

## Adding a New Telemetry Source

**Date of Publish:** 2018-12-21



# Contents

<b>Prerequisites to Adding a New Telemetry Data Source.....</b>	<b>3</b>
<b>Creating Parsers.....</b>	<b>4</b>
Create a Parser for Your New Data Source by Using the Management UI.....	4
Create a Parser for Your New Data Source by Using the CLI.....	8
Create Multiple Parsers on One Topology.....	14
Chain Parsers.....	14
Tune Parser Storm Parameters by Using the Management UI.....	20
Telemetry Data Source Parsers Bundled with HCP.....	20
Snort.....	20
Cisco Adaptive Security Appliance.....	21
Bro.....	21
ArcSight CEF.....	21
FireEye.....	21
YAF (NetFlow).....	21
Indexing.....	22
pcap.....	22
<b>Configuring Indexing.....</b>	<b>22</b>
Understanding Indexing.....	23
Default Configuration.....	23
Solr.....	24
Create a New Solr Index Collection.....	24
Elasticsearch.....	25
Create a New Elasticsearch Index Template.....	25
Upgrading to Elasticsearch 5.6.2.....	25
Add X-Pack Extension to Elasticsearch.....	28
HDFS.....	30
Index HDFS Tuning.....	30
Turn Off HDFS Writer.....	30
Troubleshooting Indexing.....	31
Understanding Global Configuration.....	31
Create Global Configurations.....	32
Verify That Events Are Indexed.....	35
<b>Streaming Data.....</b>	<b>36</b>
Stream Data Using NiFi.....	36

## Prerequisites to Adding a New Telemetry Data Source

Part of customizing your Hortonworks Cybersecurity Platform (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.

Before you add a new telemetry data source, you must ensure that your system set up meets the Hortonworks Cybersecurity Platform (HCP) requirements.

- 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-1.6.1.0)

## Creating Parsers

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 three types of parsers:

**Built-in**

HCP features several built-in parsers that support many common security devices.

**General Purpose**

HCP supports three general purpose parsers: Grok, CSV, and JSON map.

- Grok - Regular expression-based parser extracts HCP values; ideal for ingesting structured or semi-structured logs that are well understood and telemetries with lower volumes of traffic
- CSV - Maps CSV columns to HCP events
- JSON Map - Maps JSON documents into HCP events

**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

## Create a Parser for Your New Data Source by Using the Management UI

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 of creating parsers by viewing an example using the general-purpose parser Grok.

**Procedure**

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 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.

3. Launch the HCP Management module from \$METRON\_MANAGEMENT\_UI\_HOST:4200, or follow these steps:
- From the Ambari Dashboard, click **Metron**.
  - Select the **Quick Links**.
  - Select **Metron Management UI**.
4. Launch the Management UI.
5. Under Operations, click **Sensors**.
6. Click




to view the new sensor panel:







**NAME \***

  
**PARSER TYPE \***



Grok 

**GROK STATEMENT**

    
**SCHEMA**


TRANSFORMATIONS	0	 
ENRICHMENTS	0	
THREAT INTEL	0	

**THREAT TRIAGE**

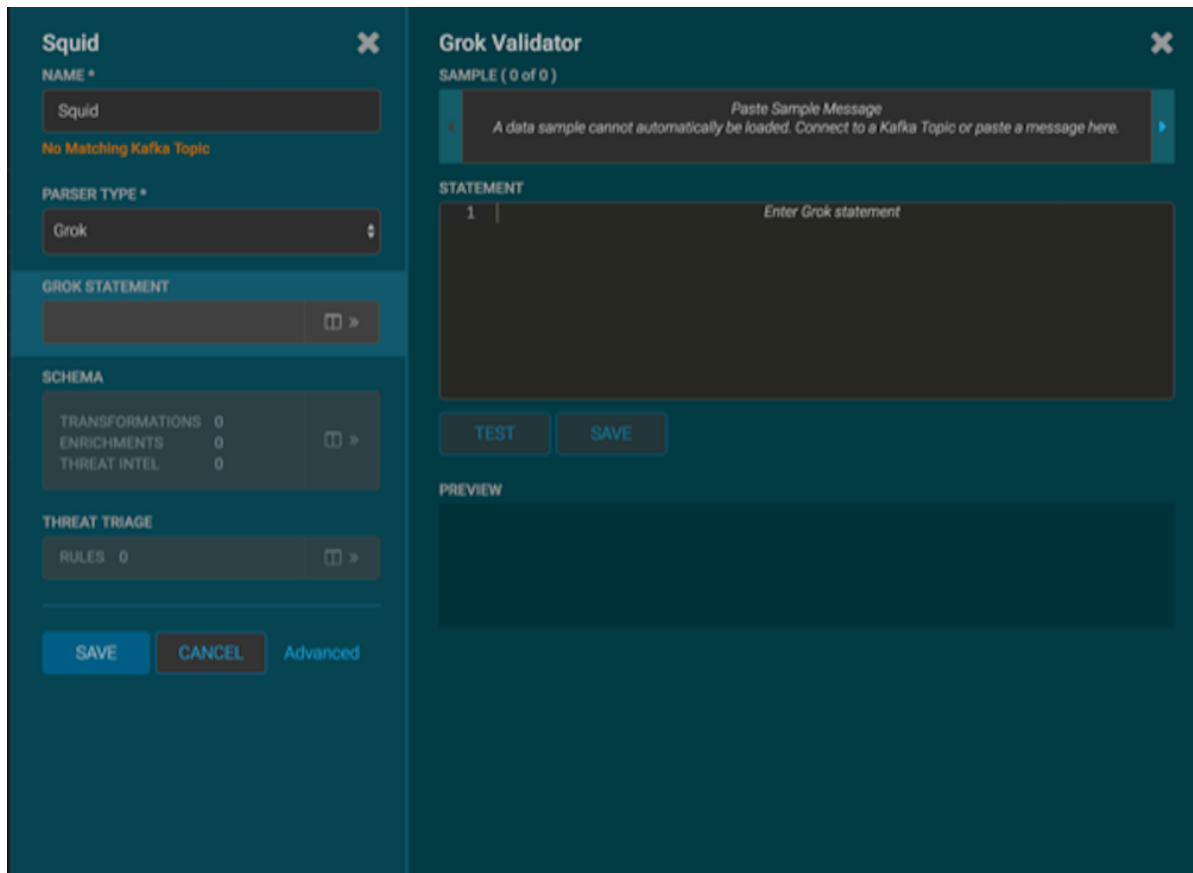
RULES	0	 
-------	---	---

---

**SAVE** **CANCEL** **Advanced**

7. In the **NAME** field, enter the name of the new sensor.
8. In the **Kafka Type** field, enter the name of the new sensor.
9. In the **Parser Type** field, choose the type of parser for the new sensor (in this example task, Grok).  
Don't worry if you see "No Matching Kafka Topic." The Kafka topic will be created automatically when you save.
10. Enter a Grok statement for the new parser:
  - a) In the Grok Statement box, click
 

(expand window) to display the Grok validator panel:



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



**Note:** You must include timestamp to ensure that the system uses the event time rather than the system time.

- d) Click **TEST**.

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.

- e) Click **SAVE** to save the sensor information and add it to the list of sensors.

11. Click the pencil icon to edit the sensor you just added.

12. Scroll down to the **Parser Config** section.

13. In the first open field, indicated by **enter field**, enter timestampField.

PARSER CONFIG

grokPath

/apps/metron/patterns/mysquid

patternLabel

MYSQUID

enter field

SAVE CANCEL

14. In next open field, enter timestamp.

15. Click **Save**.

16. Continue to build and test the Grok statement until you have entries for each element in the log entry.

### Results

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.

## 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.

### Procedure

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.



**Note:** You must include timestamp to ensure that the system uses the event time rather than the system time. For information about setting the grok parser to use the current year, see step 5c.

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 3:

```
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",
  "filterClassName": null,
  "sensorTopic": "$DATASOURCE",
  "outputTopic": null,
  "errorTopic": null,
  "readMetadata": true,
  "mergeMetadata": true,
  "numWorkers": null,
}
```

```

"numAckers": null,
"spoutParallelism": 1,
"spoutNumTasks": 1,
"parserParallelism": 1,
"parserNumTasks": 1,
"errorWriterParallelism": 1,
"errorWriterNumTasks": 1,
"spoutConfig": {},
"securityProtocol": null,
"stormConfig": {},
"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.MessageFilter<JSONObject>` interface. Message filters enable you to ignore a set of messages by using custom logic. The existing implementation is STELLAR. The Stellar implementation 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. If the topic is prefixed and suffixed by `/` then it is assumed to be a regex and will match any topic matching the pattern (for example, `/bro.*/` matches `bro_cust0`, `bro_cust1` and `bro_cust2`).

<b>readMetadata</b>	<p>A Boolean indicating whether to read metadata and make it available to field transformations (false by default).</p> <p>There are two types of metadata supported in HCP:</p> <ul style="list-style-type: none"><li>• Environmental metadata about the whole system</li></ul> <p>For example, if you have multiple Kafka topics being processed by one parser, you might want to tag the messages with the Kafka topic.</p> <ul style="list-style-type: none"><li>• Custom metadata from an individual telemetry source that you might want to use within Metron</li></ul>
<b>mergeMetadata</b>	<p>A Boolean indicating whether to merge metadata with the message (false by default).</p> <p>If this property is set to true, then every metadata field becomes part of the messages and, consequently, is also available for field transformations.</p>
<b>numWorkers</b>	<p>The number of workers to use in the topology (default is the storm default of 1).</p>
<b>numAckers</b>	<p>The number of acker executors to use in the topology (default is the Storm default of 1).</p>
<b>spoutParallelism</b>	<p>The Kafka spout parallelism (default to 1). You can override the default on the command line and if there are multiple sensors they should be in a comma separated list in the same order as the sensors.</p>
<b>spoutNumTasks</b>	<p>The number of tasks for the spout (default to 1). You can override the default on the command line, and if there are multiple sensors they should be in a comma separated list in the same order as the sensors.</p>
<b>parserParallelism</b>	<p>The parser bolt parallelism (default to 1). This can be overridden on the command line , and if there are multiple sensors should be in a comma separated list in the same order as the sensors.</p>
<b>parserNumTasks</b>	<p>The number of tasks for the parser bolt (default to 1). If there are multiple sensors, the last one's configuration will be used. This can be overridden on the command line.</p>
<b>errorWriterParallelism</b>	<p>The error writer bolt parallelism (default to 1). This can be overridden on the command line.</p>
<b>errorWriterNumTasks</b>	<p>The number of tasks for the error writer bolt (default to 1). This can be overridden on the command line.</p>
<b>spoutConfig</b>	<p>A map representing a custom spout configuration (this is a map). If there are multiple sensors, the configs will be merged with the last specified taking</p>

	precedence. This can be overridden on the command line.
<b>securityProtocol</b>	The security protocol to use for reading from Kafka (this is a string). This can be overridden on the command line and also specified in the spout configuration via the security.protocol key. If both are specified, then they are merged and the CLI will take precedence. If multiple sensors are used, any non "PLAINTEXT" value will be used.
<b>stormConfig</b>	The storm configuration to use (this is a map). This can be overridden on the command line. If both are specified, they are merged with CLI properties taking precedence.
<b>cacheConfig</b>	Cache config for stellar field transformations. This configures a least frequently used cache. This is a map with the following keys. If not explicitly configured (the default), then no cache will be used. <ul style="list-style-type: none"> <li>• stellar.cache.maxSize - The maximum number of elements in the cache. Default is to not use a cache.</li> <li>• stellar.cache.maxTimeRetain - The maximum amount of time an element is kept in the cache (in minutes). Default is to not use a cache.</li> </ul>
<b>grokPath</b>	The path for the Grok statement.
<b>patternLabel</b>	The top-level pattern of the Grok file.
<b>parserConfig</b>	<p>A JSON map defining the parser implementation configuration.</p> <p>This configuration file also includes batch sizing and timeout settings for writer configuration. If you do not define these properties, the system uses their default values.</p> <ul style="list-style-type: none"> <li>• batchSize - Number of records to batch together before sending to the writer. Default is 15.</li> <li>• batchSize - Optional. The timeout after which a batch is flushed even if the batchSize is not met.</li> </ul> <pre> "parserConfig" {   "batchSize": 15,   "batchTimeout" : 0 }, </pre>
<b>fieldTransformations</b>	<p>In addition, you can override settings for the kafka writer within the parserConfig file.</p> <p>An array of complex objects representing the transformations to be performed on the message</p>

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.

- c) If you want to set the grok parser to use the current year in its timestamp, add the following information to the transformations function in the datasource json file:

```
"fieldTransformations" : [
  {
    "transformation" : "STELLAR"
    ,"output" : [ "timestamp" ]
    ,"config" : {
      "timestamp": "TO_EPOCH_TIMESTAMP(FORMAT('%s %d',
timestamp_str , YEAR()), 'MMM dd HH:mm:ss:yyyy')"
```

For example, the datasource json file would change to:

```
"fieldTransformations" : [
  {
    "transformation" : "STELLAR"
    ,"output" : [ "full_hostname", "domain_without_subdomains" ,
"timestamp" ]
    ,"config" : {
      "full_hostname" : "URL_TO_HOST(url)"
      ,"domain_without_subdomains" :
      ,"timestamp": "TO_EPOCH_TIMESTAMP(FORMAT('%s %d',
timestamp_str , YEAR()), 'MMM dd HH:mm:ss:yyyy')"
```

"DOMAIN\_REMOVE\_SUBDOMAINS(full\_hostname)"

- d) 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
```

## 6. Deploy the new parser topology to the cluster:

If you want to deploy multiple parsers on one topology, refer to *Creating Multiple Parsers on One Topology*.

- 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.

## Create Multiple Parsers on One Topology

You can specify multiple parsers to run on one aggregated Storm topology to conserve resources. However, for performance reasons, you should group multiple parsers that have similar velocity or data flow and perform functions with similar complexity.

### Procedure

1. Use the CLI to create multiple parsers that you want to specify on a single Storm topology.

Refer to *Create a Parser for Your New Data Source by Using the CLI*.

2. Deploy the new parser topologies to the cluster:

- a) Log in to the host that has Metron installed as root user.
- b) Deploy the new parsers you want to specify onto one topology:

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



**Note:** If your parser name contains a hyphen, you must enclose the parser name in single quotes ('). If you do not enclose a hyphenated parser name in single quotes, Ambari will assume each word or character in the hyphenated parser name is a separate parser. For example, Ambari interprets `sapower-windows-x-json,bro` as seven parsers instead of one. Even the hyphen is considered a parser.

For clarity and consistency, we recommend enclosing all parser names in single quotes when you deploy the new parsers onto a topology.

For example:

```
$METRON_HOME/bin/start_parser_topology.sh -z $ZOOKEEPER_HOST:2181 -s
'bro-sourcel','yaf'
```

- c) If you want to override parser parameters, you can add the parameter and its value to the deployment command.

For a list of parser parameters, see *Create a Parser for Your New Data Source by Using the CLI*.

For example:

```
$METRON_HOME/bin/start_parser_topology.sh -z $ZOOKEEPER_HOST:2181
-s 'bro-sourcel','yaf' -spoutNumTasks 2,3 -parserParallelism 2 -
parserNumTasks 5
```

This command will create a topology with the following parameters:

- Bro - spout number of tasks = 2
- YAF - spout number of tasks = 3
- YAF - parser parallelism = 2
- YAF - parser number of tasks = 5

- d) 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 each \$DATASOURCE Kafka topic that you created earlier and then parses the event with the HCP Grok framework using the Grok pattern defined earlier.

## Chain Parsers

Many sensors contain metadata that should be ingested along with the data or contain different sensor types that need to be parsed separately. You can chain multiple parsers for a sensor to individually address the different types of information in the sensor. For example, you can parse multiple components in a Syslog log file such as timestamp,

message type, and message payload, to differentiate the information contained in the log file. To chain parsers, you need an enveloping parser and sub-parsers for one or more sensor types. For ease of explanation, the following steps use the Grok parser format example provided in Step 1c.

## Procedure

1. Before editing configurations, pull the configurations from ZooKeeper locally:

```
$METRON_HOME/bin/zk_load_configs.sh --mode PULL -z $ZOOKEEPER -o
$METRON_HOME/config/zookeeper/ -f
```

For ease of explanation, steps in this topic use the Grok parser format example provided in Step 2c.

2. Determine the format of the new data source's log entries, so you can parse them.
3. Create a statement that defines the pattern of the parser expression for the log type for your enveloping parser. For ease of explanation, we assume that we are using a Grok topology. Refer to the Grok documentation for additional details.
4. Save the Grok statement 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/$ENVELOPE_DATASOURCE
```

- b) Open \$ENVELOPE\_DATASOURCE and add the Grok statement defined in Step 3:

```
vi /tmp/$ENVELOPE_DATASOURCE
```

- c) Put the \$ENVELOPE\_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/$ENVELOPE_DATASOURCE
hdfs dfs -put /tmp/$ENVELOPE_DATASOURCE /apps/metron/patterns/
```

5. Define the enveloping parser configuration.

- a) As root, log into the host with HCP installed:

```
ssh $HCP_HOST
```

- b) Create a \$DATASOURCE envelope parser configuration file at \$METRON\_HOME/config/zookeeper/parsers/\$ENVELOPE\_DATASOURCE.json:

```
/usr/hdp/current/kafka-broker/bin/kafka-topics.sh --zookeeper $ZOOKEEPER
--create --topic $ENVELOPE_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
--list
```

- d) Populate the \$ENVELOPE\_PARSER Kafka topic with the following:

```
{
  "parserClassName": "org.apache.metron.parsers.GrokParser",
  "sensorTopic": "$ENVELOPE_DATASOURCE",
  "parserConfig": {
    "grokPath": "/apps/metron/patterns/$ENVELOPE_DATASOURCE",
    "batchSize": 1,
    "patternLabel": "$DATASOURCE_DELIMITED",
    "timestampField": "timestamp"
  }
}
```

```

    "timeFields" : [ "timestamp" ],
    "dateFormat" : "MMM dd yyyy HH:mm:ss",
    "kafka.topicField" : "logical_source_type"
  }

```

The important parameters to set for this parser are the following:

<b>parserClassName</b>	The name of the parser's class in the .jar file.
<b>sensorTopic</b>	The Kafka topic on which the telemetry is being streamed. If the topic is prefixed and suffixed by / then it is assumed to be a regex and will match any topic matching the pattern (for example, /bro.*/ matches bro_cust0, bro_cust1 and bro_cust2).
<b>parserConfig</b>	A JSON map defining the parser implementation configuration.  For an envelope parser, this parameter specifies that the parser will send messages to the topic specified in the logical_source_type field. If the field does not exist, then the message is not sent.

#### EXAMPLE for Envelope Parser

The following is an example of an envelope parser called pix\_syslog\_router configured to:

- Parse the timestamp field
- Parse the payload into a field called data (messageField" : "data)
- Parse the tag into a field called pix\_type (input": "pix\_type)
- Route the enveloped message to the appropriate Kafka topic based on the tag. In this case, it's called logical\_source\_type.

The envelope parser will send output to two sub-parsers:

- cisco-6-302 - Connection creation and teardown messages, for example, Built UDP connection for faddr 198.207.223.240/53337 gaddr 10.0.0.187/53 laddr 192.168.0.2/53
- cisco-5-304 - URL access events, for example 192.168.0.2 Accessed URL 66.102.9.99:/

In order for this parser configuration to work, you must create a file called cisco\_patterns and populate it with the following grok expressions:

```

CISCO_ACTION Built|Teardown|Deny|Denied|denied|requested|permitted|denied
  by ACL|discarded|est-allowed|Dropping|created|deleted
CISCO_REASON Duplicate TCP SYN|Failed to locate egress interface|Invalid
  transport field|No matching connection|DNS Response|DNS Query|(?
  %{WORD}\s*)*
CISCO_DIRECTION Inbound|inbound|Outbound|outbound
CISCOFW302020_302021 %{CISCO_ACTION:action}(?:
  %{CISCO_DIRECTION:direction})? %{WORD:protocol} connection
  %{GREEDYDATA:ignore} faddr %{IP:ip_dst_addr}/%{INT:icmp_seq_num}(?:
  \(%{DATA:fwuser}\)\)? gaddr %{IP:ip_src_xlated}/%{INT:icmp_code_xlated}
  laddr %{IP:ip_src_addr}/%{INT:icmp_code}(\(%{DATA:user}\)\)?
ACCESSED %{URIHOST:ip_src_addr} Accessed URL %{IP:ip_dst_addr}:
  %{URIPATHPARAM:uri_path}
CISCO_PIX %{GREEDYDATA:timestamp}: %PIX-%{NOTSPACE:pix_type}:
  %{GREEDYDATA:data}

```

Place the file at /tmp/cisco\_patterns in HDFS by using:

```
hadoop fs -put ~/cisco_patterns /tmp
```



## Parser Configuration

```
{
  "parserClassName" : "org.apache.metron.parsers.GrokParser"
  , "sensorTopic" : "pix_syslog_router"
  , "parserConfig": {
    "grokPath": "/tmp/cisco_patterns",
    "batchSize" : 1,
    "patternLabel": "CISCO_PIX",
    "timestampField": "timestamp",
    "timeFields" : [ "timestamp" ],
    "dateFormat" : "MMM dd yyyy HH:mm:ss",
    "kafka.topicField" : "logical_source_type"
  }
  , "fieldTransformations" : [
    {
      "transformation" : "REGEX_SELECT"
      , "input" : "pix_type"
      , "output" : "logical_source_type"
      , "config" : {
        "cisco-6-302" : "^6-302.*",
        "cisco-5-304" : "^5-304.*"
      }
    }
  ]
}
```

**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.

For this example, this parameter includes the following options:

- **transformation** - The REGEX\_SELECT field transformation sets the logical\_source\_type field based on the value of the input value.
- **input** - Determines the subparser type.
- **output** - The output of the field transform.
- **config** - The name of the sub-parsers and the REGEX that matches them.

**6. Define one or more sub-parser configurations.**

- a) As root, log into the host with HCP installed:

```
ssh $HCP_HOST
```

- b) Create a \$DATASOURCE sub-parser configuration file at \$METRON\_HOME/config/zookeeper/parsers/\$SUBPARSER\_DATASOURCE.json:

```
/usr/hdp/current/kafka-broker/bin/kafka-topics.sh --zookeeper $ZOOKEEPER
--create --topic $SUBPARSER_DATASOURCE --partitions 1 --replication-
factor 1
```

- c) Populate the \$SUBPARSER\_DATASOURCE.json file with the following:

```
{
  "parserClassName": "org.apache.metron.parsers.GrokParser",
  "sensorTopic": "$SUBPARSER_DATASOURCE",
```

```

"rawMessageStrategy" : "ENVELOPE"
, "rawMessageStrategyConfig" : {
  "messageField" : "data",
  "metadataPrefix" : ""
"parserConfig": {
  "grokPath": "/apps/metron/patterns/$SUBPARSER_DATASOURCE",
  "batchSize" : 1,
  "patternLabel": "$DATASOURCE_DELIMITED",
  "timestampField": "timestamp"
  "timeFields" : [ "timestamp" ],
  "dateFormat" : "MMM dd yyyy HH:mm:ss",
  "kafka.topicField" : "logical_source_type"
}
}

```

The important parameters to set for this parser are the following:

<b>parserClassName</b>	The name of the parser's class in the .jar file.
<b>sensorTopic</b>	The Kafka topic on which the telemetry is being streamed. If the topic is prefixed and suffixed by / then it is assumed to be a regex and will match any topic matching the pattern (for example, /bro.*/ matches bro_cust0, bro_cust1 and bro_cust2).
<b>rawMessageStrategyConfig</b>	<p>This is a strategy that indicates how to read data and metadata. The strategies supported are:</p> <ul style="list-style-type: none"> <li>• <b>DEFAULT</b> - Data is read directly from the Kafka record value and metadata, if any, is read from the Kafka record key. This strategy defaults to not reading metadata and not merging metadata.</li> <li>• <b>ENVELOPE</b> - Data from Kafka record value is presumed to be a JSON blob. One of these fields must contain the raw data to pass to the parser. All other fields should be considered metadata. The field containing the raw data is specified in rawMessageStrategyConfig. Data held in the Kafka key as well as the non-data fields in the JSON blob passed into the Kafka value are considered metadata. Note that the exception to this is that any original_string field is inherited from the envelope data so that the original string contains the envelope data. If you do not prefer this behavior, remove this field from the envelope data.</li> </ul>
<b>rawMessageStrategyConfig</b>	<p>The configuration (a map) for the rawMessageStrategy. Available configurations are strategy dependent:</p> <ul style="list-style-type: none"> <li>• <b>DEFAULT</b> - metadataPrefix defines the key prefix for metadata (default is metron.metadata).</li> <li>• <b>ENVELOPE</b> - metadataPrefix defines the key prefix for metadata (default is metron.metadata)</li> </ul> <p>messageField defines the field from the envelope to use as the data. All other fields are considered metadata.</p>

**parserConfig**

A JSON map defining the parser implementation configuration.

For a chained parser, this parameter specifies that the parser will send messages to the topic specified in the `logical_source_type` field. If the field does not exist, then the message is not sent.

This parameter also includes batch sizing and timeout settings for writer configuration. If you do not define these properties, the system uses their default values.

- `grokPath` - The path for the Grok statement.
- `batchSize` - Number of records to batch together before sending to the writer. Default is 15.
- `patternLabel` - The name of the Grok statement that defines the pattern of the Grok expression.
- `kafka.topicField` - Specifies the topic as the value of a particular field.

This field enables the routing capabilities necessary for handling enveloped data. If this value is unpopulated, the message is dropped.

**EXAMPLE for Sub-Parser**

The following is an example of a parser called `cisco-6-302` configured to append to the existing fields from the `pix_syslog_router` the sensor specific fields based on the tag type.

```
{
  "parserClassName" : "org.apache.metron.parsers.GrokParser"
  , "sensorTopic" : "cisco-6-302"
  , "rawMessageStrategy" : "ENVELOPE"
  , "rawMessageStrategyConfig" : {
    "messageField" : "data",
    "metadataPrefix" : ""
  }
  , "parserConfig": {
    "grokPath": "/tmp/cisco_patterns",
    "batchSize" : 1,
    "patternLabel": "CISCOFW302020_302021"
  }
}
```

7. 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
```

8. 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 -z $ZOOKEEPER -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.

## Tune Parser Storm Parameters by Using the Management UI

You can tune some of your Storm parameters using the Management UI.

### Procedure

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

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



**Note:** Your sensor must be running and producing data before you can add tuning information.

3. In the **STORM SETTINGS** box, click



(expand window).

The Management UI displays the **Configure Storm Settings** panel.

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

4. You can tune the following Storm parameters:

**Spout Parallelism**

The Kafka spout parallelism (default to 1).

**Spout Num Tasks**

The number of tasks for the spout (default to 1)

**Parser Parallelism**

The parser bolt parallelism (default to 1).

**Parser Num Tasks**

The number of tasks for the parser bolt (default to 1).

**Error Writer Parallelism**

The error writer bolt parallelism (default to 1).

**Error Writer Num Tasks**

The number of tasks for the error writer bolt (default to 1).

**Spout Config**

A map representing a custom spout configuration.

**Storm Config**

The Storm configuration to use (this is a map). If both a specified, they are merged with the CLI properties taking precedence.

5. Click **SAVE**.

## 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 in Hortonworks Cybersecurity Platform (HCP) powered by Metron. HCP features several built-in parsers that support some common security devices.

### Snort

Snort is one of the telemetry data source parsers that are bundled in Hortonworks Cybersecurity Platform (HCP).

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"  
}
```

## Cisco Adaptive Security Appliance

Cisco Adaptive Security Appliance (ASA) is one of the telemetry data source parsers that are bundled in Hortonworks Cybersecurity Platform (HCP).

Cisco Adaptive Security Appliance (ASA) Software is the core operating system for the Cisco ASA Family. It delivers firewall capabilities for ASA devices in an array of form factors - standalone appliances, blades, and virtual appliances - for any distributed network environment.

## 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.

## ArcSight CEF

Common Event Format (CEF) is an extensible, text-based format designed to support multiple device types by offering the most relevant information.

The Common Event Format (CEF) standard format, developed by ArcSight, enables vendors and their customers to quickly integrate their product information into ESM.

## FireEye

FireEye, Inc. provides products and services to protect against advanced cyber threats, such as advanced persistent threats and spear phishing.

## 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.

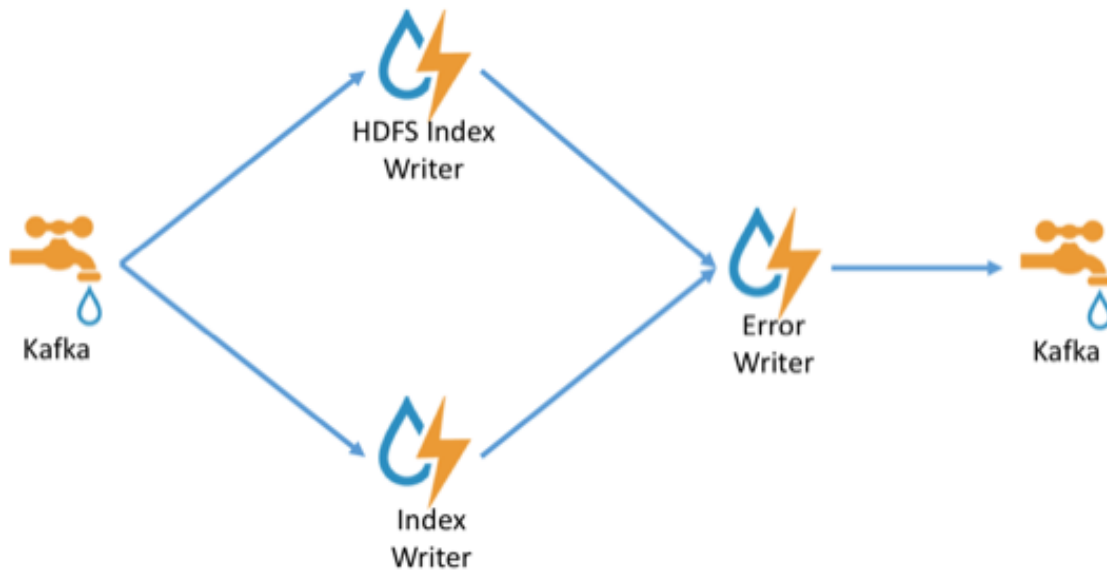
## Indexing

The Indexing topology takes data ingested into Kafka from enriched topologies and sends the data to an indexing bolt configured to write to HDFS and either Elasticsearch or Solr.

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:



## 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).

## 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

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

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

- Solr
- Elasticsearch
- 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.

If you would like to view the sensor output in the Alerts user interface, you must configure the sensor for either Solr or Elasticsearch.

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:

<b>index</b>	The name of the index to write to (defaulted is the name of the sensor).
<b>batchSize</b>	The size of the batch allowed to be written to the indices at once (defaulted is 1).
<b>enabled</b>	Whether the index or writer is enabled (default is true).

## 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

## Solr

Solr is an open source enterprise search platform. It is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more.

You can use Zoomdata to customize a dashboard for Solr data.

### Create a New Solr Index Collection

When you set up a new sensor, you must create either a new index template if you are using Elasticsearch or a new index schema if you are using Solr.

#### Procedure

1. Create a schema.xml file by copying an existing schema.xml file from another sensor and then replace the existing fields with the fields supported by your new sensor.

You can leave the common fields and type definitions in the new schema.xml file.

For example:

```
tail -n10 /usr/$METRON_HOME/config/schema/$SENSOR_DIRECTORY/schema.xml
<field name="ip_src_addr" type="ip" indexed="true" stored="true" />
<field name="ip_src_port" type="pint" indexed="true" stored="true" />
<field name="ip_src_addr" type="ip" indexed="true" stored="true" />
<field name="ip_dst_port" type="pint" indexed="true" stored="true" />
```

The schema.xml file describes the document fields, their types, and how they are indexed.



**Note:** If you have two fields with the same name, even if they are supported by different sensors and defined in different schema.xml files, they must have the same type. For example, if you have the ip\_src\_addr field defined in more than one schema.xml file, they must use the same type (such as, type="ip").

2. Create a Solrconfig.xml file by copying one from an existing sensor.  
The Solrconfig.xml file does not vary based on the content of the index.
3. Ensure that the Solr user has permission to access both the schema.xml and solrconfig.xml files.
4. Navigate to the \$SOLR\_USER and add the schema.xml file and the Solrconfig.xml file to the /usr/hcp/\$METRON\_HOME/config/schema directory.
5. As the \$SOLR\_USER, use create\_collection.sh to create the collection for your new sensor:

```
export SOLR_HOME='opt/lucidworks-hdpsearch/solr/'
export SOLR_USER=solr
export METRON_HOME=/usr/hcp/current/metron
export ZOOKEEPER=localhost:2181/solr
sudo -E su $SOLR_USER -c $METRON_HOME/bin/create_collection.sh $1
./create_solr_collection.sh $SENSOR_NAME
```

You can ignore the error logs.

6. Display the Solr UI to view your new collection.  
Refer to [Solr Index Schemas](#) for more information.



## Elasticsearch

Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.

Elasticsearch features a user interface named Kibana for viewing Elasticsearch data and search results.

### Create a New Elasticsearch Index Template

When you set up a new sensor, you must create a new index template if you are using Elasticsearch.

#### Procedure

Add the following to the properties section of the Elasticsearch template:

```
"properties": {
  "metron_field": {
    "type": "keyword"
  }
}
```

Refer to [Elastic Index Templates](#) for more information.

### Upgrading to Elasticsearch 5.6.2

Hortonworks Cybersecurity Platform (HCP) has deprecated support for Elasticsearch 2.x. You must upgrade to Elasticsearch 5.x to HCP queries in the current release. In addition to upgrading to Elasticsearch 5.x, you must also update Elasticsearch type mappings, templates, and existing sensors.

Elasticsearch 5.x 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]];
```

#### Elasticsearch Type Mapping Changes

Type mappings in Elasticsearch 5.6.2 have changed from ES 2.x.

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	<pre>"my_property" : {   "type":   "string",   "index":   "analyzed" }</pre>	<pre>"my_property" : {   "type": "text" }</pre> <p>Additional defaults: "index": "true", "fielddata": "false"</p>	"New York" handled via in-mem search as "New" and "York" buckets. No aggregation or sort.
yes	<pre>"my_property": {   "type":   "string",   "index":   "analyzed" }</pre>	<pre>"my_property": {   "type":   "text",   "fielddata":   "true" }</pre>	"New York" handled via in-mem search as "New" and "York" buckets. Can aggregate and sort.
yes	<pre>"my_property": {   "type":   "string",   "index":   "not_analyzed" }</pre>	<pre>"my_property" : {   "type":   "keyword" }</pre>	"New York" searchable as single value. Can aggregate and sort. A search for "New" or "York" will not match against the whole value.
yes	<pre>"my_property": {   "type":   "string",   "index":   "analyzed" }</pre>	<pre>"my_property": {   "type":   "text",   "fields": {     "keyword": {       "type":       "keyword",     }   }   "ignore_above":   256 }</pre>	"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

### Update Elasticsearch Templates to Work with Elasticsearch 5.x

HCP requires that all sensor templates have a nested `metron_alert` field defined to work with Elasticsearch 5.x.

#### Procedure

##### 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\
"metron_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
```

- To update existing indexes, update Elasticsearch mappings with the new field for each sensor:

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

### Update Existing Indexes to Work with Elasticsearch 5x

You must update existing indexes to work with Elasticsearch 5x.

#### Procedure

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
```

### Add X-Pack Extension to Elasticsearch

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

#### Before you begin

Ensure that Elasticsearch and Kibana are installed. You must also choose the X-pack version that matches the version of Elasticsearch that you are running.

#### Procedure

- Use the Storm UI to stop the **random\_access\_indexing** topology.
  - From **Topology Summary**, click **random\_access\_indexing**
  - Under **Topology actions**, click **Deactivate**.
- Install X-Pack on Elasticsearch and Kibana.  
See [Installing X-Pack](#) for information on installing X-Pack.
- 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 `xpack_client_user` with the password `changeme` and superuser credentials:

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

- 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. Pull the most recent HCP configuration to the local file system by running the following on the node on which HCP is installed:

```
$METRON_HOME/bin/zk_load_configs.sh -m PULL -o ${METRON_HOME}/config/
zookeeper -z $ZOOKEEPER -f
```

6. Set the X-Pack es.client.settings by adding it to \$METRON\_HOME/config/zookeeper/global.json.

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

```
{
  ...
  "es.client.settings" : {
    "xpack.username" : "xpack_client_user",
    "xpack.password.file" : "/apps/metron/elasticsearch/xpack-password"
  }
  ...
}
```

7. OPTIONAL: Set up SSL connection for Elasticsearch client:

- a) Navigate to a node that has an HDFS client, then create a file containing the password you used for your truststore file and upload it to HDFS.

For example:

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

- b) Add the following properties to es.client.settings in the \$METRON\_HOME/config/zookeeper/global.json file:

```
{
  ...
  "es.client.settings" : {
    "ssl.enabled": true,
    "keystore.path" : "$LOCAL_FILE_SYSTEM_PATH",
    "keystore.password.file" : "/apps/metron/elasticsearch/truststore-
password"
  }
  ...
}
```



**Note:** Make sure you do not overwrite the existing es.client.settings properties.

The truststore.jks file must reside on all Storm supervisor nodes as well as the REST application node. For more information about configuring Elasticsearch SSL for X-pack, see [Encrypted Communication](#).

8. Add the X-Pack changes to ZooKeeper:

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

9. Use Ambari to restart the REST API.
10. Use the Storm UI to restart the **random\_access\_indexing** topology.
- From **Topology Summary**, click **random\_access\_indexing**.
  - Under **Topology actions**, click **Start**.

## HDFS

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

## Index HDFS Tuning

For information on tuning indexing, see [General Tuning Suggestions](#).

## Turn Off HDFS Writer

You can turn off the HDFS index or writer by modifying the `index.json` file.

### Procedure

Create or modify the `index.json` file by adding the following:

```
{
  "solr": {
    "index": "foo",
    "enabled": true
  },
  "elasticsearch": {
    "index": "foo",
```

```

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

```

## 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.

## Understanding Global Configuration

The global configuration file is a repository of properties that can be used by any configurable component in the system. The global configuration file can be used to assign a property to multiple parser topologies. For example, every message from every sensor is validated against global configuration rules. The global configuration file can also be used to assign properties to enrichments and the profiler which each use a single topology. For example, you can use the global configuration to configure the enrichment topology's writer batching settings.

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.

**Table 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
profiler.writer.batchSize	Profiler	Integer	N/A
profiler.writer.batchTimeout	Profiler	Integer	N/A
update.hbase.table	REST/Indexing	String	update_hbase_table

Property Name	Subsystem	Type	Ambari Property
update.hbase.cf	REST-Indexing	String	update_hbase_cf
geo.hdfs.file	Enrichment	String	geo_hdfs_file
enrichment.writer.batchSize	Enrichment	Integer	N/A
enrichment.writer.batchTimeout	Enrichment	Integer	N/A
source.type.field	UI	String	source_type_field
threat.triage.score.field	UI	String	threat_triage_score-_field

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')"
    }
  }
]
```

## Create Global Configurations

The global configuration file is accessible to all configurable components in the system. The global configuration file can be used to assign a property to multiple parser topologies. For example, every message from every sensor is validated against global configuration rules. The global configuration file can also be used to assign properties to enrichments and the profiler which each use a single topology. For example, you can use the global configuration to configure the enrichment topology's writer batching settings.

### Procedure

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.

or 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:

---

<b>STELLAR</b>	Execute a Stellar Language statement. Expects the query string in the condition field of the config.
<b>IP</b>	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.
<b>DOMAIN</b>	Validates that the fields are all domains.
<b>EMAIL</b>	Validates that the fields are all email addresses.
<b>URL</b>	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.)

## Verify 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.

### Procedure

From the Alerts UI, search the source:type filter for the \$DATASOURCE messages.

By convention, the index of new messages is called \$DATASOURCE\_index\_[timestamp] and the document type is \$DATASOURCE\_doc.

## Streaming Data

After you add your parser and configure your indexing, you need to stream all raw events from that source into Kafka.

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 `thesnort.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.

- 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.

## Stream Data Using NiFi

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

### Procedure

1. Open the NiFi user interface canvas.
2. Drag




(processor icon) to your workspace.

NiFi displays the **Add Processor** dialog box.

3. Select the **TailFile** type of processor and click **ADD**.

NiFi displays a new TailFile processor:

 <span style="font-size: 1.2em; font-weight: bold;">TailFile</span> 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

4. Right-click the processor icon and select **Configure** to display the Configure Processor dialog box.
  - a) In the **Settings** tab, change the name to Ingest \$DATASOURCE Events:

### Configure Processor

SETTINGS

SCHEDULING

PROPERTIES

COMMENTS

Name

Id  
13a1a081-015c-1000-7972-76c6816628b0

Type  
TailFile

Penalty Duration ?

Bulletin Level ?

Enabled

Automatically Terminate Relationships ?  
 success  
All FlowFiles are routed to this Relationship.

Yield Duration ?

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

### Configure Processor

SETTINGS
SCHEDULING
PROPERTIES
COMMENTS

Required field +

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

CANCEL
APPLY

5. Click **Apply** to save your changes and dismiss the **Configure Processor** dialog box.
6. Add another processor by dragging



(processor icon) to your workspace.

7. Select the **PutKafka** type of processor and click **Add**.
8. Right-click the processor and select **Configure**.
9. In the **Settings** tab, change the name to Stream to Metron and then select the relationship check boxes for **failure** and **success**.

**Configure Processor**

**SETTINGS** | SCHEDULING | PROPERTIES | COMMENTS

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

CANCEL APPLY

10. In the **Properties** tab, set the following three properties:

<b>Known Brokers</b>	\$KAFKA_HOST:6667
<b>Topic Name</b>	\$DATAPROCESSOR
<b>Client Name</b>	nifi-\$DATAPROCESSOR

**Configure Processor**

SETTINGS
SCHEDULING
PROPERTIES
COMMENTS

Required field +

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

CANCEL
APPLY

11. Click **Apply** to save your changes and dismiss the **Configure Processor** dialog box.

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

NiFi displays Configure Connection dialog box.



**Create Connection**

**DETAILS** | **SETTINGS**

**From Processor**  
Ingest Squid Events Tailfile

**Within Group**  
NIFI Flow

**For Relationships**  
 success

**To Processor**  
Stream to Metron PutKafka

**Within Group**  
NIFI Flow

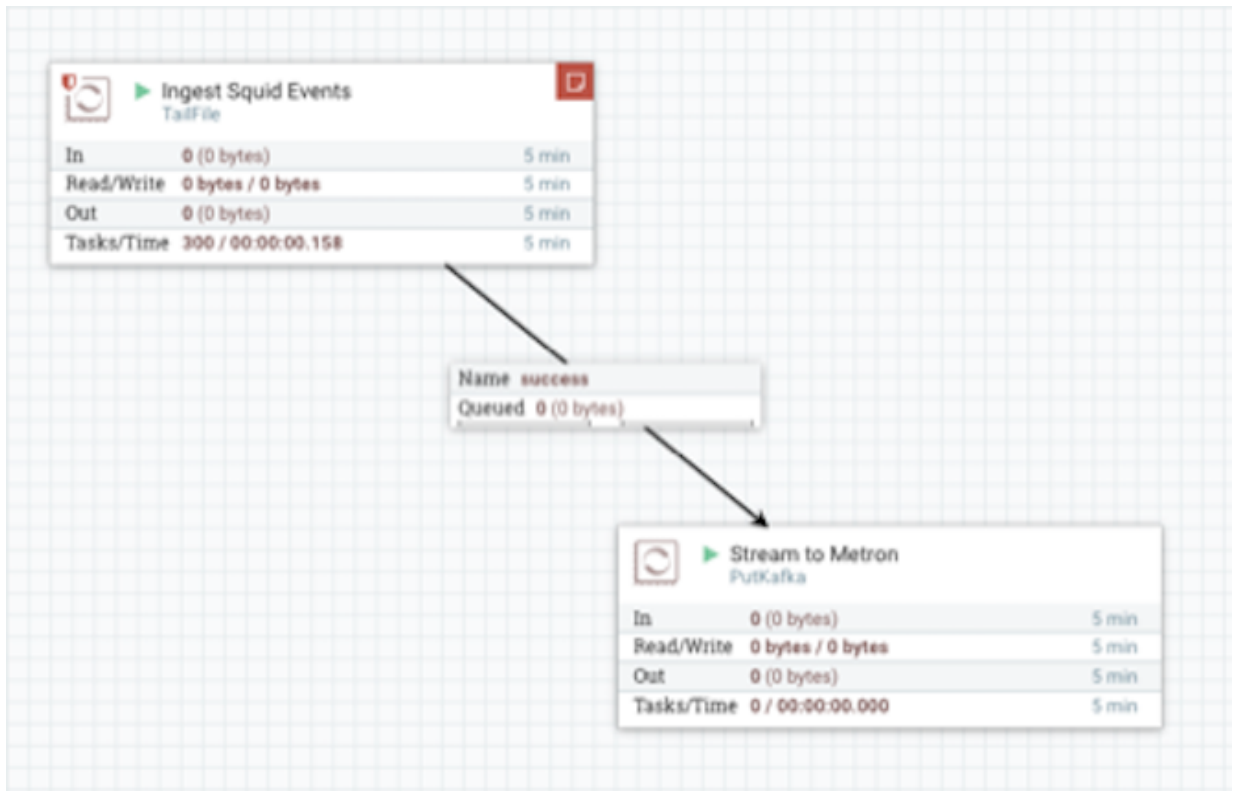
**CANCEL** **ADD**

13. In the **Details** tab, check the **failure** checkbox under **For Relationships**.

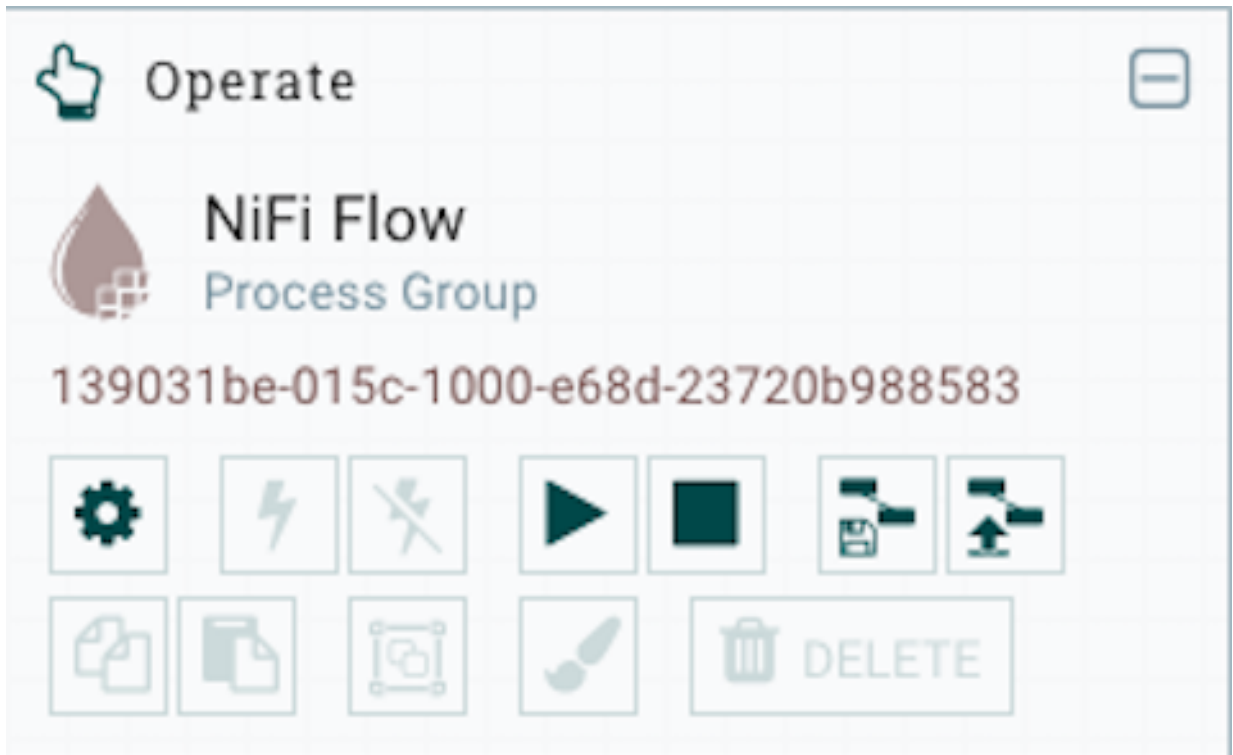
14. Click **APPLY** to accept the default settings for the connection.

15. Press **Shift** and draw a box around both parsers to select the entire flow.

All of the processor icons turn into green arrows:



16. In the Operate panel, click the arrow icon.



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

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

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