CCP Understanding Framework 2.0.0

# Understanding Enrichment

**Date of publish: 2017-11-06**

**https://docs.cloudera.com/**

# Legal Notice

# Contents

# 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 CCP data format (JSON files) and performs the following enhancements:

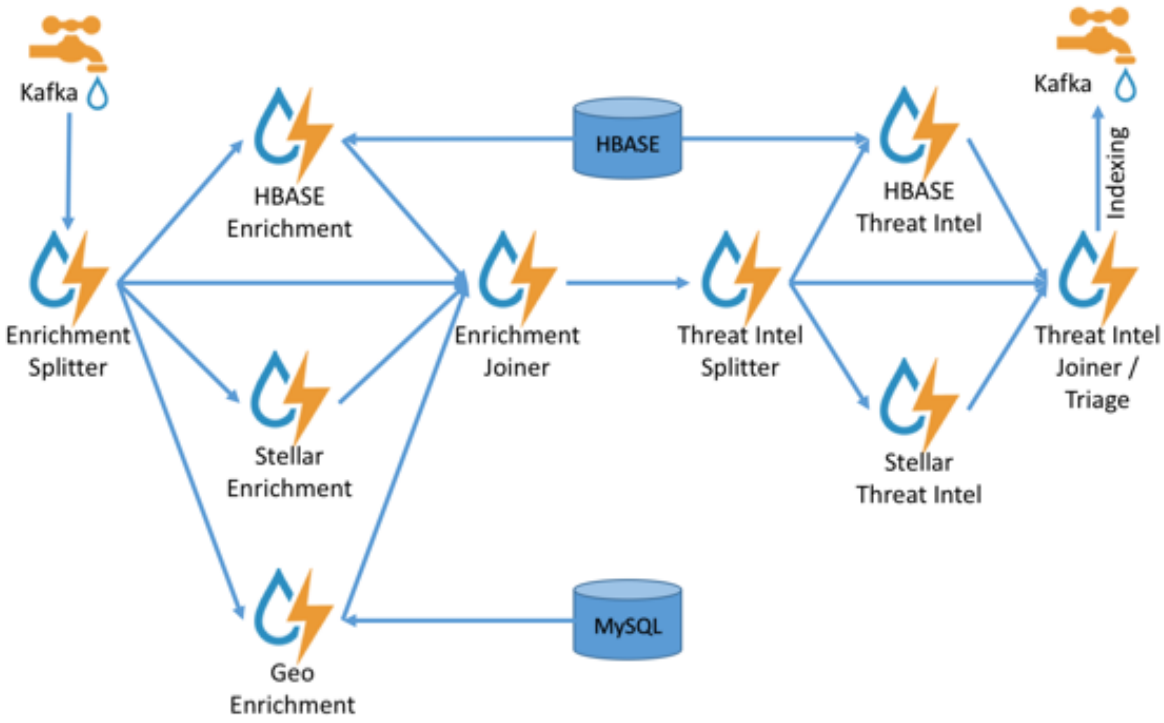
- Enriches messages with external data from data stores by adding new information based on existing fields in the messages
- Marks messages as threats based on data in external data stores
- Marks threat alerts with a numeric triage level based on a set of Stellar rules

The configuration for the enrichment topology is defined by JSON documents stored in ZooKeeper. CCP features two types of configurations:

- Sensor
- Global

The following figure illustrates the enrichment flow for both individual sensor enrichment and threat intelligence enrichment.

CCP Enrichment Flow




## 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.

## Individual Sensor Enrichments

Cloudera Cybersecurity Platform (CCP) includes three individual sensor enrichments.

CCP 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 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 tellar enrichment which provides tellar 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.

## 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 tellar 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 tellar 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'

## Threat Intelligence Enrichments

Cloudera Cybersecurity Platform (CCP) 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. CCP provides an adapter that is able to read Soltra-produced STIX/TAXII feeds and stream them into HBase. CCP

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 2: Threat Intelligence Enrichment Configuration**

| 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" : [ "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 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"
    }
  }
}
```

## 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 like so:

```
{
  ...
  ,"threatIntel" : {
            ...
          , "triageConfig" : {
                    "riskLevelRules" : {
                                "condition1" : level1
                              , "condition2" : level2
                                ...
                                }
                    ,"aggregator" : "MAX"
                            }
                  }
}
```

**riskLevelRules** — Correspond to the set of condition to numeric level mappings that define the threat triage for this particular sensor.

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

The current supported aggregation functions are:

**MAX** — The maximum of all of the associated values for matching queries

**MIN** — The minimum of all of the associated values for matching queries

**MEAN** — The mean of all of the associated values for matching queries

**POSITIVE_MEAN** — The mean of the positive associated values for the matching queries

## 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 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.

## Use Stellar for Queries

You can use Stellar to create queries.

The Stellar query language supports the following:

- Referencing fields in the enriched JSON
- Simple boolean operations: and, not, or
- Simple arithmetic operations: *, /, +, - on real numbers or integers
- Simple comparison operations <, >, <=, >=
- if/then/else comparisons (in other words, if var1 < 10 then 'less than 10' else '10 or more')
- Determining whether a field exists (via exists)
- The ability to have parenthesis to make order of operations explicit
- User defined functions

The following is an example of a Stellar query:

```
IN_SUBNET( ip, '192.168.0.0/24') or ip in [ '10.0.0.1', '10.0.0.2' ] or
 exists(is_local)
```

This query evaluates to “true” 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.

## Use 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 ou 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 CCP, see Stellar Language Quick Reference.

## Management Utility

Cloudera Cybersecurity Platform (HCP) recommends that your store your configuration on disk prior to uploading them to ZooKeeper.

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