How to parse JSON data

How-to guide

Requirements

- You should have a basic understanding of what Chronicle parsers are and why they are needed
- You should understand the basics of how a parser is structured, and how data is written to the Unified Data Model (UDM) within a parser
- You should have an understanding of the JSON format, and the syntax used to define different data types and structures

Overview

This document covers the basics on how to use the Logstash-style parsing syntax within Chronicle to parse data from a JSON log. This can be used as a guide to taking a JSON string and extracting the relevant fields for it, which can then be written to the Unified Data Model (UDM) for an event.

JSON Extraction Syntax

The base extraction syntax for a message in JSON is:

```
json {
 source => "message"
}
```

This will take the string value from the **message** variable, and attempt to extract each of the key-value pairs to a variable and value in the parser, example of using this is below:

```
# JSON message:
{
  "variableA": "a",
  "variableB": 199
}
# Parser:
filter {
  json {
    source => "message"
  }
```

```
  statedump{}
}
# Output:
Internal State (label=):
{
  "@collectionTimestamp": {
    "nanos": 845681187,
    "seconds": 1652095500
  },
  "@createTimestamp": {
    "nanos": 845681187,
    "seconds": 1652095500
  },
  "@enableCbnForLoop": true,
  "@onErrorCount": 0,
  "@output": [],
  "@timezone": "",
  "message": "{\"variableA\":\"a\",\"variableB\":199}",
  "variableA": "a",
  "variableB": 199
}
```

In the above output from **statedump{}**, we can see **variableA** and **variableB** have been extracted from the **message** variable, with the correct types (string, and integer respectively), and are now ready for us to assign to UDM fields.

Manipulating JSON Arrays

Where a JSON contains an array of values, we can use the **array_function** parameter to extract these values in a format we can then iterate through. The extraction syntax for this is:

```
json {
  source => "message"
  array_function => "split_columns"
}
```

An example of using this is shown below:

```
# JSON message:
{
  {"arrayA":["a",1,true]}
}
# Parser:
filter {
  json {
    source => "message"
    array_function => "split_columns"
  }
  statedump{}
}
# Output:
Internal State (label=):
{
  "@collectionTimestamp": {
    "nanos": 935211265,
    "seconds": 1652096143
  },
  "@createTimestamp": {
    "nanos": 935211265,
    "seconds": 1652096143
  },
  "@enableCbnForLoop": true,
  "@onErrorCount": 0,
  "@output": [],
  "@timezone": "",
  "arrayA": {
    "0": "a",
    "1": 1,
    "2": true
  },
  "message": "{\"arrayA\":[\"a\",1,true]}"
}
```

In the above output from **statedump{}**, we can see **arrayA** has been extracted from the **message** variable, and expanded into the **arrayA.0**, **arrayA.1**, and **arrayA.2** variables, with the correct types (string, integer, and boolean respectively), and are now ready for us to assign to UDM fields.

These can be iterated over with the **for in** syntax, an example of this is below:

```
for index,value in arrayA {
  mutate {
    merge => {
      "user.phone_numbers" => "value"
    }
  }
}
```

Note that both the index and the value of the array entry can be extracted, although in this example we are only using the value.

Error Handling

We can use the **on_error** parameter to handle errors in extracting the JSON, this can then be used to drive programmatic logic for handling this error. The syntax for this is:

```
json {
  source => "message"
  on_error => "_not_json"
}
```

We can then use the **_not_json** variable to drive error handling scenarios, an example of this is below:

```
if [_not_json] {
  drop { tag => "TAG_UNSUPPORTED" }
}
```

This will drop the parsing process if the **message** field passed to the **json** block was not syntactically correct JSON.