MEAN	The mean of the positive associated values for the matching queries.

3.5.3.2. Uploading the Threat Triage Configuration to ZooKeeper

To apply this example triage configuration, you must modify the configuration for the new sensor in the enrichment topology.

1. Log in as root user to the host on which Metron is installed.
2. Modify `$METRON_HOME/config/zookeeper/sensors/$DATASOURCE.json`.

   Because the configuration in ZooKeeper might be out of sync with the configuration on disk, ensure that they are in sync by downloading the ZooKeeper configuration first:

   ```
   $METRON_HOME/bin/zk_load_configs.sh -m PULL -z $ZOOKEEPER_HOST:2181 -f -o
    $METRON_HOME/config/zookeeper
   ```
3. Validate that the enrichment configuration for the data source exists:

   ```
   cat $METRON_HOME/config/zookeeper/enrichments/$DATASOURCE.json
   ```
4. In the `$METRON_HOME/config/zookeeper/enrichments/$DATASOURCE.json` file, add the following to the `triageConfig` section in the threat intelligence section.

   For example:

   ```
   "threatIntel" : {
       "fieldMap" : {
         "hbaseThreatIntel" : [ "domain_without_subdomains" ]
       },
       "fieldToTypeMap" : {
         "domain_without_subdomains" : [ "zeusList" ]
       },
       "config" : { },
       "triageConfig" : {
         "riskLevelRules" : {
            "exists(threatintels.hbaseThreatIntel.domain_without_subdomains.
   zeusList)" : 5
                 , "not(ENDS_WITH(domain_without_subdomains, '.com') or
    ENDS_WITH(domain_without_subdomains, '.net'))" : 10
                              }
           ,"aggregator" : "MAX"
           ,"aggregationConfig" : { }
                         }
   ```

```
                         }
  }
```

5. Ensure that the aggregator field indicates MAX.

6. Push the configuration back to ZooKeeper:

```
$METRON_HOME/bin/zk_load_configs.sh -m PUSH -z $ZOOKEEPER_HOST:2181 -i
 $METRON_HOME/config/zookeeper
```

3.5.3.3. Viewing Triaged or Scored Alerts

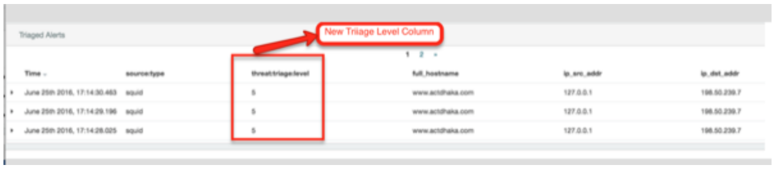
You can view triaged alerts in the indexing topic in Kafka or in the triaged alert panel in the HCP Metron dashboard.

An alert in the indexing topic in Kafka will appear similar to the following:

```
> THREAT_TRIAGE_PRINT(conf)
############################################################################################
# Name # Comment # Triage Rule # Score # Reason #
############################################################################################
# Abnormal DNS Port # # source.type == "bro" and protocol == "dns" and
 ip_dst_port != 53 # 10 # FORMAT("Abnormal DNS Port: expected: 53, found: %s:
%d", ip_dst_addr, ip_dst_port) #
##########################################################
```

The following figure shows you an example of a triaged alert panel in the HCP Metron dashboard. For URLs from cnn.com, no threat alert is shown, so no triage level is set. Notice the lack of a threat.triage.level field:

Figure 3.19. Investigation Module Triaged Alert Panel



Triaged Alerts

New Triiage Level Column

1 2 »

Time	source:type	threat:triage:level	full_hostname	ip_src_addr	ip_dst_addr
June 25th 2016, 17:14:30.463	squid	5	www.actdhaka.com	127.0.0.1	198.50.239.7
June 25th 2016, 17:14:29.196	squid	5	www.actdhaka.com	127.0.0.1	198.50.239.7
June 25th 2016, 17:14:28.025	squid	5	www.actdhaka.com	127.0.0.1	198.50.239.7

3.6. Setting Up Enrichment Configurations

The `enrichment` topology is a topology dedicated to taking the data from the parsing topologies that have been normalized into the Metron data format (for example, a JSON Map structure with `original_message` and `timestamp`) and

- Enriching messages with external data from data stores (for example, hbase) by adding new fields based on existing fields in the messages.
- Marking messages as threats based on data in external data stores.
- Marking threat alerts with a numeric triage level based on a set of Stellar rules.

The configuration for the `enrichment` topology, the topology primarily responsible for enrichment and threat intelligence enrichment, is defined by JSON documents stored in ZooKeeper.

There are two types of configurations, global and sensor specific.

3.6.1. Sensor Configuration

The sensor specific configuration is intended to configure the individual enrichments and threat intelligence enrichments for a given sensor type (for example, `snort`).

Just like the global config, the sensor configuration format is a JSON object stored in ZooKeeper. The configuration is a complex JSON object with the following top level fields:

- enrichment : A complex JSON object representing the configuration of the enrichments
- threatIntel : A complex JSON object representing the configuration of the threat intelligence enrichments

The sensor enrichment configuration uses the following fields:

- fieldToTypeMap - In the case of a simple HBase enrichment (a key/value lookup), the mapping between fields and the enrichment types associated with those fields must be known. This enrichment type is used as part of the HBase key. Note: applies to hbaseEnrichment only. | `"fieldToTypeMap" : { "ip_src_addr" : [ "asset_enrichment" ] }` |
- fieldMap - The map of enrichment bolts names to configuration handlers which know how to divide the message. The simplest of which is just a list of fields. More complex examples would be the stellar enrichment which provides stellar statements. Each field listed in the array arg is sent to the enrichment referenced in the key. Cardinality of fields to enrichments is many-to-many. | `"fieldMap": {"hbaseEnrichment": ["ip_src_addr","ip_dst_addr"]}` |
- config - The general configuration for the enrichment.

The `config` map is intended to house enrichment specific configuration. For instance, for the `hbaseEnrichment`, the mappings between the enrichment types to the column families is specified.

The `fieldMap`contents are of interest because they contain the routing and configuration information for the enrichments. When we say 'routing', we mean how the messages get split up and sent to the enrichment adapter bolts.

The simplest, by far, is just providing a simple list as in

```
"fieldMap": {
      "geo": [
        "ip_src_addr",
        "ip_dst_addr"
      ],
      "host": [
        "ip_src_addr",
        "ip_dst_addr"
      ],
      "hbaseEnrichment": [
        "ip_src_addr",
        "ip_dst_addr"
      ]
      }
```

Based on this sample config, both `ip_src_addr` and `ip_dst_addr` will go to the `geo`, `host`, and `hbaseEnrichment` adapter bolts.

3.7. Global Configuration

Global configurations are applied to all data sources as opposed to other configurations that are applied to a specific sensor. For example, every message from every sensor is validated against global configuration rules.

Various parts of the HCP stack use the global configuration and methods for using and modifying the global configuration properties are documented throughout the HCP documentation. The following is an index of the global configuration properties and their associated Ambari properties if they are managed by Ambari.

Important

Any property that is managed by Ambari should only be modified via Ambari. Otherwise, when you restart a service, Ambari might overwrite your updates. For more information, see Updating Properties [86].

Table 3.1. Global Configuration Properties

Property Name	Subsystem	Type	Ambari Property
es.clustername	Indexing	String	es_cluster_name
es.ip	Indexing	String	es_hosts
es.port	Indexing	String	es_port
es.date.format	Indexing	String	es_date_format
fieldValidations	Parsing	Object	N/A
parser.error.topic	Parsing	String	N/A
stellar.function.paths	Stellar	CSV String	N/A
stellar.function.resolver.includes	Stellar	CSV String	N/A
stellar.function.resolver.excludes	Stellar	CSV String	N/A
profiler.period.duration	Profiler	Integer	profiler_period_duration
profiler.period.duration.units	Profiler	String	profiler_period_units
update.hbase.table	REST/Indexing	String	update_hbase_table
update.hbase.cf	REST-Indexing	String	update_hbase_cf
geo.hdfs.file	Enrichment	String	geo_hdfs_file

1. To configure a global configuration file, create a file called `global.json` at `$METRON_HOME/config/zookeeper`.

2. Populate the file with enrichment configurations you want to apply to all sensors.

   The file should have the following format:

```
{
  "es.clustername": "metron",
  "es.ip": "node1",
  "es.port": "9300",
  "es.date.format": "yyyy.MM.dd.HH",
  "fieldValidations" : [
              {
                "input" : [ "ip_src_addr", "ip_dst_addr" ],
                "validation" : "IP",
                "config" : {
```

```
                            "type" : "IPV4"
                                      }
                   }
                                ]
}
```

where

es.ip

A single or collection of elastic search master nodes.

They may be specified via the widely accepted `hostname:port` syntax. If a port is not specified, then a separate global property `es.port` is required:

- Example: `es.ip` : [“10.0.0.1:1234”, “10.0.0.2:1234”]
- Example: `es.ip` : “10.0.0.1” (thus requiring `es.port` to be specified as well)
- Example: `es.ip` : “10.0.0.1:1234” (thus not requiring `es.port` to be specified)

es.port

The port of the elastic search master node.

This is not strictly required if the port is specified in the `es.ip global` property as described above. It is expected that this be an integer or a string representation of an integer.

- Example: `es.port` : “1234"
- Example: `es.port` : 1234

es.clustername

The elastic search cluster name to which you want to write.

- Example: `es.clustername` : “metron” (providing your ES cluster is configured to have metron be a valid cluster name)

es.date.format

We shard the indices first by sensor and then by date.

This provides the granularity time-wise that we shard.

- Example: `es.date.format` : “yyyy.MM.dd.HH” (this would shard by hour creating, for example, a Bro shard of bro_2016.01.01.01, bro_2016.01.01.02, etc.)
- Example: `es.date.format` : “yyyy.MM.dd” (this would shard by day, creating, for example, a Bro shard of bro_2016.01.01, bro_2016.01.02, etc.)

fieldValidations

A validation framework that enables you to construct validation rules that cross all sensors.

This is done in the form of validation plugins where assertions about fields or whole messages can be made.

input
: An array of input fields or a single field. If this is omitted, then the whole messages is passed to the validator.

config
: A String to Object map for validation configuration. This is optional if the validation function requires no configuration.

validation
: The validation function to be used. This is one of the following:

- STELLAR: Execute a Stellar Language statement. Expects the query string in the `condition` field of the config.
- IP: Validates that the input fields are an IP address. By default, if no configuration is set, it assumes IPV4, but you can specify the type by passing in type with either `IPV6` or `IPV4` or by passing in a list `[IPV4,IPV6]` in which case the input(s) will be validated against both.
- DOMAIN: Validates that the fields are all domains.
- EMAIL: Validates that the fields are all email addresses.
- URL: Validates that the fields are all URLs.
- DATE: Validates that the fields are a date. Expects `format` in the config.
- INTEGER: Validates that the fields are an integer. String representation of an integer is allowed.
- REGEX_MATCH: Validates that the fields match a regex. Expects `pattern` in the config.
- NOT_EMPTY: Validates that the fields exist and are not empty (after trimming.)

You can also create a validation using Stellar. The following validation uses Stellar to validate an `ip_src_addr` similar to the "validation":"IP"" example above:

```
"fieldValidations" : [
              {
                "validation" : "STELLAR",
                "config" : {
                    "condition" : "IS_IP(ip_src_addr, 'IPV4')"
                           }
              }
                       ]
```

3.8. Configuring the Profiler

A profile describes the behavior of an entity on a network. This feature is typically used by a data scientist and you should coordinate with the data scientist determine if they will need your assistance with customizing the Profiler values.

The Profiler is installed in the HCP install and runs as an independent Storm topology. The configuration for the Profiler topology is stored in ZooKeeper at `/metron/topology/profiler`. These properties also exist in the default installation of HCP at `$METRON_HOME/config/zookeeper/profiler.json`. The values can be changed on disk and then uploaded to ZooKeeper using `$METRON_HOME/bin/zk_load_configs.sh`.

For more information on creating a profile, see Creating Profiles.

Note

The Profiler can persist any serializable object, not just numeric values.

Table 3.2. Profiler Properties

Settings.	**Description**
profiler.workers	The number of worker processes to create for the topology.
profiler.executors	The number of executors to spawn per component.
profiler.input.topic	The name of the Kafka topic from which to consume data.
profiler.output.topic	The name of the Kafka topic to which profile data is written. Only used with profiles that use the [`triage` result field](#result).
profiler.period.duration	The duration of each profile period. This value should be define along with `profiler.period.duration.units`.
profiler.period.duration.units	The units used to specify the profile period duration. This value should be defined along with `profiler.period.duration`.
profiler.ttl	If a message has not been applied to a Profile in this period of time, the Profile will be forgotten and its resources will be cleaned up. This value should be defined along with `profiler.ttl.units`.
profiler.ttl.units	The units used to specify the `profiler.ttl`
profiler.hbase.salt.divisor	A salt is prepended to the row key to help prevent hotspotting. This constant is used to generate the sale. Ideally, this constant should be roughly equal to the number of nodes in the HBase cluster.

profiler.hbase.table	The name of the HBase table that profiles are written to.
profiler.hbase.column.family	The column family used to store profiles.
profiler.hbase.batch	The number of puts that are written in a single batch.
profiler.hbase.flush.interval.seconds	The maximum number of seconds between batch writes to HBase.

3.9. Creating an Index Template

To work with a new data source data in the Metron dashboard, you need to ensure that the data is landing in the search index (Elasticsearch) with the correct data types. This can be achieved by defining an index template.

Note

You will need to update the Index template after you add or change enrichments for a data source.

1. Run the following command to create an index template for the new data source.

   The following is an example of an index template for a new sensor called 'sensor1'.

   - The template applies to any indices that are named sensor1_index*.
   - The index has one document type that must be named sensor1_doc.
   - The index is expected to contain timestamps.
   - The properties section defines the types of each field. This example defines the five common fields that most sensors contain.
   - Additional fields can be added following the five that are already defined.

```
curl -XPOST $SEARCH_HOST:$SEARCH_PORT/_template/$DATASOURCE_index -d '
{
  "template": "sensor1_index*",
  "mappings": {
    "sensor1_doc": {
      "properties": {
        "timestamp": {
          "type": "date",
          "format": "epoch_millis"
        },
        "ip_src_addr": {
          "type": "ip"
        },
        "ip_src_port": {
          "type": "integer"
        },
        "ip_dst_addr": {
          "type": "ip"
        },
        "ip_dst_port": {
          "type": "integer"
        }
      }
    }
  }
}
```

2. By default, Elasticsearch will attempt to analyze all fields of type string. This means that Elasticsearch will tokenize the string and perform additional processing to enable free-form text search. In many cases, you want to treat each of the string fields as enumerations. This is why most fields in the index template are `not_analyzed`.

3. An index template will only apply for indices that are created after the template is created. Delete the existing indices for the new data source so that new ones can be generated with the index template.

   ```
   curl -XDELETE $SEARCH_HOST:9200/$DATSOURCE*
   ```

4. Wait for the new data source index to be re-created. This might take a minute or two based on how fast the new data source data is being consumed in your environment.

   ```
   curl -XGET $SEARCH_HOST:9200/$DATASOURCE*
   ```

3.10. Configuring the Metron Dashboard to View the New Data Source Telemetry Events

After HCP is configured to parse, index, and persist telemetry events and NiFi is pushing data to HCP, you can view streaming telemetry data in the Metron Dashboard. See HCP User Guide for information about configuring the Metron Dashboard.

3.11. Setting up pcap to View Your Raw Data

The pcap data source creates a Storm topology that can rapidly ingest raw data directly into HDFS from Kafka. As a result, you can store all of your cybersecurity data in its raw form in HDFS and review or query it at a later date. HCP supports two pcap components:

- The pycapa tool aimed at low-volume packet capture

  Pycapa is a open-source Python-based probe created by Cisco.

- The Fastcapa tool aimed at high-volume packet capture.

  Fastcapa is a probe that performs fast network packet capture by leveraging Linux kernel-bypass and user space networking technology. The probe will bind to a network interface, capture network packets, and send the raw packet data to Kafka. This provides a scalable mechanism for ingesting high-volumes of network packet data into a Hadoop cluster.

  Fastcapa leverages the Data Plane Development Kit (DPDK). DPDK is a set of libraries and drivers to perform fast packet processing in Linux user space.

The rest of this chapter provides or points to instructions for setting up pycapa and Fastcapa and using pcap and Fastcapa:

- Setting up pycapa [74]
- Starting pcap [74]

- Setting up Fastcapa [76]
- Using Fastcapa [79]

3.11.1. Setting up pycapa

You can set up pycapa by completing the following steps. This installation assumes the following environment variables:

```
PYCAPA_HOME=/opt/pycapa
PYTHON27_HOME =/opt/rh/python27/root
```

1. Install the following packages:

   ```
    epel-release
   centos-release-scl
   "@Development tools"
   python27
   python27-scldevel
   python27-python-virtualenv
   libpcap-devel
   libselinux-python
   ```

   For example:

   ```
   yum -y install epel-release centos-release-scl
   yum -y install "@Development tools" python27 python27-scldevel python27-
   python-virtualenv libpcap-devel libselinux-python
   ```

2. Set up the following directory:

   ```
   mkdir $PYCAPA_HOME && chmod 755 $PYCAPA_HOME
   ```

3. Create the following virtual environment:

   ```
   export LD_LIBRARY_PATH="/opt/rh/python27/root/usr/lib64"
   ${PYTHON27_HOME}/usr/bin/virtualenv pycapa-venv
   ```

4. Copy `incubator-metron/metron-sensors/pycapa` from the Metron source tree into $PYCAPA_HOME on the node on which you would like to install pycapa.

5. Build pycapa:

   ```
   cd ${PYCAPA_HOME}/pycapa
   activate the virtualenv
   source ${PYCAPA_HOME}/pycapa-venv/bin/activate
   pip install -r requirements.txt
   python setup.py install
   ```

6. Start the pycapa packet capture producer:

   ```
   cd ${PYCAPA_HOME}/pycapa-venv/bin
   pycapa --producer --topic pcap -i $ETH_INTERFACE -k $KAFKA_HOST:6667
   ```

3.11.2. Starting pcap

To start pcap, HCP provides a utility script. This script takes no arguments and is very simple to run. Complete the following steps to start pcap:

1. Log into the host on which you are running Metron.

2. If you are running HCP on an Ambari-managed cluster, perform the following steps. If you are running a VM or a cluster that is not managed by Ambari, skip to Step 3.

    a. Update the `$METRON_HOME/config/pcap.properties` by changing `kafka.zk` to the appropriate server.

    You can retrieve the appropriate server information from Ambari in **Kafka service** > **Configs** > **Kafka Broker** > **zookeeper.connect**.

    b. On the HDFS host, create `/apps/metron/pcap`, change its ownership to metron:hadoop, and change its permissions to 775.

    ```
    hdfs dfs -mkdir /apps/metron/pcap
    hdfs dfs -chown metron:hadoop /apps/metron/pcap
    hdfs dfs -chmod 755 /apps/metron/pcap
    ```

    c. Create a Metron user's home directory on HDFS and change its ownership to the Metron user.

    ```
    hdfs dfs -mkdir /user/metron
    hdfs dfs -chown metron:hadoop /user/metron
    hdfs dfs -chmod 755 /user/metron
    ```

    d. Create a pcap topic in Kafka.

        i. Switch to metron user:

        ```
        su - metron
        ```

        ii. Create a Kafka topic named pcap:

        ```
        /usr/hdp/current/kafka-broker/bin/kafka-topics.sh \
        --zookeeper $ZOOKEEPER_HOST:2181 \
        --create \
        --topic pcap \
        --partitions 1 \
        --replication-factor 1
        ```

        iii. List all of the Kafka topics, to ensure that the new pcap topic exists:

        ```
        /usr/hdp/current/kafka-broker/bin/kafka-topics.sh --zookeeper
         $ZOOKEEPER_HOST:2181 --list
        ```

3. Start the pcap topology:

    ```
    $METRON_HOME/bin/start_pcap_topology.sh
    ```

    If HCP is installed on an Ambari-managed cluster, proceed the previous command with `su - metron`.

4. Check the Storm topology to ensure that packets are being captured.

    After Storm has captured a sufficient number of packets, you can check to ensure it is creating files on HDFS:

    ```
    hadoop fs -ls /apps/metron/pcap
    ```

3.11.3. Setting up Fastcapa

You can install Fastcapa one of two ways: automated or manual. The automated installation is the simplest method, but it requires that you are running CentOS 7.1. If you are not running CentOS 7.1 or would like more visibility into the installation process, you can manually install Fastcapa.

- Automated Installation [76]
- Manual Installation [76]

3.11.3.1. Prerequisites

The following system requirements must be met to run the Fastcapa probe:

- Linux kernel >= 2.6.34
- A DPDK supported ethernet device; NIC
- Port(s) on the ethernet device that can be dedicated for exclusive use by Fastcapa

3.11.3.2. Automated Installation

The process of installing Fastcapa has several steps and involves building DPDK, loading specific kernel modules, enabling huge page memory, and binding compatible network interface cards.

The best documentation is code that actually does this for you. An Ansible role that performs the entire installation procedure can be found at https://github.com/apache/metron/blob/master/metron-deployment/development/fastcapa. Use this to install Fastcapa or as a guide for manual installation. The automated installation assumes CentOS 7.1 and is directly tested against bento/centos-7.1.

3.11.3.3. Manual Installation

The following manual installation steps assume that they are executed on CentOS 7.1. Some minor differences might result if you use a different Linux distribution.

- Enable Transparent Huge Pages [76]
- Install DPDK [77]
- Install Librdkafka [78]
- Install Fastcapa [79]
- Using Fastcapa in a Kerberized Environment [82]

3.11.3.4. Enable Transparent Huge Pages

The probe performs its own memory management by leveraging transparent huge pages. In Linux, Transparent Huge Pages (THP) can be enabled either dynamically or on boot.

It is recommended that these be allocated on boot to increase the chance that a larger, physically contiguous chunk of memory can be allocated.

The size of THPs that are supported will vary based on your CPU. These typically include 2 MB and 1 GB THPs. For better performance, allocate 1 GB THPs if supported by your CPU.

1. Ensure that your CPU supports 1 GB THPs. A CPU flag `pdpe1gb` indicates whether or not the CPU supports 1 GB THPs.

   ```
   grep --color=always pdpe1gb /proc/cpuinfo | uniq
   ```

2. Add the following boot parameters to the Linux kernel. Edit `/etc/default/grub` and add the additional kernel parameters to the line starting with `GRUB_CMDLINE_LINUX`.

   ```
   GRUB_CMDLINE_LINUX=... default_hugepagesz=1G hugepagesz=1G hugepages=16
   ```

3. Rebuild the grub configuration then reboot. The location of the Grub configuration file will differ across Linux distributions.

   ```
   cp /etc/grub2-efi.cfg /etc/grub2-efi.cfg.orig
   /sbin/grub2-mkconfig -o /etc/grub2-efi.cfg
   ```

4. Once the host has been rebooted, ensure that the THPs were successfully allocated.

   ```
   $ grep HugePage /proc/meminfo
   AnonHugePages:    933888 kB
   HugePages_Total:      16
   HugePages_Free:        0
   HugePages_Rsvd:        0
   HugePages_Surp:        0
   ```

   The total number of huge pages that you have been allocated should be distributed fairly evenly across each NUMA node. In the following example, a total of 16 THPs were requested and 8 have been assigned on each of the 2 NUMA nodes.

   ```
   $ cat /sys/devices/system/node/node*/hugepages/hugepages-1048576kB/
   nr_hugepages
   8
   8
   ```

5. Once the THPs have been reserved, they need to be mounted to make them available to the probe.

   ```
   cp /etc/fstab /etc/fstab.orig
   mkdir -p /mnt/huge_1GB
   echo "nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0" >> /etc/fstab
   mount -fav
   ```

3.11.3.5. Install DPDK

1. Install the required dependencies.

   ```
   yum -y install "@Development tools"
   yum -y install pciutils net-tools glib2 glib2-devel git
   yum -y install kernel kernel-devel kernel-headers
   ```

2. Decide where DPDK will be installed.

```
export DPDK_HOME=/usr/local/dpdk/
```

3. Download, build, and install DPDK.

```
wget http://fast.dpdk.org/rel/dpdk-16.11.1.tar.xz -O - | tar -xJ
cd dpdk-stable-16.11.1/
make config install T=x86_64-native-linuxapp-gcc DESTDIR=$DPDK_HOME
```

4. Find the PCI address of the ethernet device that you plan on using to capture network packets. In the following example we plan on binding `enp9s0f0` which has a PCI address of `09:00.0`.

```
$ lspci | grep "VIC Ethernet"
09:00.0 Ethernet controller: Cisco Systems Inc VIC Ethernet NIC (rev a2)
0a:00.0 Ethernet controller: Cisco Systems Inc VIC Ethernet NIC (rev a2)
```

5. Bind the device. Replace the device name and PCI address with what is appropriate for your environment.

```
ifdown enp9s0f0
modprobe uio_pci_generic
$DPDK_HOME/sbin/dpdk-devbind --bind=uio_pci_generic "09:00.0"
```

6. Ensure that the device was bound. It should be shown as a 'network device using DPDK-compatible driver.'

```
$ dpdk-devbind --status
Network devices using DPDK-compatible driver
============================================
0000:09:00.0 'VIC Ethernet NIC' drv=uio_pci_generic unused=enic
Network devices using kernel driver
===================================
0000:01:00.0 'I350 Gigabit Network Connection' if=eno1 drv=igb unused=
uio_pci_generic
```

3.11.3.6. Install Librdkafka

The probe has been tested with Librdkafka 0.9.4.

1. Choose an installation path. In the following example, the libs will actually be installed at `/usr/local/lib`; note that `lib` is appended to the prefix.

```
export RDK_PREFIX=/usr/local
```

2. Download, build, and install.

```
wget https://github.com/edenhill/librdkafka/archive/v0.9.4.tar.gz  -O - |
 tar -xz
cd librdkafka-0.9.4/
./configure --prefix=$RDK_PREFIX
make
make install
```

3. Ensure that the installation location is on the search path for the runtime shared library loader.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RDK_PREFIX/lib
```

3.11.3.7. Install Fastcapa

1. Set the required environment variables.

```
export RTE_SDK=$DPDK_HOME/share/dpdk/
export RTE_TARGET=x86_64-native-linuxapp-gcc
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RDK_HOME
```

2. Build Fastcapa. The resulting binary will be placed at `build/app/fastcapa`.

```
cd metron/metron-sensors/fastcapa
make
```

3.11.4. Using Fastcapa

Follow these steps to run Fastcapa.

1. Create a configuration file that at a minimum specifies your Kafka broker.

   An example configuration file, `conf/fastcapa.conf`, is available that documents other useful parameters.

```
[kafka-global]
metadata.broker.list = kafka-broker1:9092
```

2. Bind the capture device.

   This is only needed if the device is not already bound. In this example, the device `enp9s0f0`with a PCI address of `09:00:0` is bound. Use values specific to your environment.

```
ifdown enp9s0f0
modprobe uio_pci_generic
$DPDK_HOME/sbin/dpdk-devbind --bind=uio_pci_generic "09:00.0"
```

3. Run Fastcapa.

```
fastcapa -c 0x03 --huge-dir /mnt/huge_1GB -- -p 0x01 -t pcap -c /etc/
fastcapa.conf
```

4. Terminate Fastcapa with `SIGINT` or by entering `CTRL-C`.

   The probe will cleanly shut down all of the workers and allow the backlog of packets to drain.

   To terminate the process without clearing the queue, send a `SIGKILL` or be entering `killall -9 fastcapa`.

3.11.4.1. Parameters

Fastcapa accepts three sets of parameters.

- Command-line parameters passed directly to DPDK's Environmental Abstraction Layer (EAL)
- Command-line parameters that define how Fastcapa will interact with DPDK. These parameters are separated on the command line by a double-dash (`--`).

- A configuration file that defines how Fastcapa interacts with Librdkafka.

3.11.4.1.1. Environmental Abstraction Layer Parameters

The most commonly used EAL parameter involves specifying which logical CPU cores should be used for processing. This can be specified in any of the following ways.

```
  -c COREMASK         Hexadecimal bitmask of cores to run on
  -l CORELIST         List of cores to run on
                      The argument format is <c1>[-c2][,c3[-c4],...]
                      where c1, c2, etc are core indexes between 0 and 128
  --lcores COREMAP    Map lcore set to physical cpu set
                      The argument format is
                            '<lcores[@cpus]>[<,lcores[@cpus]>...]'
                      lcores and cpus list are grouped by '(' and ')'
                      Within the group, '-' is used for range separator,
                      ',' is used for single number separator.
                      '( )' can be omitted for single element group,
                      '@' can be omitted if cpus and lcores have the same
 value
```

To get more information about other EAL parameters, run the following.

```
fastcapa -h
```

3.11.4.1.1.1. Fastcapa-Core Parameters

Name	Command	Description	Default
Port Mask	-p PORT_MASK	A bit mask identifying which ports to bind.	0x01
Burst Size	-b BURST_SIZE	Maximum number of packets to receive at one time.	32
Receive Descriptors	-r NB_RX_DESC	The number of descriptors for each receive queue (the size of the receive queue.) Limited by the ethernet device in use.	1024
Transmission Ring Size	-x TX_RING_SIZE	The size of each transmission ring. This must be a power of 2.	2048
Number Receive Queues	-q NB_RX_QUEUE	Number of receive queues to use for each port. Limited by the ethernet device in use.	2
Kafka Topic	-t KAFKA_TOPIC	The name of the Kafka topic.	pcap
Configuration File	-c KAFKA_CONF	Path to a file containing configuration values.	
Stats	-s KAFKA_STATS	Appends performance metrics in the form of JSON strings to the specified file.	

To get more information about the Fastcapa specific parameters, run the following. Note that this puts the `-h` after the double-dash `--`.

```
fastcapa -- -h
```

3.11.4.1.1.2. Fastcapa-Kafka Configuration File

The path to the configuration file is specified with the `-c` command line argument. The file can contain any global or topic-specific, producer-focused configuration values accepted by Librdkafka.

The configuration file is a `.ini`-like Glib configuration file. The global configuration values should be placed under a `[kafka-global]` header and topic-specific values should be placed under `[kafka-topic]`.

A minimally viable configuration file would only need to include the Kafka broker to connect to.

```
[kafka-global]
metadata.broker.list = kafka-broker1:9092, kafka-broker2:9092
```

The configuration parameters that are important for either basic functioning or performance tuning of Fastcapa include the following.

Global configuration values that should be located under the `[kafka-global]` header.

Name	***Description***	***Default***
metadata.broker.list	Initial list of brokers as a CSV list of broker host or host:port	
client.id	Client identifier.	
queue.buffering.max.messages	Maximum number of messages allowed on the producer queue	100000
queue.buffering.max.ms	Maximum time, in milliseconds, for buffering data on the producer queue	1000
message.copy.max.bytes	Maximum size for message to be copied to buffer. Messages larger than this will be passed by reference (zero-copy) at the expense of larger iovecs.	65535
batch.num.messages	Maximum number of messages batched in one MessageSet	10000
statistics.interval.ms	How often statistics are emitted; 0 = never	0
compression.codec	Compression codec to use for compressing message sets; {none, gzip, snappy, lz4 }	none

Topic configuration values that should be located under the `[kafka-topic]` header.

Name	***Description***	***Default***
compression.codec	Compression codec to use for compressing message sets; {none, gzip, snappy, lz4 }	none
request.required.acks	How many acknowledgements the leader broker must receive from ISR brokers before responding to the request; { 0 = no ack, 1 = leader ack, -1 = all ISRs }	1
message.timeout.ms	Local message timeout. This value is only enforced locally and limits the time a produced message waits	300000

Name	*Description*	*Default*
	for successful delivery. A time of 0 is infinite.	
queue.buffering.max.kbytes	Maximum total message size sum allowed on the producer queue	

3.11.4.1.2. Output

When running the probe some basic counters are output to stdout. Of course during normal operation these values will be much larger.

```
     ------ in ------  --- queued --- ----- out ----- ---- drops ----
[nic]               8               -               -               -
[rx]                8               0                8                0
[tx]                8               0                8                0
[kaf]               8               1                7                0
```

- `[nic] + in` : The ethernet device is reporting that it has seen 8 packets.
- `[rx] + in` : The receive workers have consumed 8 packets from the device.
- `[rx] + out` : The receive workers have enqueued 8 packets onto the transmission rings.
- `[rx] + drops` : If the transmission rings become full it will prevent the receive workers from enqueuing additional packets. The excess packets are dropped. This value will never decrease.
- `[tx] + in` : The transmission workers have consumed 8 packets.
- `[tx] + out` : The transmission workers have packaged 8 packets into Kafka messages.
- `[tx] + drops` : If the Kafka client library accepted fewer packets than expected. This value can increase or decrease over time as additional packets are acknowledged by the Kafka client library at a later point in time.
- `[kaf] + in` : The Kafka client library has received 8 packets.
- `[kaf] + out` : A total of 7 packets has successfully reached Kafka.
- `[kaf] + queued` : There is 1 packet within the `rdkafka` queue waiting to be sent.

3.11.5. Using Fastcapa in a Kerberized Environment

The Fastcapa probe can be used in a Kerberized environment. Follow these additional steps to use Fastcapa with Kerberos. The following assumptions have been made. These might need to be altered to fit your environment.

- The Kafka broker is at `kafka1:6667`
- ZooKeeper is at `zookeeper1:2181`
- The Kafka security protocol is SASL_PLAINTEXT
- The keytab used is located at `/etc/security/keytabs/metron.headless.keytab`

- The service principal is `metron@EXAMPLE.COM`

1. Build Librdkafka with SASL support (`--enable-sasl`).

   ```
   wget https://github.com/edenhill/librdkafka/archive/v0.9.4.tar.gz  -O - |
    tar -xz
   cd librdkafka-0.9.4/
   ./configure --prefix=$RDK_PREFIX --enable-sasl
   make
   make install
   ```

2. Validate Librdkafka supports SASL. Run the following command and ensure that `sasl` is returned as a built-in feature.

   ```
   $ examples/rdkafka_example -X builtin.features
   builtin.features = gzip,snappy,ssl,sasl,regex
   ```

   If it is not, ensure that you have `libsasl` or `libsasl2` installed. On CentOS, this can be installed with the following command.

   ```
   yum install -y cyrus-sasl cyrus-sasl-devel cyrus-sasl-gssapi
   ```

3. Grant access to your Kafka topic. In this example, the Kafka topic is simply named `pcap`.

   ```
   $KAFKA_HOME/bin/kafka-acls.sh --authorizer kafka.security.auth.
   SimpleAclAuthorizer \
     --authorizer-properties zookeeper.connect=zookeeper1:2181 \
     --add --allow-principal User:metron --topic pcap
   ```

4. Obtain a Kerberos ticket.

   ```
   kinit -kt /etc/security/keytabs/metron.headless.keytab metron@EXAMPLE.COM
   ```

5. Add the following additional configuration values to your Fastcapa configuration file.

   ```
   security.protocol = SASL_PLAINTEXT
   sasl.kerberos.keytab = /etc/security/keytabs/metron.headless.keytab
   sasl.kerberos.principal = metron@EXAMPLE.COM
   ```

6. Now run Fastcapa as you normally would. It should have no problem landing packets in your kerberized Kafka broker.

3.12. Troubleshooting Parsers

This section provides some troubleshooting solutions for parser issues.

3.12.1. Storm is Not Receiving Data From a New Data Source

1. Ensure that your Grok parser statement is valid.

   You can use GrokConstructor to test your parser statement.

   If you need to modify your Grok parser statement, you must kill the topology for your new data source in the Storm UI and then resubmit your data source.

   a. Log into HOST $HOST_WITH_ENRICHMENT_TAG as root.

   b. Deploy the new parser topology:

      ```
      $METRON_HOME/bin/start_parser_topology.sh -k $KAFKA_HOST:6667 -z
       $ZOOKEEPER_HOST:2181 -s $DATASOURCE
      ```

   c. Go to the Storm UI. You should now see the new topology. Ensure that the topology has no errors.

2. Ensure that the Kafka topic you created for your new data source is receiving data.

3. Check your NiFi configuration to ensure that data is flowing between the Kafka topic for your new data source into HCP.

3.12.2. Determining Which Events Are Not Being Processed

Events that are not processed end up in a dead letter queue. There are two types of events. One, where the event could not be parsed at all. Two, where the event was parsed, but failed validation

4. Monitor and Management

Hortonworks Cybersecurity Package (HCP) powered by Apache Metron provides a number of options for monitoring and managing your system. Before you perform any of these monitoring and management tasks, we suggest that you become familiar with HCP data throughput by referring to Understanding Throughput.

The rest of this chapter provides detailed instructions on performing the following monitoring and management tasks:

4.1. Understanding Throughput

The data flow for HCP is performed in real-time and contains the following steps:

1. Information from telemetry data sources is ingested into Kafka files through the telemetry event buffer. This information is the raw telemetry data consisting of host logs, firewall logs, emails, and network data. Depending on the type of data you are streaming into HCP, you can use one of the following telemetry data collectors to ingest the data:

   NiFi

   This type of streaming method works for most types of telemetry data sources. See the NiFi documentation for more information,

   Performant network ingestion probes

   This type of streaming method is ideal for streaming high volume packet data. See Viewing pcap Data for more information.

   Real-time and batch threat intelligence feed loaders

   This type of streaming method is used for real-time and batch threat intelligence feed loaders.

2. Once the information is ingested into Kafka files, the data is parsed into a normalized JSON structure that HCP can read. This information is parsed using a Java or general purpose parser and then it is uploaded to ZooKeeper. A Kafka file containing the parser information is created for every telemetry data source.

3. The information is then enriched with asset, geo, and threat intelligence information.

4. The information is then indexed, stored, and any resulting alerts are sent to the Metron dashboard.

4.2. Updating Properties

HCP configuration information is stored in ZooKeeper as a series of JSON files. There are multiple locations from which you can populate the ZooKeeper configurations:

- $METRON_HOME/config/zookeeper
- Management UI
- Ambari
- Stellar REPL

However, Ambari explicitly manages some of these configuration properties. If you change a property explicitly managed by Ambari from one of the other mechanisms outside of Ambari, such as the Management UI, Ambari will not be aware of this change and will overwrite it whenever the Metron topology is restarted. Therefore, you should modify Ambari-managed properties only in Ambari.

For example, the `es.ip` property is managed explicitly by Ambari. If you modify `es.ip` and change the `global.json` file outside Ambari, you will not see this change in Ambari. Meanwhile, the indexing topology would be using the new value stored in ZooKeeper. You will not receive any errors notifying you of the discrepancy between ZooKeeper and Ambari. However, when you restart the Metron topology component via Ambari, the `es.ip` property would be set back to the value stored in Ambari.

See the following table for a list of Ambari-managed properties. All other properties can be managed in one of the mechanisms outside Ambari.

Table 4.1. Properties Managed by Ambari

Global Configuration Property Name	Ambari Name
es.clustername	es_cluster_name
es.ip	es_hosts
es.port	es_port
es.date.format	es_date_format
profiler.period.duration	profiler_period_duration
profiler.period.duration.units	profiler_period_units
update.hbase.table	update_hbase_table
update.hbase.cf	update_hbase_cf
geo.hdfs.file	geo_hdfs_file

4.3. Updating ZooKeeper

ZooKeeper configurations should be stored on disk in the following structure starting at `$METRON_HOME/bin/zk_load_configs.sh`:

global.json	The global config
sensors	The subdirectory containing sensor enrichment configuration JSON (for example, snort.json, bro.json)

By default, this directory as deployed by the ansible infrastructure is at `$METRON_HOME/config/zookeeper`.

While the configs are stored on disk, they must be loaded into ZooKeeper to be used. To this end, there is a utility program to assist in this called `$METRON_HOME/bin/zk_load_config.sh`.

This has the following options:

`-f,--force`	Force operation
`-h,--help`	Generate Help screen
`-i,--input_dir <DIR>`	The input directory containing the configuration files named like "$source.json"
`-m,--mode <MODE>`	The mode of operation: DUMP, PULL, PUSH
`-o,--output_dir <DIR>`	The output directory which will store the JSON configuration from ZooKeeper
`-z,--zk_quorum <host:port,[host:port]*>`	ZooKeeper Quorum URL (zk1:port,zk2:port,...)

Usage examples:

- To dump the existing configs from ZooKeeper on the single node vagrant machine:

  ```
  $METRON_HOME/bin/zk_load_configs.sh -z node1:2181 -m DUMP
  ```

- To push the configs into ZooKeeper on the single node vagrant machine:

  ```
  $METRON_HOME/bin/zk_load_configs.sh -z node1:2181 -m PUSH -i
  $METRON_HOME/config/zookeeper
  ```

- To pull the configs from ZooKeeper to the single node vagrant machine disk:

  ```
  $METRON_HOME/bin/zk_load_configs.sh -z node1:2181 -m PULL -o
  $METRON_HOME/config/zookeeper -f
  ```

4.4. Managing Sensors

You can manage your sensors and associated topologies using either the HCP Management module or the Storm UI. The following procedures use the HCP Management module to manage sensors. For information on using Storm to manage sensors, see the Storm documentation.

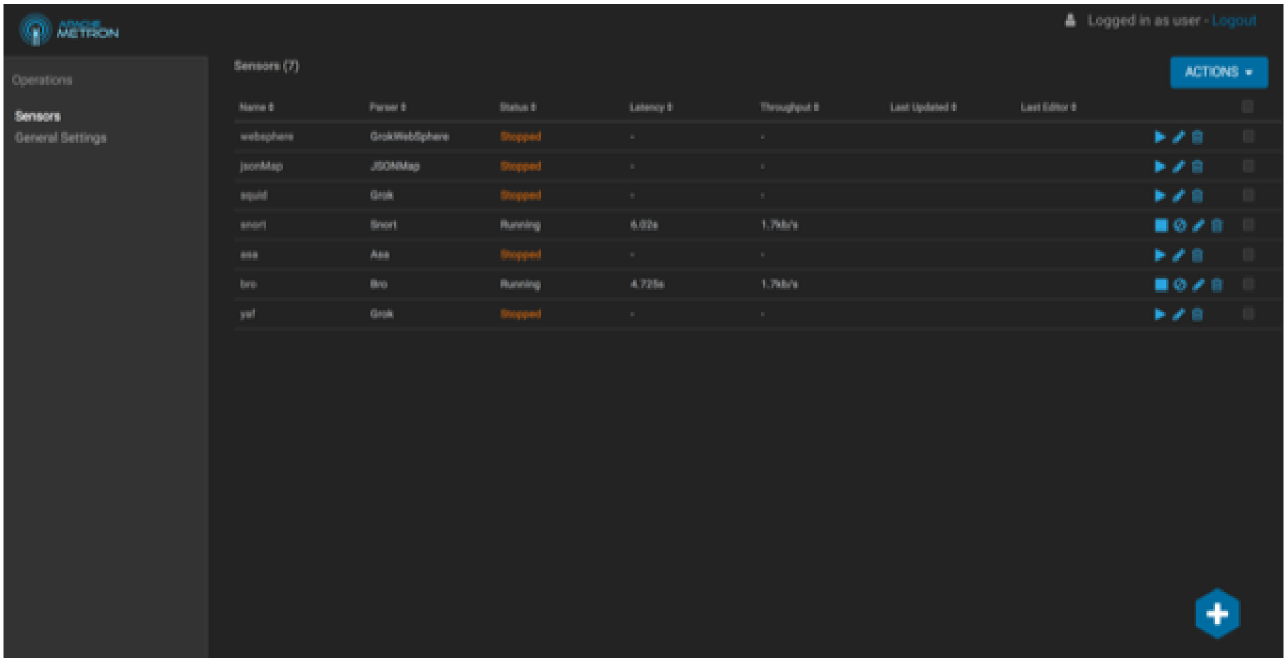
- Starting and Stopping a Sensor [88]
- Modifying a Sensor [88]
- Deleting a Sensor [90]

4.4.1. Starting and Stopping a Sensor

After you install a sensor, you can start or stop it using the Management Module.

To start or stop a sensor, select the sensor in the main window, then click ▶ (start button) to start the sensor or ■ (stop button) to stop the sensor in ■ ⊘ ✎ 🗑 (tool bar) on the right side of the window.

Figure 4.1. Management Module Main Window



Sensors (7)

Name ⇕	Parser ⇕	Status ⇕	Latency ⇕	Throughput ⇕	Last Updated ⇕	Last Editor ⇕		□
websphere	GrokWebSphere	Stopped	-	-			▶ ✎ 🗑	□
jsonMap	JSONMap	Stopped	-	-			▶ ✎ 🗑	□
squid	Grok	Stopped	-	-			▶ ✎ 🗑	□
snort	Snort	Running	6.02s	1.7kb/s			■ ⊘ ✎ 🗑	□
asa	Asa	Stopped	-	-			▶ ✎ 🗑	□
bro	Bro	Running	4.725s	1.7kb/s			■ ⊘ ✎ 🗑	□
yaf	Grok	Stopped	-	-			▶ ✎ 🗑	□

Starting or stopping the sensor might take a few minutes. When the operation completes successfully, you should see the Status for the sensor change to the Running or Stopped.

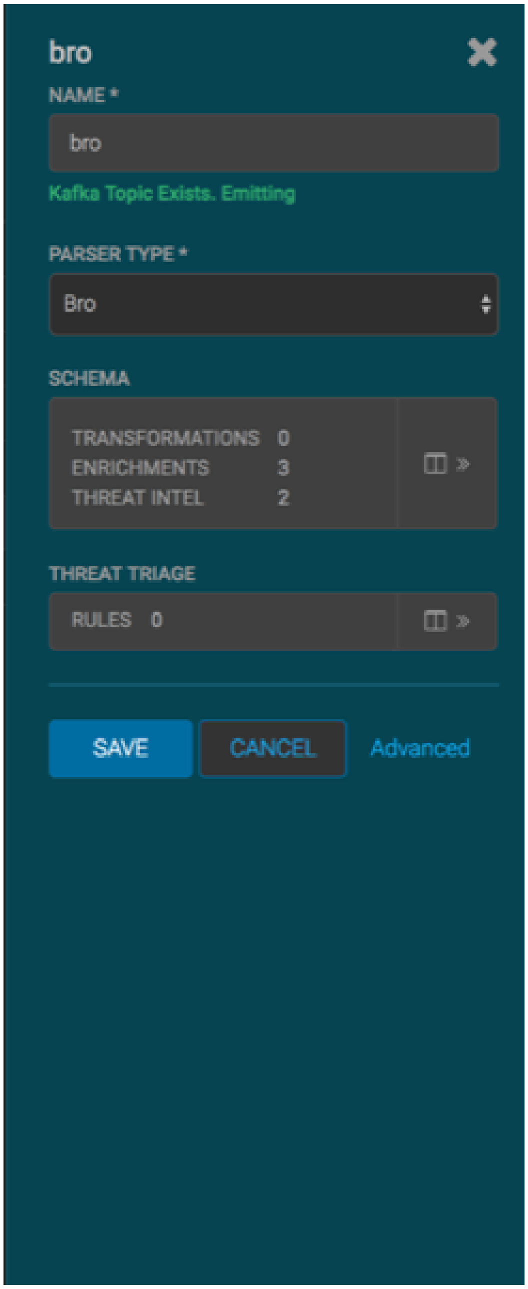
4.4.2. Modifying a Sensor

You can modify any sensor listed in HCP Management module.

1. Select **Sensors** in the **Operations** panel on the left side of the window, and then click ✎ (edit button) for the sensor you want to modify.

The Management module displays a panel populated with the sensor configuration information.

Figure 4.2. Sensor Panel



bro

NAME *

bro

Kafka Topic Exists. Emitting

PARSER TYPE *

Bro

SCHEMA

TRANSFORMATIONS 0
ENRICHMENTS 3
THREAT INTEL 2

THREAT TRIAGE

RULES 0

2. You can modify the following information for the sensor:

- Sensor name
- Parser type
- Schema information

- Threat triage information

3. Click **Save** to save your changes.

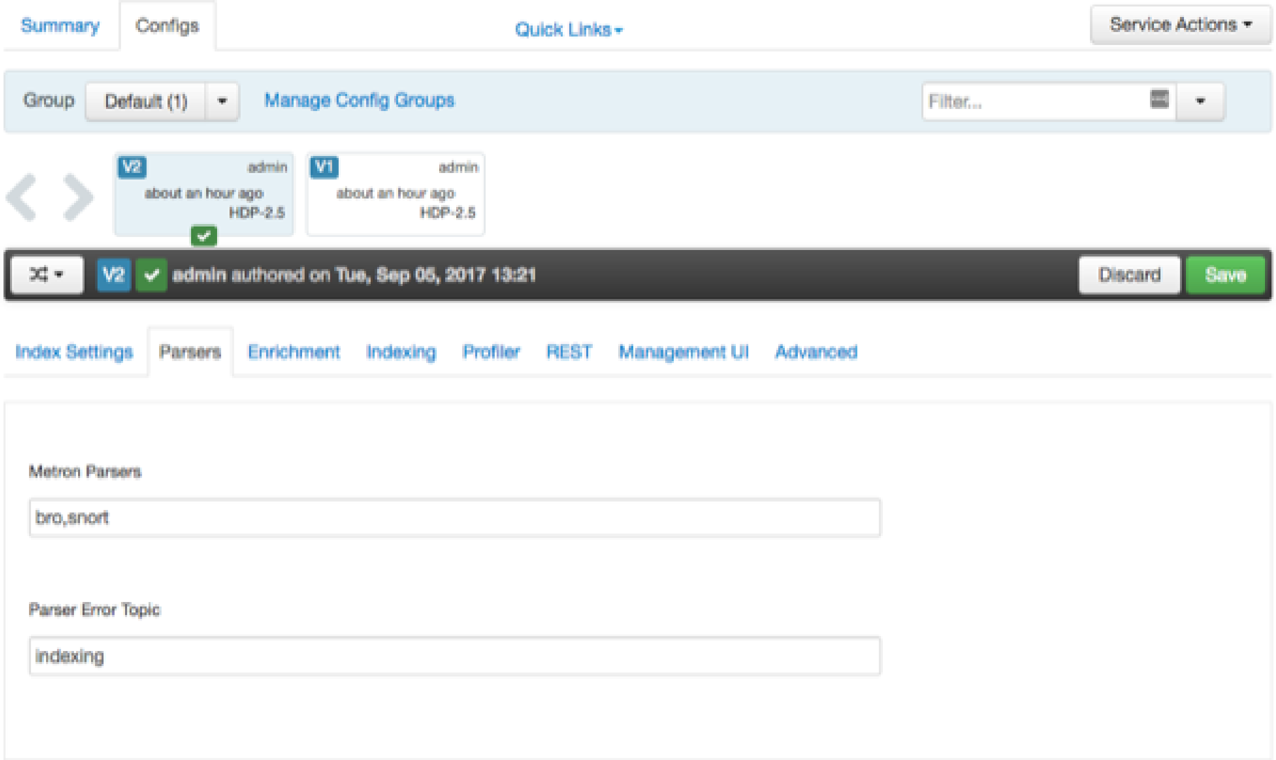
4.4.3. Deleting a Sensor

You can delete a sensor if you don't need it.

> **Important**
>
> You must take the sensor offline before deleting it.

1. In the Ambari user interface, click the **Services** tab, then **Metron** from the list of services.
2. Click the **Configs** tab and then click **Parsers**.

Figure 4.3. ambari_configs_parsers.png



Metron Parsers: bro,snort

Parser Error Topic: indexing

3. Remove the parser you want to delete from the **Metron Parsers** field.
4. Next, display the Management module.
5. Select the check box next to the appropriate sensor in the Sensors table.

   You can delete more than one sensor at a time by clicking multiple check boxes.
6. From the Actions menu, choose **Delete**.

The Management module deletes the sensor from ZooKeeper.

7. Finally, delete the json file for the sensor on the Ambari master node:

```
ssh $AMBARI_MASTER_NODE
cd $METRON_HOME/config/zookeeper/parser
rm $DATASOURCE.json
```

4.5. Monitoring Sensors

You can use the Metron Error Dashboard to monitor sensor error messages and troubleshoot them.

The Metron user interface provides two dashboards: the Metron Dashboard and the Metron Error Dashboard. The first dashboard, the Metron Dashboard, provides sensor-specific data that can be used to identify, investigate, and analyze cybersecurity data. This first dashboard is described extensively in the HCP User Guide. The second dashboard, the Metron Error Dashboard, receives information on all errors detected by HCP. This section describes the Error Dashboard in detail and provides instruction on how to use the dashboard to monitor sensor errors and troubleshoot problems and contains the following sections:

- Displaying the Metron Error Dashboard [91]
- Default Metron Error Dashboard [92]
- Loading Metron Templates [93]

4.5.1. Displaying the Metron Error Dashboard

Prior to displaying the Metron Error Dashboard, ensure that you have completed the following:

- Created an Index template

The Metron Dashboard user interface defaults to displaying the Metron Dashboard. To display the Metron Error Dashboard, complete the following steps:

1. Click 📂 (Load Saved Dashboard icon) in the upper right corner of the Metron Dashboard, then choose **Metron Error Dashboard** from the list of dashboards.

2. Click

(timeframe tab) in the upper right corner of the Metron Error Dashboard to choose the timeframe you need the error dashboard to cover

The Metron Error dashboard receives the following information for all error messages:

- Exception

- Hostname - which machine the error occurred on
- Stack trace
- Time - When the error occurred
- Message
- Raw Message - original message
- Raw_message_bytes - The bytes of the original message
- Hash - To determine if there is a duplicate message
- Source_type - Identifies source sensor
- Error type - Parser error, etc.

4.5.2. Default Metron Error Dashboard

The following list contains a description of each of the sections that display by default in the Metron Error dashboard.

Total Error Messages
The total number of error messages received during the interim you have specified.

Unique Error Messages
The total number of unique error messages received during the interim you have specified.

Errors Over Time
A **detailed message panel** displays the raw data from your search query.

Error Source
When you submit a search query, the 500 most recent documents that match the query are listed in the **Documents** table which is displayed in the center of the **Discover** window.

Errors by Error Type
A list of all of the fields associated with a selected index pattern. This list is displayed on the left side of the **Discover** window.

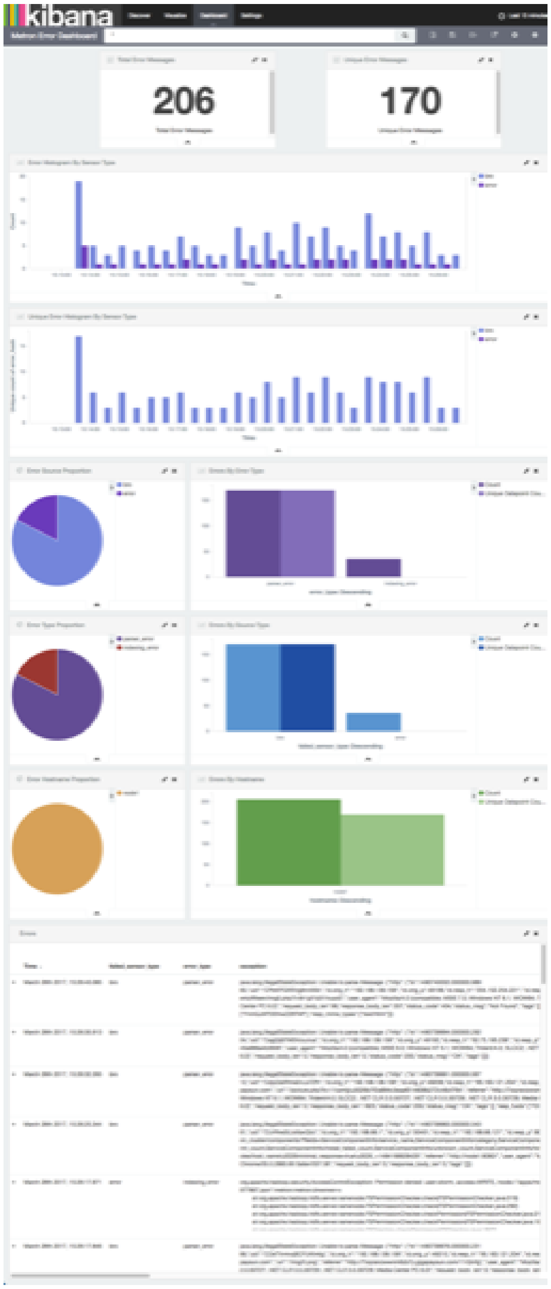
Error Type Proportion
Use the **line chart** when you want to display high density time series. This chart is useful for comparing one series with another.

Errors by Type
You can use the **mark down widget panel** to provide explanations or instructions for the dashboard.

List of Errors
You can use a **metric panel** to display a single large number such as the number of hits or the average of a numeric field.

The default Error dashboard should look similar to the following:

Figure 4.4. Error Dashboard



4.5.3. Loading Metron Templates

HCP provides templates for the Metron UI dashboards. You might want to load or reload these templates if the Metron UI is not displaying the default dashboard panes, or if you would like to return to the default format.

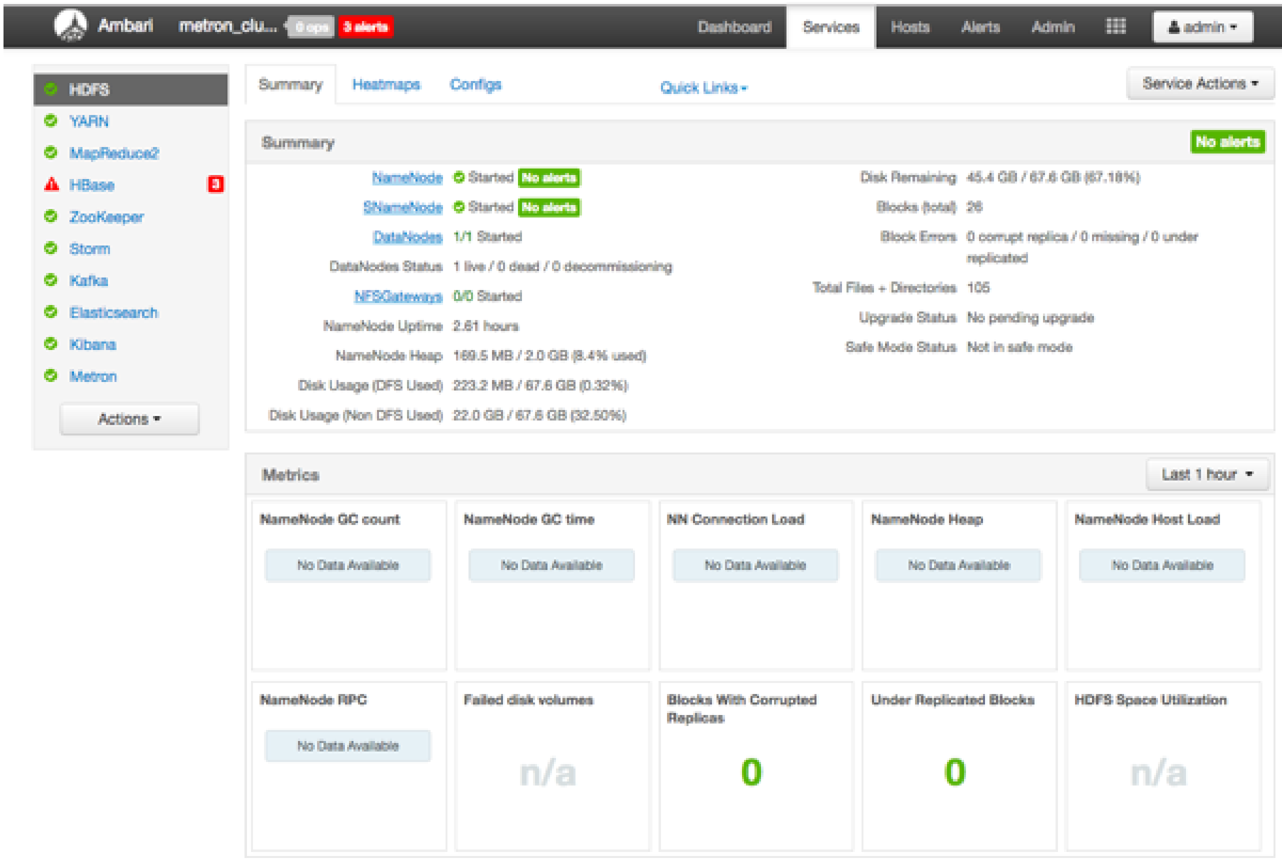
To load the Metron templates, complete the following steps:

1. Display the Ambari UI:

```
https://$METRON_HOME:8080
```

2. Click the **Services** tab and select Kibana in the left pane of the window.

Figure 4.5. Ambari Services Tab



Summary

NameNode ✓ Started No alerts
SNameNode ✓ Started No alerts
DataNodes 1/1 Started
DataNodes Status 1 live / 0 dead / 0 decommissioning
NFSGateways 0/0 Started
NameNode Uptime 2.61 hours
NameNode Heap 169.5 MB / 2.0 GB (8.4% used)
Disk Usage (DFS Used) 223.2 MB / 67.6 GB (0.32%)
Disk Usage (Non DFS Used) 22.0 GB / 67.6 GB (32.50%)
Disk Remaining 45.4 GB / 67.6 GB (67.18%)
Blocks (total) 26
Block Errors 0 corrupt replica / 0 missing / 0 under replicated
Total Files + Directories 105
Upgrade Status No pending upgrade
Safe Mode Status Not in safe mode

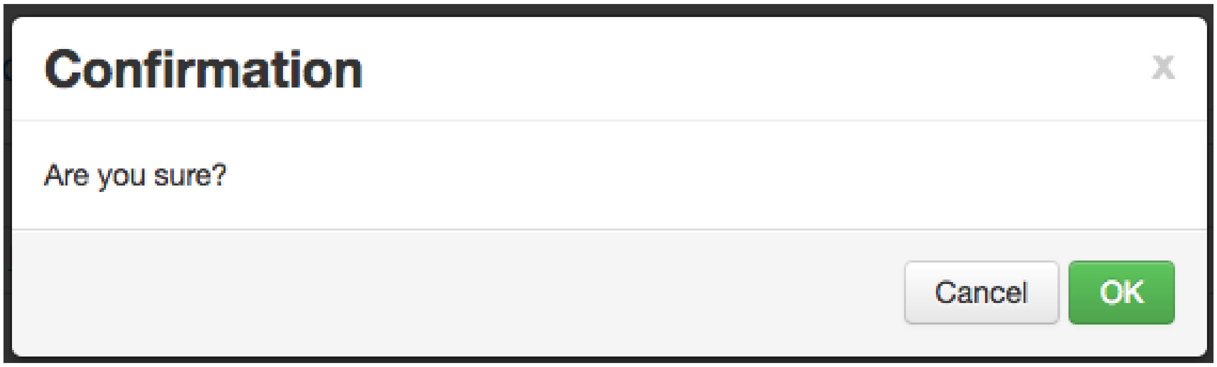
Metrics

NameNode GC count: No Data Available
NameNode GC time: No Data Available
NN Connection Load: No Data Available
NameNode Heap: No Data Available
NameNode Host Load: No Data Available
NameNode RPC: No Data Available
Failed disk volumes: n/a
Blocks With Corrupted Replicas: 0
Under Replicated Blocks: 0
HDFS Space Utilization: n/a

3. From the **Service Actions** pull down menu, select **Load Template**.

4. Click the **OK** button to confirm your selection.

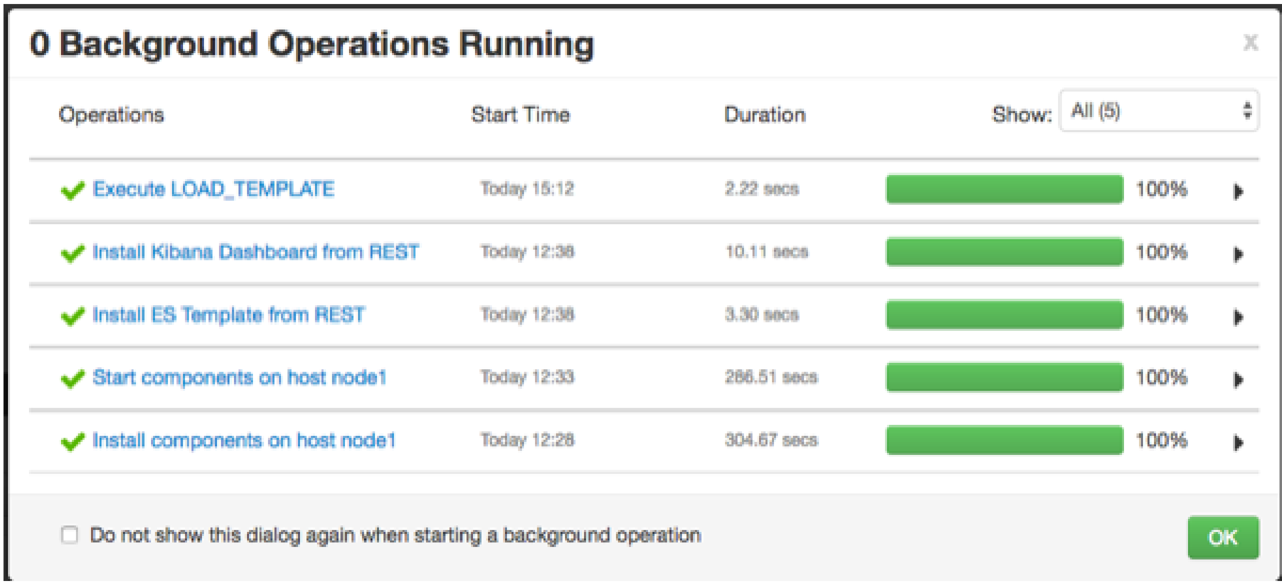
Figure 4.6. Confirmation Dialog Box


Confirmation

Are you sure?

Ambari displays a dialog box listing the background operations it is running.

Figure 4.7. Ambari Background Operations



0 Background Operations Running

Show: All (5)

Operations	Start Time	Duration		
✓ Execute LOAD_TEMPLATE	Today 15:12	2.22 secs	🟩	100%
✓ Install Kibana Dashboard from REST	Today 12:38	10.11 secs	🟩	100%
✓ Install ES Template from REST	Today 12:38	3.30 secs	🟩	100%
✓ Start components on host node1	Today 12:33	286.51 secs	🟩	100%
✓ Install components on host node1	Today 12:28	304.67 secs	🟩	100%

□ Do not show this dialog again when starting a background operation

5. Click the **OK** button to dismiss the dialog box.

   Ambari has completed loading the Metron template. You should be able to see the default formatting in the Metron dashboards.

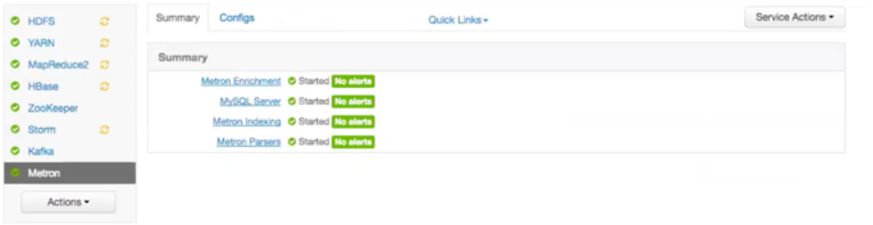
4.6. Starting and Stopping Parsers

You might want to stop or start parsers as you refine or focus your cybersecurity monitoring. You can easily stop and start parsers by using Ambari.

To start or stop a parser, complete the following steps:

1. Display the Ambari tool and navigate to **Services** > **Metron** > **Summary**.

Figure 4.8. Ambari Metron Summary Window



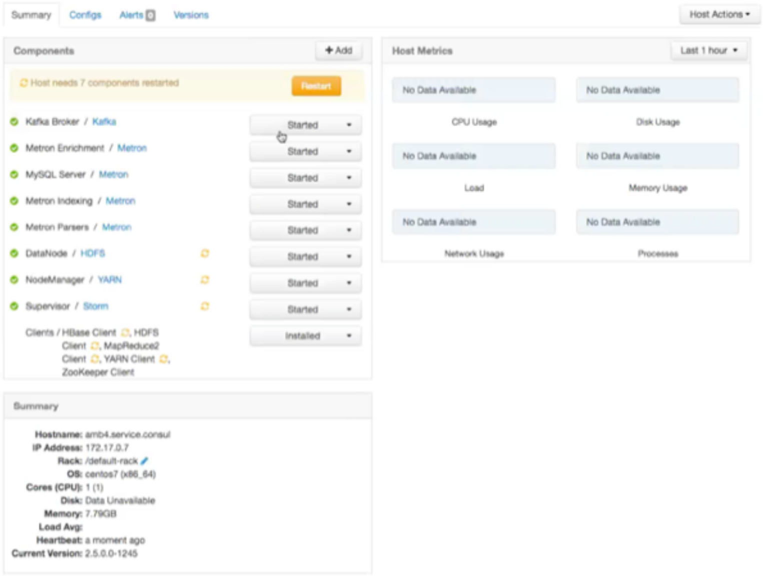
Summary

Metron Enrichment	Started	No alerts
MySQL Server	Started	No alerts
Metron Indexing	Started	No alerts
Metron Parsers	Started	No alerts

2. Under Summary, click on **Metron Parsers** to display the **Components** window.

   The Components window displays a list of Metron hosts and which components reside on each host.

Figure 4.9. Components Window



Kafka Broker / Kafka — Started ▾

Metron Enrichment / Metron — Started ▾

MySQL Server / Metron — Started ▾

Metron Indexing / Metron — Started ▾

Metron Parsers / Metron — Started ▾

DataNode / HDFS — Started ▾

NodeManager / YARN — Started ▾

Supervisor / Storm — Started ▾

Clients / HBase Client, HDFS Client, MapReduce2 Client, YARN Client, ZooKeeper Client — Installed ▾

Summary

Hostname: amb4.service.consul
IP Address: 172.17.0.7
Rack: /default-rack
OS: centos7 (x86_64)
Cores (CPU): 1 (1)
Disk: Data Unavailable
Memory: 7.79GB
Load Avg:
Heartbeat: a moment ago
Current Version: 2.5.0.0-1245

3. Click the **Started/Stopped** button by **Metron Parsers** to change the status of the Parsers then click the **Confirmation** button to verify that you want to start or stop the parsers.

   Ambari displays the **Background Operation Running** dialog box.

4. Click **Stop Metron Parsers**.

   Ambari displays the **Stop Metron Parsers** dialog box.

5. Click the entry for your Metron cluster, then click **Metron Parser Stop** again.

   Ambari displays a dialog box for your Metron cluster which lists the actions as is stops the parsers.

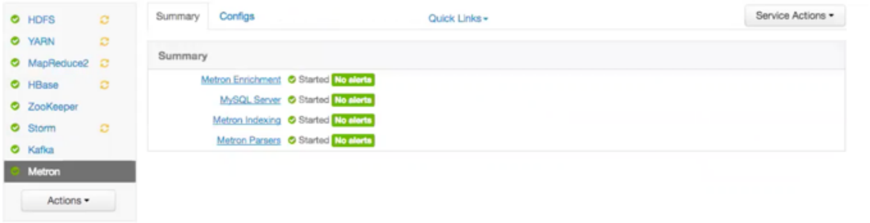
4.7. Starting and Stopping Enrichments

You might want to stop or start enrichments as you refine or focus your cybersecurity monitoring. You can easily stop and start enrichments by using Ambari.

To start or stop the enrichments, complete the following steps:

1. Display the Ambari tool and navigate to **Services** > **Metron** > **Summary**.

Figure 4.10. Ambari Metron Summary Window



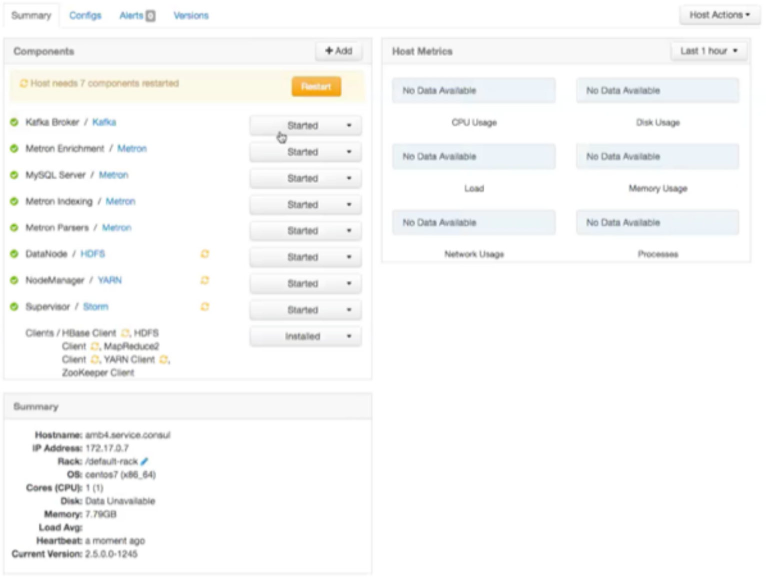
Summary

Metron Enrichment	Started	No alerts
MySQL Server	Started	No alerts
Metron Indexing	Started	No alerts
Metron Parsers	Started	No alerts

2. Under Summary, click on **Metron Enrichments** to display the **Components** window.

   This window displays a list of HCP hosts and which components reside on each host.

Figure 4.11. Components Window



Host needs 7 components restarted

Kafka Broker / Kafka	Started
Metron Enrichment / Metron	Started
MySQL Server / Metron	Started
Metron Indexing / Metron	Started
Metron Parsers / Metron	Started
DataNode / HDFS	Started
NodeManager / YARN	Started
Supervisor / Storm	Started
Clients / HBase Client, HDFS Client, MapReduce2 Client, YARN Client, ZooKeeper Client	Installed

Summary

Hostname: amb4.service.consul
IP Address: 172.17.0.7
Rack: /default-rack
OS: centos7 (x86_64)
Cores (CPU): 1 (1)
Disk: Data Unavailable
Memory: 7.79GB
Load Avg:
Heartbeat: a moment ago
Current Version: 2.5.0.0-1245

3. Click the **Started/Stopped** button by **Metron Enrichments** to change the status of the Enrichments then click the **Confirmation** button to verify that you want to start or stop the enrichments.

   Ambari displays the **Background Operation Running** dialog box.

4. Click **Stop Metron Enrichments**.

   Ambari displays the **Stop Metron Enrichments** dialog box.

5. Click the entry for your Metron cluster, then click **Metron Enrichments Stop** again.

   Ambari displays a dialog box for your Metron cluster which lists the actions as is stops the enrichments.

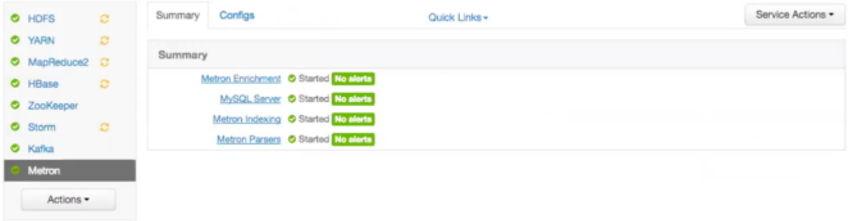
4.8. Starting and Stopping Indexing

You might want to stop or start indexing as you refine or focus your cybersecurity monitoring. You can easily stop and start indexing by using Ambari.

To start or stop indexing, complete the following steps:

1. Display the Ambari tool and navigate to **Services** > **Metron** > **Summary**.

Figure 4.12. Ambari Metron Summary Window



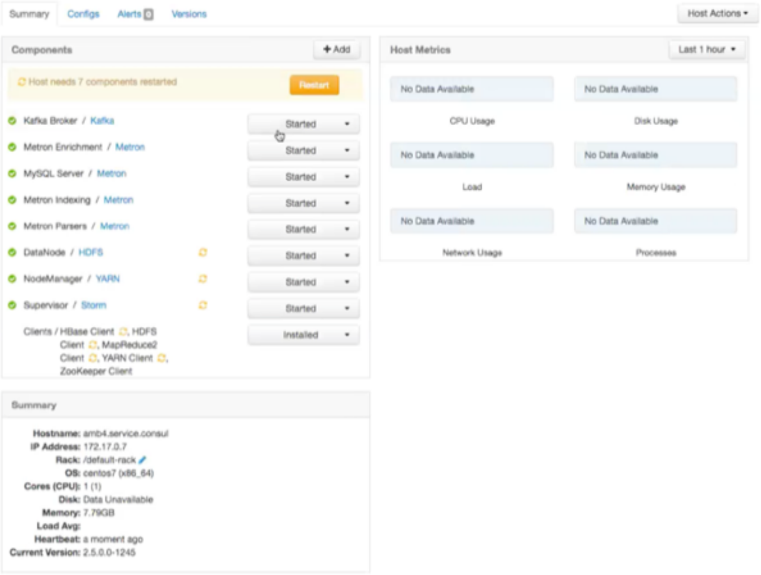
Summary

Metron Enrichment ✓ Started No alerts
MySQL Server ✓ Started No alerts
Metron Indexing ✓ Started No alerts
Metron Parsers ✓ Started No alerts

2. Under Summary, click on **Metron Indexing** to display the **Components** window.

   This window displays a list of HCP hosts and which components reside on each host.

Figure 4.13. Components Window



Components

Host needs 7 components restarted

✓ Kafka Broker / Kafka Started ▾
✓ Metron Enrichment / Metron Started ▾
✓ MySQL Server / Metron Started ▾
✓ Metron Indexing / Metron Started ▾
✓ Metron Parsers / Metron Started ▾
✓ DataNode / HDFS Started ▾
✓ NodeManager / YARN Started ▾
✓ Supervisor / Storm Started ▾
Clients / HBase Client, HDFS Client, MapReduce2 Client, YARN Client, ZooKeeper Client Installed ▾

Summary

Hostname: amb4.service.consul
IP Address: 172.17.0.7
Rack: /default-rack
OS: centos7 (x86_64)
Cores (CPU): 1 (1)
Disk: Data Unavailable
Memory: 7.79GB
Load Avg:
Heartbeat: a moment ago
Current Version: 2.5.0.0-1245

3. Click the **Started/Stopped** button by **Metron Indexing** to change the status of the Indexing then click the **Confirmation** button to verify that you want to start or stop the indexing.

   Ambari displays the **Background Operation Running** dialog box.

4. Click **Stop Metron Indexing**.

Ambari displays the **Stop Metron Indexing** dialog box.

5. Click the entry for your Metron cluster, then click **Metron Indexing Stop** again.

   Ambari displays a dialog box for your Metron cluster which lists the actions as it stops the indexing.

4.9. Modifying the Elasticsearch Template

You can modify the Elasticsearch template to change the settings in your HCP environment. Some of these settings will help optimize your system performance.

- indexing.workers
- indexing.executors
- bolt.hdfs.batch.size

1. Display the Ambari tool and navigate to **Services** > **Metron** > **Summary** > **Advanced**.
2. Click **Advanced metron-env**.

   Ambari displays the contents of the `metron-env` file which includes the elasticsearch.properties template.
3. Modify the appropriate properties, then click **Save** at the top of the window.

4.10. Pruning Data From Elasticsearch

Elasticsearch provides tooling to prune index data through its Curator utility. For more information about the Curator utility, see Curator Reference.

The following is a sample invocation that you can configure through Cron to prune indexes based on timestamp in the index name.

```
/opt/elasticsearch-curator/curator_cli --host localhost delete_indices --
filter_list '
     {
       "filtertype": "age",
       "source": "name",
       "timestring": "%Y.%m.%d",
       "unit": "days",
       "unit_count": 10,
       "direction": "older”
     }'
```

Using `name` as the `source` tells Curator to look for a `timestring` within the index or snapshot name, and convert that into an epoch timestamp (epoch implies UTC).

For finer-grained control over the indexes that will be pruned, you can also provide multiple filters as an array of JSON objects to `filter_list`. There is an implicit logical AND when chaining multiple filters.

```
--filter_list '[{"filtertype":"age","source":"creation_date",
"direction":"older","unit":"days","unit_count":13},
{"filtertype":"pattern","kind":"prefix","value":"logstash"}]'
```

4.11. Backing up the Metron Dashboard

You can back up your Metron dashboard to avoid losing your customizations by using the following command:

```
python packaging/ambari/metron-mpack/src/main/resources/common-services/
KIBANA/5.6.2/package/scripts/dashboard/dashboardindex.py \
  $ES_HOST 9200 \
 $SOME_PATH/dashboard.p -s
```

4.12. Restoring Your Metron Dashboard Backup

To restore a back up of your Metron dashboard, use the following command:

```
python packaging/ambari/metron-mpack/src/main/resources/common-services/
KIBANA/5.6.2/package/scripts/dashboard/dashboardindex.py \
  $ES_HOST 9200 \
 $SOME_PATH/dashboard.p
```

Note

This method of writing the Kibana dashboard to Elasticsearch will overwrite the entire .kibana index.

5. Concepts

This chapter provides more in-depth information about the terminology used in the rest of this guide. This chapter contains detailed information on the following:

5.1. Parsers

Parsers are pluggable components that transform raw data (textual or raw bytes) into JSON messages suitable for downstream enrichment and indexing. Data flows through the parser bolt via Kafka and into the enrichments topology in Storm. Errors are collected with the context of the error (for example stacktrace) and the original message causing the error and sent to an error queue. Invalid messages as determined by global validation functions are also treated as errors and sent to an error queue.

HCP supports two types of parsers: Java and general purpose. Each of these parsers plus the parser configuration are described in the following sections.

5.1.1. Java Parsers

The Java parser is written in Java and conforms with the MessageParser interface. This kind of parser is optimized for speed and performance and is built for use with higher-velocity topologies. Java parsers are not easily modifiable; to make changes to them, you must recompile the entire topology.

Currently, the Java adapters included with HCP are as follows:

- org.apache.metron.parsers.ise.BasicIseParser
- org.apache.metron.parsers.bro.BasicBroParser
- org.apache.metron.parsers.sourcefire.BasicSourcefireParser
- org.apache.metron.parsers.lancope.BasicLancopeParser

5.1.2. General Purpose Parsers

The general purpose parser is primarily designed for lower-velocity topologies or for quickly setting up a temporary parser for a new telemetry. General purpose parsers are defined using a config file, and you need not recompile the topology to change them. HCP supports two general purpose parsers: Grok and CSV.

Grok parser

The Grok parser class name (parserClassName) is `org.apache.metron,parsers.GrokParser`.

Grok has the following entries and predefined patterns for `parserConfig`:

`grokPath` The patch in HDFS (or in the Jar) to the Grok statement

`patternLabel` The pattern label to use from the Grok statement

`timestampField` The field to use for timestamp

`timeFields` A list of fields to be treated as time

`dateFormat` The date format to use to parse the time fields

`timezone` The timezone to use. UTC is the default.

CSV Parser

The CSV parser class name (parserClassName) is `org.apache.metron.parsers.csv.CSVParser`

CSV has the following entries and predefined patterns for `parserConfig`:

`timestampFormat` The date format of the timestamp to use. If unspecified, the parser assumes the timestamp is starts at UNIX epoch.

`columns` A map of column names you wish to extract from the CSV to their offsets. For example, `{ 'name' : 1,'profession' : 3}` would be a column map for extracting the 2nd and 4th columns from a CSV.

`separator` The column separator. The default value is ",".

JSON Map Parser

The JSON parser class name (parserClassName) is `org.apache.metron.parsers.csv.JSONMapParser`

JSON has the following entries and predefined patterns for `parserConfig`:

mapStrategy A strategy to indicate how to handle multi-dimensional Maps. This is one of:

- `DROP` Drop fields which contain maps
- `UNFOLD` Unfold inner maps. So `{ "foo" : { "bar" : 1} }` would turn into `{"foo.bar" : 1}`
- `ALLOW` Allow multidimensional maps
- `ERROR` Throw an error when a multidimensional map is encountered

`timestamp` This field is expected to exist and, if it does not, then current time is inserted.

5.1.3. Parser Configuration

The configuration for the various parser topologies is defined by JSON documents stored in ZooKeeper. The JSON document is structured in the following way:

parserClassName

The fully qualified class name for the parser to be used.

sensorTopic

The Kafka topic to send the parsed messages to.

parserConfig

A JSON Map representing the parser implementation specific configuration.

fieldTransformations

An array of complex objects representing the transformations to be done on the message generated from the parser before writing out to the Kafka topic.

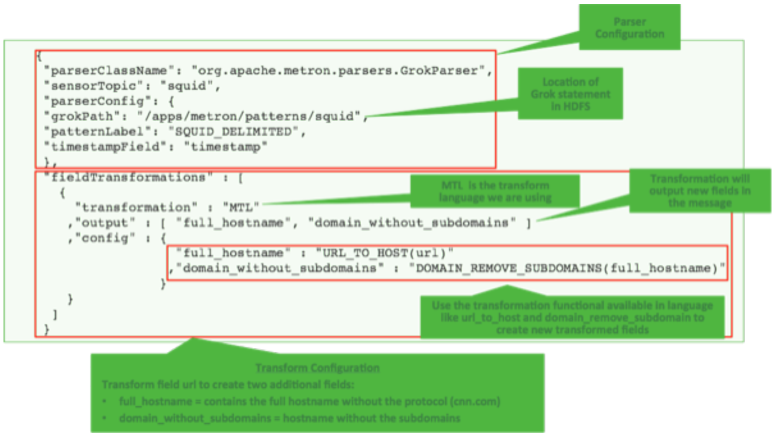
The fieldTransformations is a complex object which defines a transformation that can be done to a message. This transformation can perform the following:

- Modify existing fields to a message
- Add new fields given the values of existing fields of a message
- Remove existing fields of a message

5.1.3.1. fieldTransformation Configuration

In this example, the host name is extracted from the URL by way of the URL_TO_HOST function. Domain names are removed by using DOMAIN_REMOVE_SUBDOMAINS, thereby creating two new fields (`full_hostname` and `domain_without_subdomains`) and adding them to each message.

Figure 5.1. Configuration File with Transformation Information



```
{
 "parserClassName": "org.apache.metron.parsers.GrokParser",
 "sensorTopic": "squid",
 "parserConfig": {
 "grokPath": "/apps/metron/patterns/squid",
 "patternLabel": "SQUID_DELIMITED",
 "timestampField": "timestamp"
 },
 "fieldTransformations" : [
   {
     "transformation" : "MTL"
    ,"output" : [ "full_hostname", "domain_without_subdomains" ]
    ,"config" : {
                  "full_hostname" : "URL_TO_HOST(url)"
                 ,"domain_without_subdomains" : "DOMAIN_REMOVE_SUBDOMAINS(full_hostname)"
                 }
   }
  ]
}
```

Parser Configuration
Location of Grok statement in HDFS
MTL is the transform language we are using
Transformation will output new fields in the message
Use the transformation functional available in language like url_to_host and domain_remove_subdomain to create new transformed fields
Transform Configuration
Transform field url to create two additional fields:
• full_hostname = contains the full hostname without the protocol (cnn.com)
• domain_without_subdomains = hostname without the subdomains

The format of a fieldTransformation is as follows:

input

An array of fields or a single field representing the input. This is optional; if unspecified, then the whole message is passed as input.

output
The outputs to produce from the transformation. If unspecified, it is assumed to be the same as inputs.

transformation
The fully qualified class name of the transformation to be used. This is either a class which implements FieldTransformation or a member of the FieldTransformations enum.

config
A String to Object map of transformation specific configuration.

HCP currently implements the following fieldTransformations options:

REMOVE
This transformation removes the specified input fields. If you want a conditional removal, you can pass a Metron Query Language statement to define the conditions under which you want to remove the fields.

The following example removes `field1` unconditionally:

```
{
...
    "fieldTransformations" : [
          {
            "input" : "field1"
          , "transformation" : "REMOVE"
          }
                      ]
}
```

The following example removes field1 whenever field2 exists and has a corresponding value equal to 'foo':

```
{
...
  "fieldTransformations" : [
          {
            "input" : "field1"
          , "transformation" : "REMOVE"
          , "config" : {
              "condition" : "exists(field2) and field2 ==
 'foo'"
                       }
          }
                      ]
}
```

IP_PROTOCOL
This transformation maps IANA protocol numbers to consistent string representations.

The following example maps the `protocol` field to a textual representation of the protocol:

```
{
...
    "fieldTransformations" : [
          {
            "input" : "protocol"
          , "transformation" : "IP_PROTOCOL"
          }
                      ]
}
```

STELLAR, lo

This transformation executes a set of transformations expressed as Stellar Language statements.

The following example adds three new fields to a message:

utc_timestamp

The UNIX epoch timestamp based on the timestamp field, a dc field which is the data center the message comes from and a dc2tz map mapping data centers to timezones.

url_host

The host associated with the url in the url field.

url_protocol

The protocol associated with the url in the url field.

```
{
...
    "fieldTransformations" : [
          {
           "transformation" : "STELLAR"
          ,"output" : [ "utc_timestamp", "url_host",
 "url_protocol" ]
          ,"config" : {
            "utc_timestamp" : "TO_EPOCH_TIMESTAMP(timestamp,
 'yyyy-MM-dd
HH:mm:ss', MAP_GET(dc, dc2tz, 'UTC') )"
           ,"url_host" : "URL_TO_HOST(url)"
           ,"url_protocol" : "URL_TO_PROTOCOL(url)"
                      }
          }
                      ]
   ,"parserConfig" : {
      "dc2tz" : {
                "nyc" : "EST"
               ,"la" : "PST"
               ,"london" : "UTC"
                }
    }
}
```

Note that the dc2tz map is in the parser config, so it is accessible in the functions.

5.2. Telemetry Data Source Parsers Bundled with Hortonworks Cybersecurity Package

Telemetry data sources are sensors that provide raw events that are captured and pushed into Kafka topics to be ingested in Hortonworks Cybersecurity Package (HCP) powered by Metron. This section describes the telemetry data sources bundled with HCP 1.0:

- Snort [106]
- Bro [106]
- YAF (NetFlow) [107]

- Indexing [107]
- pcap [107]

For information about how to add telemetry data sources, see Adding a New Telemetry Data Source [6].

5.2.1. Snort

Snort is one of the more popular network intrusion prevention systems (NIPS). Snort monitors network traffic and produces alerts that are generated based on signatures from community rules. HCP plays the output of the packet capture probe to Snort and whenever Snort alerts are triggered. HCP uses the kafka-console-producer to send these alerts to a Kafka topic. After the Kafka topic receives Snort alerts, they are retrieved by the parsing topology in Storm.

By default, the Snort parser is configured to use ZoneId.systemDefault() for the source `timeZone` for the incoming data and MM/dd/yy-HH:mm:ss.SSSSSS as the default `dateFormat`. Valid timezones are per Java's ZoneId.getAvailableZoneIds(). DateFormats should be valid per options listed on the following website: https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html. Below is a sample configuration with the `dateFormat` and `timeZone` explicitly set in the parser config.

```
"parserConfig": {
"dateFormat" : "MM/dd/yy-HH:mm:ss.SSSSSS",
"timeZone" : "America/New_York"
}
```

Note

When you install and configure Snort, you must configure Snort to include the year in the timestamp by modifying the `snort.conf` file as follows:

```
# Configure Snort to show year in timestamps
config show_year
```

This is important for the proper functioning of indexing and analytics.

5.2.2. Bro

The Bro ingest data source is a custom Bro plug-in that pushes DPI (deep packet inspection) metadata into HCP.

Bro is primarily used as a DPI metadata generator. HCP does not currently use the IDS alert features of Bro. HCP integrates with Bro by way of a Bro plug-in, and does not require recompiling of Bro code.

The Bro plug-in formats Bro output messages into JSON and puts them into a Kafka topic. The JSON message output by the Bro plug-in is designed to be parsed by the HCP Bro parsing topology.

DPI metadata is not a replacement for packet capture (pcap), but rather a complement. Extracting DPI metadata (API Layer 7 visibility) is expensive, and therefore is performed on only selected protocols. You should enable DPI for HTTP and DNS protocols so that, while the pcap probe records every single packets it sees on the wire, the DPI metadata is extracted only for a subset of these packets.

5.2.3. YAF (NetFlow)

The YAF (yet another flowmeter) data source ingests NetFlow data into HCP.

Not everyone wants to ingest pcap data due to space constraints and the load exerted on all infrastructure components. NetFlow, while not a substitute for pcap, is a high-level summary of network flows that would be contained in the pcap files. If you do not want to ingest pcap, then you should at least enable NetFlow. HCP uses YAF to generate IPFIX (NetFlow) data from the HCP pcap probe, so the output of the probe is IPFIX instead of raw packets. If NetFlow is generated instead of pcap, then the NetFlow data goes to the generic parsing topology instead of the pcap topology.

5.2.4. Indexing

The Indexing topology takes data ingested into Kafka from enriched topologies and sends the data to an indexing bolt configured to write to one or more of the following indices:

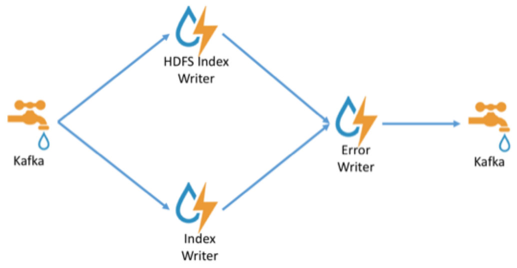
- Elasticsearch or Solr
- HDFS under `/apps/metron/enrichment/indexed`

Indices are written in batch and the batch size is specified in the Enrichment Configuration file by the batchSize parameter. This configuration is variable by sensor type.

Errors during indexing are sent to a Kafka topic named `indexing_error`.

The following figure illustrates the data flow between Kafka, the Indexing topology, and HDFS.

Figure 5.2. Indexing Architecture



5.2.5. pcap

Packet capture (pcap) is a performant C++ probe that captures network packets and streams them into Kafka. A pcap Storm topology then streams them into HCP. The purpose of including pcap source with HCP is to provide a middle tier in which to negotiate retrieving packet capture data that flows into HCP. This packet data is of a form that libpcap-based tools can read.

The network packet capture probe is designed to capture raw network packets and bulk-load them into Kafka. Kafka files are then retrieved by the pcap Storm topology and bulk-loaded into Hadoop Distributed File System (HDFS). Each file is stored in HDFS as a sequence file.

HCP provides three methods to access the pcap data:

- Rest API
- pycapa
- DPDK

There can be multiple probes into the same Kafka topic. The recommended hardware for the probe is an Intel family of network adapters that are supportable by Data Plane Development Kit (DPDK).

5.3. Enrichment Framework

Enrichments add additional context to the streaming message. The enrichment framework takes the data from the parsing topologies that have been normalized into the HCP data format (JSON files) and performs the following enhancements:

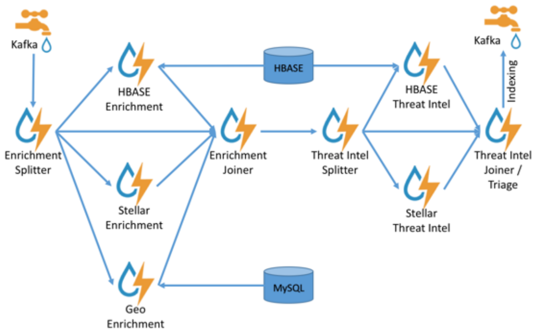
- Enriches messages with external data from data stores by adding new information based on existing fields in the messages
- Marks messages as threats based on data in external data stores
- Marks threat alerts with a numeric triage level based on a set of Stellar rules

The configuration for the enrichment topology is defined by JSON documents stored in ZooKeeper. HCP features two types of configurations:

- Sensor Enrichment Configuration [109]
- Global Configuration [115]

The following figure illustrates the enrichment flow for both individual sensor enrichment and threat intelligence enrichment.

Figure 5.3. HCP Enrichment Flow



5.3.1. Sensor Enrichment Configuration

The sensor enrichment configuration configures enrichments for a given sensor (for example, Snort). The sensor enrichment configuration configures two types of enrichments: individual sensor enrichments and threat intelligence enrichments. The configuration for both types of enrichments is a complex JSON object with the following top-level fields:

index The name of the sensor

batchSize The size of the batch that is written to the indices at once

enrichment A complex JSON object representing the configuration of the enrichments

threatIntel A complex JSON object representing the configuration of the threat intelligence enrichments

The remaining configuration differs for the two types of enrichments. See the following sections for information about both of these configuration types.

5.3.1.1. Individual Sensor Enrichments

HCP includes the following individual sensor enrichments:

Geo Provides GeoIP information, which includes coordinates, city, state, and country information, to any external IP address.

Asset Provides the host name for an IP address. If the IP address is known, then the enrichment provides everything else that is known of the asset from the LDAP, AD, or enterprise inventory stores.

User Provides the user that owns the session/alert associated with the IP-application pair.

The JSON documents for the individual enrichment configurations are structured as follows:

Table 5.1. Individual Enrichment Configuration Fields

Field	Description	Example
`fieldToTypeMap`	In the case of a simple HBase enrichment (in other words, a key/value lookup), the mapping between fields and the enrichment types associated with those fields must be known. This enrichment type is used as part of the HBase key.	`"fieldToTypeMap" : { "ip_src_addr" : [ "asset_enrichment" ] }`
`fieldMap`	The map of enrichment bolts names to configuration handlers which know how to split the message up. The simplest of which is just a list of fields. More complex examples would be the STELLAR enrichment which provides STELLAR statements. Each field is sent to the enrichment referenced in the key.	`"fieldMap": {"hbaseEnrichment": ["ip_src_addr","ip_dst_addr"]}`
`config`	The general configuration for the enrichment.	`"config": {"typeToColumnFamily": { "asset_enrichment" : "cf" } }`

The `config` map is intended to house enrichment-specific configurations. For example, `hbaseEnrichment` specifies the mappings between the enrichment types to the column families.

The `fieldMap` contents contain the routing and configuration information for the enrichments. Routing defines how the messages is split up and sent to the enrichment adapter bolts. The simplest `fieldMapcontents` provides a simple list as in:

```
 "fieldMap": {
      "geo": [
        "ip_src_addr",
        "ip_dst_addr"
      ],
      "host": [
        "ip_src_addr",
        "ip_dst_addr"
      ],
      "hbaseEnrichment": [
        "ip_src_addr",
        "ip_dst_addr"
      ]
      }
```

Based on this sample config, both `ip_src_addr` and `ip_dst_addr` will go to the `geo`, `host`, and `hbaseEnrichment` adapter bolts.

5.3.1.2. Stellar Enrichment Configuration

For the `geo`, `host`, and `hbaseEnrichment`, this is sufficient. However, more complex enrichments might contain their own configuration. Currently, the `stellar` enrichment is more adaptable and thus requires a more nuanced configuration.

At its most basic, we want to take a message and apply a couple of enrichments, such as converting the `hostname` field to lowercase. We do this by specifying the transformation inside of the `config` for the `stellar` fieldMap. There are two syntaxes that are supported, specifying the transformations as a map with the key as the field and the value the stellar expression:

```
"fieldMap": {
       ...
      "stellar" : {
        "config" : {
          "hostname" : "To_LOWER(hostname)"
        }
      }
    }
```

Another approach is to make the transformations as a list with the same `var := expr` syntax as is used in the Stellar REPL, such as:

```
"fieldMap": {
       ...
      "stellar" : {
        "config" : [
          "hostname := TO_LOWER(hostname)"
        ]
      }
```

```
   }
```

Sometimes arbitrary stellar enrichments may take enough time that you would prefer to split some of them into groups and execute the groups of stellar enrichments in parallel. Take, for instance, if you wanted to do an HBase enrichment and a profiler call which were independent of one another. This use case is supported by splitting the enrichments up as groups.

For example:

```
"fieldMap": {
      ...
     "stellar" : {
       "config" : {
         "malicious_domain_enrichment" : {
           "is_bad_domain" : "ENRICHMENT_EXISTS('malicious_domains',
ip_dst_addr, 'enrichments', 'cf')"
         },
         "login_profile" : [
           "profile_window := PROFILE_WINDOW('from 6 months ago')",
           "global_login_profile := PROFILE_GET('distinct_login_attempts',
'global', profile_window)",
           "stats := STATS_MERGE(global_login_profile)",
           "auth_attempts_median := STATS_PERCENTILE(stats, 0.5)",
           "auth_attempts_sd := STATS_SD(stats)",
           "profile_window := null",
           "global_login_profile := null",
           "stats := null"
         ]
       }
     }
   }
```

Here we want to perform two enrichments that hit HBase and we would rather not run in sequence. These enrichments are entirely independent of one another (i.e. neither relies on the output of the other). In this case, we've created a group called `malicious_domain_enrichment` to inquire about whether the destination address exists in the HBase enrichment table in the `malicious_domains` enrichment type. This is a simple enrichment, so we can express the enrichment group as a map with the new field `is_bad_domain` being a key and the stellar expression associated with that operation being the associated value.

In contrast, the stellar enrichment group `login_profile` is interacting with the profiler, has multiple temporary expressions (for example, `profile_window`, `global_login_profile`, and `stats`) that are useful only within the context of this group of stellar expressions. In this case, we would need to ensure that we use the list construct when specifying the group and remember to set the temporary variables to `null` so they are not passed along.

In general, things to note from this section are as follows:

- The stellar enrichments for the `stellar` enrichment adapter are specified in the `config` for the `stellar` enrichment adapter in the `fieldMap`
- Groups of independent (for example, no expression in any group depend on the output of an expression from an other group) may be executed in parallel

- If you have the need to use temporary variables, you may use the list construct. Ensure that you assign the variables to `null` before the end of the group.
- Ensure that you do not assign a field to a stellar expression which returns an object which JSON cannot represent.
- Fields assigned to Maps as part of stellar enrichments have the maps unfolded, similar to the HBase Enrichment
  - For example the stellar enrichment for field `foo` which assigns a map such as `foo := { 'grok' : 1, 'bar' : 'baz'}` would yield the following fields:
    - `foo.grok` == `1`
    - `foo.bar` == `'baz'`

5.3.1.3. Threat Intelligence Enrichments

HCP provides an extensible framework to plug in threat intelligence sources. Each threat intelligence source has two components: an enrichment data source and an enrichment bolt. The threat intelligence feeds are bulk loaded and streamed into a threat intelligence store similarly to how the enrichment feeds are loaded. The keys are loaded in a key-value format. The key is the indicator and the value is the JSON formatted description of what the indicator is. Hortonworks recommends using a threat feed aggregator such as Soltra to dedup and normalize the feeds via STIX/TAXII. HCP provides an adapter that is able to read Soltra-produced STIX/TAXII feeds and stream them into HBase. HCP additionally provides a flat file and STIX bulk loader that can normalize, dedup, and bulk load or stream threat intelligence data into HBase even without the use of a threat feed aggregator.

The JSON documents for the threat intelligence enrichment configurations are structured in the following way:

Table 5.2. Threat Intelligence Enrichment Configuration

Field	Description	Example
`fieldToTypeMap`	In the case of a simple HBase threat intelligence enrichment (in other words, a key/value lookup), the mapping between fields and the enrichment types associated with those fields must be known. This enrichment type is used as part of the HBase key.	`"fieldToTypeMap" : { "ip_src_addr" : [ "malicious_ips" ] }`
`fieldMap`	The map of threat intelligence enrichment bolts names to fields in the JSON messages. Each field is sent to the threat intelligence enrichment bolt referenced in the key.	`"fieldMap": {"hbaseThreatIntel": ["ip_src_addr","ip_dst_addr"]}`
`triageConfig`	The configuration of the threat triage scorer. In the situation where a threat is detected, a score is assigned to the message and embedded in the indexed message.	`"riskLevelRules" : { "IN_SUBNET(ip_dst_addr, '192.168.0.0/24')" : 10 }`
`config`	The general configuration for the threat intelligence.	`"config": {"typeToColumnFamily": { "malicious_ips" : "cf" } }`

The `config` map is intended to house threat intelligence specific configurations. For instance, the `hbaseThreatIntel` threat intelligence adapter specifies the mappings between the enrichment types and the column families.

The `triageConfig` field utilizes the following fields:

Table 5.3. triageConfig Fields

Field	Description	Example
riskLevelRules	The mapping of Metron Query Language (see above) queries to a score.	"riskLevelRules" : { "IN_SUBNET(ip_dst_addr, '192.168.0.0/24')" : 10 }
aggregator	An aggregation function that takes all non-zero scores representing the matching queries from `riskLevelRules` and aggregates them into a single score.	"MAX"

The supported aggregator functions are as follows:

MAX The maximum of all of the associated values for matching queries

MIN The minimum of all of the associated values for matching queries

MEAN The mean of all of the associated values for matching queries

POSITIVE_MEAN The mean of the positive associated values for the matching queries

The following is an example configuration for the YAF sensor:

```
{
  "index": "yaf",
  "batchSize": 5,
  "enrichment": {
    "fieldMap": {
      "geo": [
        "ip_src_addr",
        "ip_dst_addr"
      ],
      "host": [
        "ip_src_addr",
        "ip_dst_addr"
      ],
      "hbaseEnrichment": [
        "ip_src_addr",
        "ip_dst_addr"
      ]
    }
  ,"fieldToTypeMap": {
      "ip_src_addr": [
        "playful_classification"
      ],
      "ip_dst_addr": [
        "playful_classification"
      ]
    }
  },
  "threatIntel": {
    "fieldMap": {
      "hbaseThreatIntel": [
        "ip_src_addr",
        "ip_dst_addr"
      ]
    },
    "fieldToTypeMap": {
      "ip_src_addr": [
        "malicious_ip"
      ],
      "ip_dst_addr": [
        "malicious_ip"
      ]
    },
    "triageConfig" : {
      "riskLevelRules" : {
        "ip_src_addr == '10.0.2.3' or ip_dst_addr == '10.0.2.3'" : 10
      },
      "aggregator" : "MAX"
    }
  }
}
```

5.3.1.4. Using Stellar to Set up Threat Triage Configurations

The threat triage configuration defines conditions by associating them with scores. Because this is a per-sensor configuration, this fits nicely within the sensor enrichment configuration held in ZooKeeper. This configuration fits well within the threatIntel section of the configuration like so:

```
{
  ...
  ,"threatIntel" : {
            ...
           , "triageConfig" : {
                     "riskLevelRules" : {
                                 "condition1" : level1
                               , "condition2" : level2
                                  ...
                                        }
                     ,"aggregator" : "MAX"
                             }
                  }
}
```

riskLevelRules
Correspond to the set of condition to numeric level mappings that define the threat triage for this particular sensor.

aggregator
An aggregation function that takes all non-zero scores representing the matching queries from riskLevelRules and aggregates them into a single score.

The current supported aggregation functions are:

MAX
The maximum of all of the associated values for matching queries

MIN
The minimum of all of the associated values for matching queries

MEAN
The mean of all of the associated values for matching queries

POSITIVE_MEAN	The mean of the positive associated values for the matching queries

5.3.2. Global Configuration

Global enrichments are applied to all data sources as opposed to other enrichments that are applied at the field level. In other words, every message from every sensor is validated against the global configuration rules. The format of the global enrichment is a JSON string-to-object map that is stored in ZooKeeper.

This configuration is stored in ZooKeeper and looks something like the following:

```
{
  "es.clustername": "metron",
  "es.ip": "node1",
  "es.port": "9300",
  "es.date.format": "yyyy.MM.dd.HH",
  "fieldValidations" : [
              {
                "input" : [ "ip_src_addr", "ip_dst_addr" ],
                "validation" : "IP",
                "config" : {
                    "type" : "IPV4"
                           }
              }
                       ]
}
```

Inside the global configuration is a framework that validates all messages coming from all parsers. This is performed using validation plug-ins that make assertions about fields or whole messages.

The format for this framework is a `fieldValidations` field inside the global configuration. This is associated with an array of field validation objects structured that are defined in Global Configuration [68].

5.3.3. Using Stellar for Queries

You can use Stellar to create queries.

The Stellar query language supports the following:

- Referencing fields in the enriched JSON
- Simple boolean operations: and, not, or
- Simple arithmetic operations: *, /, +, - on real numbers or integers
- Simple comparison operations <, >, <=, >=
- if/then/else comparisons (in other words, if var1 < 10 then 'less than 10' else '10 or more')
- Determining whether a field exists (via `exists`)
- The ability to have parenthesis to make order of operations explicit
- User defined functions

The following is an example of a Stellar query:

```
IN_SUBNET( ip, '192.168.0.0/24') or ip in [ '10.0.0.1', '10.0.0.2' ] or
 exists(is_local)
```

This query evaluates to “true” precisely when one of the following is true:

- The value of the ip field is in the 192.168.0.0/24 subnet.
- The value of the ip field is 10.0.0.1 or 10.0.0.2.
- The field is_local exists.

5.3.4. Using Stellar to Transform Sensor Data Elements

You can use Stellar to customize sensor data elements to more useful information. For example, you can transform a timestamp to be specific to your timezone.

```
TO_EPOCH_TIMESTAMP(timestamp, 'yyyy-MM-dd HH:mm:ss', MAP_GET(dc, dc2tz,
 'UTC'))
```

For a message with a timestamp and dc field, we want to transform the timestamp to an epoch timestamp given a timezone that we will look up in a separate map, called dc2tz.

This converts the timestamp field to an epoch timestamp based on the following:

- Format yyyy-MM-dd HH:mm:ss
- The value in dc2tz associated with the value associated with field dc, defaulting to UTC

The following is a list of Stellar transformation functions currently supported by HCP:

TO_LOWER(string)	Transforms the first argument to a lowercase string
TO_UPPER(string)	Transforms the first argument to an uppercase string
TO_STRING(string)	Transforms the first argument to a string
TO_INTEGER(x)	Transforms the first argument to an integer
TO_DOUBLE(x)	Transforms the first argument to a double
TRIM(string)	Trims white space from both sides of a string
JOIN(list, delim)	Joins the components of the list with the specified delimiter
SPLIT(string, delim)	Splits the string by the delimiter. Returns a list.
GET_FIRST(list)	Returns the first element of the list
GET_LAST(list)	Returns the last element of the list
GET(list, i)	Returns the i'th element of the list (i is 0-based).

MAP_GET(key, map, default)	Returns the value associated with the key in the map. If the key does not exist, the default will be returned. If the default is unspecified, then null will be returned.
DOMAIN_TO_TLD(domain)	Returns the TLD of the domain
DOMAIN_REMOVE_TLD(domain)	Removes the TLD of the domain.
REMOVE_TLD(domain)	Removes the TLD from the domain
URL_TO_HOST(url)	Returns the host from a URL
URL_TO_PROTOCOL(url)	Returns the protocol from a URL
URL_TO_PORT(url)	Returns the port from a URL
URL_TO_PATH(url)	Returns the path from a URL
TO_EPOCH_TIMESTAMP(dateTime, format, timezone)	Returns the epoch timestamp of the dateTime given the format If the format does not have a timestamp and you wish to assume a given timestamp, you may specify the timezone optionally.

5.3.5. Management Utility

You should store your configurations on disk in the following structure, starting at $BASE_DIR:

- **global.json:** The global configuration
- **sensors:** The subdirectory containing sensor-enrichment configuration JSON (for example, snort.json or bro.json)

By default, this directory is deployed by the Ansible infrastructure at `$METRON_HOME/config/zookeeper`.

While the configs are stored on disk, they must be loaded into ZooKeeper to be used. You can use the `$METRON_HOME/bin/zk_load_config.sh utility` program to do this.

This has the following options:

-f,–force	Force operation
-h,–help	Generate Help screen
-i,–input_dir <DIR>	The input directory containing configuration files with names such as "$source.json"
-m,–mode <MODE>	The mode of operation: DUMP, PULL, or PUSH
-o,–output_dir (DIR)	The output directory that will store the JSON configuration from ZooKeeper

-z,–zk_quorum <host:port, [host:port]*>	ZooKeeper quorum URL (zk1:port,zk2:port,...)

Following are some usage examples:

- To dump the existing configs from ZooKeeper on the single-node vagrant machine:
  ```
  $METRON_HOME/bin/zk_load_configs.sh -z node1:2181 -m DUMP
  ```
- To push the configs into ZooKeeper on the single-node vagrant machine:
  ```
  $METRON_HOME/bin/zk_load_configs.sh -z node1:2181 -m PUSH -i
  $METRON_HOME/config/zookeeper
  ```
- To pull the configs from ZooKeeper to the single-node vagrant machine disk:
  ```
  $METRON_HOME/bin/zk_load_configs.sh -z node1:2181 -m PULL -o
  $METRON_HOME/config/zookeeper -f
  ```

5.4. Fastcapa

The probe leverages a poll-mode, burst-oriented mechanism to capture packets from a network interface and transmit them efficiently to a Kafka topic. Each packet is wrapped within a single Kafka message and the current timestamp, as epoch microseconds in network byte order, is attached as the message's key.

The probe leverages Receive Side Scaling (RSS), a feature provided by some ethernet devices that allows processing of received data to occur across multiple processes and logical cores. It does this by running a hash function on each packet, whose value assigns the packet to one, of possibly many, receive queues. The total number and size of these receive queues are limited by the ethernet device in use. More capable ethernet devices will offer a greater number and greater sized receive queues.

- Increasing the number of receive queues allows for greater parallelization of receive side processing.
- Increasing the size of each receive queue can allow the probe to handle larger, temporary spikes of network packets that can often occur.

A set of receive workers, each assigned to a unique logical core, are responsible for fetching packets from the receive queues. There can only be one receive worker for each receive queue. The receive workers continually poll the receive queues and attempt to fetch multiple packets on each request. The maximum number of packets fetched in one request is known as the 'burst size'. If the receive worker actually receives 'burst size' packets, then it is likely that the queue is under pressure and more packets are available. In this case the worker immediately fetches another 'burst size' set of packets. It repeats this process up to a fixed number of times while the queue is under pressure.

The receive workers then enqueue the received packets into a fixed size ring buffer known as a transmit ring. There is always one transmit ring for each receive queue. A set of transmit workers then dequeue packets from the transmit rings. There can be one or more transmit workers assigned to any single transmit ring. Each transmit worker has its own unique connection to Kafka.

- Increasing the number of transmit workers allows for greater parallelization when writing data to Kafka.

- Increasing the size of the transmit rings allows the probe to better handle temporary interruptions and latency when writing to Kafka.

After receiving the network packets from the transmit worker, the Kafka client library internally maintains its own send queue of messages. Multiple threads are then responsible for managing this queue and creating batches of messages that are sent in bulk to a Kafka broker. No control is exercised over this additional send queue and its worker threads, which can be an impediment to performance. This is an opportunity for improvement that can be addressed as follow-on work.