Hortonworks Cybersecurity Platform

Administration

(April 24, 2018)

docs.cloudera.com

Hortonworks Cybersecurity Platform: Administration

Hortonworks Cybersecurity Platform (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 dashboarding.

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

Reviewing the documentation roadmap related to administering Hortonworks Cybersecurity Platform (HCP) can help you locate the resources best suited to most immediate information needs.

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. Understanding Hortonworks Cybersecurity Suite

If you are a Platform Engineer responsible for installing, configuring, and maintaining Hortonworks Cybersecurity Platform (HCP) powered by Apache Metron, you must first understand HCP architecture and terminology.

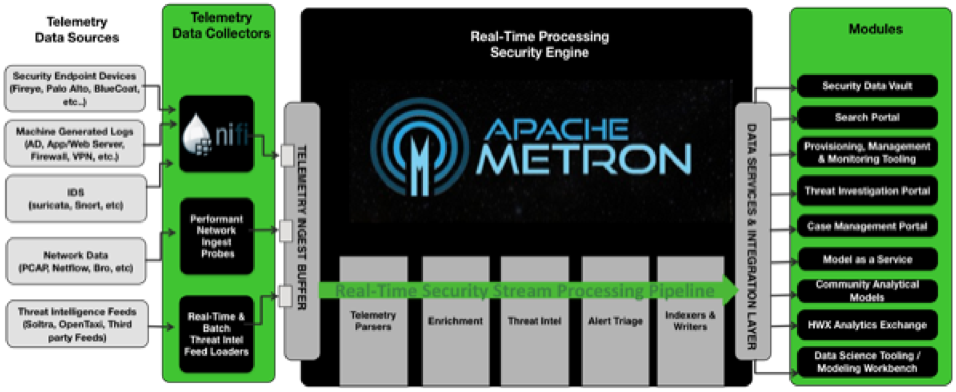
- Configuring and Customizing HCP [6]
- Monitor and Manage [87]
- Concepts [104]

2.1. HCP Architecture

Hortonworks Cybersecurity Platform (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]

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, OpenTaxi, 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 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. The data is parsed into a normalized JSON structure that Metron can read.

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

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

2.1.1. Real-Time Processing Security Engine

The core of Hortonworks Cybersecurity Platform (HCP) architecture is the Apache Metron real-time processing security engine. This component provides the ingest buffer to capture 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 by using the Stellar language). The engine then writes the events to a searchable index, as well as to HDFS, for analytics.

2.1.2. Telemetry Data Collectors

Telemetry data collectors push or stream the data source events into Apache Metron. Hortonworks Cybersecurity Platform (HCP) works with Apache 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

The data services and integration layer is a set of three HCP modules that 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	An HCP-provided management module that expedites provisioning and managing sensors. Other provisioning, management, and monitoring functions are supported through Apache Ambari.

2.2. Understanding HCP Terminology

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

alerts	Provide information about current security issues, vulnerabilities, and exploits.
Apache Kafka	A fast, scalable, durable, fault-tolerant publish-subscribe messaging system you can use for stream processing, messaging, website activity tracking, metrics collection and monitoring, log aggregation, and event sourcing.
Apache Storm	Enables data-driven, automated activity by providing a real-time, scalable, fault-tolerant, highly available, distributed solution for streaming data.

Apache ZooKeeper	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 hardware, software, and the information on them, as well as from disruption or misdirection of the services they provide.
data management	A set of utilities that get data into Apache HBase in a format that allows data flowing through Metron to be enriched with the results. Contains integrations with threat intelligence feeds exposed through 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 Apache Storm bolt that enriches the telemetry.
enrichment data loader	A streaming or a batch loader that stages data from the enrichment source into HCP so that telemetry is enriched with the information from the enrichment source in real time.
Forensic Investigator	Collects evidence on breach and attack incidents and prepares legal responses to breaches.
Model as a Service	An Apache Yarn application that deploys machine learning and statistical models, along with the associated Stellar functions, onto the cluster so that they can be retrieved in a scalable manner.
parser	An Apache Storm bolt that transforms telemetry from its native format to JSON so that Metron can use it.
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. After a profile defines normal behavior, you can build models to identify anomalous behavior.
Security Data Scientist	Works with security data, performing data munging, visualization, plotting, exploration, feature engineering, and model creation. Evaluates and monitors the correctness and currency of existing models.
Security Operations Center (SOC)	A centralized unit that manages 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 standards.

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 used throughout HCP: from simple field transformation to expressing triage rules.

telemetry data source

The source of telemetry data, from low level (packet capture), to intermediate level (deep packet analysis), to very high level (application logs).

telemetry event

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

3. Configuring and Customizing HCP

One of the key benefits of using Hortonworks Cybersecurity Platform (HCP) powered by Apache Metron is its easy extensibility. HCP comes to you 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 it.

3.1. Adding a New Telemetry Data Source

Part of customizing your HCP configuration is adding a new telemetry data source. Before HCP can process the information from your 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 Apache 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, you must first meet certain prerequisites, and then perform the following tasks:

1. Stream data into HCP
2. Create a parser for your new data source
3. Verify that events are indexed

3.1.1. Telemetry Data Source Parsers Bundled with HCP

Telemetry data sources are sensors that provide raw events that are captured and pushed into Apache Kafka topics to be ingested into Hortonworks Cybersecurity Platform (HCP) powered by Metron.

For information about how to add additional telemetry data sources, see Adding a New Telemetry Data Source [6].

3.1.1.1. Snort

Snort is a network intrusion prevention systems (NIPS). Snort monitors network traffic and generates alerts based on signatures from community rules. Hortonworks Cybersecurity Platform (HCP) sends the output of the packet capture probe to Snort. 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 uses `ZoneId.systemDefault()` as the source time zone for the incoming data and `MM/dd/yy-HH:mm:ss`.SSSSSS as the default date format. Valid time zones are determined according to the `Java ZoneId.getAvailableZoneIds()` values. DateFormats should match options at `https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html`.

Following is a sample configuration with dateFormat and timeZone explicitly set in the parser configuration file:

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

3.1.1.2. Bro

The Bro ingest data source is a custom Bro plug-in that pushes DPI (deep packet inspection) metadata into Hortonworks Cybersecurity Platform (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 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.

3.1.1.3. 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 are 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.

3.1.1.4. 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:

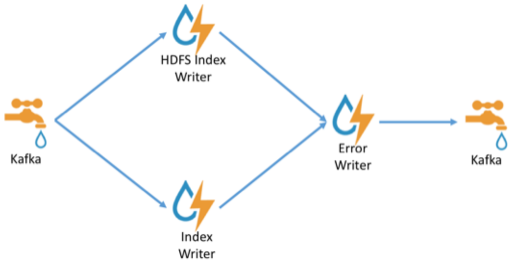
- Elasticsearch or Solr
- HDFS under `/apps/metron/enrichment/indexed`

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 3.1. Indexing Data Flow



3.1.1.5. 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).

3.1.2. Prerequisites to Adding a New Telemetry Data Source

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

- Ensure that the new sensor is installed and set up.
- Ensure that Apache NiFi or another telemetry data collection tool can feed the telemetry data source events into an Apache 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 set up several hosts. Note the locations of these hosts, their port numbers, and the Metron version for future use:

  KAFKA_HOST The host on which a Kafka broker is installed.

  ZOOKEEPER_HOST The host on which an Apache ZooKeeper server is installed.

  PROBE_HOST The host on which your sensor probes are installed. If you do not have any sensors installed, choose the host on which an Apache Storm supervisor is running.

  NIFI_HOST The host on which you install Apache NiFi.

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

  SEARCH_HOST The host on which Amazon Elasticsearch or Apache Solr is running. This is the host in your inventory hosts

	file that you put in the "search" group. 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 on which your Metron UI web application is running. This is the host in your inventory hosts file that you put in the "web" group.
METRON_VERSION	The release of the Metron binaries you are working with. (For example, HCP-1.4.2.0)

3.1.3. Understanding Streaming Data into HCP

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

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.

If you choose to use the Snort telemetry data source, you must meet the following configuration requirements:

- When you install and configure Snort, to ensure proper functioning of indexing and analytics, 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
  ```

- 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 defined in Java's `ZoneId.getAvailableZoneIds()`. DateFormats should use the options defined in `https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html`. The following sample configuration shows the `dateFormat` and `timeZone` values explicitly set in the parser configuration:

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

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

- **NiFi**

  This streaming method works for most types of data sources. To use it with HCP, you must install it manually on port 8089. 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.

- **Performant network ingestion probes**

  This 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 streaming method works for intelligence feeds that you want to view in real-time or collect batches of information to view or query at a later date. For more information see Using Threat Intelligence Feeds.

3.1.4. Streaming Data Using NiFi

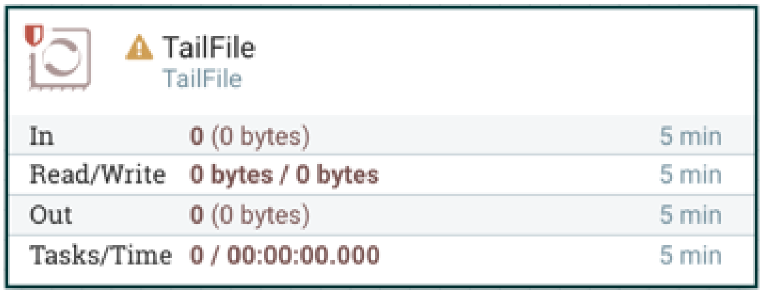
NiFi provides a highly intuitive streaming user interface that is compatible with most types of data sources.

1. Drag the 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:

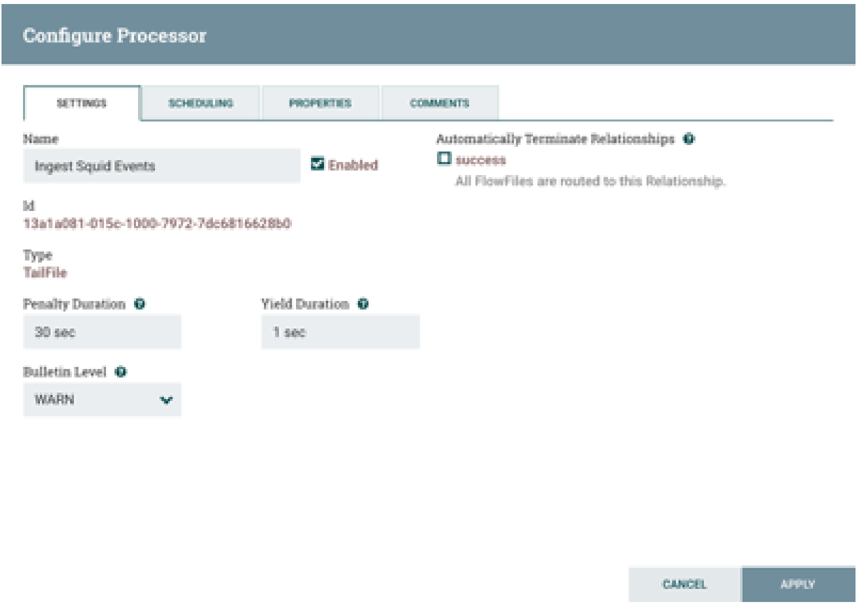


TailFile
TailFile

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 (processor icon) and select **Configure** to display the Configure Processor dialog box:

a. In the **Settings** tab, change the name to `Ingest $DATASOURCE Events`:



Name

Ingest Squid Events

☑ Enabled

Automatically Terminate Relationships

□ success

All FlowFiles are routed to this Relationship.

Id

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

Type

TailFile

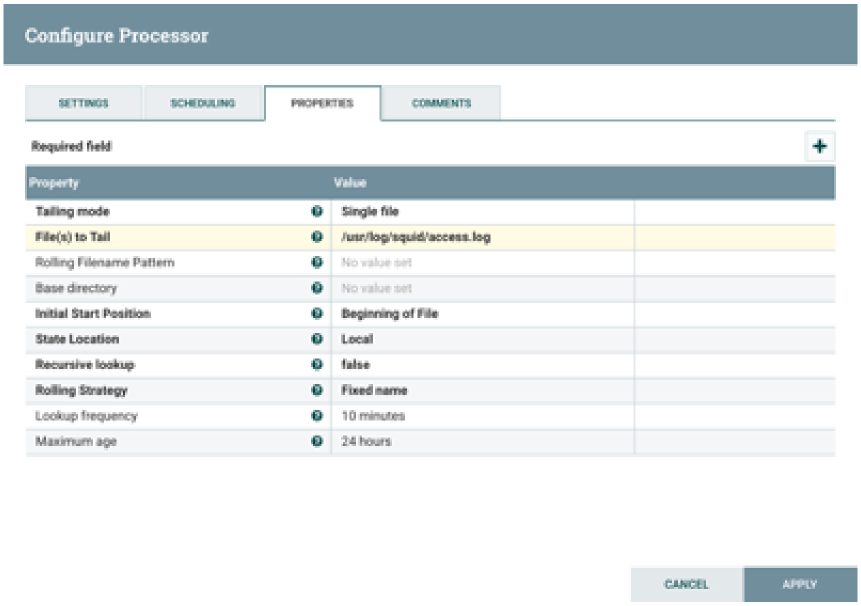
Penalty Duration: 30 sec

Yield Duration: 1 sec

Bulletin Level: WARN

b. In the **Properties** tab, enter the path to the data source file in the **Value** column for the **File(s) to Tail** property:

Figure 3.2. 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

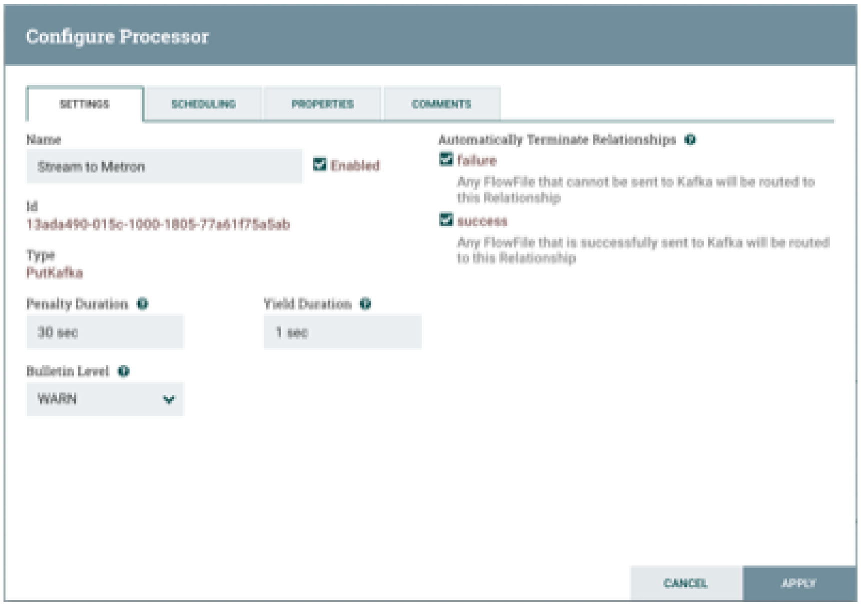
4. Add another processor by dragging the Processor icon to your workspace.

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 select the relationship check boxes for **failure** and **success**.

Figure 3.3. Configure Processor Settings Tab



Configure Processor

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

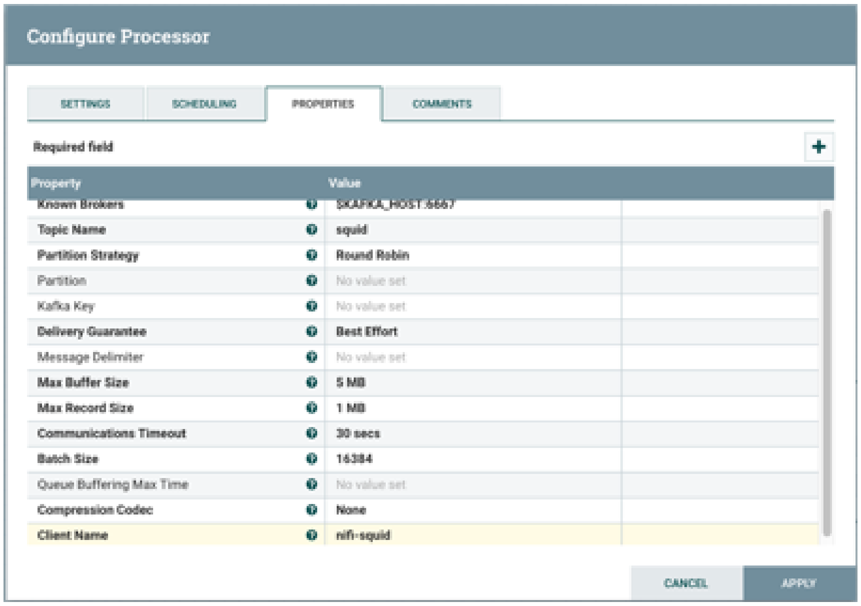
☑ success

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 3.4. Configure Processor Properties Tab



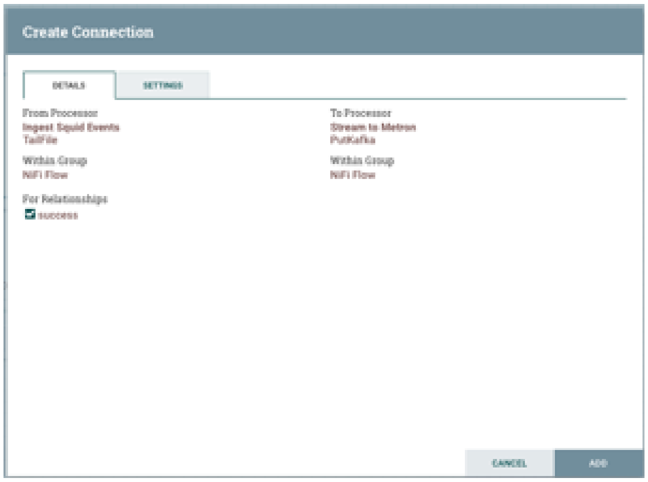
Configure Processor

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 the **Create Connection** dialog box.

   ### Figure 3.5. Create Connection Dialog Box

   Create Connection

   DETAILS | SETTINGS

   From Processor
   Ingest Squid Events
   TailFile

   Within Group
   NiFi Flow

   To Processor
   Stream to Metron
   PutKafka

   Within Group
   NiFi Flow

   For Relationships
   ☑ success

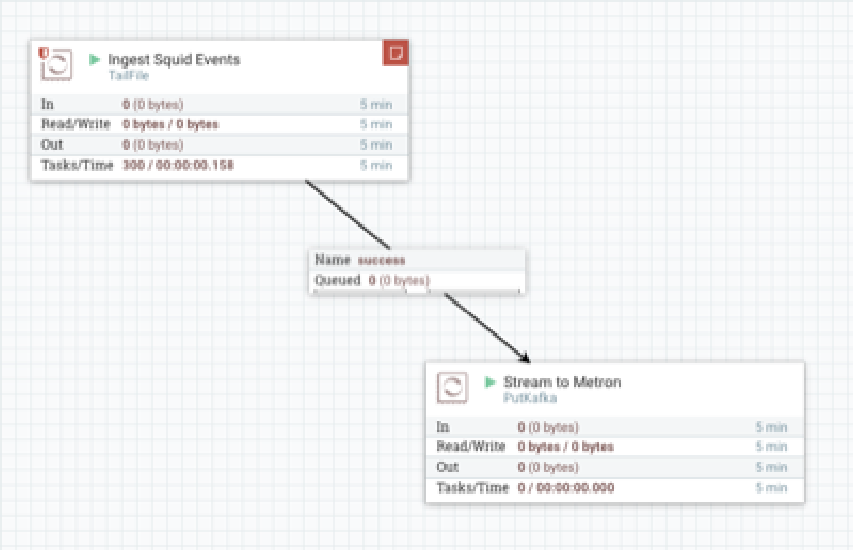
   CANCEL | ADD

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

11. Press **Shift** and draw a box around both parsers to select the entire flow; then click the green arrow.

   All of the processor icons turn into green arrows:

   ### Figure 3.6. 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

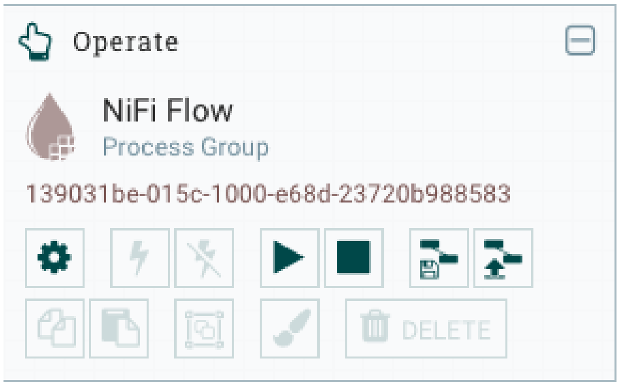
   Name success
   Queued 0 (0 bytes)

   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. In the Operate panel, click the arrow icon.

Figure 3.7. Operate Panel



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

14. Look at the Storm UI for the parser topology and confirm that tuples are coming in.

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

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

3.1.5. Understanding Parsing a New Data Source to HCP

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

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

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

- Type of parser to use

  HCP supports two types of parsers:

  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.

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

- How to parse

  HCP enables you to parse a new data source and transform data fields using the HCP Management module or the command line interface:

  - Creating a Parser for Your New Data Source by Using the Management Module [18]

- Create a Parser for Your New Data Source by Using the CLI [24]

3.1.6. Elasticsearch 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 Elasticsearch:

- 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

3.1.7. Creating a Parser for Your New Data Source by Using the Management Module

To add a new data source, you must create a parser that transforms the data source data into JSON messages suitable for downstream enrichment and indexing by HCP. Although HCP supports both Java and general-purpose parsers, you can learn the general process by viewing an example using the general-purpose parser Grok.

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

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

   b. View the different log files and determine which to parse:

   ```
   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.

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

   ```
   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 with the same name 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. Verify your new topic by listing the Kafka topics:

   ```
   /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.

   Important

   You must include `timestamp` in the Grok expression to ensure that the system uses the event time rather than the system time.

Refer to the Grok documentation for additional details.

4. Launch the HCP Management module from `$METRON_MANAGEMENT_UI_HOST:4200`, or follow these steps:

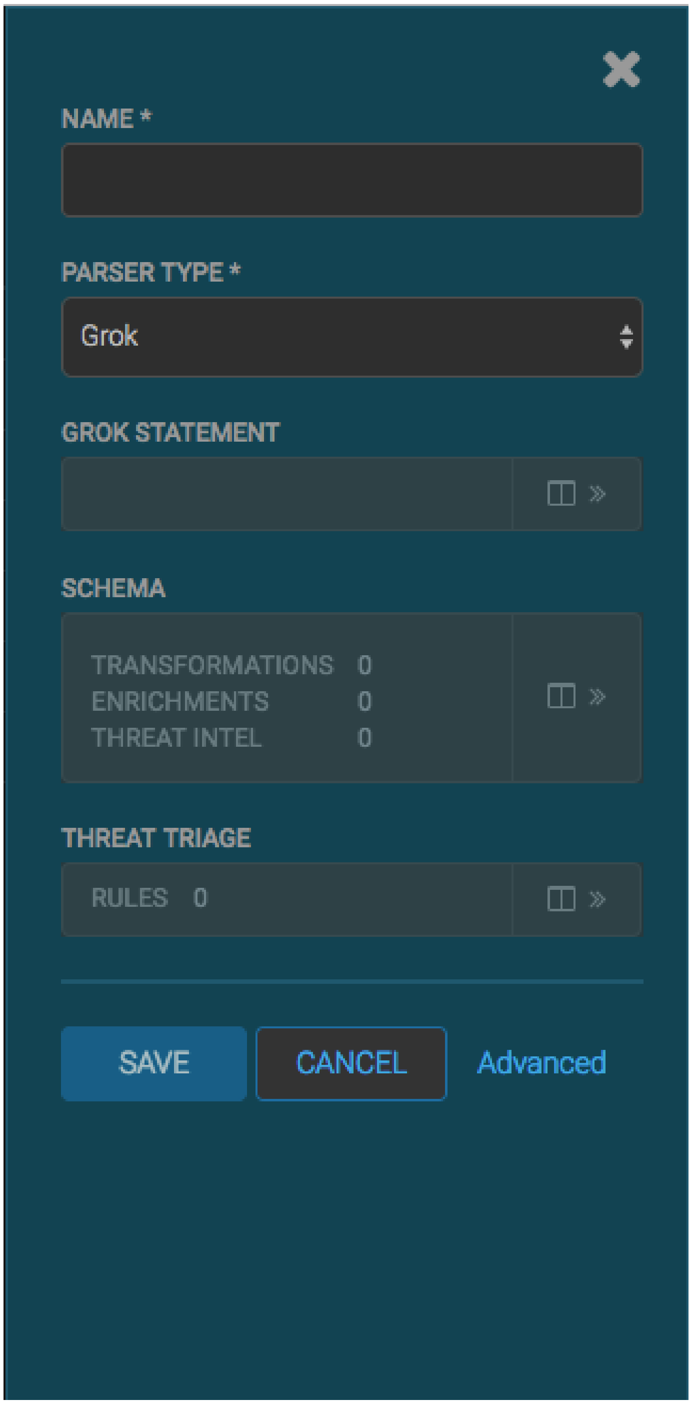
   a. From the Ambari Dashboard, click **Metron**.

   b. Select the **Summary**.

   c. Select **Metron Management UI**.

5. Under Operations, click Sensors.

6. Click + (add) to view the 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.

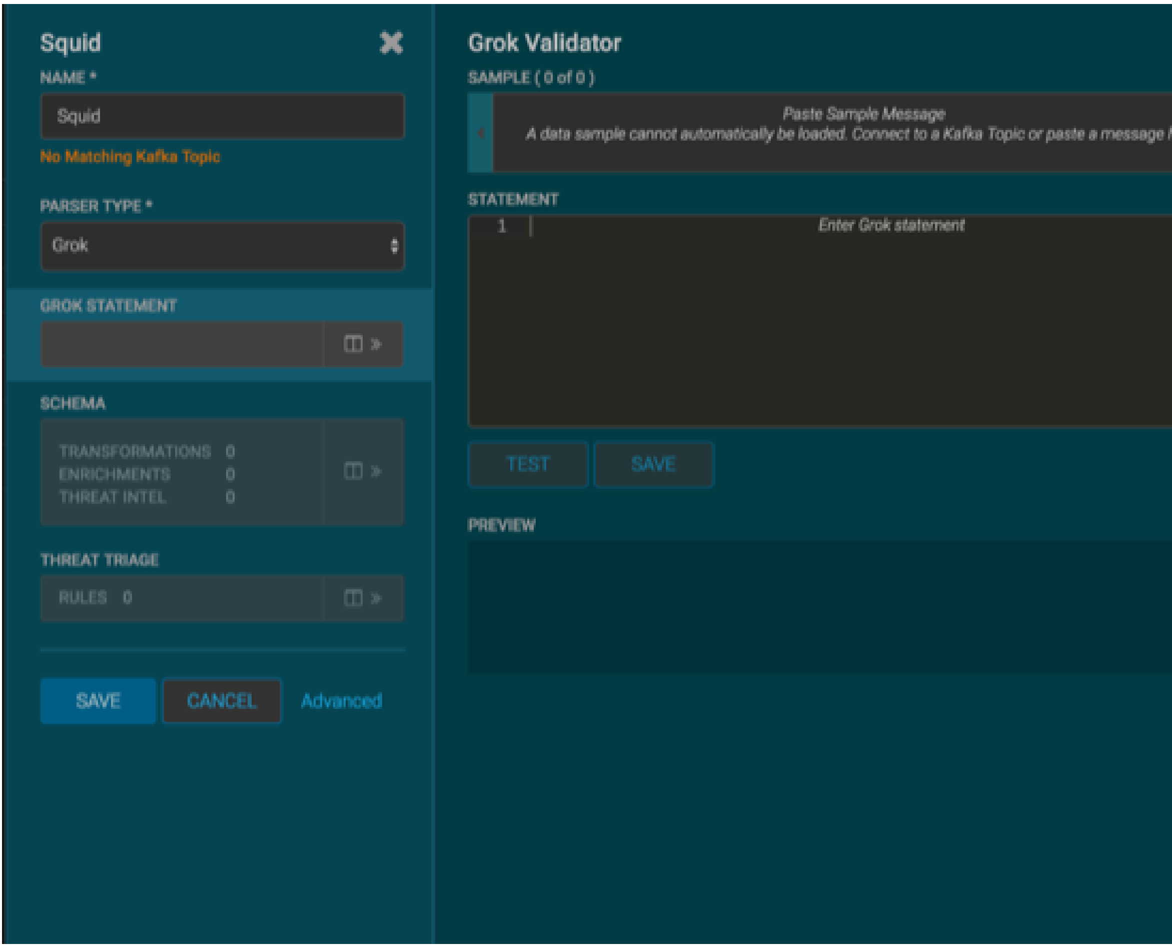
   Because you created a Kafka topic for your data source, the module should display a message similar to **Kafka Topic Exists. Emitting**. If no matching Kafka topic is found, the module displays **No Matching Kafka Topic**.

8. In the **Parser Type** field, choose the type of parser for the new sensor (in this example task, Grok).

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

9. Enter a Grok statement for the new parser:

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



Squid

No Matching Kafka Topic

Grok

TRANSFORMATIONS 0
ENRICHMENTS 0
THREAT INTEL 0

RULES 0

Paste Sample Message
A data sample cannot automatically be loaded. Connect to a Kafka Topic or paste a message

Enter Grok statement

   b. For **SAMPLE**, enter a sample log entry for the data source.

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

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

Important

You must include `timestamp` to ensure that the system uses the event time rather than the system time.

If the validator finds an error, it displays the error information; otherwise, the valid mapping displays in the **PREVIEW** field.

Consider repeating substeps a through c to ensure that your Grok statement is valid for all sensor logs.

d. Click **SAVE**.

10.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.

3.1.8. Transform Your New Data Source Parser Information by Using the Management Module

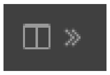
This section explains how to transform the information for your new data source and transform data fields.

1. From the list of sensors in the main window, select your new sensor.
2. Click the pencil icon in the toolbar.

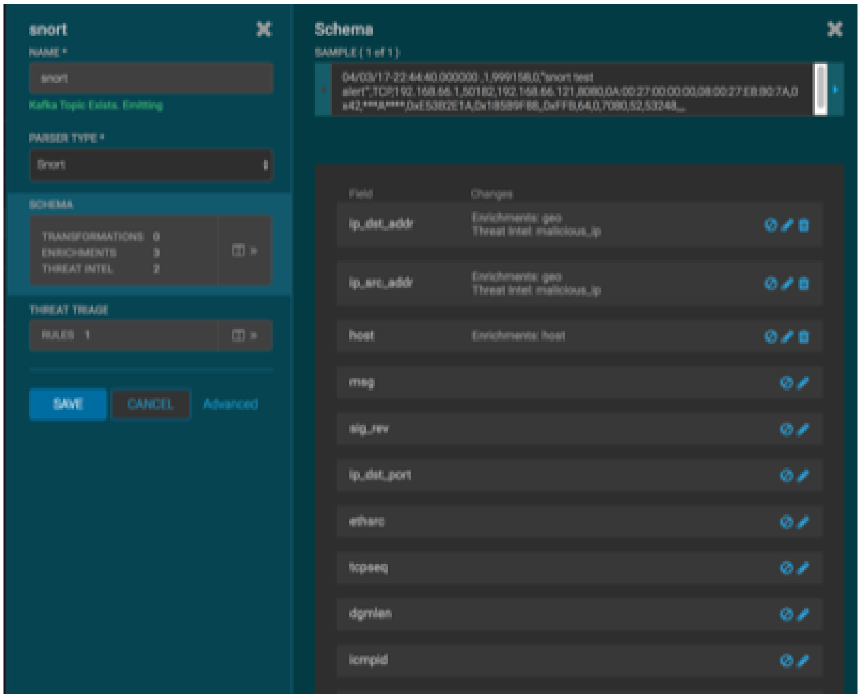
   The Management module displays the sensor panel for the new sensor.

   Note

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

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

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



NAME *: snort

Kafka Topic Exists. Emitting

PARSER TYPE *: Snort

SCHEMA

TRANSFORMATIONS	0
ENRICHMENTS	3
THREAT INTEL	2

THREAT TRIAGE

RULES 1

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,0
x42,***A****,0xE5382E1A,0x18589F88,0xFFB,64,0,7080,52,53248,,,
```

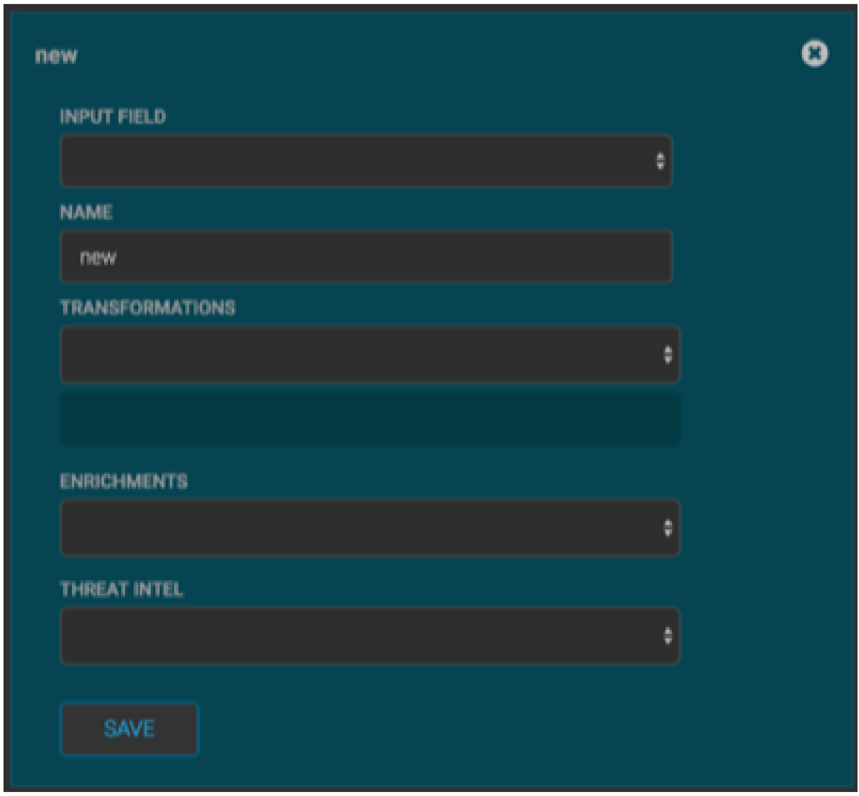
Field	Changes
ip_dst_addr	Enrichments: geo Threat Intel: malicious_ip
ip_src_addr	Enrichments: geo Threat Intel: malicious_ip
host	Enrichments: host
msg	
sig_rev	
ip_dst_port	
ethsrc	
tcpseq	
dgmlen	
icmpid	

The Sample field displays a parsed version of a sample message from the sensor. The Management module tests your transformations against these parsed messages.

You can use the right and left arrows to view the parsed version of each sample.

Although you can apply transformations to an existing field, users typically create and transform a new field.

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



pla

5. From **INPUT FIELD**, select the field to transform, enter the name of the new field in the **NAME** field, and then choose a function with the appropriate parameters in the **TRANSFORMATIONS** box.

6. Click **SAVE**.

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

7. You can also suppress fields from the transformation feature by clicking ⊘ (suppress icon).

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

8. Click **SAVE**.

3.1.9. Create a Parser for Your New Data Source by Using the CLI

As an alternative to using the HCP Management module to parse your new data source, you can use the CLI.

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

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

   b. Look at the different log files and determine which to parse:

      ```
      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.

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

      ```
      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 with the same name 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. Verify your new topic by listing the Kafka topics:

      ```
      /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.

   Important

   You must include `timestamp` to ensure that the system uses the event time rather than the system time.

   Refer to the Grok documentation for additional details.

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

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

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

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

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

   c. 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/
```

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

   a. As root, log into the host with HCP installed.

   ```
   ssh $HCP_HOST
   ```

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

   ```
   {
   "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)"
                     }
       }
      ]
   }
   ```

   parserClassName — The name of the parser's class in the .jar file.

   filterClassName — The filter to use.

   - This can be the fully qualified name of a class that implements the `org.apache.metron.parsers.interfaces.MessageFilte` interface. Message filters enable you to ignore a set of messages by using custom logic. The existing implementation is:

     - `STELLAR`: Enables you to apply a Stellar statement that returns a Boolean, which passes every message for which the statement returns `true`. The Stellar statement is specified by the `filter.query` property in the `parserConfig`. For example, the following Stellar filter includes messages that 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 to read metadata and make it available to field transformations (`false` by default).

There are two types of metadata supported in HCP:

- Environmental metadata about the whole system

  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 from an individual telemetry source that you might want to use within Metron

mergeMetadata

A Boolean indicating whether to merge metadata with the message (`false` by default).

If this property is set to `true`, then every metadata field becomes part of the messages and, consequently, is also available for 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 performed on the message generated from the parser before writing 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 enables 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 about using the fieldTransformations element in the parser configuration, see Understanding Parsers [104].
spoutParallelism	The Kafka spout parallelism (default to `1`). You can override the default on the command line.
spoutNumTasks	The number of tasks for the spout (default to `1`). You can override the default on the command line.
parserParallelism	The parser bolt parallelism (default to `1`). You can override the default on the command line.
parserNumTasks	The number of tasks for the parser bolt (default to `1`). You can override the default on the command line.
errorWriterParallelism	The error writer bolt parallelism (default to `1`). You can override the default on the command line.
errorWriterNumTasks	The number of tasks for the error writer bolt (default to `1`). You can override the default 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 configuration (this is a map). You can override the default on the command line.
securityProtocol	The security protocol to use for reading from Kafka (this is a string). You can be override this on the command line and also specify a value in the spout configuration via the `security.protocol` key. If both are specified, then they are merged and the CLI will take precedence.
stormConfig	The Storm configuration to use (this is a map). You can override this 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
```

You can safely ignore any resulting warning messages.

6. 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 verify that the new topology is listed and that it has no errors.

This new data source processor topology ingests from the $DATASOURCE Kafka topic that you created earlier and then parses the event with the HCP Grok framework using the Grok pattern defined earlier.

3.1.10. Verifying That Events Are Indexed

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

By convention, the index of new messages 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 you parse and normalize the raw security telemetry events, you must enrich the data elements of the normalized event. Enrichments add external data from data stores (such as Apache HBase) to make the data from the normalized event more useful and relevant.

Threat intelligence is another type of enrichment. For information about threat intelligence see Using Threat Intelligence.

Examples of enrichments are GEO where an external IP address is enriched with GeoIP information (lat/long coordinates + City/State/Country) and HOST 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.

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

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

- Asset
- GeoIP

- User

One of the advantages of the enrichment topology is that it groups messages by HBase key. Whenever you execute a Stellar function, you can add a caching layer, thus decreasing the need to call HBase for every event.

Prior to enabling an enrichment capability within HCP, you must load the enrichment store (which for HCP is primarily HBase) with enrichment data. You can load enrichment data from the local file system or HDFS, or you can use the parser framework to stream data into the enrichment store. The enrichment loader transforms the enrichment into a JSON format that Apache Metron can use. Additionally, the loading framework can detect and remove old data from the enrichment store automatically. .

After you load the stores, you can incorporate into the enrichment topology an enrichment bolt to a specific field or tag within a Metron message. The bolt can detect when it can enrich a field and then take an enrichment from the enrichment store and tag the message with it. The enrichment is then stored within the bolt's in-memory cache. HCP uses the underlying Storm routing capabilities to ensure that similar enrichment values are sent to the appropriate bolts that already have these values cached in-memory.

To configure an enrichment source, complete the following tasks:

- Prerequisites to Adding a New Telemetry Data Source [9]
- Bulk Loading Enrichment Information [30]
- Streaming Enrichment Information [38]

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

3.2.1. Bulk Loading Enrichment Information

If you decide to not stream enrichment data into the enrichment store, you can bulk load it from HDFS, which involves configuring an extractor file, configuring element-to-enrichment mapping, running the loader, and mapping fields to Apache HBase enrichments.

3.2.1.1. Bulk Loading Sources

You can bulk load enrichment information from the following sources:

- CSV File Ingestion
- HDFS via MapReduce
- Taxii Loader

CSV File

The shell script `$METRON_HOME/bin/flatfile_loader.sh` reads data from local disk and loads the enrichment data into an HBase table. This loader uses the special configuration parameter `inputFormatHandler` to specify 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 in which each line indicates a unit of information to 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	Required?	Description
-h		No	Generates the help screen or 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 is loaded. If this is a directory, then the files are 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 Through MapReduce

The shell script $METRON_HOME/bin/flatfile_loader.sh starts the MapReduce job to load data from HDFS to 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	Generates the help screen or 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 to
-c	–hbase_cf	Yes	The HBase table column family to import to
-i	–input	Yes	The input data location on local disk. If this is a file, then that file is loaded. If this is a directory, then the files are loaded recursively under that directory.
-l	–log4j	No	The log4j properties file to load

Short Code	Long Code	Is Required?	Description
-n	–enrichment_config	No	The JSON document describing the enrichments to configure. Unlike other loaders, this is run first if specified.

Taxii Loader

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

This Taxii server is often an aggregation server such as Soltra Edge.

This loader requires a configuration file describing the connection information to the Taxii server and Enrichment and Extractor configurations. 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" ]
}
```

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 to
columnFamily	The column family to import to
allowedIndicatorTypes	An array of acceptable threat intelligence types

The parameters for the utility are as follows:

Short Code	Long Code	Is Required?	Description
-h		No	Generates the help screen or set of options
-e	–extractor_config	Yes	JSON document describing the extractor for this input data source
-c	–taxii_connection_config	Yes	The JSON configuration 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

Short Code	Long Code	Is Required?	Description
-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

You use the extractor configuration file to bulk load the enrichment store into HBase. Complete the following steps to configure the extractor configuration file:

1. On the host on which Metron is installed, log in as root.

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:

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

4. 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:

```
"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:

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

In addition, if you prefer to piggyback your transformations, you can refer to the variable `domain`, which allows your indicator transforms to inherit transformations to this value.

`indicator_filter`

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

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

Including all of the supported Stellar extractor properties in the extractor configuration file, looks 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"
}
```

If you run a file import with this data and extractor configuration, you get 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 by using 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 looks 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 included when you cut and pasted the value_transform information:

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

The extractor_config.json file is not stored by the loader. If you want to reuse it, you must store it yourself.

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 Apache ZooKeeper.

1. On the host with Metron installed, log in as root.
2. Cut and paste the following syntax 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 data source you use 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 in the pasted syntax in Step 2:

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

3.2.1.4. Running the Enrichment Loader

After you configure the extractor configuration file and the element-enrichment mapping, you must run the loader to move the data from the enrichment source to the HCP enrichment store and store the enrichment configuration in Apache 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 the configuration is defined in the `extractor_config.json` file and populated into an HBase table called `enrichment`.

2. Verify that the logs were properly ingested to 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 Using the Management Module

After you establish dataflow to the HBase table, you must use the HCP Management module or the CLI to ensure that the enrichment topology is enriching the data flowing past.

You can use the Management module to refine the parser output in three ways:

- Transformations
- Enrichments
- Threat Intel

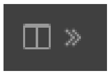
Prerequisite

Your sensor must be running and producing data to load sample data.

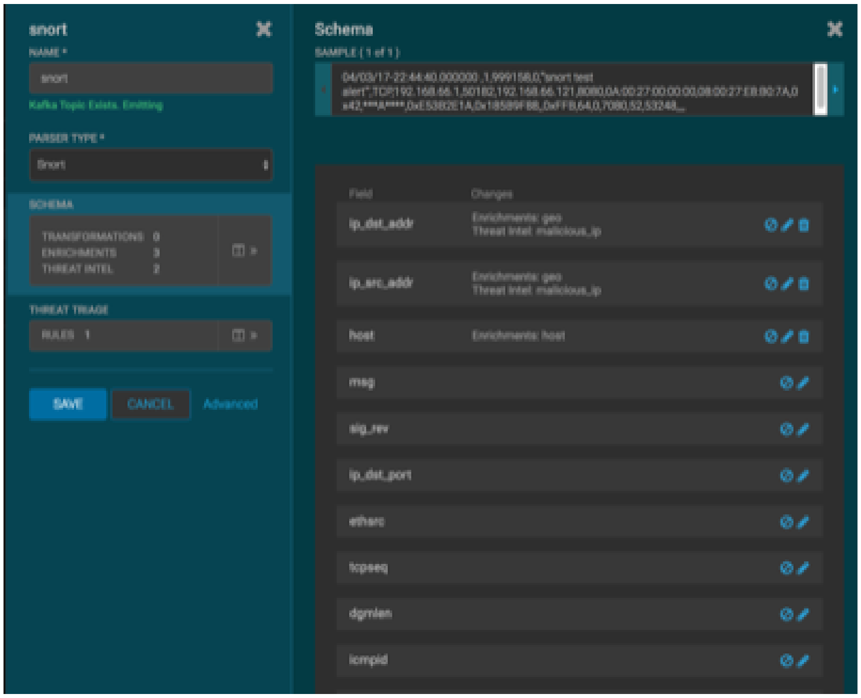
1. From the list of sensors in the main window, select your new sensor.

2. Click the pencil icon in the toolbar.

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

3. In the Schema panel, click .

4. Review the resulting message, field, and value information displayed in the Schema panel.



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,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 displays a parsed version of a sample message from the sensor. The Management module tests your transformations against these parsed messages.

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

5. Apply transformations to an existing field by clicking ✎ or create a new field by clicking + .

6. If you create a new field, complete the fields.

7. Click **SAVE**.

8. If you want to suppress fields from showing in the Index, click ⊘.

9. Click **SAVE**.

3.2.1.6. Mapping Fields to HBase Enrichments Using CLI

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:

```
{
  "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 finish enriching telemetry events, you should ensure that the enriched data is displaying on the Metron dashboard.

3.2.2. Streaming Enrichment Information

Streaming enrichment information is useful when you need enrichment information in real time. This type of information is most useful in real time as opposed to waiting for a bulk load of the enrichment information.

You incorporate streaming intelligence feeds slightly differently than when you use bulk loading. The enrichment information resides in its own parser topology instead of in an extraction configuration file. The parser file defines the input structure and how that data is used in enrichment. Streaming information goes to Apache HBase rather than to Apache Kafka, so you must configure the writer by using both the `writerClassName` and simple HBase enrichment writer (shew) parameters.

1. Define a parser topology in `$METRON_HOME/zookeeper/parsers/user.json`:

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

2. Populate the file with the parser topology definition.

   For example, the following commands associate IP addresses with user names for the Squid information.

   ```
   {
    "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
                }
 }
}
```

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 you 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. In this example, 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 Apache ZooKeeper:

   a. Create a Kafka topic sized to manage your estimated data flow:

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

   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:

   ```
   $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. Data is flowing into the HBase table, but you must 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:

```
{
  "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

You configure an indexing topology to store enriched data in one or more supported indexes. Configuration includes understanding supported indexes and the default configuration, specifying index parameters, tuning indexes, turning off HDFS writer, and, if necessary, seeking support.

- Understanding Indexing [40]
- Default Configuration [41]
- Indexing HDFS Tuning [43]
- Turning Off HDFS Writer [45]
- Support for Elasticsearch 5.x [45]

3.3.1. Understanding Indexing

Currently, Hortonworks Cybersecurity Platform (HCP) supports the following indices:

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

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

The `Indexing Configuration` file is a JSON file stored in Apache 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 following parameters are currently supported:

`index` The name of the index to write to (default is the name of the sensor).

`batchSize` The size of the batch allowed to be written to the indices at once (defaulted is 1).

`enabled` Whether the index or writer is enabled (default is `true`).

3.3.2. Default Configuration

If you do not configure the individual writers, the sensor-specific configuration uses default values. You can use this default configuration either by not creating an indexing configuration file or by entering the following in the file:

```
{
}
```

Not specifying a writer configuration causes a warning in the Storm console, such as `WARNING: Default and (likely) unoptimized writer config used for hdfs writer and sensor squid`. You can safely ignore this warning.

The default configuration has the following features:

- solr writer
  - index name the same as the sensor
  - batch size of 1
  - enabled
- 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 by Using the Management Module

If you want customized writer parameters, you can specify them using either the Management module or the CLI. Note that any properties managed by Apache Ambari must be modified within Ambari to persist.

For a list of the properties managed by Ambari, see Updating Properties [88].

1. In the Management module, edit your sensor by clicking ✎.
2. Click **Advanced**.
3. Enter index configuration information for your sensor.
4. Click the **Raw JSON** field and set the alert field to `"type": "nested"`:

```
 },
 "alert": {
 "type": "nested"
}
```

5. Click **Save**.

3.3.4. Specifying Index Parameters by 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`. Note that any properties managed by Apache Ambari must be modified within Ambari to persist.

For a list of the properties managed by Ambari, see Updating Properties [88].

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:

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

This syntax specifies the following parameter values:

- Solr writer or index
  - index name of "foo"
  - batch size of 100
  - enabled
- 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

  You must set this field to `"type": "nested"`.

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
```

3.3.5. Indexing HDFS Tuning

There are 48 partitions set for the indexing partition.

The following are the batch size settings for the Bro index.

```

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

The following are the settings 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
```

3.3.5.1. PCAP Tuning

PCAP is a Spout-only topology. Both Kafka topic consumption and HDFS writing is performed 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
                                +
                                +
```

3.3.6. Turning Off HDFS Writer

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

Create or modify

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

3.3.7. Support for Elasticsearch 5.x

Elasticsearch 5x requires that all sensor templates include a nested alert field definition. 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 5x requires the following changes to support HCP queries.

- Updating Elasticsearch Templates to Work with Elasticsearch 5.x [45]
- Updating Existing Indexes to Work with Elasticsearch 5x [46]

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

To update your existing Elasticsearch templates, perform the following steps:

1. Retrieve the template.

   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
```

3. Verify your changes:

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

4. Add the template back into Elasticsearch:

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

3.3.7.2. Updating Existing Indexes to Work with Elasticsearch 5x

To update existing indexes to work with Elasticsearch 5x, 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
```

3.3.8. Adding X-Pack Extension to Elasticsearch

You can add the X-Pack extension to Elasticsearch to enable secure connections for Elasticsearch.

Prerequisites

- Elasticsearch must be installed
- Kibana must be installed
- Choose the X-pack version that matches the version of Elasticsearch you are running

1. Use the Storm UI to stop the **random_access_indexing** topology.

   a. From **Topology Summary**, click **random_access_indexing**.

   b. Under **Topology actions**, click **Deactivate**.

2. Install X-Pack on Elasticsearch and Kibana.

   See Installing X-Pack for information on installing X-Pack.

3. After installing X-pack, navigate to the Elasticsearch node where Elasticsearch Master and the X-Pack were installed, then add a user name and password for Elasticsearch and Kibana to enable external connections from Metron components:

For example, the following creates a user `transport_client_user` with the password `changeme` and `superuser` credentials.

```
sudo /usr/share/elasticsearch/bin/x-pack/users useradd transport_client_user
 -p changeme -r superuser
```

4. Create a file containing the password you created in Step 3 and upload it to HDFS.

   For example:

```
echo changeme > /tmp/xpack-password
sudo -u hdfs hdfs dfs -mkdir /apps/metron/elasticsearch/
sudo -u hdfs hdfs dfs -put /tmp/xpack-password /apps/metron/elasticsearch/
sudo -u hdfs hdfs dfs -chown metron:metron /apps/metron/elasticsearch/xpack-
password
```

5. Set the X-Pack `es.client.class` by adding it to `$METRON_HOME/config/zookeeper/global.json`.

   For example, add the following to the `global.json` file:

```
{
...
  "es.client.settings" : {
      "es.client.class" : "org.elasticsearch.xpack.client.
PreBuiltXPackTransportClient",
      "es.xpack.username" : "transport_client_user",
      "es.xpack.password.file" : "/apps/metron/elasticsearch/xpack-password"
  }
  ...
}
```

6. Add the X-Pack changes to ZooKeeper:

```
$METRON_HOME/bin/zk_load_configs.sh -m PUSH -i METRON_HOME/config/zookeeper/
 -z $ZOOKEEPER
```

7. Create a custom X-Pack shaded and relocated jar file.

   Your jar file is specific to your licensing restrictions. However, you can use the following example for reference:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch-xpack-shaded</artifactId>
    <name>elasticsearch-xpack-shaded</name>
    <packaging>jar</packaging>
    <version>5.6.2</version>
    <repositories>
        <repository>
            <id>elasticsearch-releases</id>
            <url>https://artifacts.elastic.co/maven</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>x-pack-transport</artifactId>
            <version>5.6.2</version>
            <exclusions>
              <exclusion>
                <groupId>com.fasterxml.jackson.dataformat</groupId>
                <artifactId>jackson-dataformat-smile</artifactId>
              </exclusion>
              <exclusion>
                <groupId>com.fasterxml.jackson.dataformat</groupId>
                <artifactId>jackson-dataformat-yaml</artifactId>
              </exclusion>
              <exclusion>
                <groupId>com.fasterxml.jackson.dataformat</groupId>
                <artifactId>jackson-dataformat-cbor</artifactId>
              </exclusion>
              <exclusion>
                <groupId>com.fasterxml.jackson.core</groupId>
                <artifactId>jackson-core</artifactId>
              </exclusion>
              <exclusion>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
              </exclusion>
              <exclusion>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
              </exclusion>
              <exclusion>
                <groupId>log4j</groupId>
                <artifactId>log4j</artifactId>
              </exclusion>
              <exclusion> <!-- this is causing a weird build error if not
 excluded - Error creating shaded jar: null: IllegalArgumentException -->
                    <groupId>org.apache.logging.log4j</groupId>
                    <artifactId>log4j-api</artifactId>
                </exclusion>
            </exclusions>
          </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.4.3</version>
                <configuration>
                    <createDependencyReducedPom>true</
createDependencyReducedPom>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                          <filters>
                            <filter>
                              <artifact>*:*</artifact>
                              <excludes>
                                <exclude>META-INF/*.SF</exclude>
                                <exclude>META-INF/*.DSA</exclude>
                                <exclude>META-INF/*.RSA</exclude>
                              </excludes>
                            </filter>
                          </filters>
                          <relocations>
                                <relocation>
                                    <pattern>org.apache.logging.log4j</
pattern>
                                    <shadedPattern>org.apache.metron.
logging.log4j</shadedPattern>
                                </relocation>
                            </relocations>
                            <artifactSet>
                                <excludes>
                                    <exclude>org.slf4j.impl*</exclude>
                                    <exclude>org.slf4j:slf4j-log4j*</
exclude>
                                </excludes>
                            </artifactSet>
                            <transformers>
                                <transformer
                                  implementation="org.apache.maven.plugins.
shade.resource.DontIncludeResourceTransformer">
                                     <resources>
                                        <resource>.yaml</resource>
                                        <resource>LICENSE.txt</resource>
                                        <resource>ASL2.0</resource>
                                        <resource>NOTICE.txt</resource>
                                      </resources>
                                </transformer>
                                <transformer
                                        implementation="org.apache.maven.
plugins.shade.resource.ServicesResourceTransformer"/>
                                <transformer
                                        implementation="org.apache.maven.
plugins.shade.resource.ManifestResourceTransformer">
                                    <mainClass></mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

8. After you build the `elasticsearch-xpack-shaded-5.6.2.jar` file, you must make the file available to Storm when you submit the topology.

Create a `contrib` directory for indexing and then put the `elasticsearch-xpack-shaded-5.6.2.jar` file in this directory:

```
$METRON_HOME/indexing_contrib/elasticsearch-xpack-shaded-5.6.2.jar
```

9. Use the Storm UI to restart the **random_access_indexing** topology.

   a. From **Topology Summary**, click **random_access_indexing**.

   b. Under **Topology actions**, click **Start**.

3.3.9. 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 ensure that indexing has started for your topologies. If the Storm UI indicates that the indexing topology has started, you might need to install the latest version 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.

3.4. Preparing to Configure Threat Intelligence

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 [50]
- Bulk Loading Enrichment Information [30]
- Creating a Streaming Threat Intel Feed Source [60]

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. As a best practice, 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.

3.4.2.1. Bulk Loading Threat Intelligence Sources

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

- CSV Ingestion
- HDFS through MapReduce
- Taxii Loader

CSV File

The shell script `$METRON_HOME/bin/flatfile_loader.sh` reads data from local disk and loads the threat intelligence data into an HBase table. This loader uses the special configuration parameter `inputFormatHandler` to specify 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 in which each line indicates a unit of information to 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	Generates 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

Short Code	Long Code	Is Required?	Description
			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 the 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	Generates 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 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	Generates 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 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
   ```

3.4.2.3. Configure Mapping for the Intelligence Feed

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

1. On the host with Metron installed, log in as root.

2. 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" ]
          }
     }
   }
}
```

3. Remove any non-ASCII invisible characters in the pasted syntax in Step 2:

```
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 define the threat intelligence source, threat intelligence extractor, and threat intelligence mapping configuration, you must 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
   ```

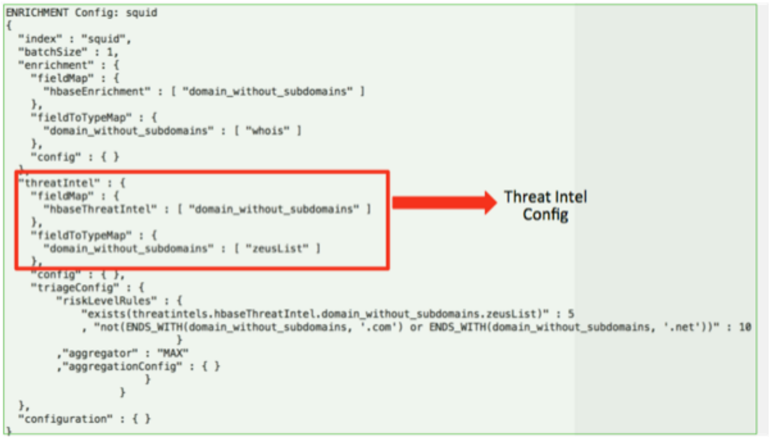
   This command adds the threat intelligence data into HBase and establishes a ZooKeeper mapping. The data is extracted using the extractor and configuration defined in the `extractor_config.json` file and populated into an HBase table called `threatintel`.

3. Verify that the logs are properly ingested to HBase:

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

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

   Figure 3.8. 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 by Using the Management Module

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

You can add or modify each of the parser outputs in the **Schema** field.

Note

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

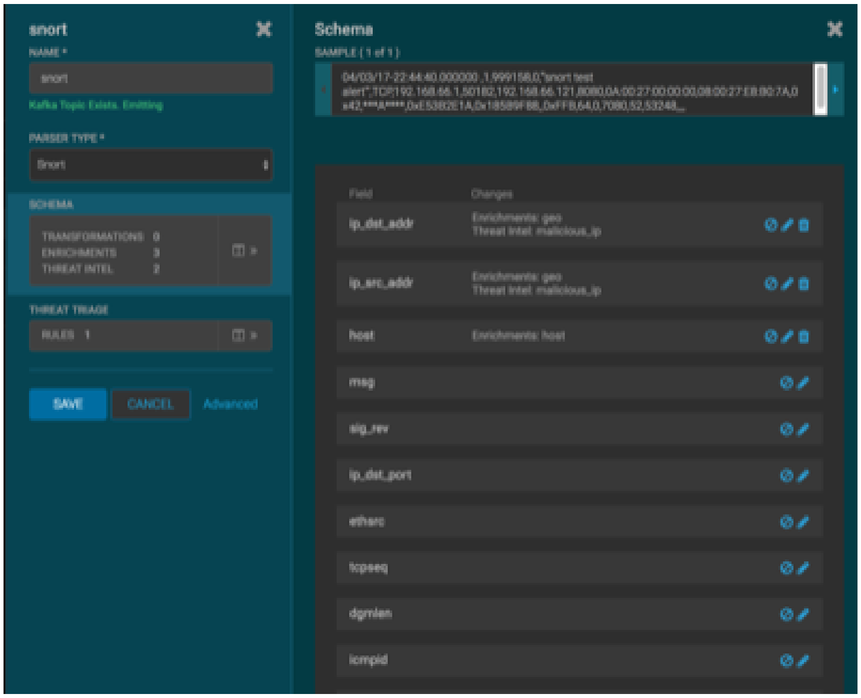
1. Select the new sensor from the list of sensors on the main window.

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,0
x42,***A****,0xE5382E1A,0x185B9F88,,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 intelligence 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.

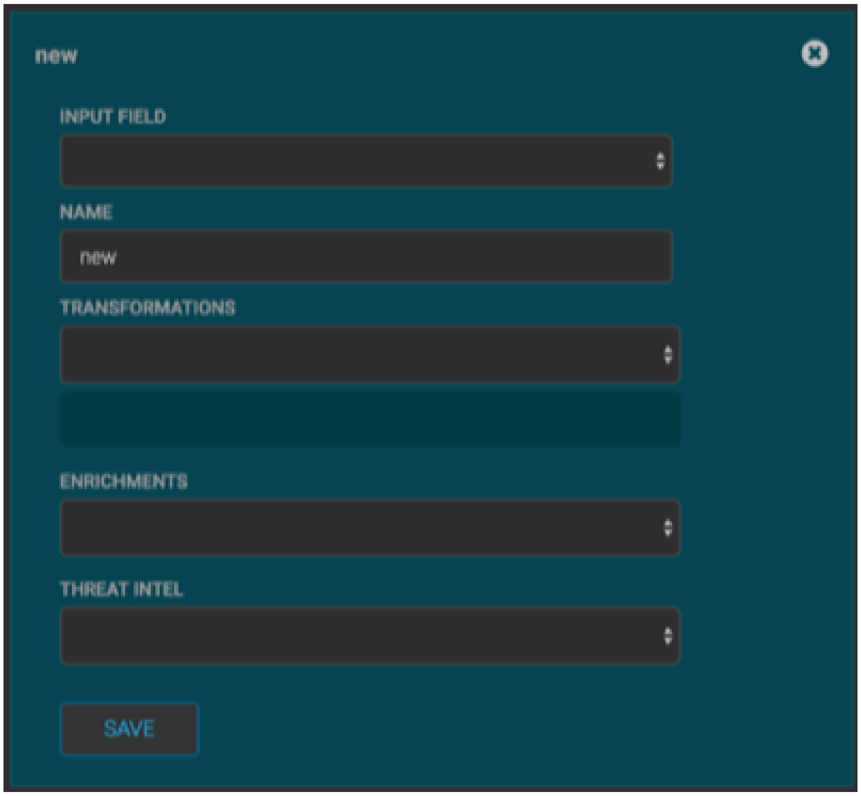
4. You can apply threat intelligence to an existing field or create a new field. Click the ✎ (edit icon) next to a field to apply transformations to that field. Or click

(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 3.9. New Schema Information Panel



NAME

new

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.

3.4.2.6. Mapping Fields to HBase Threat Intel by Using the CLI

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

1. Edit the new data source threat intelligence 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 are incorporated slightly differently than data from a flat CSV file. This section describes how to define a streaming source.

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

1. Define a parser topology 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" : "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/config/zookeeper
```

4. 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:

```
{
  "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/config/zookeeper
```

3.5. Prioritizing Threat Intelligence

Not all threat intelligence indicators are equal. Some require immediate response, while others can be addressed as time and availability permits. As a result, you must 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:

- Performing Threat Triage Using the Management Module [63]
- Uploading the Threat Triage Configuration to ZooKeeper [67]

- Viewing Triaged or Scored Alerts [68]

3.5.1. 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.

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

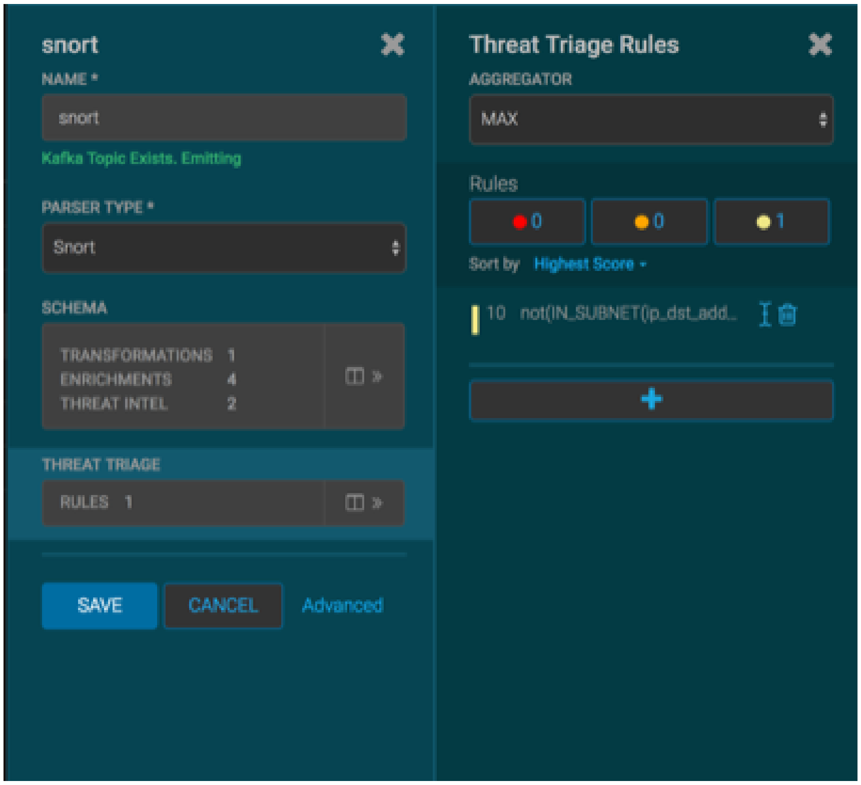
Following are some example rules with example values:

Rule 1	If a threat intelligence enrichment type is alerted, you receive an alert score of 5.
Rule 2	If the URL does not end in .com or .net, you receive an alert score of 10.

To create similar rules, complete the following steps:

1. On the sensor panel, in the Threat Triage field, click ◫ »

Figure 3.10. 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

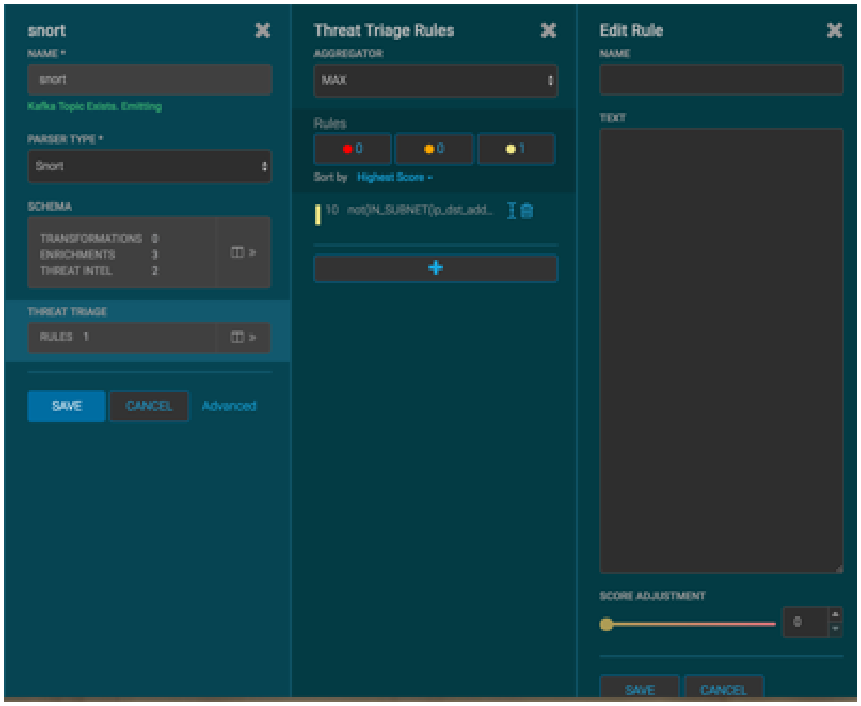
● 0 ● 0 ● 1

Sort by Highest Score

10 not(IN_SUBNET(ip_dst_add...

2. To add a rule, click +.

Figure 3.11. 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 in the **NAME** field.

4. In the **TEXT** field, enter the syntax for the new rule:

```
Exists(IsAlert)
```

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

6. Click **SAVE**

   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 among the following:

   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. If you want to filter the threat triage display, use the **Rules** section and the **Sort by** menu below it.

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, select **Highest Score** from the **Sort by** menu.

9. Click **SAVE**.

3.5.2. Performing Threat Triage Using the CLI

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

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

- Understanding Threat Triage Rule Configuration [65]
- Uploading the Threat Triage Configuration to ZooKeeper [67]
- Viewing Triaged or Scored Alerts [68]

3.5.2.1. Understanding 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, if the value exceeds threshold of value-threshold, then it receives 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

3.5.2.2. Uploading the Threat Triage Configuration to ZooKeeper

To apply 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` to match the configuration on disk:

   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:

   ```
   "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
```

3.5.2.3. Viewing Triaged or Scored Alerts

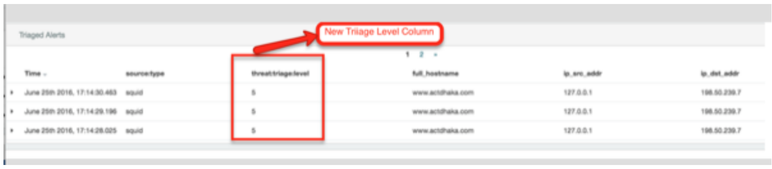
You can view triaged alerts in the indexing topic in Apache Kafka or in the triaged alert panel in the HCP Metron dashboard.

An alert in the indexing topic in Kafka looks 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 shows you an example of a triaged alert panel in the HCP Metron dashboard:

Figure 3.12. Investigation Module Triaged Alert Panel



Triaged Alerts

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

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.

3.6. Setting Up Enrichment Configurations

The enrichment topology is a topology dedicated to performing the following:

- Taking the data from the parsing topologies normalized into the Metron data format (for example, a JSON Map structure with `original_messageand` `timestamp`).

- 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

You can use the sensor-specific configuration to configure the individual enrichments and threat intelligence enrichments for a given sensor type (for example, Snort). The sensor configuration format is a JSON object stored in Apache ZooKeeper.

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. Understanding 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.

The following is an index of the global configuration properties and their associated Apache 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 [88].

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`

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. Creating Global Configurations

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

2. Using the following format, populate the file with enrichment values that you want to apply to all sensors:

```
{
  "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"
                           }
              }
                       ]
}
```

`es.ip`

A single or collection of elastic search master nodes.

They might be specified using the `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`

The format of the date that specifies how the information is parsed time-wise.

For 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.)
- `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.

The `fieldValidations` enrichment value use validation plugins or assertions about fields or whole messages

`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 is 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 configuration.
`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 configuration.
`NOT_EMPTY`	Validates that the fields exist and are not empty (after trimming.)

3.9. Understanding 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 to determine if they need your assistance with customizing the Profiler values.

HCP installs the Profiler which runs as an independent Apache Storm topology. The configuration for the Profiler topology is stored in Apache ZooKeeper at `/metron/topology/profiler`. These properties also exist in the default installation of HCP at `$METRON_HOME/config/zookeeper/profiler.json`. You can change the values on disk and then upload them to ZooKeeper using `$METRON_HOME/bin/zk_load_configs.sh`.

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

Note

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

HCP supports the following profiler properties:

`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. Define this value along with `profiler.period.duration.units`.

`profiler.period.duration.units` The units used to specify the profile period duration. Define this value along with `profiler.period.duration`.

`profiler.ttl` If a message has not been applied to a Profile in this period of time, the Profile is forgotten and its resources cleaned up. Define this value along with `profiler.ttl.units`.

`profiler.ttl.units` The units used to specify `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 Apache 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 written in a single batch.

`profiler.hbase.flush.interval.seconds` The maximum number of seconds between batch writes to HBase.

3.10. Creating an Index Template

To work with a new data source data in the Metron dashboard, you must ensure that the data is sent to the search index (Solr or Elasticsearch) with the correct data types. You achieve this by defining an index template.

Prerequisite

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

Note

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

```
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"
        }
      }
    }
  }
}
```

This example shows 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.
- You can add fields following the five that are already defined.

By default, Elasticsearch attempts to analyze all fields of type `string`. This means that Elasticsearch tokenizes the string and performs 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 for Elasticsearch have the value `not_analyzed`.

2. Delete existing indices to enable updated replacements using the new template:

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

3. Wait for the new data source index to be re-created:

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

   This might take a minute or two based on how fast the new data source data is being consumed in your environment.

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

After Hortonworks Cybersecurity Platform (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.

3.12. Setting up pcap to View Your Raw Data

Because the pcap data source creates an Apache Storm topology that can rapidly ingest raw data directly into HDFS from Apache Kafka, 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, for low-volume packet capture
- The Fastcapa tool, for 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 [76]
- Starting pcap [77]
- Installing Fastcapa [78]
- Using Fastcapa [81]

3.12.1. Setting up pycapa

You can set up pycapa by completing the following steps.

Prerequisite

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 want 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
   ```

3.12.2. 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 in to the host on which you are running Metron.

2. If you are running HCP on an Ambari-managed cluster, perform the following steps; otherwise proceed with 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. If HCP is installed on an Ambari-managed cluster, use the following command to start the pcap topology:

```
su - metron $METRON_HOME/bin/start_pcap_topology.sh
```

4. If HCP is installed by CLI, use the following command to start the pcap topology.

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

5. 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
```

3.12.3. Installing Fastcapa

You can install Fastcapa either automatically or manually. The automated installation is the simplest but it requires 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.

- Installing Fastcapa Automatically [79]
- Installing Fastcapa Manually [79]

3.12.3.1. Requirements for Installing Fastcapa

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

- Linux kernel 2.6.34 or later
- A DPDK supported ethernet device; NIC

- Ports on the Ethernet device that can be dedicated for exclusive use by Fastcapa

3.12.3.2. Installing Fastcapa Automatically

Installing Fastcapa has several steps and involves building Data Plan Development Kit (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.

3.12.3.3. Installing Fastcapa Manually

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 [79]
- Install DPDK [80]
- Install Librdkafka [81]
- Install Fastcapa [81]
- Using Fastcapa in a Kerberized Environment [85]

3.12.3.4. Enable Transparent Huge Pages

The probe performs its own memory management by leveraging Transparent Huge Pages (THP). In Linux, you can use the Transparent Huge Pages to enable either dynamically or automatically upon startup. It is recommended that these be allocated on boot to increase the chance that a larger, physically contiguous chunk of memory can be allocated.

Prerequisite

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. Edit `/etc/default/grub` to add the following book parameters at 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. After the host reboots, 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 should be distributed fairly evenly across each non-uniform memory access (NUMA) node. In the following example, a total of 16 requested THPs are distributed as 8 to each of 2 NUMA nodes:

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

5. After the THPs are reserved, mount them 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
   ```

3.12.3.5. 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. Specify where you want DPDK 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. Specify the PCI address of the Ethernet device that you want to use to capture network packets:

   ```
   $ 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, using a device name and PCI address appropriate to 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:

```
$ 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
```

3.12.3.6. Install Librdkafka

The probe has been tested with Librdkafka 0.9.4.

1. Specify an installation path:

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

   In the example, the libs will actually be installed at `/usr/local/lib`; note that `lib` is appended to the prefix.

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 is on the search path for the runtime shared library loader:

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

3.12.3.7. 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:

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

   The resulting binary is placed at `build/app/fastcapa`.

3.12.4. Using Fastcapa

Follow these steps to run Fastcapa.

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

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

You can view the example configuration file `conf/fastcapa.conf` to learn other useful parameters.

2. If the capture device is not bound, bind it:

   ```
   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 and clear the queue by using`SIGINT` or by typing `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, use `SIGKILL` or enter`killall -9 fastcapa`.

3.12.4.1. Fastcapa Parameters

Fastcapa accepts three sets of parameters.

- Command-line parameters passed directly to the DPDK Environmental Abstraction Layer (EAL).
- Command-line parameters that define how Fastcapa 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.

3.12.4.1.1. 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
```

For more information about EAL parameters, run the following command:

```
fastcapa -h
```

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. 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.	

For more information about Fastcapa-specific parameters, run the following command:.

```
fastcapa -- -h
```

Fastcapa-Kafka Configuration File

You specify the path to the configuration file 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. Place the global configuration values under a `[kafka-global]` header and place topic-specific values under `[kafka-topic]`.

A minimally viable configuration file only needs 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.

Name	*Description*	*Default*
metadata.broker.list	Initial list of brokers as a CSV list of broker host or host:port	NA
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

Name	*Description*	*Default*
message.copy.max.bytes	Maximum size for the message to be copied to buffer. Messages larger than this are 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

Local global configuration values under the `[kafka-global]` header.

Locate topic configuration values under the `[kafka-topic]` header.

	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 for successful delivery. A value of 0 represents infinity.	300000
queue.buffering.max.kbytes	Maximum total message size sum allowed on the producer queue	none

Fastcapa Counters Output

When running the probe, some basic counters are output to `stdout`. During normal operation these values are 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 eight packets.
- `[rx] + in` : The receive workers have consumed eight 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 prevents receive workers from enqueuing additional packets. The excess packets are dropped. This value never decreases.
- `[tx] + in` : The transmission workers consumed 8 packets.

- `[tx] + out` : The transmission workers packaged 8 packets into Kafka messages.
- `[tx] + drops` : If the Kafka client library accepted fewer packets than expected. This value might change as additional packets are acknowledged by the Kafka client library
- `[kaf] + in` : The Kafka client library received 8 packets.
- `[kaf] + out` : A total of 7 packets successfully reached Kafka.
- `[kaf] + queued` : There is 1 packet within the `rdkafka` queue

3.12.5. Using Fastcapa in a Kerberized Environment

You can use the Fastcapa probe in a Kerberized environment. Follow these steps to use Fastcapa with Kerberos.

Prerequisites

The following task assumes that you have configured the following values. If necessary, change these values to match 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. Verify that Librdkafka supports SASL:

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

3. If Librdkafka does not support SASL, install `libsasl` or `libsasl2`. Use the following command to install `libsasl` on your CentOS environment:

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

4. Grant access to your Kafka topic (in this example, 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
```

5. Obtain a Kerberos ticket:

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

6. 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
   ```

7. Run Fastcapa.

3.13. Troubleshooting Parsers

This section provides some troubleshooting solutions for parser issues.

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

1. Ensure that your Grok parser statement is valid.

   a. Log in to HOST $HOST_WITH_ENRICHMENT_TAG as root.

   b. Deploy a new, valid parser topology:

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

   c. Navigate to the Apache Storm UI to validate that the new topology is displayed and has no errors.

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

3. Check your Apache NiFi configuration to ensure that data is flowing between the Kafka topic for your new data source and Hortonworks Cybersecurity Platform (HCP).

3.13.2. 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 Manage

Hortonworks Cybersecurity Platform (HCP) powered by Apache Metron provides you with several options for monitoring and managing your system. Before you perform any of these tasks, you should become familiar with HCP data throughput.

4.1. Understanding Throughput

Data flow for HCP occurs in real-time and involves Apache Kafka files ingesting raw telemetry data; parsing it into a structure that HCP can read; enriching it with asset, geo, and threat intelligence information; and indexing and storing the enriched data. Depending on the type of data streaming into HCP, streaming occurs using Apache NiFi, performance networking ingestion probes, or real-time and batch threat intelligence feed loaders.

1. Apache Kafka ingests information from telemetry data sources trough 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 works for most types of telemetry data sources. See the NiFi documentation for more information.

   Performant network ingestion probes

   This type of streaming works 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 works for real-time and batch threat intelligence feed loaders.

2. After the data is ingested into Kafka files, it 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 Apache ZooKeeper. A Kafka file containing the parser information is created for every telemetry data source.

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

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

4.2. Updating Properties

HCP configuration information is stored in Apache ZooKeeper as a series of JSON files. You can populate your ZooKeeper configurations from multiple locations:

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

Because Ambari explicitly manages some of these configuration properties, if you change a property explicitly managed by Ambari from a mechanism outside of Ambari, such as the Management UI, Ambari is aware of this change and overwrites 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.

Following are the Ambari-managed properties:

Table 4.1. Ambari-Managed Properties

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. Understanding ZooKeeper Configurations

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 the sensor enrichment configuration JSON (for example, `snort.json` or `bro.json`

By default, the sensors directory as deployed by the Ansible infrastructure is located at `$METRON_HOME/config/zookeeper`.

Although the configurations are stored on disk, they must be loaded into ZooKeeper to be used. You can use the utility program `$METRON_HOME/bin/zk_load_config.sh` to load configurations into ZooKeeper.

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

See the following list for examples of usage:

- To dump the existing configs from ZooKeeper on the single-ode 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 Hortonworks Cybersecurity Platform (HCP) Management module or the Apache Storm UI. The following procedures use the HCP Management module to manage sensors. For information about using Storm to manage sensors, see the Storm documentation.

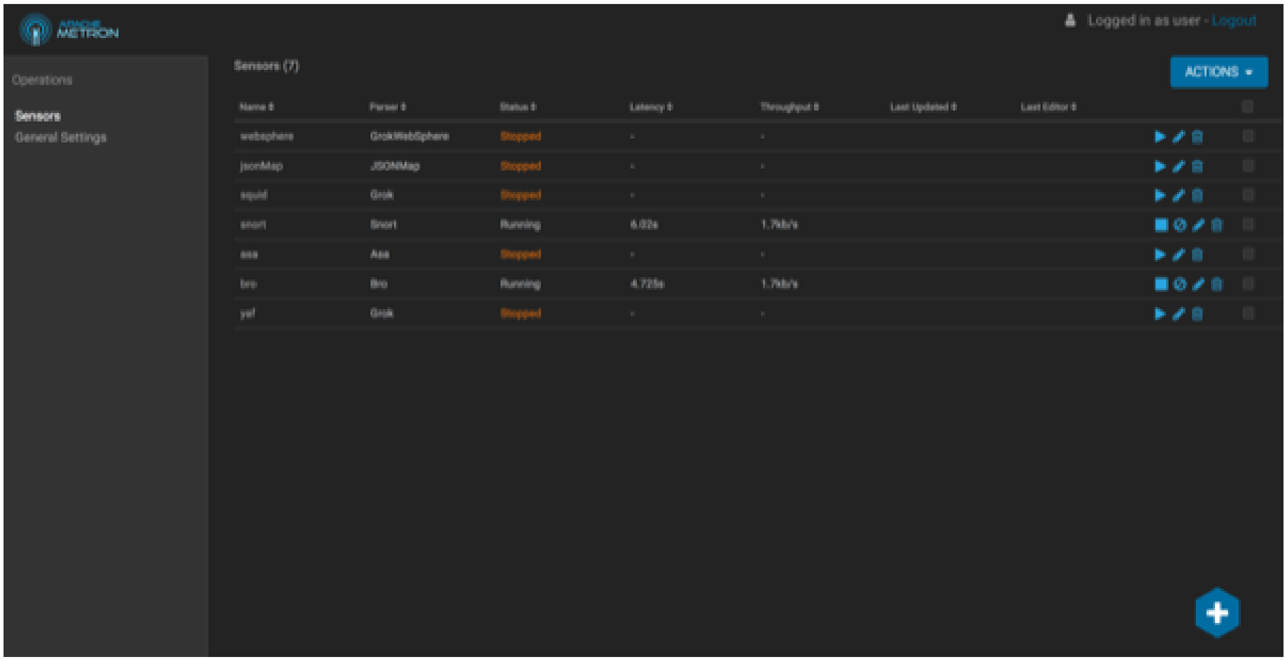
- Starting a Sensor [90]
- Modifying a Sensor [91]
- Deleting a Sensor [93]

4.4.1. Starting a Sensor

After you install a sensor, you can start it using Management module.

1. From the main window, click ▶ (start) in the ■ ⊘ ✎ 🗑 (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	-	-			▶ ✎ 🗑	□

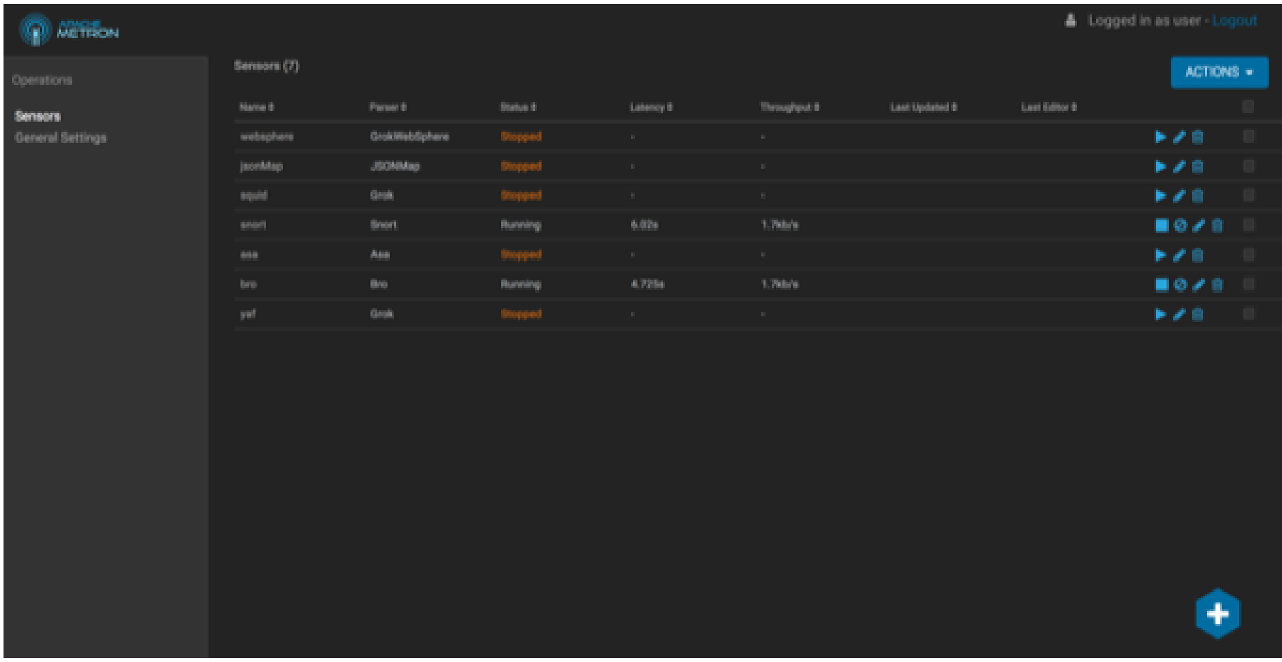
Starting the sensor might take a few minutes. When the operation completes successfully, the Status value for the sensor changes to Running.

4.4.2. Stopping a Sensor

After you install a sensor, you can stop it using Management module.

1. From the main window, click ■ (stop) in ■ ⊘ ✎ 🗑 (tool bar) on the right side of the window.

Figure 4.2. 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	-	-		

Stopping the sensor might take a few minutes. When the operation completes successfully, the Status value for the sensor changes to Stopped

4.4.3. Modifying a Sensor

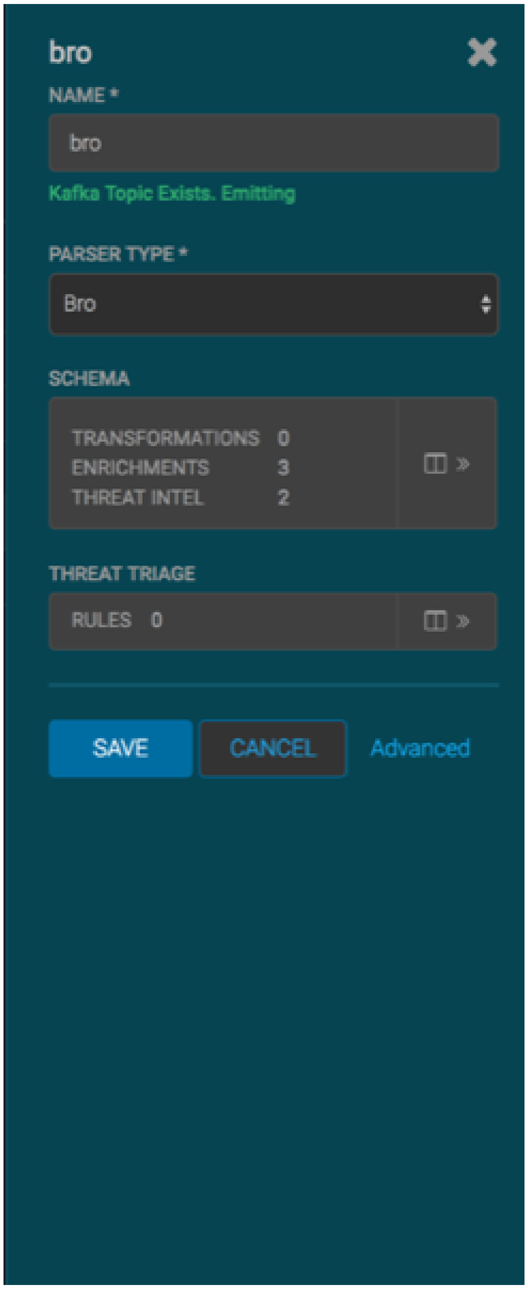
You can modify any sensor listed in Hortonworks Cybersecurity Platform (HCP) Management module.

1. From the **Operations** panel of the main window, select **Sensors**

2. Click ✎ (edit) for the sensor you want to modify.

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

Figure 4.3. Sensor Panel



bro

NAME * bro

Kafka Topic Exists. Emitting

PARSER TYPE * Bro

SCHEMA

TRANSFORMATIONS 0

ENRICHMENTS 3

THREAT INTEL 2

THREAT TRIAGE

RULES 0

3. Modify the following information for the sensor, as necessary:

   - Sensor name
   - Parser type
   - Schema information
   - Threat triage information

4. Click **Save**.

4.4.4. Deleting a Sensor

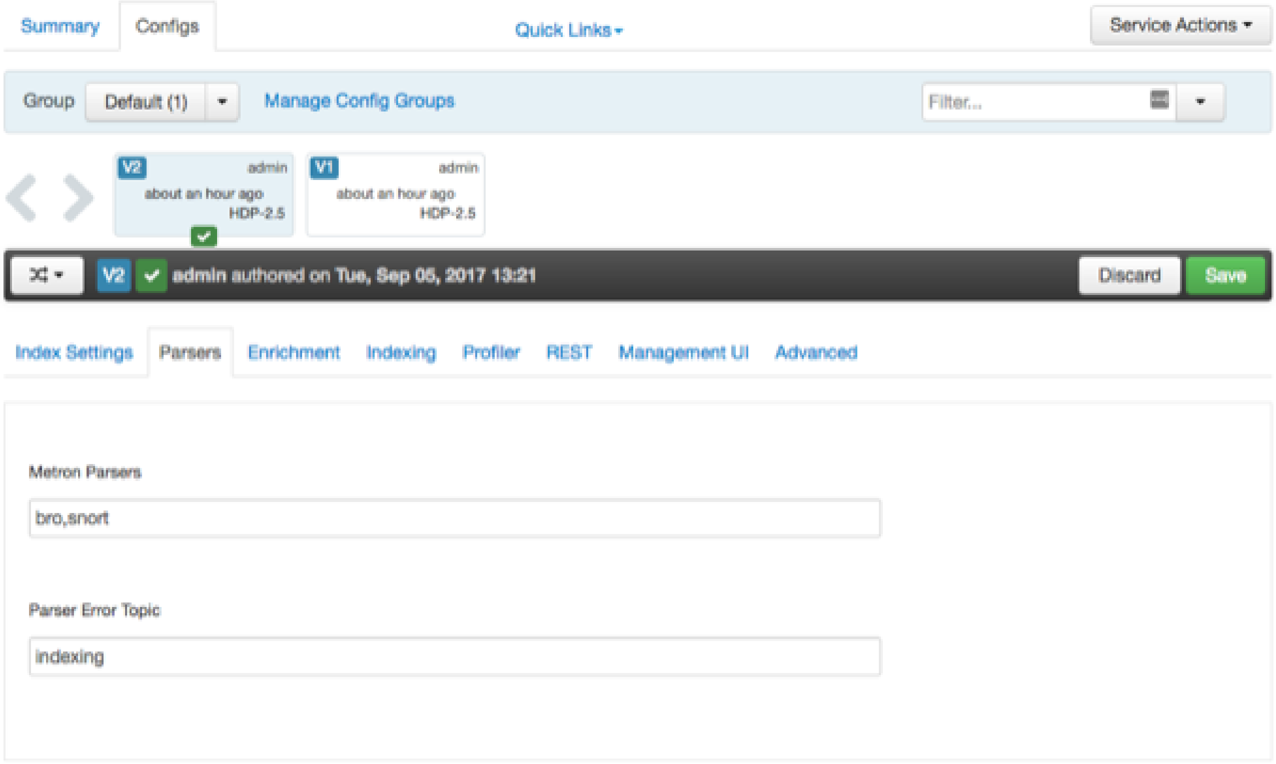
You can delete a sensor if you don't need it.

Prerequisite

You must take the sensor offline before deleting it.

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

Figure 4.4. ambari_configs_parsers.png



Metron Parsers: bro,snort

Parser Error Topic: indexing

4. Delete the name of the parser you want to delete from the **Metron Parsers** field.
5. Display the Management module.
6. Select the check box next to the appropriate sensor in the Sensors table.

   You can delete more than one sensor by clicking multiple check boxes.
7. From the **Actions** menu, select **Delete**.

The Management module deletes the sensor from ZooKeeper.

8. 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 Apache 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.

Metron Dashboard

Displays sensor-specific data that you can use to identify, investigate, and analyze cybersecurity data. This dashboard is described extensively in the HCP User Guide.

Metron Error Dashboard

Displays information on all errors detected by HCP.

4.5.1. Displaying the Metron Error Dashboard

Prior to displaying the Metron Error Dashboard, ensure that you have created an index template. See Creating an Index template for more information.

To display the error dashboard, you must navigate to it from the main Metron dashboard:

1. In the main Metron dashboard, click 📂 (load saved dashboard) in the upper right corner of the Metron dashboard.

2. Select **Metron Error Dashboard** from the list of dashboards.

3. Click (timeframe tab) in the upper right corner of the Metron Error Dashboard and choose the timeframe you want to use.

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

- Exception
- Hostname - The machine on which the error occurred
- Stack trace
- Time - When the error occurred
- Message

- Raw Message - Original message
- Raw_message_bytes - The bytes of the original message
- Hash - Determines if there is a duplicate message
- Source_type - Identifies source sensor
- Error type - Defines the error type; for example parser error

4.5.2. Default Metron Error Dashboard Section Descriptions

Total Error Messages

The total number of error messages received during an interim you specify.

Unique Error Messages

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

Errors Over Time

A **detailed message panel** that 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.

Errors by Error Type

A list of all of the fields associated with a selected index pattern.

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

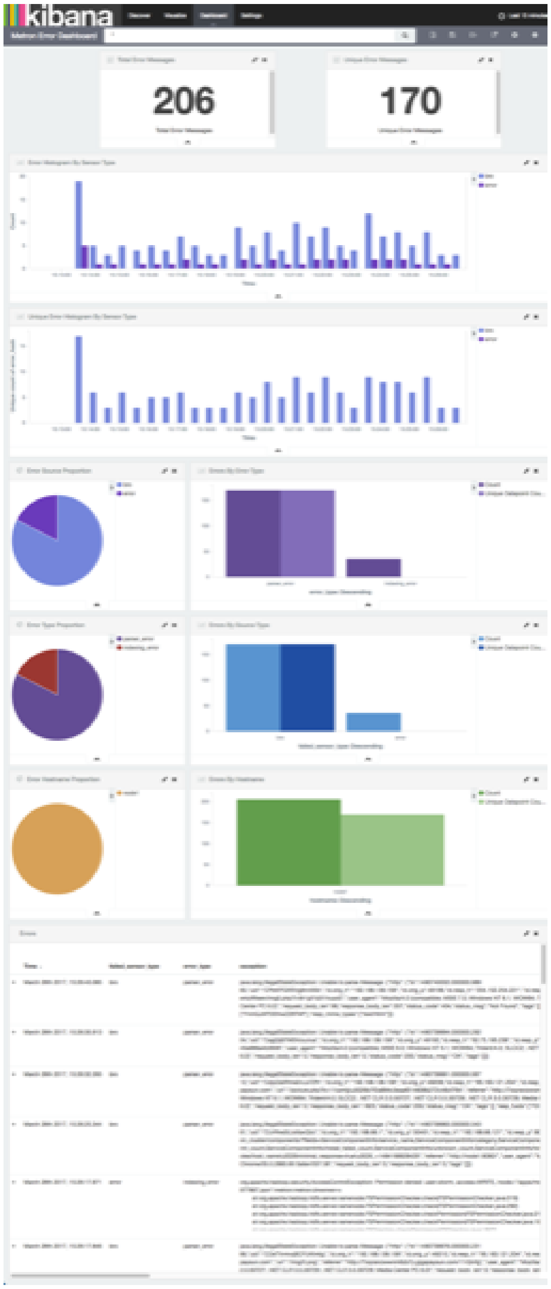
You can use the **mark down widget panel** to provide explanations or instructions for the dashboard.

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 4.5. Error Dashboard



4.5.3. Reloading Metron Templates

Hortonworks Cybersecurity Platform (HCP) provides templates that display the default format for the Metron UI dashboards. You might want to 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.

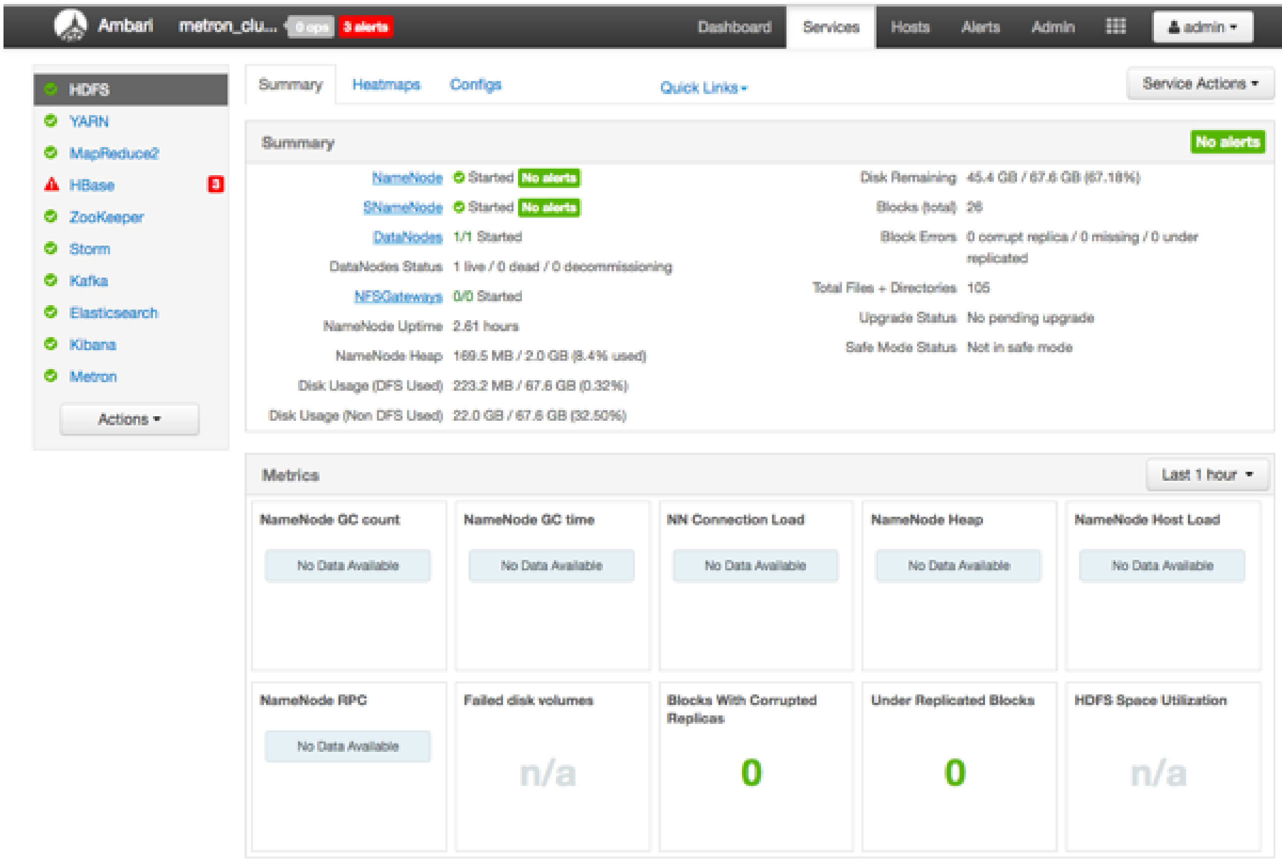
1. From web browser, display the Ambari UI:

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

2. Click the **Services** tab.

3. Select Kibana in the left pane of the window.

Figure 4.6. Ambari Services Tab



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%)

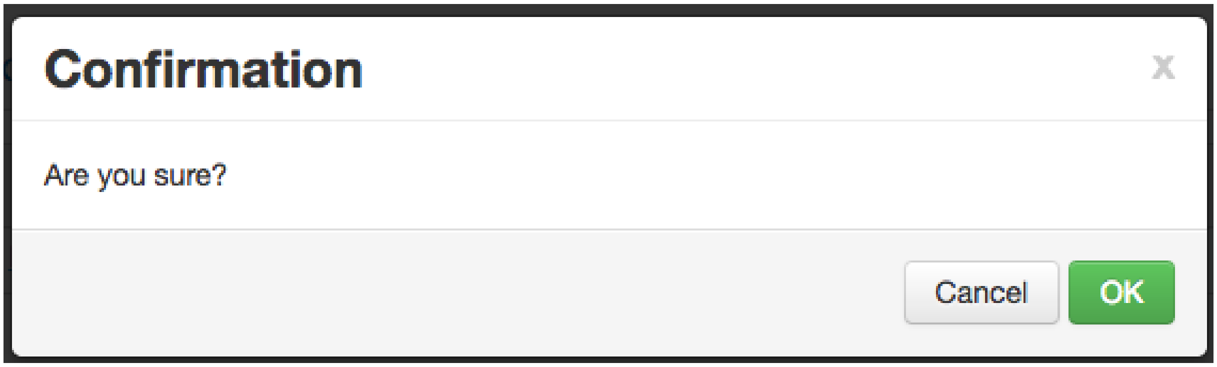
Disk Remaining 45.4 GB / 67.6 GB (67.18%)
Blocks (total) 26
Block Errors 0 corrupt replica / 0 missing / 0 under replicated
Total Files + Directories 105
Upgrade Status No pending upgrade
Safe Mode Status Not in safe mode

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

4. From the **Service Actions** menu, select **Load Template**.

5. In the Confirmation dialog box, click **OK**.

Figure 4.7. Confirmation Dialog Box

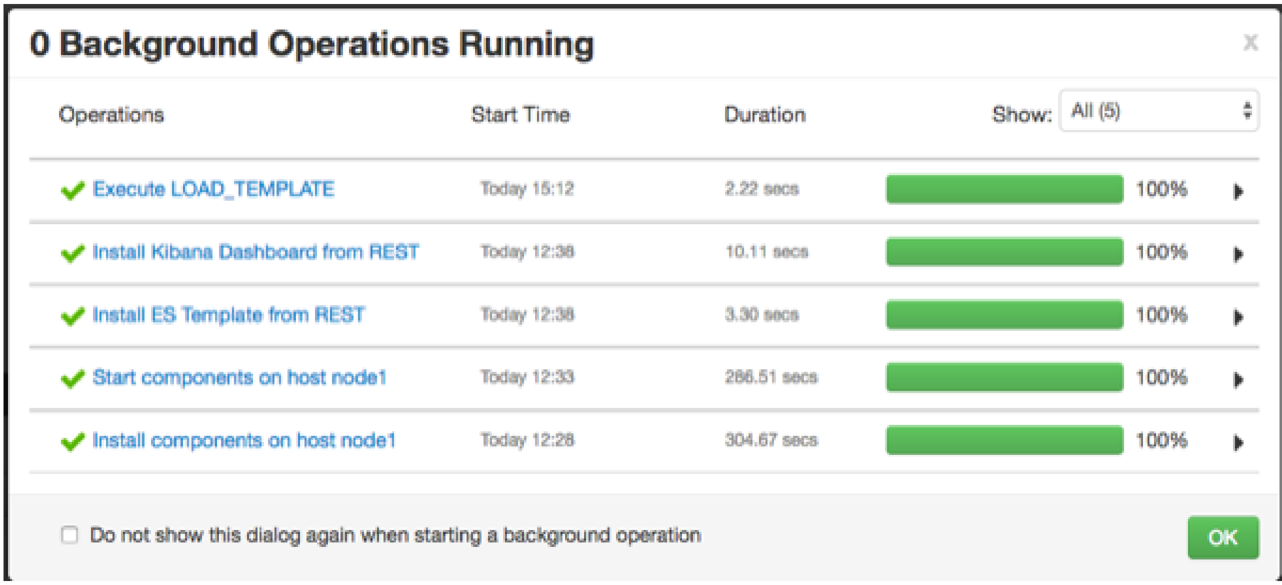


Confirmation

Are you sure?

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

Figure 4.8. Ambari Background Operations



0 Background Operations Running

Operations	Start Time	Duration	Show: All (5)		
✓ 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

6. In the **Background Operation Running** dialog box, click **OK** 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.

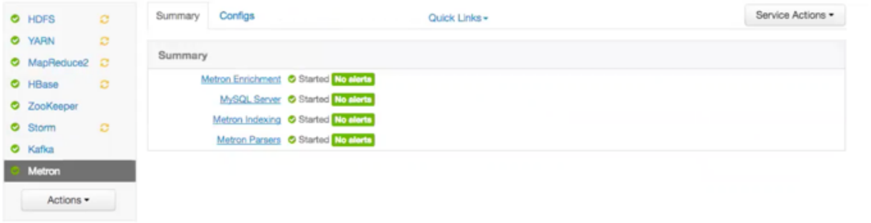
4.6. Starting and Stopping Parsers

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 4.9. Ambari Metron Summary Window

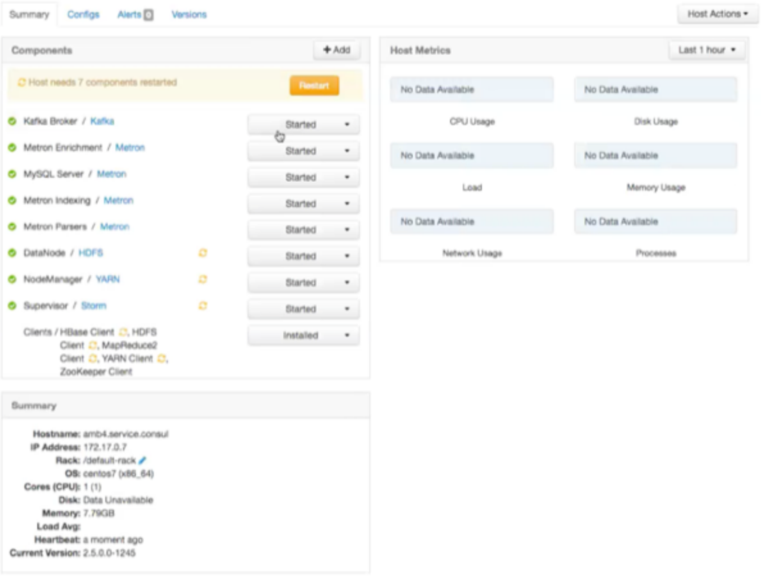
   Summary

   Metron Enrichment ✓ Started No alerts
   MySQL Server ✓ Started No alerts
   Metron Indexing ✓ Started No alerts
   Metron Parsers ✓ Started No alerts

2. Click **Metron Parsers** to display the **Components** window.

   The Components window displays a list of Metron hosts and which components reside on each host.

Figure 4.10. 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 **Started/Stopped** to change the status of the Parsers; then click **Confirmation**.
4. In the **Background Operation Running** dialog box, click **Stop Metron Parsers**.
5. In the **Stop Metron Parsers** dialog box, click the entry for your Metron cluster; then click **Metron Parser Stop**.

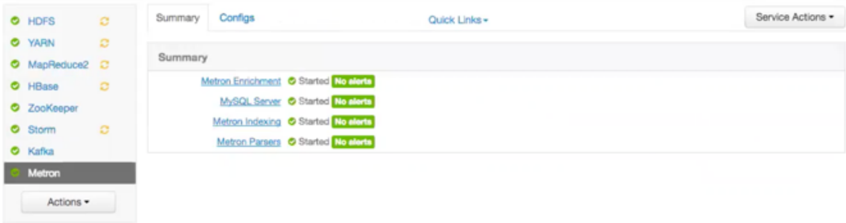
   Ambari displays a dialog box for your Metron cluster which lists the actions as is stops the parsers.

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.

1. Display the Ambari tool and navigate to **Services** > **Metron** > **Summary**.

   Figure 4.11. Ambari Metron Summary Window

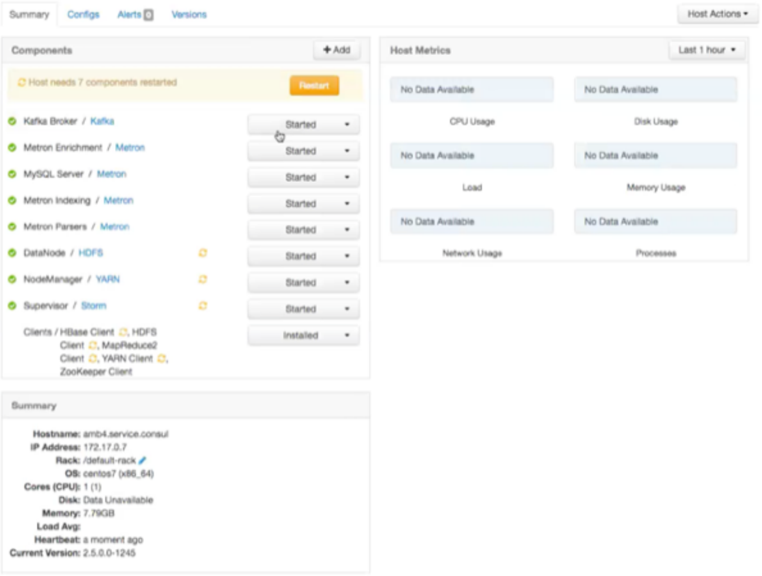


Metron Enrichment	Started	No alerts
MySQL Server	Started	No alerts
Metron Indexing	Started	No alerts
Metron Parsers	Started	No alerts

2. Click **Metron Enrichments** to display the **Components** window.

This window displays a list of HCP hosts and which components reside on each host.

Figure 4.12. 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 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.

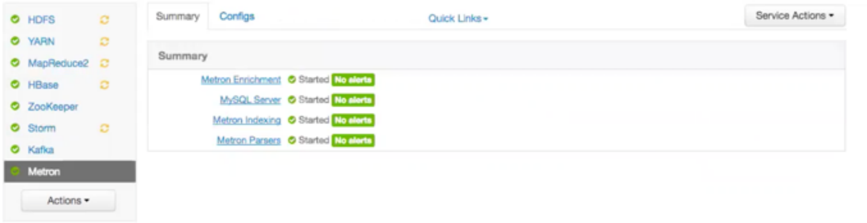
4.8. Starting and Stopping Indexing

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 4.13. Ambari Metron Summary Window

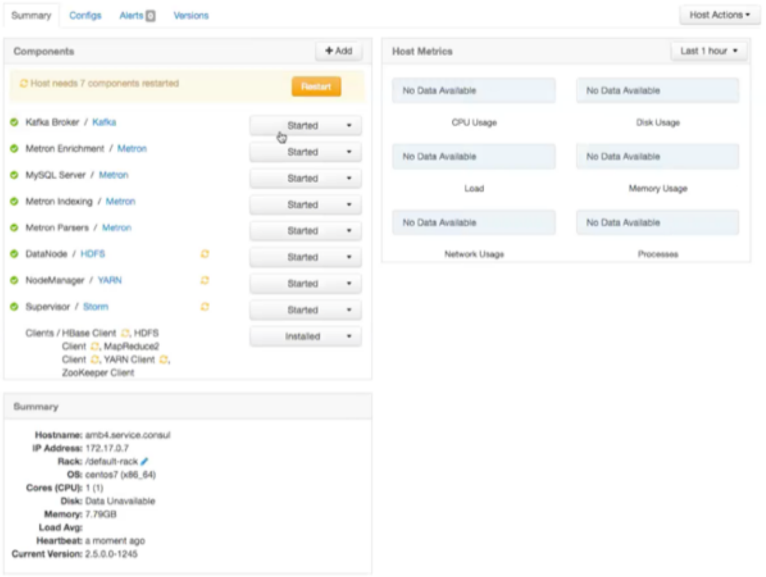


Metron Enrichment ✓ Started No alerts
MySQL Server ✓ Started No alerts
Metron Indexing ✓ Started No alerts
Metron Parsers ✓ Started No alerts

2. Click **Metron Indexing**.

   This window displays a list of HCP hosts and which components reside on each host.

Figure 4.14. 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 **Started/Stopped** by **Metron Indexing** to change the status of the Indexing.

   Ambari displays the **Background Operation Running** dialog box.

4. Click the **Confirmation** button to verify that you want to start or stop the indexing.

5. Click **Stop Metron Indexing**.

   Ambari displays the **Stop Metron Indexing** dialog box.

6. 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. 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.

1. Use the following command to prune the Elasticsearch data:

   The following is a sample invocation that you can configure through Cron to prune indexes based on the 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` value causes Curator to look for a `timestring` value within the index or snapshot name, and to convert that into an epoch timestamp (epoch implies UTC).

2. For finer-grained control over indexes pruning, provide multiple filters as an array of JSON objects to `filter_list`. Chaining multiple filters implies logical AND.

   ```
   --filter_list '[{"filtertype":"age","source":"creation_date",
   "direction":"older","unit":"days","unit_count":13},
   {"filtertype":"pattern","kind":"prefix","value":"logstash"}]'
   ```

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`. Chaining multiple filters implies 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"}]'
```

4.10. Tuning Apache Solr

You can customize and tune Apache Solr to reflect your environment and improve its performance. See Apache Solr Reference Guide for more information.

4.11. Backing Up the Metron Dashboard

You can back up your Metron dashboard to avoid losing your customizations.

For example:

```
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, you can write the Kibana dashboard to Solr or Elasticsearch.

For example:

```
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 that this overwrites the `.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 about the following:

5.1. Understanding 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 Apache Kafka and into the enrichments topology in Apache Storm. Errors are collected with the context of the error (for example, stacktrace) and the original message causing the error and are 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
- General purpose

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 consists of the following attributes:

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.

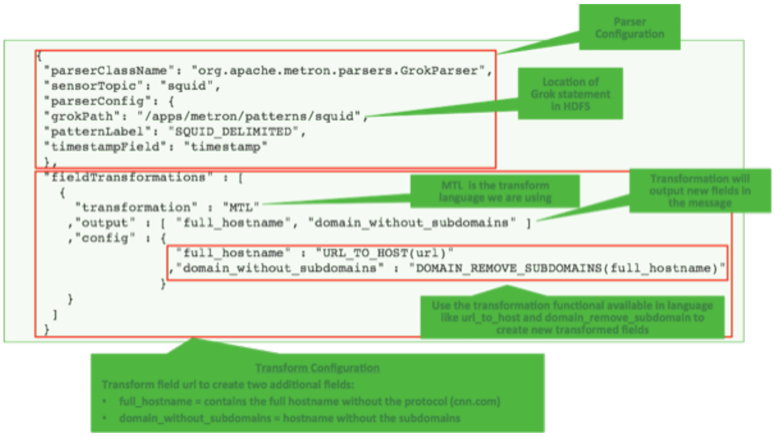
The fieldTransformations is a complex object which defines a transformation that can be done to a message. This transformation can perform the following:

- 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. Example: 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. Enrichment Framework

Enrichments add 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:

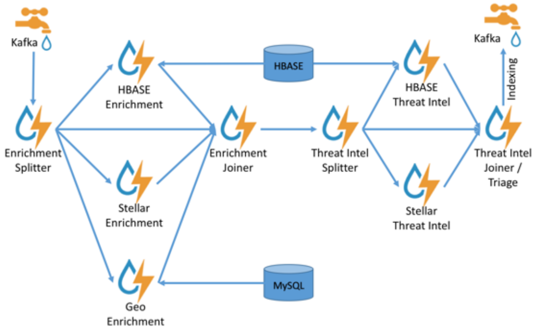
- Enriches messages with external data from data stores by adding new information based on existing fields in the messages
- Marks messages as threats based on data in external data stores
- Marks threat alerts with a numeric triage level based on a set of Stellar rules

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 5.2. HCP Enrichment Flow



5.2.1. Sensor Enrichment Configuration

The sensor enrichment configuration provides enrichments for a given sensor (for example, Snort). The sensor enrichment configuration includes 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.

5.2.1.1. 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 or 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, you must specify the mapping between fields and the enrichment types associated with those fields. You must also specify the enrichment type for the HBase key.	`"fieldToTypeMap" : { "ip_src_addr" : [ "asset_enrichment" ] }`
`fieldMap`	The map of enrichment bolts names to configuration handlers that know how to divide the message. The simplest map is just a list of fields. More complex maps consist of 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 of the enrichment.	`"config": {"typeToColumnFamily": { "asset_enrichment" : "cf" } }`

The `config` map contains 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 message is divided and sent to the enrichment adapter bolts. The simplest `fieldMapcontents` provides a simple list, such as the following

```
"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 configuration, both `ip_src_addr` and `ip_dst_addr` go to the `geo`, `host`, and `hbaseEnrichment` adapter bolts.

5.2.1.2. Stellar Enrichments

Individual sensor enrichments are sufficient for the `geo`, `host`, and `hbaseEnrichment`, sensor topologies However, more complex enrichments might contain their own configuration. Currently, the `stellar` enrichment is more adaptable and thus requires a more nuanced configuration.

Consider the basic example of taking a message and applying a couple of enrichments, such as converting the `hostname` field to lowercase. For this conversion, you must specify the transformation inside of the `config` file for the `stellar` fieldMap option. Two syntaxes are supported, specifying the transformations as a map with the key as the field and the value as the Stellar expression:

```
"fieldMap": {
       ...
      "stellar" : {
        "config" : {
          "hostname" : "To_LOWER(hostname)"
        }
      }
    }
```

Another approach is to make the transformations a list with the same `var := expr` syntax used in the Stellar REPL:

```
"fieldMap": {
       ...
      "stellar" : {
        "config" : [
          "hostname := TO_LOWER(hostname)"
        ]
      }
    }
```

Sometimes arbitrary Stellar enrichments running in sequence run so slowly that you want to group them and run them in parallel: for instance, performing an HBase enrichment and a profiler call

:

```
 "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"
          ]
        }
      }
    }
```

In the previous example, a group called `malicious_domain_enrichment` determines whether the destination address exists in the HBase enrichment table in the `malicious_domains` enrichment type. Because this is a simple enrichment, the group is expressed 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` that interacts 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, you must use the list construct when specifying the group and 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'`

5.2.1.3. 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 enrichment, you must specify the	`"fieldToTypeMap" : {`
		`"ip_src_addr" : [`

Field	Description	Example
	mapping between fields and the enrichment types associated with those fields. You must also specify the enrichment type for the HBase key.	`"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 houses 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"
    }
  }
}
```

5.2.1.4. 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 the sensor enrichment configuration held in ZooKeeper. This configuration fits within the threatIntel section of the configuration as follows:

```
{
  ...
  ,"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

5.2.2. 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 Understanding Global Configuration [70].

5.2.3. 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 (using `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” 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.

5.2.4. 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, transform the timestamp to an epoch timestamp given a timezone that you can 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

For a list of Stellar transformation functions supported by HCP, see https://docs.hortonworks.com/HDPDocuments/HCP1/HCP-1.4.2/bk_stellar-quick-ref/content/index.html.

5.2.5. 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 configurations 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
  ```

5.3. Fastcapa

The Fastcapa 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 receive queues. The total number and size of these receive queues are limited by the Ethernet device in use. More capable Ethernet devices 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, is responsible for fetching packets from the receive queues. There can be only 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.