CRIBL NOTES on

STREAM CHEAT SHEET

U.S. $FREE

Version 2.0

©2020-2022 Cribl, Inc.
## Basic Concepts

### Sources

Cribl Stream can process data from various metrics, logs, traces, and generic event sources, including Splunk, HTTP, Elastic Beats, Kinesis, Kafka, TCP JSON etc. Depending on the Source, both push and pull methods are supported.

### Destinations

Cribl Stream can send data to various Destinations, including Exabeam, Splunk, SignalFx, Kafka, Elasticsearch, Kinesis, InfluxDB, Snowflake, Databricks, Honeycomb, Azure Blob Store, Azure EventHubs, TCP JSON, Wavefront, and many others. Destinations can be streaming (events are sent in real time) or non-streaming (events are sent in batches).

### Routes

Routes evaluate incoming events against filter expressions to find the appropriate Pipeline to send them to. Routes are evaluated in order, and a Route can be associated only with one Pipeline and one Destination.

### Pipelines

A series of Functions is called a Pipeline, and the order in which they are executed matters. Events are delivered to the beginning of a Pipeline by a Route, and as they’re processed by a Function, they are passed onto the next Function down the line. Events only move forward, towards the end of the Pipeline and eventually out of the system.

### Functions

A Function is a piece of JavaScript code that executes on an event, and it encapsulates the smallest amount of processing that can happen to that event. E.g., a Function can replace the term foo with bar on each event. Another one can hash bar, and yet another can add a field, say, dc=jfk-42 to any event from host us-nyc-42.cribl.io.

### Packs

Packs enable Cribl Stream administrators and developers to pack up and share complex configurations and workflows across multiple Worker Groups, or across organizations. Packs can contain everything between a Source and a Destination: Routes (Pack-level), Pipelines (Pack-level), Functions (built-in and custom), Sample data files, Knowledge objects (Lookups, Parsers, Global Variables, Grok Patterns, and Schemas). Wherever you can reference a Pipeline, you can specify a Pack!

### QuickConnect

QuickConnect is a visual rapid-development UI. With it, you can connect Cribl Stream inputs (Sources) to outputs (Destinations) through simple drag-and-drop. You can then insert Pipelines or Packs into the connections, to take advantage of Cribl Stream’s full range of data-transformation Functions. Or you can omit these processing stages entirely, to send incoming data directly to Destinations – with minimal configuration fuss.

### Scaling and Sizing

Expected resource utilization will be proportional to how much overall processing is occurring. For instance, a Function that adds a static field will likely perform faster than one that applies a regex to finding and replacing a string. Cribl’s current sizing guidance is 400GB thru/day/CPU. For example:

- Input: 4TB/day
- Outputs: 4TB/day to S3
- 2TB/day to Splunk
- Total thru: 10TB/day
- Est. CPUs: 25 (10TB/400GB)
Basic Concepts (cont.)

Cribl.Cloud

The fast alternative to downloading and self-hosting Cribl Stream software is to launch Cribl.Cloud. This SaaS version, whether free or paid, places the Leader and the Worker Nodes/Edge Nodes in Cribl.Cloud, where Cribl assumes responsibility for managing the infrastructure.

Deployments

When data volume is low, and/or the amount of processing is light, a single-instance deployment may be sufficient. To accommodate higher volume, increased processing complexity, and increased availability, Cribl Stream can be scaled up and out across multiple instances. This is known as a distributed deployment. Distributed deployments are not limited to on-premises only or Cribl.Cloud only; Stream is also capable of hybrid deployments.

Event Model

All data processing is based on discrete data entities commonly known as events. An event is generally defined as a collection of key/value pairs (fields). Some Sources deliver discrete events directly, while others might deliver bytestreams that need to be broken up by Event Breakers. The internal representation of a Cribl Stream event looks like this:

```json
{
  "_raw": "<body of non JSON parse-able event>",
  "_time": "<timestamp in UNIX epoch format>",
  "__inputId": "<Source of the event>",
  "__other1": "<Internal field1>",
  "__other2": "<Internal field2>",
  "__otherN": "<Internal fieldN>",
  "key1": "<value1>",
  "key2": "<value2>",
  "key3": "<value3>",
  "keyN": "<valueN>",
  ... : "..."
}
```

Fields that with start with a double underscore are internal to Stream. For example, syslog sources add both an __inputId and a __srlcpPort field to each event. Internal fields can be used in a Pipeline, but are not passed down to Destinations. If an event cannot be JSON-parsed, all of its content will be assigned to the _raw field. If a timestamp is not configured, or cannot be extracted from an event, Stream will assign the current time (in UNIX epoch format) to _time.

Filters and Value Expressions

You can use JavaScript filters and other value expressions to configure Stream’s Routes and built-in Functions. Expressions are syntactically valid units of code that resolve to a value. Conceptually, Stream supports two types of expressions: Some assign a value to a field – e.g., myAnswer=42. Others evaluate to a value, – e.g., (Math.random() * 42). Filters are expressions that must evaluate to either true (or truthy) or false (or falsy). You can use filters in Routes to select a subset of incoming data flow, and in Functions to scope or narrow down their applicability.

Some simple examples:

**Filter:** Check if incoming events are from host foo or filename ends in .log:

```
host==’foo’ || source.endsWith(‘.log’)
```

**Expression:** Assign field sourcetype the value of cisco:asa if string %ASA is in _raw, else leave it as is.

```
/%ASA/.test(_raw) ? ’cisco:asa’ : sourcetype
```
Performance Tips

Start On Boot (systemd)

Cribl Stream can be configured to start at boot time with systemd. To do this, run the boot-start command. Make sure you first create a user you want to specify to run Cribl Stream.

Example: To run Cribl Stream on boot as existing user cribl use:
```
sudo $CRIBL_HOME/bin/cribl boot-start enable -m systemd -u cribl
```

The above will install a unit file named cribl.service, and will start Cribl Stream at boot time as user cribl.

Note: A `--configDir` option can be used to specify where to install the unit file. If not specified, this location defaults to /etc/systemd/system/*. 

It may be necessary to change ownership for the Cribl Stream installation ([sudo] chown -R cribl $CRIBL_HOME). Finally, enable Stream to ensure that the service starts on system boot ([sudo] systemctl enable cribl)

Available systemctl commands are: systemctl [start|stop|restart|status] cribl

Filtering Function Performance

The filters in your Cribl Stream Routes and Pipelines are like the gasoline in your car’s engine. The better the fuel, the better the engine runs. The better your Stream filters, the faster your Routes and Pipelines can process observability data. Observability at scale requires careful attention to minor performance items, such as the choice of the function in your filter. A good rule of thumb is: regex matching functions such as `match`, `test`, `search` will typically take more CPU to process than string matching functions such as `indexOf`, `includes`, and `startsWith`, etc.

JSON.parse(_raw)

JSON events are received as strings in Cribl Stream. In order to improve performance (and reduce event size) try adding an Eval function to the beginning of your pipeline where Name is _raw and Value Expression is set to `JSON.parse(_raw)`. This will quickly convert the JSON back from a string (faster than using a parser) AND slightly shrink the overall event!

_regex_.split

Multi-line logs and multi-value arrays can be tough to deal with. Navigate the logs easier by adding an Eval to the beginning of your pipeline where Name is _raw and Value Expression is set to `_raw.split('\n')`. This will quickly cut up the multi-line log file for easy parsing later on. You can even reference specific lines in the log by using `_raw.split('\n')[i]` and substituting `i` with the desired line!

Regex Lookups

When using C.LookupRegex be wary of empty new lines. Regex lookups return true when a file has a dangling new line since these are treated as a wildcard. So remember: When making lookup files -- don’t leave an empty line at the end!

Using Built-in Functions

Stream ships with a growing collection of highly configurable, out-of-the-box Functions. Below, we list key built-in Functions by purpose, followed by available JavaScript methods and Cribl expressions.

Create, remove, update, rename fields
- Functions: Eval, Rename, Lookup, Regex Extract, Grok
- Find & Replace, including basic sed-like, obfuscate, redact, hash etc.: Mask, Eval
- Add GeoIP information to events: Lookup, GeoIP
- Extract fields from structured and unstructured events: Regex Extract, Parser
- Extract and assign timestamps: Auto Timestamp
- Drop events: Drop, Regex Filter, Sampling, Suppress, Dynamic Sampling
- Sample events (e.g., high volume, low value data): Sampling, Dynamic Sampling
- Suppress events (e.g., remove duplicates etc.): Suppress
- Convert JSON arrays or XML elements into own events: Unroll, JSON Unroll, XML Unroll
- Serialize events to CEF format (send to various SIEMs): CEF Serializer
- Serialize / change format (e.g., convert JSON to CSV): Serialize
- Flatten nested structures (e.g., nested JSON): Flatten
- Aggregate events in real-time (i.e. statistical aggregations): Aggregations
- Convert events to metrics format: Publish Metrics.
# Commonly Used Functions

## Eval

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>FIELD NAME</th>
<th>VALUE EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a field called source and set it to mySource</td>
<td>source</td>
<td>'mySource'</td>
</tr>
<tr>
<td>Change status to success if code is 200</td>
<td>status</td>
<td>code==200 ? 'success' : status</td>
</tr>
<tr>
<td>Set location field to city, state</td>
<td>location</td>
<td><code>${city}, ${state}</code></td>
</tr>
<tr>
<td>Replace .com with .net in url</td>
<td>url</td>
<td>url.replace(./com/, ./net/)</td>
</tr>
<tr>
<td>Set private to yes if myip is on private range</td>
<td>private</td>
<td>C.Net.isPrivate('10.10.2.0') ? 'yes' : 'no'</td>
</tr>
<tr>
<td>Create an array field called myField</td>
<td>myField</td>
<td>['aa', 'bb', 'cc', 'dd']</td>
</tr>
<tr>
<td>Convert value of classname to lowercase</td>
<td>classname</td>
<td>classname.toLowerCase()</td>
</tr>
</tbody>
</table>

## Mask

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>MATCH REGEX</th>
<th>REPLACE EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove phone numbers from events</td>
<td>phonenumber=\d+</td>
<td>''</td>
</tr>
<tr>
<td>Remove comments (lines that start with #)</td>
<td>^#.*$ /m</td>
<td>''</td>
</tr>
<tr>
<td>Replace sensitive with REDACTED</td>
<td>sensitive</td>
<td>'REDACTED'</td>
</tr>
<tr>
<td>Remove description from wineventlogs</td>
<td>This event is generated[\s\S]+$</td>
<td>'DESCRIPTION REMOVED'</td>
</tr>
<tr>
<td>Redact last octet of an IPv4 address</td>
<td>(\d+.\d+.\d+.\d+)</td>
<td><code>${g1}xxx</code></td>
</tr>
<tr>
<td>MD5 hash a credit card number</td>
<td>(creditcard=)(\d+)</td>
<td><code>${g1}${C.Mask.md5(g2)}</code></td>
</tr>
</tbody>
</table>

## Regex Extract

Sample Event: 2020-04-20 16:20:00 input=tcpjsonin rate=42 host=555.example.com region=us-east state=nj city=edgewater

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>MATCH REGEX</th>
<th>VALUE EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract only input and rate fields</td>
<td>input=(&lt;?input&gt;\w+)&lt;s+rate=(&lt;?rate&gt;\d+)</td>
<td></td>
</tr>
<tr>
<td>Extract all KV pairs in the event</td>
<td>(?&lt;_NAME_0&gt;[^\s]+)=(&lt;?_VALUE_0&gt;[^\s]+)</td>
<td></td>
</tr>
</tbody>
</table>

Cribl Stream ships with a Regex Library that contains a set of pre-built common regex patterns.

Example: IPv4 Address - (?<=\d)(?:[0-1]?\d\d?|2[0-4]\d|25\d|25[0-5])\.(?<=\d+)(?<!\d+)(\d+|\d+)
Commonly Used Functions (cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Drop</strong></td>
<td><strong>DESCRIPTION</strong></td>
</tr>
<tr>
<td>Drop all DEBUG events</td>
<td>_raw.includes('DEBUG')</td>
</tr>
<tr>
<td>Drop all DEBUG events from host myhost</td>
<td>_raw.includes('DEBUG') &amp;&amp; host=='myHost'</td>
</tr>
<tr>
<td>Drop 50% of all events (poor man's sampler)</td>
<td>Math.random() &gt; 0.5</td>
</tr>
<tr>
<td>Drop all events with length less than 42 bytes</td>
<td>_raw.length &lt; 42</td>
</tr>
</tbody>
</table>

**Custom Code Function**

If you need to operate on data in a way that can’t be accomplished with Cribl Stream’s out-of-the-box Functions, the Code Function enables you to encapsulate your own JavaScript code. This Function is available in Cribl Stream 3.1+, and imposes some restrictions for security reasons.

**Considerations**

- Generally speaking, anything forbidden in JavaScript strict mode is forbidden in the context of the Code Function. Specifically, the following are not allowed: console, eval, uneval, Function (constructor), Promises, setTimeout, setInterval, global, globalThis, and window. All JavaScript loops and statements are allowed: for, for-of, while, do-while, switch, etc.

The Code Function watches user-defined functions to detect infinite loops that would cause processing to hang. To limit the number of iterations allowed per instance of your Code Function, adjust the Advanced Settings > Maximum number of iterations option. This defaults to 5,000; the maximum number allowed is 10,000. Once the limit is reached, the Code Function will stop processing whatever is after the statement that exhausted the allowed maximum.

**Example Event Data**

Below is an example event that will be referenced in the Function Examples table. You can try out the below functions by pasting the cpus array from the example event into an Eval function at the top of a pipeline with a field titled cpus.

```json
{
  "cpus": [  
    {"number": 1, "name": "CPU1", "value": 2.3},  
    {"number": 2, "name": "CPU2", "value": 3.1},  
    {"number": 3, "name": "CPU3", "value": 5.1},  
    {"number": 4, "name": "CPU4", "value": 1.3}  
  ],  
  "arch": "Intel x64"
}
```

**Function Examples**

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>EXPRESSION</th>
<th>NOTES</th>
</tr>
</thead>
</table>
| Accessing Fields in an Event | __e['field-name'] | In other words, think of your code executing in a context like this: function(__e: Event) {
 // your code here
} |
| Eval a Field | __e['test'] = 'Hello, Goats!' | Create a field test with the value Hello, Goats! |
| JSON Filter | __e['cpus_filtered'] = __e['cpus'].filter(entry => entry.value >= 3) | Filter the cpus array inside the event, only keep values greater than or equal to 3 and place in a new field cpu_filtered |
| Reduce | __e['cpus_reduce'] = __e['cpus'].reduce((accumulator, entry) => accumulator + entry.value, 0) | Summarize data across an array, with a returned accumulator value (cpus_reduce: 11.8) |
| Some | __e['cpus_some'] = __e['cpus'].some(entry => entry.value >= 3) | cpus_some set to true because there is at least one object with a value greater than or equal to 3 |
| Every | __e['cpus_every'] = __e['cpus'].every(entry => entry.value >= 10) | cpus_every is false, because not all values in the event are greater than or equal to 10 |
### Function Examples (cont.)

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>EXPRESSION</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform a Specific Field</td>
<td>__e['cpus'] = __e['cpus'].map(entry =&gt; Object.assign(entry, {'name': entry.name.toLowerCase()}))</td>
<td>Each cpus member will have its name field transformed to lowercase.</td>
</tr>
<tr>
<td>Debugging</td>
<td>debug(&quot;this is the event &quot; + '{JSON.stringify(__e)}&quot;);</td>
<td>Adds message <strong>this is the event</strong> to events viewed in the Preview Log. Regular logs will also show this message if the function log level is set to <strong>debug</strong>.</td>
</tr>
</tbody>
</table>

### Useful JS methods:

- **String**
  - `.startsWith()`, `.endsWith()`, `.trim()`, `.trimEnd()`, `.trimStart()`, `.substring()`, `.split()`, `.indexOf()`, `.length`, etc.

- **Number**
  - `.isInteger()`, `.toFixed()`, `.parseFloat()`, `.toString()`, etc.

- **Math**

### Cribl Expressions:

- **Decode**
  - C.Decode.base64(), C.Decode.gzip(), C.Decode.hex(), C.Decode.uri()

- **Encode**
  - C.Encode.base64(), C.Encode.gzip(), C.Encode.hex(), C.Encode.uri()

- **Inline Lookup**
  - C.Lookup(), C.LookupCIDR(), C.LookupRegex()

- **Mask**
  - C.Mask.isCC(), C.Mask.luhn(), C.Mask.md5(), C.Mask.sha1(), C.Mask.random(), etc.

- **Net**
  - C.Net.cidrMatch(), C.Net.ipv6Normalize(), C.Net.isPrivate()

- **Text**
  - C.Text.entropy(), C.Text.hashCode(), C.Text.isASCII(), C.Text.isUTF8(), C.Text.relativeEntropy()

- **Time**
  - C.Time.strftime(), C.Time.strptime(), C.Time.timestampFinder()

- **Others**
  - C.vars, C.env, C.Misc.xxx(), C.version etc.