ExtraHop 6.0
Trigger API Reference

For more documentation, see https://docs.extrahop.com/.

Published: 2016-12-21

ExtraHop Networks
Seattle, WA 98101
877-333-9872 (US)
+44 (0)203 7016850 (EMEA)
+65-31585513 (APAC)
www.extrahop.com

Contents

Overview 5

ExtraHop data types 6

Global functions 7

General purpose classes 10

Application 10
Buffer 13
Device 15
Flow 17
FlowInterface 33
FlowNetwork 34
GeoIP 35
IPAddress 36
Network 37
Session 38
System 40
Trigger 41
VLAN 41

Protocol and network data classes 42

AAA 44
ActiveMQ 48
CIFS 51
DB 55
DHCP 58
DICOM 61
DNS 63
FIX 67
FTP 69
HL7 73
HTTP 75
IBMMQ 80
ICA 83
ICMP 89
Kerberos 95
LDAP 98
LLDP 101
Memcache 102
MongoDB 105
MSMQ 108
NetFlow 110
NFS 114
POP3 117
Redis 120
RTCP 122
RTP 129
SDP 131

SIP 133
SMPP 138
SMTP 140
SSH 143
SSL 146
TCP 155
Telnet 162
Turn 165
UDP 166
WebSocket 166

Open data stream classes 169

Remote.HTTP 169
Remote.Kafka 172
Remote.MongoDB 174
Remote.Raw 176
Remote.Syslog 177

Datastore classes 179

AlertRecord 179
Dataset 180
Discover 180
MetricCycle 181
MetricRecord 181
Sampleset 182
Topnset 182

Deprecated API elements 184

Advanced trigger options 186

Performance optimization tips 189

Examples 191

Example: Collect ActiveMQ metrics 191
Example: Send data to Azure with Remote.HTTP 192
Example: Monitor CIFS actions on devices 193
Example: Track 500-level HTTP responses by customer ID and URI 194
Example: Collect response metrics on database queries 195
Example: Send discovered device data to a remote syslog server 195
Example: Send data to Elasticsearch with Remote.HTTP 196
Example: Access HTTP header attributes 196
Example: Collect IBMMQ metrics 197
Example: Record Memcache hits and misses 198
Example: Parse memcache keys 199
Example: Add metrics to the metric cycle store 201
Example: Parse custom PoS messages with universal payload analysis 202
Example: Parse syslog over TCP with universal payload analysis 203
Example: Parse NTP with universal payload analysis 206
Example: Record data to a session table 207
Example: Track SOAP requests 208
Example: Matching topnset keys 209
Example: Create an application container 211

Overview

Application Inspection triggers are composed of user-defined code that automatically executes on system events through the ExtraHop trigger API. By writing triggers, you can collect custom metric data about the activities on your network. In addition, triggers can perform operations on protocol messages (such as an HTTP request) before the packet is discarded.

The ExtraHop system monitors, extracts, and records a core set of Layer 7 (L7) metrics for devices on the network, such as response counts, error counts, and processing times. After these metrics are recorded for a given L7 protocol, the packets are discarded, freeing resources for continued processing.

Triggers enable you to:

- Generate and store custom metrics to the internal datastore of the ExtraHop system. For example, while the ExtraHop system does not collect information about which user agent generated an HTTP request, you can generate and collect that level of detail by writing a trigger and committing the data to the datastore. You can also view custom data that is stored in the datastore by creating custom metrics pages and displaying those metrics through the Metric Explorer and dashboards.
- Generate and sends records to an Explore appliance for long-term storage and retrieval.
- Create a user-defined application that collects metrics across multiple types of network traffic to capture information with cross-tier impact. For example, to gain a unified view of all the network traffic associated with a website—from web transactions to DNS requests and responses to database transactions—you can create an application that contains all of these website-related metrics.
- Generate custom metrics and send the information to syslog consumers such as Splunk, or to third party databases such as MongoDB or Kafka.
- Initiate a packet capture to record individual flows based on user-specified critera. You can download captured flows and process them through third-party tools. Your ExtraHop system must be licensed for packet capture to access this feature.

The purpose of this guide is to provide reference material when writing the blocks of JavaScript code that run when trigger conditions are met. See Get started with triggers in the ExtraHop Web UI Guide for a comprehensive overview of trigger concepts and procedures.

ExtraHop data types

ExtraHop data types record custom metrics using the Network, Application, and Device, FlowNetwork, and FlowInterface classes.

There are two kinds of metrics in the ExtraHop system:

Top-level metrics

Represent an aggregate of all activity for a particular object type, such as network, application or device.

count

Number (e.g., HTTP requests).

snapshot

A special type of count metric that, when queried over time, returns the most recent value (e.g., TCP established connections).

dataset

Statistical summary of timing information (5-number summary: min, 25th-percentile, median, 75th-percentile, max).

sampleset

Statistical summary of timing information (mean and standard deviation).

max

A special type of count metric that preserves the maximum.

Detail metrics

Represents activity that is broken down by specific keys such as IP addresses or URIs. For each key, there is a value that corresponds to the top-level metric types such as count or snapshot. Detail metrics provide drill-down information for top level metrics.

Examples:

- To record information about the number of HTTP requests over time, use a top-level count metric.
- To record information about HTTP processing time over time, use a top-level sampleset (mean and average) or dataset (5-number summary) metric.
- To record information about the number of times each client IP address accessed the server, use a detail count metric with the IPAddress key and an integer representing the number of accesses as a value.
- To record information about the length of time it took the server to process each URI, use a detail sampleset or dataset metric with the URI string key and an integer representing processing time as a value.
- To record the slowest HTTP statements over time without relying on a Session table, use a top-level and a detail max metric.

Global functions

Global functions can be called on any event.

cache (key: *String*, valueFn: () => *any*): *any*

Caches the specified parameters in a table to enable efficient lookup and return of large data sets.

key: *String*

An identifier that indicates the location of the cached value. Keys must be unique within a trigger.

valueFn: () => *any*

A zero-argument function that returns a non-null value.

In the following example, a list of known user agents in a JBoss trigger needs to be normalized before comparison with the observed user agent. The trigger converts the list to lowercase and trims excess whitespace, and then caches the entries:

```
function jbossUserAgents() {
    return [
        // Add your own user agents here, followed by a comma
        "Gecko-like (Edge 14.0; Windows 10; Silverlight or similar)",
        "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)
 AppleWebKit/537.36
         (KHTML, like Gecko) Chrome/51.0.2704.79 Safari/537.36",
        "Mozilla/5.0 (Android)"
    ].map(ua => ua.trim().toLowerCase());
}

// Added further in the trigger within an if-statement designed to
 filter out most HTTP traffic
var badUserAgents = cache("badUserAgents", jbossUserAgents);
```

The following example shows the `cache` method called in a trigger with large amounts of data hard-coded into the source code:

```
let hashTable = cache("hashTable", () => ({
    1 : "Assignment Queue Newark",
    2 : "Assignment Queue St Paul",
    3 : "Authorization Queue Newark",
    4 : "Authorization Queue St Paul"// 620 lines omitted
}));
```

commitRecord (id: *String*, record: *Object*): *void*

Commits a custom record object to the ExtraHop Explore appliance.

id: *String*

The ID of the record type to be created. The ID cannot begin with a tilde (~).

record: *Object*

An object containing a list of property and value pairs to be committed to the ExtraHop Explore appliance as a custom record.

The following properties are automatically added to records and are not represented on the objects returned by the built-in record accessors, such as `HTTP.record`:

- ex
- flowID
- client

- clientAddr
- clientPort
- receiver
- receiverAddr
- receiverPort
- sender
- senderAddr
- senderPort
- server
- serverAddr
- serverPort
- timestamp
- vlan

For example, to access the `flowID` property in an HTTP record, you would include `HTTP.record.Flow.id` in your statement.

Important: To avoid unexpected data in the record or an exception when the method is called, the property names listed above cannot be specified as a property name in custom records.

In addition, a property name in custom records cannot contain any of the following characters:

- "." - period
- ":" - colon
- "[" and "]" - square brackets

In the following example, the two property and value pairs that have been added to the `record` variable are committed to a custom record by the `commitRecord` function:

```
var record = {
   field1: 'myfield1',
   field2: 'myfield2'
};
commitRecord('record_type_id', record);
```

For built-in protocols that support committing metrics, you can access records that contain default properties. For example, you can access the `HTTP.record` object on an HTTP response event. A built-in record can be the basis for a custom record. In the following example, the `HTTP.record` object and two properties are added to the record variable. All of the default properties in the `HTTP.record` object and the two additional properties are committed to a custom record by the `commitRecord` function.

```
var record = {
   HTTP.record,
   field1: 'myfield1',
   field2: 'myfield2'
};
commitRecord('record_type_id', record);
```

debug (message: *String*): *void*

Writes to the runtime log if debugging is enabled.

getTimestamp (): *Number*

Returns the timestamp from the packet that caused the trigger event to run, expressed in milliseconds with microseconds as the fractional part after the decimal.

log (message: *String*): *void*

Writes to the runtime log regardless of whether debugging is enabled.

Multiple calls to debug and log statements in which the message is the same value will display once every 30 seconds.

The limit for runtime log entries is 2048 bytes. Refer to Remote.Syslog to log larger entries.

md5 (message: *String*): *String*

Hashes the UTF-8 representation of the specified message string and returns the MD5 sum of the string..

sha1 (message: *String*): *String*

Hashes the UTF-8 representation of the specified message string and returns the SHA1 sum of the string..

uuid (): *String*

Returns a random version 4 Universally Unique Identifier (UUID).

General purpose classes

The Trigger API classes in this section provide functionality that is broadly applicable across all events.

Class	Description
Application	Enables you to create new applications and adds custom metrics at the application level.
Buffer	Enables you to access to buffer content.
Device	Enables you to retrieve device attributes and add custom metrics at the device level.
Flow	Flow refers to a conversation between two endpoints over a protocol such as TCP, UDP or ICMP. The Flow class provides access to elements of these conversations, such as endpoint IP addresses and age of the flow. The Flow class also contains a flow store designed to pass objects from request to response on the same flow.
FlowInterface	Enables you to retrieve flow interface attributes and add custom metrics at the interface level.
FlowNetwork	Enables you to retrieve flow network attributes and add custom metrics at the flow network level.
GeoIP	Enables you to retrieve the approximate country-level or city-level location of a specific IP address.
IPAddress	Enables you to retrieve IP address attributes.
Network	Enables you to add custom metrics at the global level.
Session	Enables you to access to the session table which supports coordination across multiple independently executing triggers.
System	Enables you to access properties that identify the ExtraHop Discover appliance on which a trigger is running.
Trigger	Enables you to access details about a running trigger.
VLAN	Enables you to access information about a VLAN on the network.

Application

The Application class enables you to create new applications and add metrics at the application level. Applications are user-defined, arbitrary groups of traffic. Applications are defined through triggers only; they cannot be defined in the Web UI.

Instance methods

commit(id: *String*): *void*

Creates an application, commits built-in metrics associated with the event to the application, and adds the application to any built-in or custom records committed during the event.

The application ID must be a string. For built-in application metrics, the metrics are committed only once, even if the `commit()` method is called multiple times on the same event.

The following statement creates an application named "myApp" and commits built-in metrics to the application:

```
Application("myApp").commit();
```

Note: The initial call of a `metricAdd*` method on an application enables you to create the application without calling the `commit()` method. For more information, see the *Method Notes* section below.

You can call the `Application.commit` method only on the following events:

Metric types	Event
AAA	`AAA_REQUEST` -and- `AAA_RESPONSE`
CIFS	`CIFS_RESPONSE`
DB	`DB_RESPONSE`
DHCP	`DHCP_REQUEST` -and- `DHCP_RESPONSE`
DNS	`DNS_REQUEST` -and- `DNS_RESPONSE`
FIX	`FIX_REQUEST` -and- `FIX_RESPONSE`
FTP	`FTP_RESPONSE`
HTTP	`HTTP_RESPONSE`
IBMMQ	`IBMMQ_REQUEST` -and- `IBMMQ_RESPONSE`
ICA	`ICA_TICK` -and- `ICA_CLOSE`
Kerberos	`KERBEROS_REQUEST` -and- `KERBEROS_RESPONSE`
LDAP	`LDAP_REQUEST` -and- `LDAP_RESPONSE`
Memcache	`MEMCACHE_REQUEST` -and- `MEMCACHE_RESPONSE`
MongoDB	`MONGODB_REQUEST` -and- `MONGODB_RESPONSE`
NAS	`CIFS_RESPONSE` -and/or- `NFS_RESPONSE`
NetFlow	`NETFLOW_RECORD` Note that the commit will not occur if enterprise IDs are present in the NetFlow record.
NFS	`NFS_RESPONSE`
Redis	`REDIS_REQUEST` -and- `REDIS_RESPONSE`
RTP	`RTP_TICK`
RTCP	`RTCP_MESSAGE`

Metric types	Event
SIP	SIP_REQUEST -and- SIP_RESPONSE
SMTP	SMTP_RESPONSE
SSH	SSH_CLOSE -and- SSH_TICK
SSL	SSL_RECORD -and- SSL_CLOSE
WebSocket	WEBSOCKET_OPEN, WEBSOCKET_CLOSE, and WEBSOCKET_MESSAGE

Note: Calling the `Application.commit()` function on `TCP_OPEN` or `FLOW_CLASSIFY` events is invalid and results in an error. Instead, call the `Flow.addApplication()` method to create and assign an L4 application to the flow. This method also commits built-in metrics to the application for the life of the flow. You can call the `Flow.addApplication()` method on any device event.

The following functions enable you to record custom application metrics:

- `metricAddCount(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDataset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailCount(metric_name:`*`String`*`, key:`*`String | IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailSnap(metric_name:`*`String`*`, key:`*`String | IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailDataset(metric_name:`*`String`*`, key:`*`String | IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailMax(metric_name:`*`String`*`, key:`*`String | IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`])`*`void`*
- `metricAddDetailSampleset(metric_name:`*`String`*`, key:`*`String | IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddMax(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSampleset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSnap(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*

Method notes

- The above methods cannot not be called directly on the Application class. You can only call these methods on specific Application class instances. For example, the following statement is valid:

  ```
  Application("myApp").metricAddCount("requests", 1);
  ```

  However, the following statement is invalid:

  ```
  Application.metricAddCount("requests", 1);
  ```

- If you plan to commit custom metrics to an application, you can create the application without calling the `commit()` method. For example, if the application does not already exist, the following statement creates the application and commits the custom metric to the application:

  ```
  Application("myApp").metricAddCount("requests", 1);
  ```

  Otherwise, the statement adds the custom metric to existing application.

- The `options` object can contain one or both of the following optional properties:

  freq: *Number*

  The number of occurrences of the value passed in to the method. If no value is passed in, the default value is 1. Enables you to simultaneously record multiple occurrences of particular values in a dataset.

  Available only on the `metricAddDataset` and `metricAddDetailDataset` methods.

  highPrecision: *Boolean*

  A flag that enables one-second granularity for the metrics when set to `true`.
- `NaN` is silently discarded when passed as a value to a `metricAdd*` method. `null` is silently discard when passed as a key to a `metricAddDetail*` method.
- All count parameters for `metricAdd*` methods accept only a non-zero, positive signed 64-bit integer.
- Refer to ExtraHop data types for an overview of the data types.

Instance properties

id: *String*

The unique ID of the application, as shown in the ExtraHop Web UI on the page for that application.

Trigger examples

- Example: Create an application container

Buffer

The Buffer class provides access to binary data.

A buffer is an object with the characteristics of an array. Each element in the array is a number between 0 and 255, representing one byte. Each buffer object has a length property (the number of items in an array) and a square bracket operator.

Encrypted payload is not decrypted for TCP and UDP payload analysis.

`UDP_PAYLOAD` requires a matching string but `TCP_PAYLOAD` does not. If you do not specify a matching string for `TCP_PAYLOAD`, the trigger runs one time after the first N bytes of payload.

Instance methods

decode(type: *String*): *String*

Interprets the contents of the buffer and returns a string with one of the following options:

- utf-8
- ucs2
- hex

equals(): *Boolean*

Performs an equality test between buffer objects. The compared buffers are considered equal if the length and content are exactly the same.

slice(start: *Number*, [end: *Number*]): *Buffer*

Returns the specified bytes in a buffer as a new buffer. Bytes are selected starting at the given start argument and ending at (but not including) the end argument.

start: *Number*

Integer that specifies where to start the selection. Use negative numbers to select from the end of a buffer. This is zero-based.

end: ***Number***

Optional integer that specifies where to end the selection. If omitted, all elements from the start position and to the end of the buffer will be selected. Use negative numbers to select from the end of a buffer. This is zero-based.

toString(): ***String***

Converts the buffer to a string.

unpack(format: ***String*****, [offset:** ***Number*****]):** ***Array***

Processes binary or fixed-width data from any buffer object, such as one returned by `HTTP.payload`, `Flow.client.payload`, or `Flow.sender.payload`, according to the given format string and, optionally, at the specified offset.

Returns a JavaScript array that contains one or more unpacked fields and contains the absolute payload byte position +1 of the last byte in the unpacked object. The bytes value can be specified as the offset in further calls to unpack a buffer.

Note:
- `Buffer.unpack` uses big-endian, standard alignment, by default.
- The format does not have to consume the entire buffer.
- Null bytes are not included in unpacked strings. For example: `buf.unpack('4s')[0] - > 'example'`.
- The z format character represents variable-length, null-terminated strings. If the last field is z, the string is produced whether or not the null character is present.
- An exception is throw when all of the fields cannot be unpacked because the buffer does not contain enough data.

The table below displays supported buffer string formats:

Format	C type	JavaScript type	Standard size
x	pad type	no value	
A	struct in6_addr	IPAddress	16
a	struct in_addr	IPAddress	4
b	signed char	string of length 1	1
B	unsigned char	number	1
?	_Bool	boolean	1
h	short	number	2
H	unsigned short	number	2
i	int	number	4
I	unsigned int	number	4
l	long	number	4
L	unsigned long	number	4
q	long long	number	8
Q	unsigned long long	number	8
f	number	number	4
d	double	number	4
s	char[]	string	

Format	C type	JavaScript type	Standard size
z	char[]	string	

Instance Properties

length: *Number*

The number of bytes in the buffer.

Trigger Examples

- Example: Parse NTP with universal payload analysis
- Example: Parse syslog over TCP with universal payload analysis

Device

The Device class enables you to retrieve device attributes and add custom metrics at the device level.

Instance methods

The following method is present only on instances of the Device class:

Device(id: *String*)

Constructor for the device object that accepts one parameter, which is a unique 16-character string ID. If supplied with an ID from an existing device, the constructor creates a copy of that object with all the properties. Committing metrics on this object with the `metricAdd*` functions will persist them in the datastore. For example:

```
myDevice = new Device(Flow.server.device.id);
debug("myDevice MAC: " + myDevice.hwaddr);
```

equals(): *Boolean*

Performs an equality test between Device objects.

The following functions enable you to record device-level custom metrics:

- `metricAddCount(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDataset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailCount(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailSnap(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailDataset(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailMax(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`])`*`void`*
- `metricAddDetailSampleset(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddMax(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSampleset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSnap(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*

Method notes

- Calling a `Device.metricAdd*` method records metrics for both devices on the flow, even if the trigger is assigned to only one device on the flow.
- Calling a `Flow.client.device.metricAdd*` method records metrics only for the client device, regardless of whether the trigger is assigned to the client or the server.
- Calling a `Flow.server.device.metricAdd*` method records metrics only for the server device, regardless of whether the trigger is assigned to the client or the server.
- The `options` object can contain one or both of the following optional properties:

  freq: *Number*

  The number of occurrences of the value passed in to the method. If no value is passed in, the default value is 1. Enables you to simultaneously record multiple occurrences of particular values in a dataset.

  Available only on the `metricAddDataset` and `metricAddDetailDataset` methods.

  highPrecision: *Boolean*

  A flag that enables one-second granularity for the metrics when set to `true`.
- `NaN` is silently discarded when passed as a value to a `metricAdd*` method. `null` is silently discard when passed as a key to a `metricAddDetail*` method.
- All count parameters for `metricAdd*` methods accept only a non-zero, positive signed 64-bit integer.
- Refer to ExtraHop data types for an overview of the data types.

Instance properties

The following properties enable you to retrieve device attributes and are present only on instances of the Device class.

cdpName: *String*

The CDP name associated with the device, if present.

dhcpName: *String*

The DHCP name associated with the device, if present.

discoverTime: *Number*

The last time the capture process discovered the device (not the original discover time), expressed in milliseconds since the epoch (January 1, 1970). Previously discovered devices may be rediscovered by the capture process if they go idle and later become active again, or if the capture process is restarted.

To take trigger action only on the initial discovery of a device, see the `NEW_DEVICE` trigger event discussed in the Discover class.

dnsNames: *Array*

The DNS names associated with the device, if present.

hasTrigger: *Boolean*

The value is `true` if a trigger assigned to the Device object is currently running.

If the trigger is running on an event associated with a Flow object, the `hasTrigger` property value is `true` on at least one of the Device objects in the flow.

The `hasTrigger` property is useful to distinguish device roles. For example, if a trigger is assigned to a group of proxy servers, you can easily determine whether a device is acting as the client or the server, rather than checking for IP addresses or device IDs, such as in the following example:

```
//Event: HTTP_REQUEST
if (Flow.server.device.hasTrigger) {
    // Incoming request
} else {
    // Outgoing request
```

```
}
```

hwaddr: ***String***

The MAC address of the device, if present.

id: ***String***

The 16-character unique ID of the device, as shown in the ExtraHop Web UI on the page for that device.

ipaddrs: ***Array***

An array of IPAddress objects representing the device's known IP addresses. This will always be an array of one IP Address for L3 devices.

isGateway: ***Boolean***

The value is `true` if the device is a gateway.

isL3: ***Boolean***

The value is `true` if the device is an L3 device.

netbiosName: ***String***

The NetBIOS name associated with the device, if present.

vlanId: ***Number***

The VLAN ID for the device.

Trigger Examples

- Example: Monitor CIFS actions on devices
- Example: Track 500-level HTTP responses by customer ID and URI
- Example: Collect response metrics on database queries
- Example: Send discovered device data to a remote syslog server
- Example: Access HTTP header attributes
- Example: Record Memcache hits and misses
- Example: Parse memcache keys
- Example: Parse custom PoS messages with universal payload analysis
- Example: Add metrics to the metric cycle store

Flow

Flow refers to a conversation between two endpoints over a protocol such as TCP, UDP or ICMP. The Flow class provides access to elements of these conversations, such as endpoint IP addresses and age of the flow. The Flow class also contains a flow store designed to pass objects from request to response on the same flow.

Note: You can apply the Flow class on most L7 protocol events, but it is not supported on session or datastore events.

Events

If a flow is associated with an ExtraHop-monitored L7 protocol, events that correlate to the protocol will run in addition to flow events. For example, a flow associated with HTTP will also run the `HTTP_REQUEST` and `HTTP_RESPONSE` events.

FLOW_CLASSIFY

Runs whenever the ExtraHop system initially classifies a flow as being associated with a specific protocol.

Note: For TCP flows, the `FLOW_CLASSIFY` event runs after the `TCP_OPEN` event.

Through a combination of L7 payload analysis, observation of TCP handshakes, and port number-based heuristics, the FLOW_CLASSIFY event identifies the L7 protocol and the device roles for the endpoints in a flow such as client/server or sender/receiver.

The nature of a flow can change over its lifetime, for example, tunneling over HTTP or switching from SMTP to SMTP-TLS. In these cases, FLOW_CLASSIFY runs again after the protocol change.

The FLOW_CLASSIFY event is useful for initiating an action on a flow based on the earliest knowledge of flow information such as the L7 protocol, client/server IP addresses, or sender/receiver ports.

Common actions initiated upon FLOW_CLASSIFY include starting a packet capture through the `captureStart()` method or associating the flow with an application container through the `addApplication()` method.

Additional options are available when you create a trigger that runs on this event. By default, FLOW_CLASSIFY does not run upon flow expiration; however, you can configure a trigger to do so in order to accumulate metrics for flows that were not classified before expiring. See Advanced trigger options for more information.

FLOW_DETACH

Runs when the parser has encountered an unexpected error or has run out of memory and stops following the flow.

FLOW_DETACH can be used to detect malicious content sent by clients and servers. The following is an example of how a trigger can detect bad DNS responses upon FLOW_DETACH events:

```
if (event == "FLOW_DETACH" && Flow.l7proto== "DNS") {
    Flow.addApplication("Malformed DNS");
}
```

FLOW_RECORD

Enables you to record information about a flow at timed intervals. Once FLOW_CLASSIFY has run, the FLOW_RECORD event will run every *N* seconds and whenever a flow closes. The default value for *N*, known as the publish interval, is 30 minutes; the minimum value is 60 seconds. You can set the publish interval from the ExtraHop Admin UI through the Automatic Flow Record Settings.

FLOW_TICK

Enables you to record information about a flow per amount of data or per turn. The FLOW_TICK event will run on every FLOW_TURN or every 128 packets, whichever occurs first. Also, L2 data is reset on every FLOW_TICK event which enables you to add data together at each tick. If counting throughput, collect data from FLOW_TICK events which provide more complete metrics than FLOW_TURN.

FLOW_TICK provides a means to periodically check for certain conditions on the flow, such as zero windows and Nagle delays, and then take an action, such as initiating a packet capture or sending a syslog message.

The following is an example of FLOW_TICK:

```
log("RTT " + Flow.roundTripTime);
Remote.Syslog.info(
  " eh_event=FLOW_TICK" +
  " ClientIP="+Flow.client.ipaddr+
  " ServerIP="+Flow.server.ipaddr+
  " ServerPort="+Flow.server.port+
  " ServerName="+Flow.server.device.dnsNames[0]+
  " RTT="+Flow.roundTripTime);
```

FLOW_TURN

Runs on every TCP or UDP turn. A turn represents one full cycle of a client transferring request data followed by a server transferring a response.

`FLOW_TURN` also exposes a Turn object.

Endpoints

Flow refers to a conversation between two endpoints over a protocol; an endpoint can be one of the following components:

- client
- server
- sender
- receiver

The methods and properties described in this section are called or accessed for a specified endpoint on the flow. For example, to access the `device` property from an HTTP client, the syntax is `Flow.client.device`.

The endpoint that you specify depends on the events associated with the trigger. For example, the `ACTIVEMQ_MESSAGE` event only supports sender and receiver endpoints. The following table displays a list of events that can be associated with a flow and the endpoints supported for each event:

Event	Client / Server	Sender / Receiver
AAA_REQUEST	yes	yes
AAA_RESPONSE	yes	yes
ACTIVEMQ_MESSAGE	no	yes
CIFS_REQUEST	yes	yes
CIFS_RESPONSE	yes	yes
DB_REQUEST	yes	yes
DB_RESPONSE	yes	yes
DHCP_REQUEST	yes	yes
DHCP_RESPONSE	yes	yes
DNS_REQUEST	yes	yes
DNS_RESPONSE	yes	yes
HTTP_REQUEST	yes	yes
HTTP_RESPONSE	yes	yes
IBMMQ_REQUEST	yes	yes
IBMMQ_RESPONSE	yes	yes
ICA_AUTH	yes	no
ICA_CLOSE	yes	no
ICA_OPEN	yes	no
ICA_TICK	yes	no
FIX_REQUEST	yes	yes

Event	Client / Server	Sender / Receiver
FIX_RESPONSE	yes	yes
FLOW_CLASSIFY	yes	no
FLOW_DETACH	yes	no
FLOW_TICK	yes	no
FLOW_TURN	yes	no
FTP_REQUEST	yes	yes
FTP_RESPONSE	yes	yes
HL7_REQUEST	yes	yes
HL7_RESPONSE	yes	yes
ICMP_MESSAGE	no	yes
KERBEROS_REQUEST	yes	yes
KERBEROS_RESPONSE	yes	yes
LDAP_REQUEST	yes	yes
LDAP_RESPONSE	yes	yes
MEMCACHE_REQUEST	yes	yes
MEMCACHE_RESPONSE	yes	yes
MONGODB_REQUEST	yes	yes
MONGODB_RESPONSE	yes	yes
MSMQ_MESSAGE	no	yes
NFS_REQUEST	yes	yes
NFS_RESPONSE	yes	yes
RTCP_MESSAGE	no	yes
RTP_CLOSE	no	yes
RTP_OPEN	no	yes
RTP_TICK	no	yes
SIP_REQUEST	yes	yes
SIP_RESPONSE	yes	yes
SMPP_REQUEST	yes	yes
SMPP_RESPONSE	yes	yes
SMTP_REQUEST	yes	yes
SMTP_RESPONSE	yes	yes
SSL_ALERT	yes	yes
SSL_CLOSE	yes	no
SSL_HEARTBEAT	yes	yes

Event	Client / Server	Sender / Receiver
SSL_OPEN	yes	no
SSL_PAYLOAD	yes	yes
SSL_RECORD	yes	yes
SSL_RENEGOTIATE	yes	no
TCP_CLOSE	yes	no
TCP_OPEN	yes	no
TCP_PAYLOAD	yes	yes
UDP_PAYLOAD	yes	yes
TELNET_MESSAGE	yes	yes
WEBSOCKET_OPEN	yes	no
WEBSOCKET_CLOSE	yes	no
WEBSOCKET_MESSAGE	yes	yes

Endpoint methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `FLOW_RECORD` event. Record commits are not supported on `FLOW_CLASSIFY`, `FLOW_DETACH`, `FLOW_TICK`, or `FLOW_TURN` events.

On a flow, traffic moves in each direction between two endpoints. The `commitRecord()` method only records flow details in one direction, such as from the client to the server. To record details about the entire flow you must call `commitRecord()` twice, once for each direction, and specify the endpoint in the syntax—for example, `Flow.client.commitRecord()` and `Flow.server.commitRecord()`.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

To view the default properties committed to the record object, see the `record` property below.

Endpoint properties

bytes: *Number*

The number of L4 payload bytes transmitted by a device. Specify the device role in the syntax—for example, `Flow.client.bytes` or `Flow.receiver.bytes`.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

customDevices: *Array*

An array of custom devices in the flow. Specify the device role in the syntax—for example, `Flow.client.customDevices` or `Flow.receiver.customDevices`.

device: *Device*

The Device object associated with a device. Specify the device role in the syntax. For example, to access the MAC address of the client device, specify `Flow.client.device.hwaddr`.

equals: *Boolean*

Performs an equality test between Device objects.

dscp: ***Number***

The number representing the last differentiated services code point (DSCP) value of the flow packet.

Specify the device role in the syntax—for example, `Flow.client.dscp` or `Flow.server.dscp`.

dscpBytes: ***Array***

An array that contains the number of L2 bytes for a specific Differentiated Services Code Point (DSCP) value transmitted by a device in the flow. Specify the device role in the syntax—for example, `Flow.client.dscpBytes` or `Flow.server.dscpBytes`.

The value is zero for each entry that has no bytes of the specific DSCP since the last `FLOW_TICK` event.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

dscpName: ***String***

The name associated with the DSCP value transmitted by a device in the flow. The following table displays well-known DSCP names:

Number	Name
8	CS1
10	AF11
12	AF12
14	AF13
16	CS2
18	AF21
20	AF22
22	AF23
24	CS3
26	AF31
28	AF32
30	AF33
32	CS4
34	AF41
36	AF42
38	AF43
40	CS5
44	VA
46	EF
48	CS6
56	CS7

Specify the device role in the syntax—for example, `Flow.client.dscpName` or `Flow.receiver.dscpName`.

dscpPkts: ***Array***

An array that contains the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by a device in the flow. Specify the device role in the syntax—for example, `Flow.client.dscpPkts` or `Flow.server.dscpPkts`.

The value is zero for each entry that has no packets of the specific DSCP since the last `FLOW_TICK` event.

Applies only to `FLOW_TICK` or `FLOW_TURN` events.

fragPkts: ***Number***

The number of packets resulting from IP fragmentation transmitted by a client or server device in the flow. Specify the device role in the syntax—for example, `Flow.client.fragPkts` or `Flow.server.fragPkts`.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

ipaddr: ***IPAddress***

The IPAddress object associated with a device in the flow. Specify the device role in the syntax—for example, `Flow.client.ipaddr` or `Flow.receiver.ipaddr`.

equals: ***Boolean***

Performs an equality test between IPAddress objects.

isAborted: ***Boolean***

The value is `true` if a TCP flow has been aborted through a TCP reset (RST). The flow can be aborted by a device. If applicable, specify the device role in the syntax—for example, `Flow.client.isAborted` or `Flow.receiver.isAborted`.

This condition may be detected in the `TCP_CLOSE` event and in any impacted L7 events (for example, `HTTP_REQUEST` or `DB_RESPONSE`).

Note:
- An L4 abort occurs when a TCP connection is closed with a RST instead of a graceful shutdown.
- An L7 response abort occurs when a connection closes while in the middle of a response. This can be due to a RST, a graceful FIN shutdown, or an expiration.
- An L7 request abort occurs when a connection closes in the middle of a request. This can also be due to a RST, a graceful FIN shutdown, or an expiration.

isShutdown: ***Boolean***

The value is `true` if the device initiated the shutdown of the TCP connection. Specify the device role in the syntax—for example, `Flow.client.isShutdown` or `Flow.receiver.isShutdown`.

l2Bytes: ***Number***

The number of L2 bytes, including the ethernet headers, transmitted by a device in the flow. Specify the device role in the syntax—for example, `Flow.client.l2Bytes` or `Flow.server.l2Bytes`.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

nagleDelay: ***Number***

The number of Nagle delays associated with a device in the flow. Specify the device role in the syntax—for example, `Flow.client.nagleDelay` or `Flow.server.nagleDelay`.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

payload: ***Buffer***

The payload Buffer associated with a device in the flow. Specify the device role in the syntax—for example, `Flow.client.payload` or `Flow.receiver.payload`.

Access only on TCP_PAYLOAD, UDP_PAYLOAD, or SSL_PAYLOAD events or an error will occur.

pkts: *Number*

The number of packets transmitted by a device in the flow. Specify the device role in the syntax—for example, `Flow.client.pkts` or `Flow.server.pkts`.

Access only on FLOW_TICK or FLOW_TURN events or an error will occur.

port: *Number*

The port number associated with a device in the flow. Specify the device role in the syntax—for example, `Flow.client.port` or `Flow.receiver.port`.

rcvWndThrottle: *Number*

The number of receive window throttles sent from a device in the flow. Specify the device role in the syntax—for example, `Flow.client.rcvWndThrottle` or `Flow.server.rcvWndThrottle`.

Access only on FLOW_TICK or FLOW_TURN events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `Flow.commitRecord` on a FLOW_RECORD event. The record object represents data from a single direction on the flow.

The record object contains the following default properties:

- bytes (L3)
- dscpName
- first
- last
- pkts
- proto
- senderAddr
- senderPort
- receiverAddr
- receiverPort
- tcpFlags

Specify the device role in the syntax—for example, `Flow.client.record` or `Flow.server.record`.

Access the record object only on FLOW_RECORD events or an error will occur.

rto: *Number*

The number of retransmission timeouts (RTOs) associated with a device in the flow. Specify the device role in the syntax—for example, `Flow.client.rto` or `Flow.server.rto`.

Access only on FLOW_TICK or FLOW_TURN events or an error will occur.

zeroWnd: *Number*

The number of zero windows sent from a device in the flow. Specify the device role in the syntax—for example, `Flow.client.zeroWnd` or `Flow.server.zeroWnd`.

Access only on FLOW_TICK or FLOW_TURN events or an error will occur.

Methods

addApplication(name: *String*, [turnTiming: *Boolean*]): *void*

Creates an application with the specified name and collects L2-L4 metrics from the flow. The application can be viewed from the Web UI and the metrics are displayed on an L4 page in the

application. A flow can be associated with one or more applications at a given instant; the L2-L4 metrics collected by each application will be the same.

Calling `Flow.addApplication(name)` on a `FLOW_CLASSIFY` event is common on unsupported protocols. For flows on supported protocols with L7 trigger events, it is recommended to call the `Application(name).commit()` method, which collects a larger set of protocol metrics.

The `turnTiming` flag is set to false by default. If set to true, the ExtraHop system collects additional turn timing metrics for the flow. If this flag is omitted, no turn timing metrics are recorded for the application on the associated flow. Turn timing analysis analyzes L4 behavior in order to infer L7 processing times when the monitored protocol follows a client-request, server-response pattern and in which the client sends the first message. "Banner" protocols (where the server sends the first message) and protocols where data flows in both directions concurrently are not recommended for turn timing analysis.

captureStart(name: *String*, [options: *Object*]): *String*

Initiates a Precision Packet Capture (PPCAP) for the flow and returns a unique identifier of the packet capture in the format of a decimal number as a string. Returns `null` if the packet capture fails to start.

name: *String*

The name of the packet capture file.

- The maximum length is 256 characters
- A separate capture is created for each flow.
- Capture files with the same name are differentiated by timestamps.

options: *Object*

The options contained in the capture object. Omit any of the options to indicate unlimited size for that option. All options apply to the entire flow except the "lookback" options which apply only to the part of the flow before the trigger event that started the packet capture.

maxBytes: *Number*

The total maximum number of bytes.

maxBytesLookback: *Number*

The total maximum number of bytes from the lookback buffer. The lookback buffer refers to packets captured before the call to `Flow.captureStart()`.

maxDurationMSec: *Number*

The maximum duration of the packet capture, expressed in milliseconds.

maxPackets: *Number*

The total maximum number of packets.

maxPacketsLookback: *Number*

The maximum number of packets from the lookback buffer. The lookback buffer refers to packets captured before the call to `Flow.captureStart()`.

The following is an example of `Flow.captureStart()`:

```
// EVENT: HTTP_REQUEST
// capture facebook HTTP traffic flows
if (HTTP.uri.indexOf("www.facebook.com") !== -1) {
   var name = "facebook-" + HTTP.uri;
   //packet capture options: capture 20 packets, up to 10 from the
lookback buffer
   var opts = {
      maxPackets: 20,
      maxPacketsLookback: 10
   };
   Flow.captureStart(name, opts);
}
```

Note:
- The `Flow.captureStart()` function call requires that you have a license for precision packet capture.
- You can specify the number of bytes per packet (snaplen) you want to capture when configuring the trigger in the ExtraHop Web UI. This option is available only on some events. See Advanced trigger options for more information.
- Captured files are available in the ExtraHop Admin UI.
- Once the packet capture drive is full, no new captures will be recorded until the user deletes the files manually.
- The maximum file name string length is 256 characters. If the name exceeds 256 characters, it will be truncated and a warning message will be visible in the debug log, but the trigger will continue to execute.
- The capture file size is the whichever maximum is reached first between the `maxPackets` and `maxBytes` options.
- The size of the capture lookback buffer is whichever maximum is reached first between the `maxPacketsLookback` and `maxBytesLookback` options.
- Each passed `max*` parameter will capture up to the next packet boundary.
- If the packet capture was already started on the current flow, `Flow.captureStart()` calls result in a warning visible in the debug log, but the trigger will continue to run.
- There is a maximum of 128 concurrent packet captures in the system. If that limit is reached, subsequent calls to `Flow.captureStart()` will generate a warning visible in the debug log, but the trigger will continue to execute.

captureStop(): *Boolean*

Stops a packet capture that is in progress on the current flow.

commitRecord1(): *void*

Commits a record object to the ExtraHop Explore appliance that represents data sent from `device1` in a single direction on the flow.

You can call this method only on `FLOW_RECORD` events, and each unique record is committed only once for built-in records.

To view the properties committed to the record object, see the `record` property below.

commitRecord2(): *void*

Commits a record object to the ExtraHop Explore appliance that represents data sent from `device2` in a single direction on the flow.

You can call this method only on `FLOW_RECORD` events, and each unique record is committed only once for built-in records.

To view the properties committed to the record object, see the `record` property below.

findCustomDevice(deviceID: *String*): *Device*

Returns a single Device object that corresponds to the specified deviceID parameter if the device is located on either side of the flow. Returns `null` if no corresponding device is found.

getApplications(): *String*

Retrieves all applications associated with the flow.

Properties

The Flow object properties and methods discussed in this section are available to every L7 trigger event associated with the flow.

By default, the ExtraHop system uses loosely-initiated protocol classification, so it will try to classify flows even after the connection was initiated. Loose initiation can be turned off for ports that do not always carry the protocol traffic (e.g., the wildcard port 0). For such flows, `device1`, `port1`, and `ipaddr1` represent

the device with the numerically lower IP address and `device2`, `port2`, and `ipaddr2` represent the device with the numerically higher IP address.

age: *Number*

The time elapsed since the flow was initiated, expressed in seconds.

bytes1: *Number*

The number of L4 payload bytes transmitted by one of two devices in the flow; the other device is represented by `bytes2`. The device represented by `bytes1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

bytes2: *Number*

The number of L4 payload bytes transmitted by one of two devices in the flow; the other device is represented by `bytes1`. The device represented by `bytes2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

customDevices1: *Array*

An array of custom Device objects on a flow. Custom devices on the other side of the flow are available by accessing `customDevices2`. The device represented by `customDevices1` remains consistent for the flow.

customDevices2: *Array*

An array of custom Device objects on a flow. Custom devices on the other side of the flow are available by accessing `customDevices1`. The device represented by `customDevices2` remains consistent for the flow.

device1: *Device*

The Device object associated with one of two devices in the flow; the other device is represented by `device2`. The device represented by `device1` remains consistent for the flow. For example, `Flow.device1.hwaddr` accesses the MAC addresses of this device in the flow.

equals: *Boolean*

Performs an equality test between Device objects.

device2: *Device*

The Device object associated with one of two devices in the flow; the other device is represented by `device1`. The device represented by `device2` remains consistent for the flow. For example, `Flow.device2.hwaddr` accesses the MAC addresses of this device in the flow.

equals: *Boolean*

Performs an equality test between Device objects.

dscp1: *Number*

The number representing the last Differentiated Services Code Point (DSCP) value transmitted by one of two devices in the flow; the other device is represented by `dscp2`. The device represented by `dscp1` remains consistent for the flow.

dscp2: *Number*

The lnumber representing the last Differentiated Services Code Point (DSCP) value transmitted by one of two devices in the flow; the other device is represented by `dscp1`. The device represented by `dscp2` remains consistent for the flow.

dscpBytes1: *Array*

An array that contains the number of L2 bytes for a specific Differentiated Services Code Point (DSCP) value transmitted by one of two devices in the flow; the other device is represented by `dscpBytes2`. The device represented by `dscpBytes1` remains consistent for the flow.

The value is zero for each entry that has no bytes of the specific DSCP since the last `FLOW_TICK` event.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

dscpBytes2: ***Array***

An array that contains the number of L2 bytes for a specific Differentiated Services Code Point (DSCP) value transmitted by one of two devices in the flow; the other device is represented by `dscpBytes1`. The device represented by `dscpBytes2` remains consistent for the flow.

The value is zero for each entry that has no bytes of the specific DSCP since the last `FLOW_TICK` event.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

dcspName1: ***String***

The name associated with the DSCP value transmitted by one of two devices in the flow; the other device is represented by `dscpName2`. The device represented by `dscpName1` remains consistent for the flow.

See the `dscpName` property in the Endpoints section for a list of supported DSCP code names.

dcspName1: ***String***

The name associated with the DSCP value transmitted by one of two devices in the flow; the other device is represented by `dscpName1`. The device represented by `dscpName2` remains consistent for the flow.

See the `dscpName` property in the Endpoints section for a list of supported DSCP code names.

dscpPkts1: ***Array***

An array that contains the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by one of two devices in the flow; the other device is represented by `dscpPkts2`. The device represented by `dscpPkts1` remains consistent for the flow.

The value is zero for each entry that has no packets of the specific DSCP since the last `FLOW_TICK` event.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

dscpPkts2: ***Array***

An array that contains the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by one of two devices in the flow; the other device is represented by `dscpPkts1`. The device represented by `dscpPkts2` remains consistent for the flow.

The value is zero for each entry that has no packets of the specific DSCP since the last `FLOW_TICK` event.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

fragPkts1: ***Number***

The number of packets resulting from IP fragmentation transmitted by one of two devices in the flow; the other device is represented by `fragPkts2`. The device represented by `fragPkts1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

fragPkts2: ***Number***

The number of packets resulting from IP fragmentation transmitted by one of two devices in the flow; the other device is represented by `fragPkts1`. The device represented by `fragPkts2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

id: ***String***

The unique identifier of a Flow record.

ipaddr: ***IPAddress***

TheIPAddress object associated with a device in the flow. Specify the device role in the syntax—for example, `Flow.client.ipaddr` or `Flow.receiver.ipaddr`.

equals: *Boolean*

Performs an equality test between IPAddress objects.

ipproto: *String*

The IP protocol associated with the flow, such as TCP or UDP.

ipver: *String*

The IP version associated with the flow, such as IPv4 or IPv6.

isAborted: *Boolean*

The value is `true` if a TCP flow has been aborted through a TCP reset (RST). The flow can be aborted by a device. If applicable, specify the device role in the syntax—for example, `Flow.client.isAborted` or `Flow.receiver.isAborted`.

This condition may be detected in the `TCP_CLOSE` event and in any impacted L7 events (for example, `HTTP_REQUEST` or `DB_RESPONSE`).

Note:
- An L4 abort occurs when a TCP connection is closed with a RST instead of a graceful shutdown.
- An L7 response abort occurs when a connection closes while in the middle of a response. This can be due to a RST, a graceful FIN shutdown, or an expiration.
- An L7 request abort occurs when a connection closes in the middle of a request. This can also be due to a RST, a graceful FIN shutdown, or an expiration.

isExpired: *Boolean*

The value is `true` if the flow expired at the time of the event.

isShutdown: *Boolean*

The value is `true` if the device initiated the shutdown of the TCP connection. Specify the device role in the syntax—for example, `Flow.client.isShutdown` or `Flow.receiver.isShutdown`.

l2Bytes1: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by one of two devices in the flow; the other device is represented by `l2Bytes2`. The device represented by `l2Bytes1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

l2Bytes2: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by one of two devices in the flow; the other device is represented by `l2Bytes1`. The device represented by `l2Bytes2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

l7proto: *String*

The L7 protocol associated with the flow. For known protocols, the property returns a string representing the protocol name, such as HTTP, DB, Memcache. For lesser-known protocols, the property returns a string formatted as `ipproto:port`—`tcp:13724` or `udp:11258` For custom protocol names, the property returns a string representing the name set through the Protocol Classification section in the Admin UI.

This property is not valid during `TCP_OPEN` events.

nagleDelay1: *Number*

The number of Nagle delays associated with one of two devices in the flow; the other device is represented by `nagleDelay2`. The device represented by `nagleDelay1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

nagleDelay2: ***Number***

The number of Nagle delays associated with one of two devices in the flow; the other device is represented by `nagleDelay1`. The device represented by `nagleDelay2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

payload1: ***Buffer***

The payload Buffer associated with one of two devices in the flow; the other device is represented by `payload2`. The device represented by `payload1` remains consistent for the flow.

Access only on `TCP_PAYLOAD`, `UDP_PAYLOAD`, and `SSL_PAYLOAD` events or an error will occur.

payload2: ***Buffer***

The payload Buffer associated with one of two devices in the flow; the other device is represented by `payload1`. The device represented by `payload2` remains consistent for the flow.

Access only on `TCP_PAYLOAD`, `UDP_PAYLOAD`, or `SSL_PAYLOAD` events or an error will occur.

pkts1: ***Number***

The number of packets transmitted by one of two devices in the flow; the other device is represented by `pkts2`. The device represented by `pkts1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

pkts2: ***Number***

The number of packets transmitted by one of two devices in the flow; the other device is represented by `pkts1`. The device represented by `pkts2` remains consistent for the flow.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

port1: ***Number***

The port number associated with one of two devices in a flow; the other device is represented by `port2`. The device represented by `port1` remains consistent for the flow.

port2: ***Number***

The port number associated with one of two devices in a flow; the other device is represented by `port1`. The device represented by `port2` remains consistent for the flow.

rcvWndThrottle1: ***Number***

The number of receive window throttles sent from one of two devices in the flow; the other device is represented by `rcvWndThrottle2`. The device represented by `rcvWndThrottle1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

rcvWndThrottle2: ***Number***

The number of receive window throttles sent from one of two devices in the flow; the other device is represented by `rcvWndThrottle1`. The device represented by `rcvWndThrottle2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

record1: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `Flow.commitRecord1` on a `FLOW_RECORD` event.

The object represents traffic sent in a single direction from one of two devices in the flow; the other device is represented by the `record2` property. The device represented by the `record1` property remains consistent for the flow.

Access the record object only on `FLOW_RECORD` events or an error will occur.

The record object contains the following default properties:

- bytes (L3)

- dscpName
- first
- last
- pkts
- proto
- senderAddr
- senderPort
- receiverAddr
- receiverPort
- tcpFlags

record2: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `Flow.commitRecord2` on a `FLOW_RECORD` event.

The object represents traffic sent in a single direction from one of two devices in the flow; the other device is represented by the `record1` property. The device represented by the `record2` property remains consistent for the flow.

Access the record object only on `FLOW_RECORD` events or an error will occur.

The record object contains the following default properties:

- bytes (L3)
- dscpName
- first
- last
- pkts
- proto
- senderAddr
- senderPort
- receiverAddr
- receiverPort
- tcpFlags

roundTripTime: *Number*

The median round-trip time (RTT) for the duration of the event, expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

rto1: *Number*

The number of RTOs associated with one of two devices in the flow; the other device is represented by `rto2`. The device represented by `rto1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

rto2: *Number*

The number of RTOs associated with one of two devices in the flow; the other device is represented by `rto1`. The device represented by `rto2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

store: *Object*

The flow store is designed to pass objects from request to response on the same flow. The `store` object is an instance of an empty JavaScript object. Objects can be attached to the store as properties by defining the property key and property value. For example:

```
Flow.store.myobject = "myvalue";
```

For events that occur on the same flow, you can apply the flow store instead of the session table to share information. For example:

```
/* request */
Flow.store.userAgent = HTTP.userAgent;

/* response */
var userAgent = Flow.store.userAgent;
```

Important: Flow store values persist across all requests and responses carried on that flow. When working with the flow store, it is a best practice to set the flow store variable to `null` when its value should not be conveyed to the next request or response. This practice has the added benefit of conserving flow store memory.

Most flow store triggers should have a structure similar to the following example:

```
if (event === 'DB_REQUEST') {
                  if (DB.statement) {
                  Flow.store.stmt = DB.statement;
} else {
                  Flow.store.stmt = null;
}
}
else if (event === 'DB_RESPONSE') {
        var stmt = Flow.store.stmt;
        Flow.store.stmt = null;
        if (stmt) {
                  // Do something with ‘stmt’;
                  // e.g., commit a metric
        }
}
```

vlan: ***Number***

The VLAN number associated with the flow. If no VLAN tag is present, this value is set to `0`.

zeroWnd1: ***Number***

The number of zero windows associated with one of two devices in the flow; the other device is represented by `zeroWnd2`. The device represented by `zeroWnd1` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

zeroWnd2: ***Number***

The number of zero windows associated with one of two devices in the flow; the other device is represented by `zeroWnd1`. The device represented by `zeroWnd2` remains consistent for the flow.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

Trigger Examples

- Example: Monitor CIFS actions on devices
- Example: Track 500-level HTTP responses by customer ID and URI
- Example: Parse custom PoS messages with universal payload analysis
- Example: Parse syslog over TCP with universal payload analysis
- Example: Parse NTP with universal payload analysis
- Example: Track SOAP requests

FlowInterface

The FlowInterface class enables you to retrieve flow interface attributes and to add custom metrics at the interface level.

Methods

The following method is only present on instances of the FlowInterface class:

FlowInterface(id: *string*)

A constructor for the FlowInterface object that accepts a flow interface ID. An error occurs if the flow interface ID does not exist on the ExtraHop appliance.

You can call a FlowInterface method on any event as an instance method through the NetFlow class. You can call a FlowInterface method as a static method only on `NETFLOW_RECORD` events.

- `metricAddCount(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDataset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailCount(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailSnap(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailDataset(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailMax(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`])`*`void`*
- `metricAddDetailSampleset(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddMax(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSampleset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSnap(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*

Method notes

- To add metrics to both ingress and egress interfaces on a NetFlow, you can call a `FlowInterface.metricAdd*` method. Otherwise, you can add metrics to an individual interface by calling a `NetFlow.ingressInterface.metricAdd*` or a `NetFlow.ingressInterface.metricAdd*` method.
- The `metricAddMax` and `metricAddDetailMax` methods commit metrics that preserve a maximum. For instance, the `metricAddMax` method can record maximum values of database server processing times over time.
- The `options` object can contain one or both of the following optional properties:

  freq: *Number*

  The number of occurrences of the value passed in to the method. If no value is passed in, the default value is 1. Enables you to simultaneously record multiple occurrences of particular values in a dataset.

  Available only on the `metricAddDataset` and `metricAddDetailDataset` methods.

  highPrecision: *Boolean*

  A flag that enables one-second granularity for the metrics when set to `true`.
- Parameters that accept a string value will return NULL if information is unavailable or not applicable. Parameters that accept a number value will return NaN if information is unavailable or not applicable.
- `NaN` is silently discarded when passed as a value to a `metricAdd*` method. `null` is silently discard when passed as a key to a `metricAddDetail*` method.

- All count parameters for `metricAdd*` methods accept only a non-zero, positive signed 64-bit integer.
- Refer to ExtraHop data types for an overview of the data types.

Instance properties

id: *String*

A string that uniquely identifies the flow interface.

number: *Number*

The flow interface number reported by the NetFlow record.

FlowNetwork

The FlowNetwork class enables you to retrieve flow network attributes and to add custom metrics at the flow network level.

Methods

The following method is only present on instances of the FlowNetwork class:

FlowNetwork(id: *string*)

A constructor for the FlowNetwork object that accepts a flow network ID. An error occurs if the flow network ID does not exist on the ExtraHop appliance.

You can call a FlowNetwork method on any event as an instance method through the NetFlow class. You can call a FlowNetwork method as a static method only on `NETFLOW_RECORD` events.

The following functions enable you to record custom metrics associated with flow networks:

- `metricAddCount(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDataset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailCount(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailSnap(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailDataset(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddDetailMax(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`])`*`void`*
- `metricAddDetailSampleset(metric_name:`*`String`*`, key:`*`String`* `|` *`IPAddress`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddMax(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSampleset(metric_name:`*`String`*`, val:`*`Number`*`, [options:`*`Object`*`]):`*`void`*
- `metricAddSnap(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*

Method notes

- To add metrics to both network devices on a NetFlow, you can call a `FlowNetwork.metricAdd*` method. Otherwise, you can add metrics to a specific network device by calling a `NetFlow.network.metricAdd*` method.
- The `metricAddMax` and `metricAddDetailMax` methods commit metrics that preserve a maximum. For instance, the `metricAddMax` method can record maximum values of database server processing times over time.
- The `options` object can contain one or both of the following optional parameters:

freq: *Number*

The number of occurrences of the value passed in to the method. If no value is passed in, the default value is 1. Enables you to simultaneously record multiple occurrences of particular values in a dataset.

Available only on the `metricAddDataset` and `metricAddDetailDataset` methods.

highPrecision: *Boolean*

A flag that enables one-second granularity for the metrics when set to `true`.

- `NaN` is silently discarded when passed as a value to a `metricAdd*` method. `null` is silently discard when passed as a key to a `metricAddDetail*` method.
- All count parameters for `metricAdd*` methods accept only a non-zero, positive signed 64-bit integer.
- Refer to ExtraHop data types for an overview of the data types.

Instance properties

id: *String*

A string that uniquely identifies the flow network.

ipaddr: *IPAddress*

The IP address of the management interface on the flow network.

GeoIP

The GeoIP class enables you to retrieve the approximate country-level or city-level location of a specific IPv4 or IPv6 address.

Methods

Methods in this class give precedence to GeoMap Data Source settings configured in the ExtraHop Admin UI.

getCountry(ipaddr: *IPAddress*): *RoughLocation*

Returns country-level detail for the specified IP address in the `RoughLocation` object, which contains the following fields:

countryName: *string*

The name of the country from which the specified IP address originates, such as `United States`. The value is the same as the `countryName` field returned by the `getPreciseLocation` method.

countryCode: *string*

The code associated with the country, according to ISO 3166, such as US. The value is the same as the `countryCode` field returned by the `getPreciseLocation` method.

Field values are obtained from the MaxMind GeoLite2 Country database, which supports both IPv4 and IPv6 lookups.

Returns `null` in any field for which no data is available, or returns a `null` object if all field data is unavailable.

Note: Calling this method in any trigger reserves 20 MB of total RAM, not per trigger, on the ExtraHop Discover appliance and might impact system performance. However, the memory is not reserved if the `getPreciseLocation` method is called in any trigger.

In the following code example, the `getCountry` method is called on each specified event and retrieves rough location data for each client IP address:

```
// ignore if the IP address is non-routable
if (Flow.client.ipaddr.isRFC1918) return;
var results=GeoIP.getCountry(Flow.client.ipaddr);
if (results) {
    countryCode=results.countryCode;
    // log the 2-letter country code of each IP address
    debug ("Country Code is " + results.countryCode);
}
```

getPreciseLocation(ipaddr: *IPAddress*): *PreciseLocation*

Returns city-level detail for the specified IP address in the `PreciseLocation` object, which contains the following fields:

countryName: *string*

The name of the country from which the specified IP address originates, such as `United States`. The value is the same as the `countryName` field returned by the `getCountry` method.

countryCode: *string*

The code associated with the country, according to ISO 3166, such as US. The value is the same as the `countryCode` field returned by the `getCountry` method.

region: *string*

The region, such as a state or province, such as `Washington`.

city: *string*

The city from which the IP address originates, such as `Seattle`.

latitude: *number*

The latitude of the IP address location.

longitude: *number*

The longitude of the of the IP address location.

radius: *number*

The radius, expressed in km, around the longitude and latitude coordinates of the IP address location.

Field values are obtained from the MaxMind GeoLite2 City database, which supports both IPv4 and IPv6 lookups.

Returns `null` in any field for which no data is available, or returns a `null` object if all field data is unavailable.

Note: Calling this method in any trigger reserves 100 MB of total RAM, not per trigger, on the ExtraHop Discover appliance and might impact system performance.

IPAddress

The IPAddress class enables you to retrieve IP address attributes. The IPAddress class is also available as a property for the Flow class.

Methods

IPAddress (ip: *String* | *Number*, [mask: *Number*])

Constructor for the IPAddress class that takes two parameters:

ip: *String*

The IP address string in CIDR format.

mask: *Number*

The subnet mask in a numerical format, representing the number of leftmost '1' bits in the mask (optional).

Instance methods

equals (equals: *IPAddress*): *Boolean*

Performs an equality test between IPAddress objects as shown in the following example:

```
if (Flow.client.ipaddr.toString() === "10.10.10.10")
{ // perform a task }
```

mask (mask: Number): *IPAddress*

Sets the subnet mask of the IPAddress object as shown in the following example:

```
if ((Flow.ipaddr1.mask(24).toString() === "173.194.33.0")||
(Flow.ipaddr2.mask(24).toString() === "173.194.33.0"))
{Flow.setApplication("My L4 App");}
```

The `mask` parameter specifies the subnet mask in a numerical format, representing the number of leftmost '1' bits in the mask (optional).

toJSON(): *string*

Converts the IPAddress object to JSON format.

toString(): *String*

Converts the IPAddress object to a printable string.

Properties

hostNames: *Array of Strings*

An array of hostnames associated with the IPAddress.

isBroadcast: *Boolean*

The value is `true` if the IP address is a broadcast address.

isLinkLocal: *Boolean*

The value is `true` if the IP address is a link local address (169.254.0.0/16).

isMulticast: *Boolean*

The value is `true` if the IP address is a multicast address.

isRFC1918: *Boolean*

The value is `true` if the IP address belongs to one of the RFC1918 private IP ranges (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16). Always returns false for IPv6 addresses.

isV4: *Boolean*

The value is `true` if the IP address is an IPv4 address.

isV6: *Boolean*

The value is `true` if the IP address is an IPv6 address.

Network

The Network class enables you to add custom metrics at the global level.

Methods

The following functions enable you to record custom network metrics:

- `metricAddCount(metric_name:`*`String`*`, count:`*`Number`*`, [options:`*`Object`*`]):`*`void`*

- `metricAddDataset(metric_name:String, val:Number, [options:Object]):void`
- `metricAddDetailCount(metric_name:String, key:String | IPAddress, count:Number, [options:Object]):void`
- `metricAddDetailSnap(metric_name:String, key:String | IPAddress, count:Number, [options:Object]):void`
- `metricAddDetailDataset(metric_name:String, key:String | IPAddress, val:Number, [options:Object]):void`
- `metricAddDetailMax(metric_name:String, key:String | IPAddress, val:Number, [options:Object])void`
- `metricAddDetailSampleset(metric_name:String, key:String | IPAddress, val:Number, [options:Object]):void`
- `metricAddMax(metric_name:String, val:Number, [options:Object]):void`
- `metricAddSampleset(metric_name:String, val:Number, [options:Object]):void`
- `metricAddSnap(metric_name:String, count:Number, [options:Object]):void`

Method notes

- The `options` object can contain one or both of the following optional properties:

  freq: *Number*

  The number of occurrences of the value passed in to the method. If no value is passed in, the default value is 1. Enables you to simultaneously record multiple occurrences of particular values in a dataset.

  Available only on the `metricAddDataset` and `metricAddDetailDataset` methods.

  highPrecision: *Boolean*

  A flag that enables one-second granularity for the metrics when set to `true`.
- `NaN` is silently discarded when passed as a value to a `metricAdd*` method. `null` is silently discard when passed as a key to a `metricAddDetail*` method.
- All count parameters for `metricAdd*` methods accept only a non-zero, positive signed 64-bit integer.
- Refer to ExtraHop data types for an overview of the data types.

Trigger Examples

- Example: Parse syslog over TCP with universal payload analysis
- Example: Record data to a session table
- Example: Track SOAP requests

Session

The Session class provides access to the session table. It is designed to support coordination across multiple independently executing triggers. The session table's global state means any changes by a trigger or external process become visible to all other users of the session table. Because the session table is in-memory, changes are not saved when you restart the ExtraHop appliance or the capture process.

Session table entries can be evicted when the table grows too large or when the configured expiration is reached.

Note:
- The ExtraHop Command appliance cluster nodes do not share their global states. The ECA does not run triggers; it only manages them.
- The ExtraHop Open Data Context API exposes the session table via the management network, enabling coordination with external processes through the memcache protocol.

Events

The Session class is not limited only to the `SESSION_EXPIRE` event. You can apply the Session class to any ExtraHop event.

SESSION_EXPIRE

Runs periodically (in approximately 30 second increments) as long as the session table is in use. When the SESSION_EXPIRE event fires, keys that have expired in the previous 30 second interval are available through the `Session.expiredKeys` property.

The `SESSION_EXPIRE` event is not associated with any particular flow, so triggers on `SESSION_EXPIRE` events cannot commit device metrics through `Device.metricAdd*` methods or `Flow.client.device.metricAdd*` methods. To commit device metrics on this event, you must add Device objects to the session table through the `Device()` instance method.

Methods

add(key: *String*, value*, [options: *Object*]): *

Adds the specified key in the session table. If the key is present, the corresponding value is returned without modifying the key entry in the table. If the key is not present, a new entry is created for the key and value, and the new value is returned.

You can configure an Options object for the specified key.

getOptions(key: *String*): *Object*

Returns the Options object for the specified key. You configure options during calls to `Session.add()`, `Session.modify()`, or `Session.replace()`.

increment(key: *String*, [count: *Number*]): *Number* | *Null*

Looks up the specified key and increments the key value by the specified number. The default count value is 1. Returns the new key value if the call is successful. Returns `null` if the lookup fails. Returns an error if the key value is not a number.

lookup(key: *String*): *

Looks up the specified key in the session table and returns the corresponding value. Returns `null` if the key is not present.

modify(key: *String*, value: *, [options: *Object*]): *

Modifies the specified key value, if the key is present in the session table, and returns the previous value. If the key is not present, no new entry is created.

If changes to the Options object are included, the key options are updated. and old options are merged with new ones. If the `expire` option is modified, the expiration timer is reset.

remove(key: *String*): *

Removes the entry for the given key and returns the associated value.

replace(key: *String*, value: *, [options: *Object*]): *

Updates the entry associated with the given key. If the key is present, update the value and return the previous value. If the key is not present, add the entry and return the previous value (null).

If changes to the Options object is included, the key options are updated, and old options are merged with new ones. If the `expire` option is provided, the expiration timer is reset.

Options

expire: *Number*

The duration after which eviction occurrs, expressed in seconds. If the value is `null` or `undefined`, the entry is evicted only when the session table grows too large.

notify: *Boolean*

Indicates whether the key is available on `SESSION_EXPIRE` events. The default value is false.

priority: *String*

Priority level that determines which entries to evict if the session table grows too large. Valid values are `PRIORITY_LOW`, `PRIORITY_NORMAL`, and `PRIORITY_HIGH`. The default value is `PRIORITY_NORMAL`.

Constants

PRIORITY_LOW: *Number*

Default value is 0.

PRIORITY_NORMAL: *Number*

Default value is 1.

PRIORITY_HIGH: *Number*

Default value is 2.

Properties

expiredKeys :*Array*

An array of objects with the following properties:

age: *Number*

The age of the expired object, expressed in milliseconds. Age is the amount of time elapsed between when the object in the session table was added or modified, and the `SESSION_EXPIRE` event. The age determines whether the key was evicted or expired.

name: *String*

The key of the expired object.

value: *Number* | *String* | *IPAddress* | *Boolean* | *Device*

The value of the entry in the session table.

Expired keys include keys that were evicted because the table grew too large.

The `expiredKeys` property can be accessed only on `SESSION_EXPIRE`events or an error will occur.

Trigger Examples

- Example: Record data to a session table

System

The System class enables you to access properties that identify the ExtraHop Discover appliance on which a trigger is running. This information in useful in environments with multiple Discover appliances.

Properties

uuid: *string*

The universally unique identifier (UUID) of the ExtraHop Discover appliance.

ipaddr: *IPAddress*

The IPAddress object of the primary management interface (Interface 1) on the ExtraHop Discover appliance.

hostname: *string*

The hostname for the ExtraHop Discover appliance configured in the ExtraHop Admin UI.

Trigger

The Trigger class enables you to access details about a running trigger.

Properties

isDebugEnabled: *boolean*

The value is `true` if debugging is enabled for the trigger. The value is determined by the state of the **Enable Debugging** checkbox in the Trigger Configuration window of the ExtraHop Web UI.

VLAN

The VLAN class represents a VLAN on the network.

Instance properties

id: *Number*

Retrieves the numerical ID of a newly discovered VLAN.

Access only on `NEW_VLAN` events, as described in the Discover section, or an error will occur.

The following example retrieves the ID on a new VLAN:

```
var newVlan = VLAN;
Remote.Syslog.notice("eh_event=NEW_VLAN vlan_id=" + newVlan.id);
```

To retrieve the numerical ID of an existing VLAN, you can run `Flow.id` on Flow, TCP, UDP and L7 protocol events.

The following example retrieves the VLAN ID on a `DHCP_REQUEST` event:

```
/*
 * Monitor a set of VLANs to watch for DHCP requests that might
 indicate incorrect network configuration.
 * Relay logs over the Kafka messaging system API.
 */

var staticIpVlanIds = [1, 2, 3];
if(event === 'DHCP_REQUEST')
{
  if(staticIpVlanIds.indexOf(Flow.vlan) > -1)
  {
     Remote.Kafka.send({"topic": "dhcp_violations", "messages":
 [Flow.client.ipaddr, Flow.vlan], "partition": 1});
  }
}
```

Protocol and network data classes

The Trigger API classes in this section enable you to access properties and record metrics from protocol, message, and flow activity that occurs on the ExtraHop appliance.

Class	Description
AAA	Enables you to access properties and record metrics from AAA_REQUEST or AAA_RESPONSE events.
ActiveMQ	Enables you to access properties and record metrics from ACTIVEMQ_MESSAGE events.
CIFS	Enables you to access properties and record metrics from CIFS_REQUEST and CIFS_RESPONSE events.
DB	Enables you to access properties and record metrics metrics from DB_REQUEST and DB_RESPONSE events.
DHCP	Enables you to access properties and record metrics from DHCP_REQUEST and DHCP_RESPONSE events.
DICOM	Enables you to access properties and record metrics from DICOM_REQUEST and DICOM_RESPONSE events.
DNS	Enables you to access properties and record metrics from DNS_REQUEST and DNS_RESPONSE events.
FIX	Enables you to access properties and record metrics from FIX_REQUEST and FIX_RESPONSE events.
FTP	Enables you to access properties and record metrics from FTP_REQUEST and FTP_RESPONSE events.
HL7	Enables you to access properties and record metrics from HL7_REQUEST and HL7_RESPONSE events.
HTTP	Enables you to access properties and record metrics from HTTP_REQUEST and HTTP_RESPONSE events.
IBMMQ	Enables you to access properties and record metrics that are available from IBMMQ_REQUEST and IBMMQ_RESPONSE events.
ICA	Enables you to access properties and record metrics from ICA_OPEN, ICA_AUTH, ICA_TICK, and ICA_CLOSE events.
ICMP	Enables you to access properties and record metrics from ICMP_MESSAGE events.

Class	Description
Kerberos	Enables you to access properties and record metrics from KERBEROS_REQUEST and KERBEROS_RESPONSE events.
LDAP	Enables you to access properties and record metrics from LDAP_REQUEST and LDAP_RESPONSE events.
LLDP	Enables you to access properties and record metrics from LLDP_FRAME events.
Memcache	Enables you to access properties and record metrics from MEMCACHE_REQUEST and MEMCACHE_RESPONSE events.
MongoDB	The MongoDB class enables you to access properties and record metrics from MONGODB_REQUEST and MONGODB_RESPONSE events.
MSMQ	The MSMQ class enables you to access properties and record metrics from MSMQ_MESSAGE event.
NetFlow	Enables you to access properties and record metrics from NETFLOW_RECORD events.
NFS	Enables you to access properties and record metrics from NFS_REQUEST and NFS_RESPONSE events.
POP3	Enables you to access properties and record metrics from POP3_REQUEST and POP3_RESPONSE events.
Redis	Enables you to access properties and record metrics from REDIS_REQUEST and REDIS_RESPONSE events.
RTCP	Enables you to access properties and record metrics from RTCP_MESSAGE events.
RTP	Enables you to access properties and record metrics from RTP_OPEN, RTP_CLOSE, and RTP_TICK events.
SDP	Enables you to access SDP properties from SIP_REQUEST and SIP_RESPONSE events.
SIP	Enables you to access properties and record metrics from SIP_REQUEST and SIP_RESPONSE events.
SMPP	Enables you to access properties and record metrics from SMPP_REQUEST and SMPP_RESPONSE events.
SMTP	Enables you to access properties and record metrics from SMTP_REQUEST and SMTP_RESPONSE events.

Class	Description
SSH	Enables you to access properties and record metrics from SSH_CLOSE, SSH_OPEN and SSH_TICK events.
SSL	Enables you to access properties and record metrics from SSL_OPEN, SSL_CLOSE, SSL_ALERT, SSL_RECORD, SSL_HEARTBEAT, and SSL_RENEGOTIATE events.
TCP	Enables you to access properties and retrieve metrics from TCP_OPEN, TCP_CLOSE, FLOW_TICK and FLOW_TURN events.
Telnet	Enables you to access properties and record metrics from TELNET_MESSAGE events.
Turn	Enables you to access properties and record metrics available on FLOW_TURN events.
UDP	Enables you to access properties and retrieve metrics from FLOW_TICK and FLOW_TURN events.
WebSocket	Enables youto access properties from WEBSOCKET_OPEN, WEBSOCKET_CLOSE, and WEBSOCKET_MESSAGE events.

AAA

The AAA (Authentication, Authorization, and Accounting) class enables you to access properties and record metrics from AAA_REQUEST or AAA_RESPONSE events.

Events

AAA_REQUEST

Runs when the ExtraHop system finishes processing an AAA request .

AAA_RESPONSE

Runs on every AAA response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either an AAA_REQUEST or AAA_RESPONSE event.

The event determines which properties are committed to the record object. To view the default properties committed on each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

authenticator: *String*

The value of the authenticator field (RADIUS only).

avps:*Array*

avpLength: *Number*

The size of the AVP, expressed in bytes. This value includes the AVP header data, as well as the value.

id: *Number*

The numeric ID of the attribute represented as an integer.

isGrouped: *Boolean*

The value is `true` if this is a grouped AVP (Diameter only).

name: *String*

The name for the given AVP.

vendor: *String*

The vendor name for vendor AVPs (Diameter only).

value: *String* | *Array* | *Number*

For single AVPs, a string or numeric value. For grouped AVPs (Diameter only), an array of objects.

isDiameter: *Boolean*

The value is `true` if the request or response is Diameter.

isError: *Boolean*

The value is `true` if the response is an error. To retrieve the error details in Diameter, check `AAA.statusCode`. To retrieve the error details in RADIUS, check the AVP with code 18 (Reply-Message).

Access only on `AAA_RESPONSE` events or an error will occur.

isRadius: *Boolean*

The value is `true` if the request or response is RADIUS.

isRspAborted: *Boolean*

The value is `true` if the `AAA_RESPONSE` event is aborted.

Access only on `AAA_RESPONSE` events or an error will occur.

method: *Number*

The method that corresponds to the command code in either RADIUS or Diameter.

The following table contains valid Diameter command codes:

Command name	Abbr.	Code
AA-Request	AAR	265
AA-Answer	AAA	265
Diameter-EAP-Request	DER	268
Diameter-EAP-Answer	DEA	268
Abort-Session-Request	ASR	274
Abort-Session-Answer	ASA	274
Accounting-Request	ACR	271
Credit-Control-Request	CCR	272
Credit-Control-Answer	CCA	272
Capabilities-Exchange-Request	CER	257

Command name	Abbr.	Code
Capabilities-Exchange-Answer	CEA	257
Device-Watchdog-Request	DWR	280
Device-Watchdog-Answer	DWA	280
Disconnect-Peer-Request	DPR	282
Disconnect-Peer-Answer	DPA	282
Re-Auth-Request	RAR	258
Re-Auth-Answer	RAA	258
Session-Termination-Request	STR	275
Session-Termination-Answer	STA	275
User-Authorization-Request	UAR	300
User-Authorization-Answer	UAA	300
Server-Assignment-Request	SAR	301
Server-Assignment-Answer	SAA	301
Location-Info-Request	LIR	302
Location-Info-Answer	LIA	302
Multimedia-Auth-Request	MAR	303
Multimedia-Auth-Answer	MAA	303
Registration-Termination-Request	RTR	304
Registration-Termination-Answer	RTA	304
Push-Profile-Request	PPR	305
Push-Profile-Answer	PPA	305
User-Data-Request	UDR	306
User-Data-Answer	UDA	306
Profile-Update-Request	PUR	307
Profile-Update-Answer	PUA	307
Subscribe-Notifications-Request	SNR	308
Subscribe-Notifications-Answer	SNA	308
Push-Notification-Request	PNR	309
Push-Notification-Answer	PNA	309
Bootstrapping-Info-Request	BIR	310
Bootstrapping-Info-Answer	BIA	310
Message-Process-Request	MPR	311
Message-Process-Answer	MPA	311
Update-Location-Request	ULR	316

Command name	Abbr.	Code
Update-Location-Answer	ULA	316
Authentication-Information-Request	AIR	318
Authentication-Information-Answer	AIA	318
Notify-Request	NR	323
Notify-Answer	NA	323

The following table contains valid RADIUS command codes:

Command name	Code
Access-Request	1
Access-Accept	2
Access-Reject	3
Accounting-Request	4
Accounting-Response	5
Access-Challenge	11
Status-Server (experimental)	12
Status-Client (experimental)	13
Reserved	255

processingTime: ***Number***

The server processing time, expressed in milliseconds. The value is `NaN` if the timing is invalid.

Access only on `AAA_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `AAA.commitRecord` on either an `AAA_REQUEST` or `AAA_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

AAA_Request	AAA_Response
authenticator	authenticator
clientZeroWnd	clientZeroWnd
method	isError
reqBytes	isRspAborted
reqL2Bytes	method
reqPkts	processingTime
reqRTO	roundTripTime
serverZeroWnd	rspBytes
txId	rspL2Bytes
	rspPkts

AAA_Request	AAA_Response
	rspRTO
	statusCode
	serverZeroWnd
	txId

reqBytes: *Number*

The number of application-level request bytes.

reqL2Bytes: *Number*

The number of request L2 bytes.

reqPkts: *Number*

The number of request packets.

reqRTO: *Number*

The number of request retransmission timeouts (RTOs).

Access only on `AAA_REQUEST` events or an error will occur.

reqZeroWnd: *Number*

The number of zero windows in the request.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: *Number*

The number of application-level response bytes.

rspL2Bytes: *Number*

The number of response L2 bytes.

rspPkts: *Number*

The number of response packets.

rspRTO: *Number*

The number of response retransmission timeouts (RTOs).

Access only on `AAA_RESPONSE` events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

statusCode: *String*

A string representation of the AVP identifier 268 (Result-Code).

Access only on `AAA_RESPONSE` events or an error will occur.

txId: *Number*

A value that corresponds to the hop-by-hop identifier in Diameter and msg-id in RADIUS.

ActiveMQ

The ActiveMQ class enables you to access properties and record metrics from `ACTIVEMQ_MESSAGE` events. ActiveMQ is an implementation of the Java Messaging Service (JMS).

Events

ACTIVEMQ_MESSAGE

Runs on every JMS message processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `ACTIVEMQ_MESSAGE` event.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

correlationId: *String*

The JMSCorrelationID field of the message.

expiration: *Number*

The JMSExpiration field of the message.

msg: *Buffer*

The message body. For TEXT_MESSAGE format messages, this returns the body of the message as a UTF-8 string. For all other message formats, this returns the raw bytes.

msgFormat: *String*

The message format. Possible values are:

- BYTES_MESSAGE
- MAP_MESSAGE
- MESSAGE
- OBJECT_MESSAGE
- STREAM_MESSAGE
- TEXT_MESSAGE
- BLOG_MESSAGE

msgId: *String*

The JMSMessageID field of the message.

persistent: *Boolean*

The value is `true` if the JMSDeliveryMode is PERSISTENT.

priority: *Number*

The JMSPriority field of the message.

- 0 is the lowest priority.
- 9 is the highest priority.
- 0-4 are gradations of normal priority.
- 5-9 are gradations of expedited priority.

properties: *Object*

Zero or more properties attached to the message. The keys are arbitrary strings and the values may be booleans, numbers, or strings.

queue: *String*

The JMSDestination field of the message.

receiverBytes: *Number*

The number of application-level bytes from the receiver.

receiverIsBroker: *Boolean*

The value is `true` if the flow-level receiver of the message is a broker.

receiverL2Bytes: *Number*

The number of L2 bytes from the receiver.

receiverPkts: *Number*

The number of packets from the receiver.

receiverRTO: *Number*

The number of RTOs from the receiver.

receiverZeroWnd: *Number*

The number of zero windows sent by the receiver.

record: *Object*

The record object that was committed to the ExtraHop Explore appliance through a call to `ActiveMQ.commitRecord` on an `ACTIVEMQ_MESSAGE` event.

The record object contains the following default properties:

- correlationId
- expiration
- msgFormat
- msgId
- persistent
- priority
- queue
- receiverBytes
- receiverIsBroker
- receiverL2Bytes
- receiverPkts
- receiverRTO
- receiverZeroWnd
- redeliveryCount
- replyTo
- roundTripTime
- senderBytes
- senderIsBroker
- senderL2Bytes
- senderPkts
- senderRTO
- senderZeroWnd
- timeStamp
- totalMsgLength

redeliveryCount: *Number*

The number of redeliveries.

replyTo: *String*

The JMSReplyTo field of the message, converted to a string.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

senderBytes: *Number*

The number of application-level bytes from the sender.

senderIsBroker: *Boolean*

The value is `true` if the flow-level sender of the message is a broker.

senderL2Bytes: *Number*

The number of L2 bytes from the sender.

senderPkts: *Number*

The number of packets from the sender.

senderRTO: *Number*

The number of RTOs from the sender.

senderZeroWnd: *Number*

The number of zero windows sent by the sender.

timeStamp: *Number*

The time when the message was handed off to a provider to be sent, expressed in GMT. This is the JMSTimestamp field of the message.

totalMsgLength: *Number*

The length of the message, expressed in bytes.

CIFS

The CIFS class enables you to access properties and record metrics from `CIFS_REQUEST` and `CIFS_RESPONSE` events.

Events

CIFS_REQUEST

Runs on every CIFS request processed by the device.

CIFS_RESPONSE

Runs on every CIFS response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `CIFS_RESPONSE` event. Record commits on `CIFS_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

Important: Access time is the time it takes for a CIFS server to receive a requested block. There is no access time for operations that do not access actual block data within a file. Processing time is the time it takes for a CIFS server to respond to the operation requested by the client, such as a metadata retrieval request.

There are no access times for SMB2_CREATE. SMB2_CREATE creates a file that is referenced in the response by an SMB2_FILEID. The referenced file blocks are then read from or written to the NAS-storage device. These file read and write operations are calculated as access times.

accessTime: ***Number***

The amount of time taken by the server to access a file on disk, expressed in milliseconds. For CIFS, this is the time from the first READ command in a CIFS flow until the first byte of the response payload. The value is `NaN` if the measurement or timing is invalid.

Access only on `CIFS_RESPONSE` events or an error will occur.

encryptedBytes: ***Number***

The number of encrypted bytes in the request or response.

error: ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `CIFS_RESPONSE` events or an error will occur.

isCommandCreate: ***Boolean***

The value is `true` if the message contains an SMB file creation command.

isCommandDelete: ***Boolean***

The value is `true` if the message contains an SMB DELETE command.

isCommandFileInfo: ***Boolean***

The value is `true` if the message contains an SMB file info command.

isCommandLock: ***Boolean***

The value is `true` if the message contains an SMB locking command.

isCommandRead: ***Boolean***

The value is `true` if the message contains an SMB READ command.

isCommandRename: ***Boolean***

The value is `true` if the message contains an SMB RENAME command.

isCommandWrite: ***Boolean***

The value is `true` if the message contains an SMB WRITE command.

method: ***String***

The CIFS method. Correlates to the methods listed under the CIFS metric in the ExtraHop Web UI.

payload: ***Buffer***

The Buffer object containing the payload bytes starting from the READ or WRITE command in the CIFS message.

The buffer contains the *N* first bytes of the payload, where *N* is the number of payload bytes specified by the Bytes to Buffer field when the trigger was configured through the ExtraHop WebUI. The default number of bytes is 2048. For more information, see Advanced trigger options.

For larger volumes of payload bytes, the payload might be spread across a series of READ or WRITE commands so that no single trigger event contains the entire requested payload. You can reassemble the payload into a single, consolidated buffer through the `Flow.store` and `payloadOffset` properties.

payloadOffset: ***Number***

The file offset, expressed in bytes, within the `resource` property. The payload property is obtained from the `resource` property at the offset.

processingTime: ***Number***

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `CIFS_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `CIFS.commitRecord` on a `CIFS_RESPONSE` event.

The record object contains the following default properties:

- accessTime
- clientZeroWnd
- error
- isCommandCreate
- isCommandFileInfo
- isCommandLock
- isCommandRead
- isCommandWrite
- method
- processingTime
- reqSize
- reqXfer
- resource
- rspBytes
- rspXfer
- serverZeroWnd
- share
- statusCode
- user
- warning

Access only on `CIFS_RESPONSE` events or an error will occur.

reqBytes: *Number*

The number of L4 request bytes.

Access only on `CIFS_RESPONSE` events or an error will occur.

reqL2Bytes: *Number*

The number of L2 request bytes.

Access only on `CIFS_RESPONSE` events or an error will occur.

reqPkts: *Number*

The number of request packets.

Access only on `CIFS_RESPONSE` events or an error will occur.

reqRTO: *Number*

The number of request retransmission timeouts (RTOs).

Access only on `CIFS_RESPONSE` events or an error will occur.

reqSize: *Number*

The size of the request payload, expressed in bytes.

reqTransferTime: *Number*

The request transfer time, expressed in milliseconds. If the request is contained in a single packet, the transfer time is zero. If the request spans multiple packets, the value is the amount of time between detection of the first CIFS request packet and detection of the last packet by the ExtraHop system. A high value might indicate a large CIFS request or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

Access only on `CIFS_REQUEST` events or an error will occur.

reqZeroWnd: *Number*

The number of zero windows in the request.

resource: ***String***

The share, path, and filename, concatenated together.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `CIFS_RESPONSE` events or an error will occur.

rspBytes: ***Number***

The number of L4 response bytes.

Access only on `CIFS_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of L2 response bytes.

Access only on `CIFS_RESPONSE` events or an error will occur.

rspPkts: ***Number***

The number of response packets.

Access only on `CIFS_RESPONSE` events or an error will occur.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `CIFS_RESPONSE` events or an error will occur.

rspSize: ***Number***

The size of the response payload, expressed in bytes.

Access only on `CIFS_RESPONSE` events or an error will occur.

rspTransferTime: ***Number***

The response transfer time, expressed in milliseconds. If the response is contained in a single packet, the transfer time is zero. If the response spans multiple packets, the value is the amount of time between detection of the first CIFS response packet and detection of the last packet by the ExtraHop system. A high value might indicate a large CIFS response or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

Access only on `CIFS_RESPONSE` events or an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

share: ***String***

The name of the share the user is connected to.

statusCode: ***Number***

The numeric status code of the response (SMB2 only).

Access only on `CIFS_RESPONSE` events or an error will occur.

user: ***String***

The username, if available. In some cases, such as when the login event was not visible or the access was anonymous, the username is not available.

warning: ***String***

The detailed warning message recorded by the ExtraHop system.

Access only on `CIFS_RESPONSE` events or an error will occur.

Trigger Examples

- Example: Monitor CIFS actions on devices

DB

The DB, or database, class enables you to access properties and record metrics from `DB_REQUEST` and `DB_RESPONSE` events.

Events

DB_REQUEST

Runs on every database request processed by the device.

DB_RESPONSE

Runs on every database response processed by the device.

Method

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `DB_RESPONSE` event. Record commits on `DB_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

appName: *String*

The client application name, which is extracted only for MS SQL connections.

correlationId: *Number*

The correlation ID for DB2 applications. The value is `null` for non-DB2 applications.

database: *String*

The database instance. In some cases, such as when login events are encrypted, the database name is not available.

error: *String*

The detailed error messages recorded by the ExtraHop system in string format. If there are multiple errors in one response, the errors are concatenated into one string.

Access only on `DB_RESPONSE` events or an error will occur.

errors: *Array of strings*

The detailed error messages recorded by the ExtraHop system in array format. If there is only a single error in the response, the error is returned as an array containing one string.

Access only on `DB_RESPONSE` events or an error will occur.

isReqAborted: *Boolean*

The value is `true` if the connection is closed before the DB request is complete.

isRspAborted: *Boolean*

The value is `true` if the connection is closed before the DB response is complete.

Access only on `DB_RESPONSE` events or an error will occur.

method: *String*

The database method which correlates to the methods listed under the Database metric in the ExtraHop Web UI.

params: *Array*

An array of remote procedure call (RPC) parameters which are only available for Microsoft SQL, PostgreSQL, and DB2 databases.

The array contains each of the following parameters:

name: *String*

The optional name of the supplied RPC parameter.

value: *String* | *Number*

A text, integer, or time and date field. If the value is not a text, integer, or time and date field, the value is converted into HEX/ASCII form.

The value of the `params` property is the same when accessed on either the `DB_REQUEST` or the `DB_RESPONSE` event.

procedure: *String*

The stored procedure name. Correlates to the procedures listed under the Database methods in the ExtraHop Web UI.

processingTime: *Number*

The server processing time, expressed in milliseconds (equivalent to `rspTimeToFirstByte - reqTimeToLastByte`). The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `DB_RESPONSE` events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `DB.commitRecord` on a `DB_RESPONSE` event.

The record object contains the following default properties:

- appName
- clientZeroWnd
- correlationId
- database
- error
- isReqAborted
- isRspAborted
- method
- procedure
- reqSize
- reqTimeToLastByte
- rspSize
- rspTimeToFirstByte
- rspTimeToLastByte
- processingTime
- serverZeroWnd
- statement
- table
- user

Access only on `DB_RESPONSE` events or an error will occur.

reqBytes: *Number*

The number of L4 request bytes.

Access only on `DB_REQUEST` events or an error will occur.

reqL2Bytes: *Number*

The number of L2 request bytes.

Access only on `DB_REQUEST` events or an error will occur.

reqPkts: ***Number***

The number of request packets.

Access only on `DB_REQUEST` events or an error will occur.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

Access only on `DB_REQUEST` events or an error will occur.

reqSize: ***Number***

The size of the request payload, expressed in bytes

reqTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the request, expressed in milliseconds. Returns NaN on malformed and aborted requests, or if the timing is invalid.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `DB_RESPONSE` events or an error will occur.

rspBytes: ***Number***

The number of L4 response bytes.

Access only on `DB_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of L2 response bytes.

Access only on `DB_RESPONSE` events or an error will occur.

rspPkts: ***Number***

The number of response packets.

Access only on `DB_RESPONSE` events or an error will occur.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `DB_RESPONSE` events or an error will occur.

rspSize: ***Number***

The size of the response payload, expressed in bytes

Access only on `DB_RESPONSE` events or an error will occur.

rspTimeToFirstByte: ***Number***

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `DB_RESPONSE` events or an error will occur.

rspTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `DB_RESPONSE` events or an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

statement: ***String***

The full SQL statement, which might not be available for all database methods.

table: *String*

The name of the database table specified in the current statement. The following databases are supported:

- Sybase
- Sybase IQ
- MySQL
- PostgreSQL
- IBM Informix
- MS SQL TDS
- Oracle TNS
- DB2

Returns an empty field if there is no table name in the request.

user: *String*

The username, if available. In some cases, such as when login events are encrypted, the username is unavailable.

Trigger Examples

- Example: Collect response metrics on database queries
- Example: Create an application container

DHCP

The DHCP class enables you to access properties and record metrics from `DHCP_REQUEST` and `DHCP_RESPONSE` events.

Events

DHCP_REQUEST

Runs on every DHCP request processed by the device.

DHCP_RESPONSE

Runs on every DHCP response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either a`DHCP_REQUEST` or `DHCP_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed on each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

getOption(optionCode: *Number*): *Object*

Accepts a DHCP option code integer as input and returns an object containing the following fields:

code: *Number*

The DHCP option code.

name: *String*

The DHCP option name.

payload: *Number* | *String*

The type of payload returned will be whatever the type is for that specific option such as an IP address, an array of IP addresses, or a buffer object.

Returns `null` if the specified option code is not present in the message.

Properties

clientReqDelay: *Number*

The time elapsed before the client attempts to acquire or renew a DHCP lease, expressed in seconds.

Access only on `DHCP_REQUEST` events or an error will occur.

error: *String*

The error message associated with option code 56. The value is `null` if there is no error.

Access only on `DHCP_RESPONSE` events or an error will occur.

gwAddr: *IPAddress*

The IP address used by routers to relay request and response messages.

htype: *Number*

The hardware type code.

msgType: *String*

The DHCP message type. Supported message types are:

- DHCPDISCOVER
- DHCPOFFER
- DHCPREQUEST
- DHCPDECLINE
- DHCPACK
- DHCPNAK
- DHCPRELEASE
- DHCPINFORM
- DHCPFORCERENEW
- DHCPLEASEQUERY
- DHCPLEASEUNASSIGNED
- DHCPLEASEUNKNOWN
- DHCPLEASEACTIVE
- DHCPBULKLEASEQUERY
- DHCPLEASEQUERYDONE

offeredAddr: *IPAddress*

The IP address the DHCP server is offering or assigning to the client.

Access only on `DHCP_RESPONSE` events or an error will occur.

options: *Array of Objects*

An array of objects with each object containing the following fields:

code: *Number*

The DHCP option code.

name: *String*

The DHCP option name.

payload: ***Number*** **|** ***String***

The type of payload returned will be whatever the type is for that specific option such as an IP address, an array of IP addresses, or a buffer object. IP addresses will be parsed into an array but if the number of bytes is not divisible by 4, it will instead be returned as a buffer.

processingTime: ***Number***

The process time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `DHCP_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `DHCP.commitRecord` on either a `DHCP_REQUEST` or `DHCP_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

DHCP_REQUEST	DHCP_RESPONSE
clientReqDelay	msgType
gwAddr	error
htype	gwAddr
msgType	htype
reqBytes	offeredAddr
reqL2Bytes	processingTime
reqPkts	rspBytes
txId	rspL2Bytes
	rspPkts
	txId

reqBytes: ***Number***

The number of request bytes.

Access only on `DHCP_RESPONSE` events or an error will occur.

reqL2Bytes: ***Number***

The number of request L2 bytes.

Access only on `DHCP_RESPONSE` events or an error will occur.

reqPkts: ***Number***

The number of request packets.

Access only on `DHCP_RESPONSE` events or an error will occur.

rspBytes: ***Number***

The number of L4 response bytes.

Access only on `DHCP_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of L2 response bytes.

Access only on `DHCP_RESPONSE` events or an error will occur.

rspPkts: *Number*

The number of response packets.

Access only on `DHCP_RESPONSE` events or an error will occur.

txId: *Number*

The transaction ID.

DICOM

The DICOM (Digital Imaging and Communications in Medicine) class enables you to access properties and record metrics from `DICOM_REQUEST` and `DICOM_ RESPONSE` events.

Events

DICOM_REQUEST

Runs on every DICOM request processed by the device.

DICOM_RESPONSE

Runs on every DICOM response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `DICOM_REQUEST` or `DICOM_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed on each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

findElement(groupTag: *Number*, elementTag: *Number*): *Buffer*

Returns a buffer that contains the DICOM data element specified by the passed group and element tag numbers.

The data element is represented by a unique ordered pair of integers that represent the group tag and element tag numbers. For example, the ordered pair "0008, 0008" represents the "image type" element. A Registry of DICOM Data Elements and defined tags is available at dicom.nema.org.

groupTag: *Number*

The first number in the unique ordered pair of integers that represent a specific data element.

elementTag: *Number*

The second number in the unique ordered pair or integers that represent a specific data element.

Properties

calledAETitle: *String*

The application entity (AE) title of the destination device or program.

callingAETitle: *String*

The application entity (AE) title of the source device or program.

elements: *Array*

An array of presentation data values (PDV) command elements and data elements that comprise a DICOM message.

error: *String*

The detailed error message recorded by the ExtraHop system.

isReqAborted: *Boolean*

Returns The value is `true` if the connection is closed before the DICOM request is complete.

Access only on `DICOM_REQUEST` events or an error will occur.

isRspAborted: *Boolean*

The value is `true` if the connection is closed before the DICOM response is complete.

Access only on `DICOM_RESPONSE` events or an error will occur.

methods: *Array of Strings*

An array of command fields in the message. Each command field specifies a DIMSE operation name, such as N-CREATE-RSP.

processingTime: *Number*

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `DICOM_RESPONSE` events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `DICOM.commitRecord` on either a `DICOM_REQUEST` or `DICOM_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

DICOM_REQUEST	DICOM_RESPONSE
calledAETitle	calledAETitle
callingAETitle	callingAETitle
clientZeroWnd	clientZeroWnd
error	error
isReqAborted	isRspAborted
method	method
reqPDU	processingTime
reqSize	rspPDU
reqTransferTime	rspSize
serverZeroWnd	rspTransferTime
version	serverZeroWnd
	version

reqBytes: *Number*

The number of application-level request bytes.

Access only on `DICOM_REQUEST` events or an error will occur.

reqL2Bytes: *Number*

The number of L2 request bytes.

reqPDU: *String*

The Protocol Data Unit (PDU), or message format, of the request.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

reqSize: ***Number***

The size of the request, expressed in bytes.

Access only on `DICOM_REQUEST` events or an error will occur.

reqTransferTime: ***Number***

The request transfer time, expressed in milliseconds.

Access only on `DICOM_REQUEST` events or an error will occur.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `DICOM_RESPONSE` events or an error will occur.

rspBytes: ***Number***

The number of application-level response bytes.

Access only on `DICOM_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of L2 response bytes.

rspPDU: ***String***

The Protocol Data Unit (PDU), or message format, of the response.

Access only on `DICOM_RESPONSE` events or an error will occur.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

rspSize: ***Number***

The size of the response, expressed in bytes.

Access only on `DICOM_RESPONSE` events or an error will occur.

rspTransferTime: ***Number***

The response transfer time, expressed in milliseconds.

Access only on `DICOM_RESPONSE` events or an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

version: ***Number***

The DICOM version number.

DNS

The DNS class enables you to access properties and record metrics from `DNS_REQUEST` and `DNS_RESPONSE` events.

Events

DNS_REQUEST

Runs on every DNS request processed by the device.

DNS_RESPONSE

Runs on every DNS response processed by the device.

Methods

answersInclude(term: *String* | *IPAddress*): *Boolean*

Returns `true` if the specified term is present in a DNS response. For string terms, the method checks both the name and data record in the answer section of the response. For IPAddress terms, the method checks only the data record in the answer section.

Can be called only on `DNS_RESPONSE` events.

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `DNS_REQUEST` or `DNS_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed on each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

answers: *Array*

An array of objects corresponding to answer resource records.

Access only on `DNS_RESPONSE` events or an error will occur.

The objects contain the following properties:

data: *String*

The value of data depends on the type. The value is `null` for unsupported record types. Supported record types include:

- A
- AAAA
- NS
- PTR
- CNAME
- MX
- SRV
- SOA
- TXT

name: *String*

The record name.

ttl: *Number*

The time-to-live value.

type: *String*

The DNS record type.

typeNum: *Number*

The numeric representation of the DNS record type.

error: ***String***

The name of the DNS error code, in accordance with IANA DNS parameters, recorded by the ExtraHop system.

Returns OTHER for error codes that are unrecognized by the system; however, `errorNum` specifies the numeric code value.

Access only on `DNS_RESPONSE` events or an error will occur.

errorNum: ***Number***

The numeric representation of the DNS error code in accordance with IANA DNS parameters.

Access only on `DNS_RESPONSE` events or an error will occur.

isAuthoritative: ***Boolean***

The value is `true` if the authoritative answer is set in the response.

Access only on `DNS_RESPONSE` events or an error will occur.

isReqTimeout: ***Boolean***

The value is `true` if the request timed out.

Access only on `DNS_REQUEST` events or an error will occur.

isRspTruncated: ***Boolean***

The value is `true` if the response is truncated.

Access only on `DNS_RESPONSE` events or an error will occur.

opcode: ***String***

The name of the DNS operation code in accordance with IANA DNS parameters. The following codes are recognized by the ExtraHop system:

OpCode	Name
0	Query
1	IQuery (Inverse Query - Obsolete)
2	Status
3	Unassigned
4	Notify
5	Update
6-15	Unassigned

Returns OTHER for codes that are unrecognized by the system; however, the `opcodeNum` property specifies the numeric code value.

opcodeNum: ***Number***

The numeric representation of the DNS operation code in accordance with IANA DNS parameters.

processingTime: ***Number***

The server processing time, expressed in bytes. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `DNS_RESPONSE` events or an error will occur.

qname: ***String***

The hostname queried.

qtype: ***String***

The name of the DNS request record type in accordance with IANA DNS parameters.

Returns OTHER for types that are unrecognized by the system; however, the `qtypeNum` property specifies the numeric type value.

qtypeNum: ***Number***

The numeric representation of the DNS request record type in accordance with IANA DNS parameters.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `DNS.commitRecord` on either a `DNS_REQUEST` or `DNS_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

DNS_REQUEST	DNS_RESPONSE
clientZeroWnd	answers
IsReqTimeout	clientZeroWnd
opcode	error
qname	isAuthoritative
qtype	isRspTruncated
reqBytes	opcode
reqL2Bytes	processingTime
reqPkts	qname
serverZeroWnd	qtype
	rspBytes
	rspL2Bytes
	rspPkts
	serverZeroWnd

reqBytes: ***Number***

The number of application-level request bytes.

Access only on `DNS_REQUEST` events or an error will occur.

reqL2Bytes: ***Number***

The number of request L2 bytes.

Access only on `DNS_REQUEST` events or an error will occur.

reqPkts: ***Number***

The number of request packets.

Access only on `DNS_REQUEST` events or an error will occur.

rspBytes: ***Number***

The number of response bytes.

Access only on `DNS_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of response L2 bytes.

Access only on `DNS_RESPONSE` events or an error will occur.

rspPkts: *Number*

The number of application-level response bytes.

Access only on DNS_RESPONSE events or an error will occur.

FIX

The FIX class enables you to access properties and record metrics from FIX_REQUEST and FIX_RESPONSE events.

Events

FIX_REQUEST

Runs on every FIX request processed by the device.

FIX_RESPONSE

Runs on every FIX response processed by the device.

Note: FIX_RESPONSE is matched with request based on order ID. There is no one-to-one correlation between request and response. There could be requests without a response and sometimes data is pushed to the client. That limits request data availability on response event, however the session table could be used to solve any complex scenarios like submission order id, etc.

Method

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either a FIX_REQUEST or FIX_RESPONSE event.

The event determines which properties are committed to the record object. To view the default properties committed for each event see the record property below.

For built-in records, each unique record is committed only once, even if the commitRecord() method is called multiple times for the same unique record.

Properties

fields: *Array*

A list of FIX fields. Since they are text-based, the key-value protocol fields are exposed as an array of objects with name and value properties containing strings. For example:

```
8=FIX.4.2<SOH>9=233<SOH>35=G<SOH>34=206657...
```

translates to:

```
{"BeginString": "FIX.4.2", "BodyLength": "233", "MsgType": "G",
 "MsgSeqNum":
"206657"}
```

Key string representation is translated, if possible. With extensions, a numeric representation is used. For example, it is not possible to determine 9178=0 (as seen in actual captures). The key is instead translated to "9178". Fields are extracted after message length and version fields are extracted all the way to the checksum (last field). The checksum is not extracted.

For another example, the trigger debug(JSON.stringify(FIX.fields)); shows the following fields:

```
[ {"name":"MsgType","value":"0"},
```

```
{"name":"MsgSeqNum","value":"2"},
{"name":"SenderCompID","value":"AA"},
{"name":"SendingTime","value":"20140904-03:49:58.600"},
{"name":"TargetCompID","value":"GG"}
]
```

To debug and print all FIX fields, enable debugging on the trigger and use the following code:

```
var fields = '';
for (var i = 0; i < FIX.fields.length; i++) {
fields += '"' + FIX.fields[i].name + '" : "' + FIX.fields[i].value +
'"\n';
} debug(fields);
```

The following output prints to the trigger's Runtime Log:

```
"MsgType" : "5"
"MsgSeqNum" : "3"
"SenderCompID" : "GRAPE"
"SendingTime" : "20140905-00:10:23.814"
"TargetCompID" : "APPLE"
```

msgType: ***String***

The value of the MessageCompID key.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `FIX.commitRecord` on either an `FIX_REQUEST` or `FIX_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

FIX_REQUEST	FIX_RESPONSE
clientZeroWnd	clientZeroWnd
msgType	msgType
reqBytes	rspBytes
reqL2Bytes	rspL2Bytes
reqPkts	rspPkts
reqRTO	rspRTO
sender	sender
serverZeroWnd	serverZeroWnd
target	target
version	version

reqBytes: ***Number***

The number of application-level request bytes.

reqL2Bytes: ***Number***

The number of request L2 bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: *Number*

The number of request RTOs.

reqZeroWnd: *Number*

The number of zero windows in the request.

rspBytes: *Number*

The number of application-level response bytes.

rspL2Bytes: *Number*

The number of response L2 bytes.

rspPkts: *Number*

The number of response packets.

rspRTO: *Number*

The number of response RTOs.

rspZeroWnd: *Number*

The number of zero windows in the response.

sender: *String*

The value of the SenderCompID key.

target: *String*

The value of the TargetCompID key.

version: *String*

The protocol version.

FTP

The FTP class enables you to access properties and record metrics from `FTP_REQUEST` and `FTP_RESPONSE` events.

Events

FTP_REQUEST

Runs on every FTP request processed by the device.

FTP_RESPONSE

Runs on every FTP response processed by the device.

Method

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `FTP_RESPONSE` event. Record commits on `FTP_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

args: *String*

The arguments to the command.

Access only on `FTP_RESPONSE` events or an error will occur.

cwd: ***String***

In the case of a user at /, when the client sends "`CWD subdir`":

- FTP.cwd will be / when method == "CWD".
- FTP.cwd will be /subdir for subsequent commands (rather than CWD becoming the changed to directory as part of the CWD response trigger).

Includes "..." at the beginning of the path in the event of a resync or the path is truncated.

Includes "..." at the end of the path if the path is too long. Path truncates at 4096 characters.

Access only on `FTP_RESPONSE` events or an error will occur.

error: ***string***

The detailed error message recorded by the ExtraHop system.

Access only on `FTP_RESPONSE` events or an error will occur.

isReqAborted: ***Boolean***

The value is `true` the connection is closed before the FTP request was complete.

isRspAborted: ***Boolean***

The value is `true` if the connection is closed before the FTP response was complete.

Access only on `FTP_RESPONSE` events or an error will occur.

method: ***String***

The FTP method.

path: ***String***

The path for FTP commands. Includes "..." at the beginning of the path in the event of a resync or the path is truncated. Includes "..." at the end of the path if the path is too long. Path truncates at 4096 characters.

Access only on `FTP_RESPONSE` events or an error will occur.

processingTime: ***Number***

The server processing time, expressed in milliseconds (equivalent to `rspTimeToFirstPayload - reqTimeToLastByte`). The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `FTP_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `FTP.commitRecord` on an `FTP_RESPONSE` event.

The record object contains the following default properties:

- args
- clientZeroWnd
- cwd
- error
- isReqAborted
- isRspAborted
- method
- path
- processingTime
- reqBytes
- reqL2Bytes
- reqPkts
- reqRTO

- roundTripTime
- rspBytes
- rspL2Bytes
- rspPkts
- rspRTO
- serverZeroWnd
- statusCode
- transferBytes
- user

Access the record object only on FTP_RESPONSE events or an error will occur.

reqBytes: *Number*

The number of L4 request bytes.

Access only on FTP_RESPONSE events or an error will occur.

reqL2Bytes: *Number*

The number of L2 request bytes.

Access only on FTP_RESPONSE events or an error will occur.

reqPkts: *Number*

The number of request packets.

Access only on FTP_RESPONSE events or an error will occur.

reqRTO: *Number*

The number of request RTOs.

Access only on FTP_RESPONSE events or an error will occur.

reqZeroWnd: *Number*

The number of zero windows in the request.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is NaN if there are no RTT samples.

Access only on FTP_RESPONSE events or an error will occur.

rspBytes: *Number*

The number of L4 response bytes.

Access only on FTP_RESPONSE events or an error will occur.

rspL2Bytes: *Number*

The number of L2 response bytes.

Access only on FTP_RESPONSE events or an error will occur.

rspPkts: *Number*

The number of response packets.

Access only on FTP_RESPONSE events or an error will occur.

rspRTO: *Number*

The number of response RTOs.

Access only on FTP_RESPONSE events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

statusCode: *Number*

The FTP status code of the response.

Access only on FTP_RESPONSE events or an error will occur.

The following codes are valid:

Code	Description
110	Restart marker replay.
120	Service ready in *nnn* minutes.
125	Data connection already open; transfer starting.
150	File status okay; about to open data connection.
202	Command not implemented, superfluous at this site.
211	System status, or system help reply.
212	Directory status.
213	File status.
214	Help message.
215	NAME system type.
220	Service ready for new user.
221	Service closing control connection.
225	Data connection open; no transfer in progress.
226	Closing data connection. Requested file action successful.
227	Entering Passive Mode.
228	Entering Long Passive Mode.
229	Entering Extended Passive Mode.
230	User logged in, proceed. Logged out if appropriate.
231	User logged out; service terminated.
232	Logout command noted, will complete when transfer done
250	Requested file action okay, completed.
257	"PATHNAME" created.
331	User name okay, need password.
332	Need account for login.
350	Requested file action pending further information.
421	Service not available, closing control connection.
425	Can't open data connection.
426	Connection closed; transfer aborted.
430	Invalid username or password.
434	Requested host unavailable.
450	Requested file action not taken.
451	Requested action aborted. Local error in processing.

Code	Description
452	Requested action not taken.
501	Syntax error in parameters or arguments.
502	Command not implemented.
503	Bad sequence of commands.
504	Command not implemented for that parameter.
530	Not logged in.
532	Need account for storing files.
550	Requested action not taken. File unavailable.
551	Requested action aborted. Page type unknown.
552	Requested file action aborted. Exceeded storage allocation.
553	Requested action not taken. File name not allowed.
631	Integrity protected reply.
632	Confidentiality and integrity protected reply.
633	Confidentiality protected reply.
10054	Connection reset by peer.
10060	Cannot connect to remote server.
10061	Cannot connect to remote server. The connection is active refused.
10066	Directory not empty.
10068	Too many users, server is full.

transferBytes: ***Number***

The number of bytes transferred over the data channel during an `FTP_RESPONSE` event.

Access only on `FTP_RESPONSE` events or an error will occur.

user: ***String***

The user name, if available. In some cases, such as when login events are encrypted, the user name is not available.

HL7

The HL7 class enables you to access properties and record metrics from `HL7_REQUEST` and `HL7_RESPONSE` events.

Events

HL7_REQUEST

Runs on every HL7 request processed by the device.

HL7_RESPONSE

Runs on every HL7 response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `HL7_RESPONSE` event. Record commits on `HL7_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

ackCode: *String*

The two character acknowledgment code.

Access only on `HL7_RESPONSE` events or an error will occur.

ackId: *String*

The identifier for the message being acknowledged.

Access only on `HL7_RESPONSE` events or an error will occur.

msgId: *String*

The unique identifier for this message.

msgType: *String*

The entire message type field, including the msgId subfield.

processingTime: *Number*

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `HL7_RESPONSE` events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `HL7.commitRecord` on an `HL7_RESPONSE` event.

The record object contains the following default properties:

- ackCode
- ackId
- clientZeroWnd
- msgId
- msgType
- roundTripTime
- processingTime
- serverZeroWnd
- version

Access the record object only on `HL7_RESPONSE` events or an error will occur.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `HL7_RESPONSE` events or an error will occur.

segments: *Array*

An array of objects where each object is of type (name: XYZ, fields: array of strings).

subfieldDelimiter: *String*

Supports non-standard field delimiters.

version: *String*

The version advertised in the MSH segment.

Note: The amount of buffered data is limited by the following capture option: `("message_length_max": number)`

HTTP

The HTTP class enables you to access properties and record metrics from `HTTP_REQUEST` and `HTTP_RESPONSE` events.

Events

HTTP_REQUEST

Runs on every HTTP request processed by the device.

HTTP_RESPONSE

Runs on every HTTP response processed by the device.

Additional payload options are available when you create a trigger that runs on either of these events. See Advanced trigger options for more information.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `HTTP_RESPONSE` event. Record commits on `HTTP_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

findHeaders(name: *String*): *Array*

Allows access to HTTP header values and returns an array of header objects (with name and value properties) where the names match the prefix of the string value. See Example: Access HTTP header attributes for more information.

parseQuery(String): *Object*

Accepts a query string and returns an object with names and values corresponding to those in the query string as shown in the following example:

```
var query = HTTP.parseQuery(HTTP.query);
debug("user id: " + query.userid);
```

Properties

age: *Number*

For `HTTP_REQUEST` events, the time from the first byte of the request until the last seen byte of the request. For `HTTP_RESPONSE` events, the time from the first byte of the request until the last seen byte of the response. The time is expressed in milliseconds. Specifies a valid value on malformed and aborted requests. The value is `NaN` on expired requests and responses, or if the timing is invalid.

contentType: *String*

The value of the content-type HTTP header.

cookies: *Array*

An array of objects that represents cookies and contains properties such as "domain" and "expires." The properties correspond to the attributes of each cookie as shown in the following example:

```
var cookies = HTTP.cookies,
    cookie,
    i;
for (i = 0; i < cookies.length; i++) {
    cookie = cookies[i];
    if (cookie.domain) {
        debug("domain: " + cookie.domain);
    }
}
```

headers: *Object*

An array-like object that allows access to HTTP header names and values. Header information is available through one of the following properties:

length: *Number*

The number of headers.

string property:

The name of the header, accessible in a dictionary-like fashion, as shown in the following example:

```
var headers = HTTP.headers;
    session = headers["X-Session-Id"];
    accept = headers.accept;
```

numeric property:

Corresponds to the order in which the headers appear on the wire. The returned object has a name and a value property. Numeric properties are useful for iterating over all the headers and disambiguating headers with duplicate names as shown in the following example:

```
var headers = HTTP.headers;
for (i = 0; i < headers.length; i++) {
    hdr = headers[i];
    debug("headers[" + i + "].name: " + hdr.name);
    debug("headers[" + i + "].value: " + hdr.value);
}
```

Note: Saving `HTTP.headers` to the Flow store does not save all of the individual header values. It is a best practice to save the individual header values to the Flow store. Refer to the Flow class section for details.

headersRaw: *String*

The unmodified block of HTTP headers, expressed as a string.

host: *String*

The value in the HTTP host header.

isDesync: *Boolean*

The value is `true` if the protocol parser became desynchronized due to missing packets.

isEncrypted: *Boolean*

Specifies The value is `true` if the transaction is over secure HTTP.

isPipelined: *Boolean*

The value is `true` if the transaction is pipelined.

isReqAborted: *Boolean*

The value is `true` if the connection is closed before the HTTP request was complete.

isRspAborted: *Boolean*

The value is `true` if the connection is closed before the HTTP response was complete.

Access only on `HTTP_RESPONSE` events or an error will occur.

isRspChunked: *Boolean*

The value is `true` if the response is chunked.

Access only on `HTTP_RESPONSE` events or an error will occur.

isRspCompressed: *Boolean*

The value is `true` if the response is compressed.

isServerPush: *Boolean*

The value is `true` if the transaction is the result of a server push.

method: *String*

The HTTP method of the transaction such as POST and GET.

origin: *IPAddress* | *String*

The value in the X-Forwarded-For or the true-client-ip header.

path: *String*

The path portion of the URI: `/path/`.

payload: *Buffer*

The Buffer object containing the raw payload bytes of the event transaction. If the payload was compressed, the decompressed content is returned.

The buffer contains the *N* first bytes of the payload, where *N* is the number of payload bytes specified by the Bytes to Buffer field when the trigger was configured through the ExtraHop WebUI. The default number of bytes is 2048. For more information, see Advanced trigger options.

The following script is an example of HTTP payload analysis:

```
/* Extract the user name based on a pattern "user=*&" from payload of a
login URI that has "auth/login" as a URI substring. */

if (HTTP.payload && /auth\/login/i.test(HTTP.uri)) {
    var user = /user=(.*?)\&/i.exec(HTTP.payload);
    if (user !== null) {
        debug("user: " + user[1]);
    }
}
```

Note: If two HTTP payload buffering triggers are assigned to the same device, the higher value is used and the value of `HTTP.payload` will be the same for both triggers.

processingTime: *Number*

The server processing time, expressed in milliseconds (equivalent to `rspTimeToFirstPayload - reqTimeToLastByte`). The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `HTTP_RESPONSE` events or an error will occur.

query: *String*

The query string portion of the `URI: query=string`. This typically follows the URL and is separated from it by a question mark. Multiple query strings are separated by an ampersand (&) or semicolon (;) delimiter.

record: *Object*

The record object that was committed to the ExtraHop Explore appliance through a call to `HTTP.commitRecord` on an `HTTP_RESPONSE` event.

The record object contains the following default properties:

- clientZeroWnd
- contentType
- host
- isPipelined
- isReqAborted
- isRspAborted
- isRspChunked
- isRspCompressed
- method
- origin
- query
- referer
- reqBytes
- reqL2Bytes
- reqPkts
- reqRTO
- reqSize
- reqTimeToLastByte
- roundTripTime
- rspBytes
- rspL2Bytes
- rspPkts
- rspRTO
- rspSize
- rspTimeToFirstHeader
- rspTimeToFirstPayload
- rspTimeToLastByte
- rspVersion
- serverZeroWnd
- statusCode
- thinkTime
- title
- processingTime
- uri
- userAgent

Access the record object only on HTTP_RESPONSE events or an error will occur.

referer: *String*

The value in the HTTP referrer header.

reqBytes: *Number*

The number of L4 request bytes.

Access only on HTTP_RESPONSE events or an error will occur.

reqL2Bytes: *Number*

The number of request L2 bytes.

Access only on HTTP_RESPONSE events or an error will occur.

reqPkts: *Number*

The number of request packets.

Access only on HTTP_RESPONSE events or an error will occur.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

Access only on HTTP_RESPONSE events or an error will occur.

reqSize: ***Number***

The size of the request payload, expressed in bytes. The size does not include headers.

reqTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the request, expressed in milliseconds. The value is NaN on expired requests and responses, or if the timing is invalid.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median TCP round-trip time (RTT), expressed in milliseconds. The value is NaN if there are no RTT samples.

Access only on HTTP_RESPONSE events or an error will occur.

rspBytes: ***Number***

The number of response L4 bytes.

Access only on HTTP_RESPONSE events or an error will occur.

rspL2Bytes: ***Number***

The number of response L2 bytes.

Access only on HTTP_RESPONSE events or an error will occur.

rspPkts: ***Number***

The number of response packets.

Access only on HTTP_RESPONSE events or an error will occur.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

Access only on HTTP_RESPONSE events or an error will occur.

rspSize: ***Number***

The size of the response payload, expressed in bytes. The size does not include headers.

Access only on HTTP_RESPONSE events or an error will occur.

rspTimeToFirstHeader: ***Number***

The time from the first byte of the request until the status line that precedes the response headers, expressed in milliseconds. The value is NaN on malformed and aborted responses, or if the timing is invalid.

Access only on HTTP_RESPONSE events or an error will occur.

rspTimeToFirstPayload: ***Number***

The time from the first byte of the request until the first payload byte of the response, expressed in milliseconds. Returns zero value when the response does not contain payload. The value is NaN on malformed and aborted responses, or if the timing is invalid.

Access only on HTTP_RESPONSE events or an error will occur.

rspTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. The value is NaN on malformed and aborted responses, or if the timing is invalid.

Access only on HTTP_RESPONSE events or an error will occur.

rspVersion: *String*

The HTTP version of the response.

Access only on `HTTP_RESPONSE` events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

statusCode: *Number*

The HTTP status code of the response.

Access only on `HTTP_RESPONSE` events or an error will occur.

Note: Returns a status code of 0 if no valid `HTTP_RESPONSE` is received.

streamID: *Number*

The ID of the stream that transferred the resource. Because responses might be returned out of order, this property is required for HTTP/2 transactions to match requests with responses. The value is `1` for the HTTP/1.1 upgrade request and `null` for previous HTTP versions.

title: *String*

The value in the title element of the HTML content, if present.

thinkTime: *Number*

The time elapsed between the server having transferred the response to the client and the client transferring a new request to the server, expressed in milliseconds. The value is `NaN` if there is no valid measurement.

uri: *String*

The URI without a query string: `f.q.d.n/path/`.

userAgent: *String*

The value in the HTTP user-agent header.

Trigger Examples

- Example: Track 500-level HTTP responses by customer ID and URI
- Example: Track SOAP requests
- Example: Access HTTP header attributes
- Example: Record data to a session table
- Example: Create an application container

IBMMQ

The IBMMQ class enables you to access properties and record metrics that are available from `IBMMQ_REQUEST` and `IBMMQ_ RESPONSE` events.

Note: The IBMMQ protocol supports EBCDIC encoding.

Events

IBMMQ_REQUEST

Runs on every IBMMQ request processed by the device.

IBMMQ_RESPONSE

Runs on every IBMMQ response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either an `IBMMQ_REQUEST` or `IBMMQ_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

channel: *String*

The communication channel name.

correlationId: *String*

The IBMMQ correlation ID.

error:*String*

The error string that corresponds to the error code on the wire.

method:*String*

The wire protocol request or response method name.

The following ExtraHop method names differ from the Wireshark method names:

ExtraHop	Wireshark
ASYNC_MSG_V7	ASYNC_MESSAGE
MQCLOSEv7	SOCKET_ACTION
MQGETv7	REQUEST_MSGS
MQGETv7_REPLY	NOTIFICATION

msg: *Buffer*

An instance of the Buffer class for MQPUT, MQPUT1, MQGET_REPLY, ASYNC_MSG_V7, and MESSAGE_DATA messages.

Queue messages that are greater than 32K might be broken into more than one segment. A trigger is run for each segment and only the first segment has a non-null message.

Buffer data can be converted to a printable string through the `toString()` function or formatted through unpack commands.

msgFormat: *String*

The message format.

msgId: *Buffer*

The IBMMQ message ID.

pcfError: *String*

The error string that corresponds to the error code on the wire for the programmable command formats (PCF) channel.

pcfMethod: *String*

The wire protocol request or response method name for the programmable command formats (PCF) channel.

pcfWarning: *String*

The warning string that corresponds to the warning string on the wire for the programmable command formats (PCF) channel.

queue: ***String***

The local queue name. The value is `null` if there is no `MQOPEN`, `MQOPEN_REPLY`, `MQSP1(Open)`, or `MQSP1_REPLY` message.

queueMgr: ***String***

The local queue manager. The value is `null` if there is no `INITIAL_DATA` message at the start of the connection.

record: ***Object***

The record object that was committed to the ExtraHop Explore appliance through a call to `IBMMQ.commitRecord` on either an `IBMMQ_REQUEST` or `IBMMQ_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

IBMMQ_REQUEST	IBMMQ_RESPONSE
channel	channel
clientZeroWnd	clientZeroWnd
correlationId	correlationId
msgId	error
method	msgId
msgFormat	method
msgSize	msgFormat
queue	msgSize
queueMgr	queue
reqBytes	queueMgr
reqL2Bytes	resolvedQueue
reqPkts	resolvedQueueMgr
reqRTO	roundTripTime
resolvedQueue	rspBytes
resolvedQueueMgr	rspL2Bytes
serverZeroWnd	rspPkts
	rspRTO
	serverZeroWnd
	warning

reqBytes: ***Number***

The number of application-level request bytes.

reqL2Bytes: ***Number***

The number of L2 request bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

reqZeroWnd: ***Number***

The number of zero windows in the request.

resolvedQueue: ***String***

The resolved queue name from `MQGET_REPLY`, `MQPUT_REPLY`, or `MQPUT1_REPLY` messages. If the queue is remote, the value is different than the value returned by `IBMMQ.queue`.

resolvedQueueMgr: ***String***

The resolved queue manager from `MQGET_REPLY`, `MQPUT_REPLY`, or `MQPUT1_REPLY`. If the queue is remote, the value is different than the value returned by `IBMMQ.queueMgr`.

rfh: ***Array of Strings***

An array of strings located in the optional rules and formatting header (RFH). If there is no RFH header or the header is empty, the array will be empty.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: ***Number***

The number of application-level response bytes.

rspL2Bytes: ***Number***

The number of L2 response bytes.

rspPkts: ***Number***

The number of request packets.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

rspZeroWnd: ***Number***

The number of zero windows in the response.

totalMsgLength: ***Number***

The total length of the message, expressed in bytes.

warning: ***String***

The warning string that corresponds to the warning string on the wire.

Trigger Examples

- Example: Collect IBMMQ metrics

ICA

The ICA class enables you to access properties and record metrics from `ICA_OPEN`, `ICA_AUTH`, `ICA_TICK`, and `ICA_ CLOSE` events.

Events

ICA_AUTH

Runs when the ICA authentication is complete.

ICA_CLOSE

Runs when the ICA session is closed.

ICA_OPEN

Runs immediately after the ICA application is initially loaded.

ICA_TICK

Runs periodically while the user interacts with the ICA application.

After the ICA_OPEN event has run at least once, the ICA_TICK event is run anytime latency is reported and returned by the clientLatency or networkLatency properties described below.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either an ICA_OPEN, ICA_TICK, or ICA_CLOSE event. Record commits on ICA_AUTH events are not supported.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the record property below.

For built-in records, each unique record is committed only once, even if the commitRecord() method is called multiple times for the same unique record.

Properties

application: *String*

The name of the application being launched.

authDomain: *String*

The Windows authentication domain to which the user belongs.

channels: *Array*

An array of objects containing information about virtual channels observed since the last ICA_TICK event.

Access only on ICA_TICK events or an error will occur.

Each object contains the following properties:

name: *String*

The name of the virtual channel.

description: *String*

The friendly description of the channel name.

clientBytes: *Number*

The number of bytes sent by the client for that channel.

serverBytes: *Number*

The number of bytes sent by the server for the channel.

clientMachine: *String*

The name of the client machine. This is a name that is advertised by the ICA client and is usually the hostname of the client machine.

clientBytes: *Number*

Upon an ICA_CLOSE event, the incremental number of application-level client bytes observed since the last ICA_TICK event. Does not specify the total number of bytes for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

clientCGPMsgCount: *Number*

The number of client CGP messages since the last ICA_TICK event.

Access only on ICA_TICK events or an error will occur.

clientLatency: *Number*

The latency of the client, expressed in milliseconds, as reported by End User Experience Management (EUEM) beacon.

Client latency is reported when a packet from the client on the EUEM channel reports the result of a single ICA round-trip measurement.

Access only on ICA_TICK events or an error will occur.

clientL2Bytes: ***Number***

Upon an ICA_CLOSE event, the incremental number of L2 client bytes observed since the last ICA_TICK event. Does not specify the total number of bytes for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

clientMsgCount: ***Number***

The number of client messages since the last ICA_TICK event.

Access only on ICA_TICK events or an error will occur.

clientPkts: ***Number***

Upon an ICA_CLOSE event, the incremental number of client packets observed since the last ICA_TICK event. Does not specify the total number of packets for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

clientRTO: ***Number***

Upon an ICA_CLOSE event, the incremental number of client retransmission timeouts (RTOs) observed since the last ICA_TICK event. Does not specify the total number of RTOs for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

clientZeroWnd: ***Number***

The number of zero windows sent by the client.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

clientType: ***String***

The type of the ICA client which is the user-agent equivalent to ICA.

clipboardData: ***Buffer***

The buffer object that contains raw data from the clipboard transfer.

The value is null if the ICA_TICK event did not result from a clipboard data transfer, or if or if the channel specified by the tickChannel property is not a clipboard channel.

The maximum number of bytes in the buffer is specified by the Clipboard Bytes to Buffer field when the trigger was configured through the ExtraHop Web UI. The default maximum object size is 1024 bytes. For more information, see the Advanced trigger options.

To determine the direction of the clipboard data transfer, access this property through Flow.sender, Flow.receiver, Flow.client, or Flow.server.

Access only on ICA_TICK events or an error will occur.

clipboardDataType: ***String***

The type of data on the clipboard transfer. The following clipboard types are supported:

- TEXT
- BITMAP
- METAFILEPICT
- SYMLINK
- DIF
- TIFF
- OEMTEXT
- DIB
- PALLETTE
- PENDATA
- RIFF
- WAVE

- UNICODETEXT
- EHNMETAFILE
- OWNERDISPLAY
- DSPTEXT
- DSPBITMAP
- DSPMETAFILEPICT
- DSPENHMETAFILE

The value is `null` if the `ICA_TICK` event did not result from a clipboard data transfer, or if or if the channel specified by the `tickChannel` property is not a clipboard channel.

Access only on `ICA_TICK` events or an error will occur.

frameCutDuration: ***Number***

The frame cut duration, as reported by EUEM beacon.

Access only on `ICA_TICK` events or an error will occur.

frameSendDuration: ***Number***

The frame send duration, as reported by EUEM beacon.

Access only on `ICA_TICK` events or an error will occur.

host: ***String***

The host name of the Citrix server.

isAborted: ***Boolean***

The value is `true` if the application fails to launch successfully.

Access only on `ICA_CLOSE` events or an error will occur.

isCleanShutdown: ***Boolean***

The value is `true` if the application shuts down cleanly.

Access only on `ICA_CLOSE` events or an error will occur.

isClientDiskRead: ***Boolean***

The value is `true` if a file was read from the client disk to the Citrix server. The value is `null` if the command is not a file operation, or if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events or an error will occur.

isClientDiskWrite: ***Boolean***

The value is `true` if a file was written from the Citrix server to the client disk. The value is `null` if the command is not a file operation, or if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events or an error will occur.

isEncrypted: ***Boolean***

The value is `true` if the application is encrypted with RC5 encryption.

isSharedSession: ***Boolean***

The value is `true` if the application is launched over an existing connection.

launchParams: ***String***

The string that represents the parameters.

loadTime: ***Number***

The load time of the given application, expressed in milliseconds.

Note: The load time is recorded only for the initial application load. The ExtraHop system does not measure load time for applications launched over existing sessions and instead reports the initial load time on subsequent application loads. Choose

`ICA.isSharedSession` to distinguish between initial and subsequent application loads.

loginTime: *Number*

The user login time, expressed in milliseconds.

Access only on `ICA_OPEN`, `ICA_CLOSE`, or `ICA_TICK` events or an error will occur.

Note: The login time is recorded only for the initial application load. The ExtraHop system does not measure login time for applications launched over existing sessions and instead reports the initial login time on subsequent application loads. Choose `ICA.isSharedSession` to distinguish between initial and subsequent application loads.

networkLatency: *Number*

The current latency advertised by the client, expressed in milliseconds.

Network latency is reported when a specific ICA packet from the client contains latency information.

Access only on `ICA_TICK` events or an error will occur.

payload: *Buffer*

The Buffer object containing the raw payload bytes of the file that was read or written on the event.

The buffer contains the *N* first bytes of the payload, where *N* is the number of payload bytes specified by the Bytes to Buffer field when the trigger was configured through the ExtraHop WebUI. The default number of bytes is 2048. For more information, see Advanced trigger options.

The value is `null` if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events or an error will occur.

program: *String*

The name of the program, or application, that is being launched.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `ICA.commitRecord` on either an `ICA_OPEN`, `ICA_TICK`, or `ICA_CLOSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

ICA_CLOSE	ICA_OPEN	ICA_TICK
authDomain	authDomain	authDomain
clientBytes	clientMachine	
clientL2Bytes	clientType	clientBytes
clientMachine	clientZeroWnd	clientCGPMsgCount
clientPkts	host	clientL2Bytes
clientRTO	isEncrypted	clientLatency
clientType	isSharedSession	clientMachine
clientZeroWnd	launchParams	clientMsgCount
host	loadTime	clientPkts
isAborted	loginTime	clientRTO
isCleanShutdown	program	clientType
isEncypted	serverZeroWnd	clientZeroWnd

ICA_CLOSE	ICA_OPEN	ICA_TICK
isSharedSession	user	frameCutDuration
launchParams		frameSendDuration
loadTime		host
loginTime		isClientDiskRead
program		isClientDiskWrite
roundTripTime		isEncrypted
serverBytes		isSharedSession
serverL2Bytes		launchParams
serverPkts		loadTime
serverRTO		loginTime
serverZeroWnd		networkLatency
user		program
		resource
		roundTripTime
		serverBytes
		serverCGPMsgCount
		serverL2Bytes
		serverMsgCount
		serverPkts
		serverRTO
		serverZeroWnd
		tickChannel
		user

Access the record object only on `ICA_OPEN`, `ICA_CLOSE`, and `ICA_TICK` events or an error will occur.

resource: ***String***

The path of the file that was read or written on the event, if known. The value is `null` if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events or an error will occur.

resourceOffset: ***Number***

The offset of the file that was read or written on the event, if known. The value is `null` if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events or an error will occur.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `ICA_CLOSE` or `ICA_TICK` events or an error will occur.

serverBytes: *Number*

Upon an ICA_CLOSE event, the incremental number of application-level server bytes observed since the last ICA_TICK event. Does not specify the total number of bytes for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

serverCGPMsgCount: *Number*

The number of CGP server messages since the last ICA_TICK event.

Access only on ICA_TICK events or an error will occur.

serverL2Bytes: *Number*

Upon an ICA_CLOSE event, the incremental number of L2 server bytes observed since the last ICA_TICK event. Does not specify the total number of bytes for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

serverMsgCount: *Number*

The number of server messages since the last ICA_TICK event.

Access only on ICA_TICK events or an error will occur.

serverPkts: *Number*

Upon an ICA_CLOSE event, the incremental number of server packets observed since the last ICA_TICK event. Does not specify the total number of packets for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

serverRTO: *Number*

Upon an ICA_CLOSE event, the incremental number of server retransmission timeouts (RTOs) observed since the last ICA_TICK event. Does not specify the total number of RTOs for the session.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

serverZeroWnd: *Number*

The number of zero windows sent by the server.

Access only on ICA_CLOSE or ICA_TICK events or an error will occur.

tickChannel: *String*

The name of the virtual channel that resulted in the current ICA_TICK event. The following channels are supported:

- CTXCLI: clipboard
- CTXCDM: file
- CTXEUE: end user experience monitoring

Access only on ICA_TICK events or an error will occur.

user: *String*

The name of the user, if available.

ICMP

The ICMP class enables you to access properties and record metrics from ICMP_MESSAGE events.

Events

ICMP_MESSAGE

Runs on every ICMP message processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `ICMP_MESSAGE` event.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

gwAddr: *IPAddress*

For a redirect message, returns the address of the gateway to which traffic for the network specified in the internet destination network field of the original datagram's data should be sent. Returns null for all other messages.

Message	ICMPv4 Type	ICMPv6 Type
Redirect Message	5	n/a

hopLimit: *Number*

The ICMP packet time to live or hop count.

isError: *Boolean*

The value is `true` for message types in the following table.

Message	ICMPv4 Type	ICMPv6 Type
Destination Unreachable	3	1
Redirect	5	n/a
Source Quench	4	n/a
Time Exceeded	11	3
Parameter Problem	12	4
Packet Too Big	n/a	2

isQuery: *Boolean*

The value is `true` for message types in the following table.

Message	ICMPv4 Type	ICMPv6 Type
Echo Request	8	128
Information Request	15	n/a
Timestamp request	13	n/a
Address Mask Request	17	n/a
Router Discovery	10	151
Multicast Listener Query	n/a	130
Router Solicitation (NDP)	n/a	133
Neighbor Solicitation	n/a	135
ICMP Node Information Query	n/a	139
Inverse Neighbor Discovery Solicitation	n/a	141

Message	ICMPv4 Type	ICMPv6 Type
Home Agent Address Discovery Solicitation	n/a	144
Mobile Prefix Solicitation	n/a	146
Certification Path Solicitation	n/a	148

isReply: *Boolean*

The value is `true` for message types in the following table.

Message	ICMPv4 Type	ICMPv6 Type
Echo Reply	0	129
Information Reply	16	n/a
Timestamp Reply	14	n/a
Address Mask Reply	18	n/a
Multicast Listener Done	n/a	132
Multicast Listener Report	n/a	131
Router Advertisement (NDP)	n/a	134
Neighbor Advertisement	n/a	136
ICMP Node Information Response	n/a	140
Inverse Neighbor Discovery Advertisement	n/a	142
Home Agent Address Discovery Reply Message	n/a	145
Mobile Prefix Advertisement	n/a	147
Certification Path Advertisement	n/a	149

msg: *Buffer*

A buffer object containing up to `message_length_max` bytes of the ICMP message. The `message_length_ max` option is configured in the ICMP profile in the running config.

The following running config example changes the ICMP `message_length_ max` from its default of 4096 bytes to 1234 bytes:

```
"capture": {
    "app_proto": {
        "ICMP": {
            "message_length_max": 1234
         }
     }
}
```

msgCode: *Number*

The ICMP message code.

msgID: *Number*

The ICMP message identifier for Echo Request, Echo Reply, Timestamp Request, Timestamp Reply, Information Request, and Information Reply messages. The value is `null` for all other message types.

The following table displays type IDs for the ICMP messages:

Message	ICMPv4 Type	ICMPv6 Type
Echo Request	8	128
Echo Reply	0	129
Timestamp Request	13	n/a
Timestamp Reply	14	n/a
Information Request	15	n/a
Information Reply	16	n/a

msgLength: ***Number***

The length of the ICMP message, expressed in bytes.

msgText: ***String***

The descriptive text for the message (e.g., echo request or port unreachable).

msgType: ***Number***

The ICMP message type.

The following table displays the ICMPv4 message types available:

Type	Message
0	Echo Reply
1 and 2	Reserved
3	Destination Unreachable
4	Source Quench
5	Redirect Message
6	Alternate Host Address (deprecated)
7	Reserved
8	Echo Request
9	Router Advertisement
10	Router Solicitation
11	Time Exceeded
12	Parameter Problem: Bad IP header
13	Timestamp
14	Timestamp Reply
15	Information Request (deprecated)
16	Information Reply (deprecated)
17	Address Mask Request (deprecated)
18	Address Mask Reply (deprecated)
19	Reserved
20-29	Reserved
30	Traceroute (deprecated)

Type	Message
31	Datagram Conversion Error (deprecated)
32	Mobile Host Redirect (deprecated)
33	Where Are You (deprecated)
34	Here I Am (deprecated)
35	Mobile Registration Request (deprecated)
36	Mobile Registration Reply (deprecated)
37	Domain Name Request (deprecated)
38	Domain Name Reply (deprecated)
39	Simple Key-Management for Internet Protocol (deprecated)
40	Photuris (deprecated)
41	ICMP experimental
42-255	Reserved

The following table displays the ICMPv6 message types available:

Type	Message
1	Destination Unreachable
2	Packet Too Big
3	Time Exceeded
4	Parameter Problem
100	Private Experimentation
101	Private Experimentation
127	Reserved for expansion of ICMPv6 error messages
128	Echo Request
129	Echo Reply
130	Multicast Listener Query
131	Multicast Listener Report
132	Multicast Listener Done
133	Router Solicitation
134	Router Advertisement
135	Neighbor Solicitation
136	Neighbor Advertisement
137	Redirect Message
138	Router Renumbering
139	ICMP Node Information Query
140	ICMP Node Information Response

Type	Message
141	Inverse Neighbor Discovery Solicitation Message
142	Inverse Neighbor Discovery Advertisement Message
143	Multicast Listener Discovery (MLDv2) reports
144	Home Agent Address Discovery Request Message
145	Home Agent Address Discovery Reply Message
146	Mobile Prefix Solicitation
147	Mobile Prefix Advertisement
148	Certification Path Solicitation
149	Certification Path Advertisement
151	Multicast Router Advertisement
152	Multicast Router Solicitation
153	Multicast Router Termination
155	RPL Control Message
200	Private Experimentation
201	Private Experimentation
255	Reserved for expansion of ICMPv6 informational messages

nextHopMTU: ***Number***

An ICMPv4 `Destination Unreachable` or an ICMPv6 `Packet Too Big` message, the maximum transmission unit of the next-hop link. The value is `null` for all other messages.

Message	ICMPv4 Type	ICMPv6 Type
Destination Unreachable	3	n/a
Packet Too Big	n/a	2

original: ***Object***

An object containing the following elements from the IP datagram that caused the ICMP message to be sent:

ipproto: ***String***

The IP protocol of the datagram, such as TCP, UDP, ICMP, or ICMPv6.

ipver: ***String***

The IP version of the datagram, such as IPv4 or IPv6.

srcAddr: ***IPAddress***

The IPAddress of the datagram sender.

srcPort: ***Number***

The port number of the datagram sender.

dstAddr: ***IPAddress***

The IPAddress of the datagram receiver.

dstPort: ***Number***

The port number of the datagram receiver.

The value is `null` if the internet header and 64 bits of the Original Data datagram is not present in the message or if the IP protocol is not TCP or UDP.

Access only on `ICMP_MESSAGE` events or an error will occur.

pointer: *Number*

For a Parameter Problem message, the octet of the original datagram's header where the error was detected. The value is `null` for all other messages.

Message	ICMPv4 Type	ICMPv6 Type
Parameter Problem	12	4

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `ICMP.commitRecord` on an `ICMP_MESSAGE` event.

The record object contains the following default properties:

- gwAddr
- hopLimit
- msgCode
- msgId
- msgLength
- msgText
- msgType
- nextHopMTU
- pointer
- seqNum
- version

seqNum: *Number*

The ICMP sequence number for Echo Request, Echo Reply, Timestamp Request, Timestamp Reply, Information Request, and Information Reply messages. The value is `null` for all other messages.

version: *Number*

The version of the ICMP message type, which can be ICMPv4 or ICMPv6.

Kerberos

The Kerberos class enables you to access properties and record metrics from `KERBEROS_REQUEST` and `KERBEROS_ RESPONSE` events.

Events

KERBEROS_REQUEST

Runs on every Kerberos AS-REQ and TGS-REQ message type processed by the device.

KERBEROS_RESPONSE

Runs on every Kerberos AS-REP and TGS-REP message type processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either a `KERBEROS_REQUEST` or `KERBEROS_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

addresses: *Array of Objects*

The addresses from which the requested ticket is valid.

Access only on KERBEROS_REQUEST events or an error will occur.

cNames: *Array of Strings*

The name portions of the principal identifier.

cNameType: *String*

The type for the cNames field.

cRealm: *String*

The client realm.

error: *String*

The error returned.

Access only on KERBEROS_RESPONSE events or an error will occur.

eType: *Array of Numbers*

An array of the preferred encryption methods.

Access only on KERBEROS_REQUEST events or an error will occur.

from: *String*

In AS_REQ and TGS_REQ message types, the time when the requested ticket is to be postdated to.

Access only on KERBEROS_REQUEST events or an error will occur.

kdcOptions: *Object*

An object containing booleans for each option flag in AS_REQ and TGS_REQ messages.

Access only on KERBEROS_REQUEST events or an error will occur.

msgType: *String*

The message type. Possible values are:

- AP_REP
- AP_REQ
- AS_REP
- AS_REQAUTHENTICATOR
- ENC_AS_REP_PART
- ENC_KRB_CRED_PART
- ENC_KRB_PRIV_PART
- ENC_P_REP_PART
- ENC_TGS_REP_PART
- ENC_TICKET_PART
- KRB_CRED
- KRB_ERROR
- KRB_PRIV
- KRB_SAFE
- TGS_REP
- TGS_REQ

- TICKET

paData: ***Array of Objects***

The pre-authentication data.

processingTime: ***Number***

The processing time, expressed in milliseconds.

Access only on KERBEROS_RESPONSE events or an error will occur.

realm: ***String***

The server realm. In an AS_REQ message type, this is the client realm.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `Kerberos.commitRecord` on either a KERBEROS_REQUEST or KERBEROS_RESPONSE event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

KERBEROS_REQUEST	KERBEROS_RESPONSE
cNames	cNames
cNameType	cNameType
cRealm	cRealm
clientZeroWnd	clientZeroWnd
eType	error
from	msgType
msgType	processingTime
realm	realm
reqBytes	roundTripTime
reqL2Bytes	rspBytes
reqPkts	rspL2Bytes
reqRTO	rspPkts
sNames	rspRTO
sNameType	sNames
serverZeroWnd	sNameType
till	serverZeroWnd

sNames: ***Array of Strings***

The name portions of the server principal identifier

sNameType: ***String***

The type for the sNames field.

ticket: ***Object***

A newly issued ticket in RESPONSE or a ticket to authenticate the client to the server in an AP_REQ message.

Access only on KERBEROS_REQUEST events or an error will occur.

till: *String*

The expiration date requested by the client in a ticket request.

Access only on KERBEROS_REQUEST events or an error will occur.

LDAP

The LDAP class enables you to access properties and record metrics from LDAP_REQUEST and LDAP_RESPONSE events.

Events

LDAP_REQUEST

Runs on every LDAP request processed by the device.

LDAP_RESPONSE

Runs on every LDAP response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either an LDAP_REQUEST or LDAP_RESPONSE event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

bindDN: *String*

The bind DN of the LDAP request.

Access only on LDAP_REQUEST events or an error will occur.

dn: *String*

The LDAP distinguished name (DN). If no DN is set, <ROOT> will be returned instead.

error: *String*

The LDAP short error string as defined in the protocol (e.g., noSuchObject).

Access only on LDAP_RESPONSE events or an error will occur.

Result Code	Result String
1	operationsError
2	protocolError
3	timeLimitExceeded
4	sizeLimitExceeded
7	authMethodNotSupported
8	strongerAuthRequired
11	adminLimitExceeded
12	unavailableCriticalExtension

Result Code	Result String
13	confidentialityRequired
16	noSuchAttribute
17	undefinedAttributeType
18	inappropriateMatching
19	constraintViolation
20	attributeOrValueExists
21	invalidAttributeSyntax
32	NoSuchObject
33	aliasProblem
34	invalidDNSSyntax
36	aliasDeferencingProblem
48	inappropriateAuthentication
49	invalidCredentials
50	insufficientAccessRights
51	busy
52	unavailable
53	unwillingToPerform
54	loopDetect
64	namingViolation
65	objectClassViolation
66	notAllowedOnNonLeaf
67	notAllowedOnRDN
68	entryAlreadyExists
69	objectClassModsProhibited
71	affectsMultipleDSAs
80	other

errorDetail: *String*

The LDAP error detail, when available for that error type (e.g., protocolError : historical procotol version requested, use LDAPv3 instead).

Access only on `LDAP_RESPONSE` events or an error will occur.

method: *String*

The LDAP method.

msgSize: *Number*

The size of the LDAP message, expressed in bytes.

processingTime: ***Number***

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid or is not available. Available for the following:

- BindRequest
- SearchRequest
- ModifyRequest
- AddRequest
- DelRequest
- ModifyDNRequest
- CompareRequest
- ExtendedRequest

Applies only to `LDAP_RESPONSE` events.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `LDAP.commitRecord` on either an `LDAP_REQUEST` or `LDAP_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

LDAP_REQUEST	LDAP_RESPONSE
bindDN	clientZeroWnd
clientZeroWnd	dn
dn	error
method	errorDetail
msgSize	method
reqBytes	msgSize
reqL2Bytes	processingTime
reqPkts	roundTripTime
reqRTO	rspBytes
saslMechanism	rspL2Bytes
searchFilter	rspPkts
searchScope	rspRTO
serverZeroWnd	saslMechanism
	serverZeroWnd

reqBytes: ***Number***

The number of request bytes.

reqL2Bytes: ***Number***

The number of request L2 bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request RTOs.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: ***Number***

The number of response bytes.

rspL2Bytes: ***Number***

The number of response L2 bytes.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response RTOs.

rspZeroWnd: ***Number***

The number of zero windows in the response.

saslMechanism: ***String***

The string that defines the SASL mechanism to identify and authenticate a user to a server.

searchAttributes: ***Array***

The attributes to return from objects that match the filter criteria.

Access only on `LDAP_REQUEST` events or an error will occur.

searchFilter: ***String***

The mechanism to allow certain entries in the subtree and exclude others.

Access only on `LDAP_REQUEST` events or an error will occur.

searchScope: ***String***

The depth of a search within the search base.

Access only on `LDAP_REQUEST` events or an error will occur.

LLDP

The LLDP class enables you to access properties and record metrics from `LLDP_FRAME` events.

Events

LLDP_FRAME

Runs on every LLDP frame processed by the device.

Properties

chassisId: ***Buffer***

The chassis ID, obtained from the chassisId data field, or type-length-value (TLV).

chassisIdSubtype: ***Number***

The chassis ID subtype, obtained from the chassisID TLV.

destination: ***String***

The destination MAC address.

optTLVs: ***Array***

An array containing the optional TLVs. Each TLV is an object with the following properties:

customSubtype: *Number*

The subtype of an organizationally specific TLV.

isCustom: *Boolean*

Returns true if the object is an organizationally specific TLV.

oui: *Number*

The organizationally unique identifier for organizationally specific TLVs.

type: *Number*

The type of TLV.

value: *String*

The value of the TLV.

portId: *Buffer*

The port ID, obtained from the portId TLV.

portIdSubtype: *Number*

The port ID subtype, obtained from the portId TLV.

source: *Device*

The device sending the LLDP frame.

ttl: *Number*

The time to live, expressed in seconds. This is the length of time during which the information in this frame is valid, starting with when the information is received.

Memcache

The Memcache class enables you to access properties and record metrics from `MEMCACHE_REQUEST` and `MEMCACHE_RESPONSE` events.

Events

MEMCACHE_REQUEST

Runs on every memcache request processed by the device.

MEMCACHE_RESPONSE

Runs on every memcache response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either a `MEMCACHE_REQUEST` or `MEMCACHE_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

accessTime: *Number*

The access time, expressed in milliseconds. Available only if the first key that was requested produced a hit.

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

error: ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

hits: ***Array***

An array of objects containing the Memcache key and key size.

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

key: ***String*** **|** ***Null***

The Memcache key for which this was a hit, if available.

size: ***Number***

The size of the value returned for the key, expressed in bytes.

isBinaryProtocol: ***Boolean***

The value is `true` if the request/response corresponds to the binary version of the memcache protocol.

isNoReply: ***Boolean***

The value is `true` if the request has the "noreply" keyword and therefore should never receive a response (text protocol only).

Access only on `MEMCACHE_REQUEST` events or an error will occur.

isRspImplicit: ***Boolean***

The value is `true` if the response was implied by a subsequent response from the server (binary protocol only).

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

method: ***String***

The Memcache method as recorded in Metrics section of the ExtraHop Web UI.

misses: ***Array***

An array of objects containing the Memcache key.

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

key: ***String*** **|** ***Null***

The Memcache key for which this was a miss, if available.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `Memcache.commitRecord` on either a `MEMCACHE_REQUEST` or `MEMCACHE_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

MEMCACHE_REQUEST	MEMCACHE_RESPONSE
clientZeroWnd	accessTime
isBinaryProtocol	clientZeroWnd
isNoReply	error
method	hits
reqBytes	isBinaryProtocol
reqL2Bytes	isRspImplicit
reqPkts	method

MEMCACHE_REQUEST	MEMCACHE_RESPONSE
reqRTO	misses
reqSize	roundTripTime
serverZeroWnd	rspBytes
vbucket	rspL2Bytes
	rspPkts
	rspRTO
	serverZeroWnd
	statusCode
	vbucket

reqBytes: *Number*

The number of application-level request bytes.

reqKeys: *Array*

An array containing the Memcache key strings sent with the request.

The value of the `reqKeys` property is the same when accessed on either the `MEMCACHE_REQUEST` or the `MEMCACHE_RESPONSE` event.

reqL2Bytes: *Number*

The number of request L2 bytes.

reqPkts: *Number*

The number of request packets.

reqRTO: *Number*

The number of request RTOs.

Access only on `MEMCACHE_REQUEST` events or an error will occur.

reqSize: *Number*

The size of the request payload, expressed in bytes. The value is `NaN` for requests with no playload, such as GET and DELETE.

reqZeroWnd: *Number*

The number of zero windows in the request.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: *Number*

The number of application-level response bytes.

rspL2Bytes: *Number*

The number of response L2 bytes.

rspPkts: *Number*

The number of response packets.

rspRTO: *Number*

The number of response RTOs.

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

statusCode: *String*

The Memcache status code. For the binary protocol, the ExtraHop system metrics prepend the method to status codes other than `NO_ERROR`, but the statusCode property does not. Refer to the examples for code that matches the behavior of the ExtraHop system metrics.

Access only on `MEMCACHE_RESPONSE` events or an error will occur.

vbucket: *Number*

The Memcache vbucket, if available (binary protocol only).

Trigger Examples

- Example: Record Memcache hits and misses
- Example: Parse memcache keys

MongoDB

The MongoDB class enables you to access properties and record metrics from `MONGODB_REQUEST` and `MONGODB_ RESPONSE` events.

Events

MONGODB_REQUEST

Runs on every MongoDB request processed by the device.

MONGODB_RESPONSE

Runs on every MongoDB response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either a `MONGODB_REQUEST` or `MONGODB_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

collection: *String*

The name of the database collection specified in the current request.

database: *String*

The MongoDB database instance. In some cases, such as when login events are encrypted, the database name is not available.

error: *String*

The detailed error message recorded by the ExtraHop system.

Access only on `MONGODB_RESPONSE` events or an error will occur.

isReqAborted: *Boolean*

The value is `true` if the connection is closed before the MongoDB request was complete.

isReqTruncated: ***Boolean***

The value is `true` if the request document(s) size is greater than the maximum payload document size.

isRspAborted: ***Boolean***

The value is `true` if the connection is closed before the MongoDB response was complete.

Access only on `MONGODB_RESPONSE` events or an error will occur.

method: ***String***

The MongoDB database method (appears under **Methods** in the user interface).

opcode: ***String***

The MongoDB operational code on the wire protocol, which might differ from the MongoDB method used.

processingTime: ***Number***

The time to process the request, expressed in milliseconds (equivalent to `rspTimeToFirstByte - reqTimeToLastByte`). The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `MONGODB_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `MongoDB.commitRecord` on either an `MONGODB_REQUEST` or `MONGODB_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

MONGODB_REQUEST	MONGODB_RESPONSE
clientZeroWnd	clientZeroWnd
collection	collection
database	database
isReqAborted	error
isReqTruncated	isRspAborted
method	method
opcode	opcode
reqBytes	processingTime
reqL2Bytes	roundTripTime
reqPkts	rspBytes
reqRTO	rspL2Bytes
reqSize	rspPkts
reqTimeToLastByte	rspRTO
serverZeroWnd	rspSize
user	rspTimeToFirstByte
	rspTimeToLastByte
	serverZeroWnd

MONGODB_REQUEST	MONGODB_RESPONSE
	user

reqBytes: ***Number***

The number of application-level request bytes.

reqL2Bytes: ***Number***

The number of request L2 bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request RTOs.

reqSize: ***Number***

The size of the request payload, expressed in bytes.

reqTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the request, expressed in milliseconds.

reqZeroWnd: ***Number***

The number of zero windows in the request.

request: ***Array***

An array of JS objects parsed from MongoDB request payload documents. Total document size is limited to 4K.

If BSON documents are truncated, `isReqTruncated` flag is set. Truncated values are represented as follows:

- Primitive string values like code, code with scope, and binary data are partially extracted.
- Objects and Arrays are partially extracted.
- All other primitive values like Numbers, Dates, RegExp, etc., are substituted with `null`.

If no documents are included in the request, an empty array is returned.

The value of the `request` property is the same when accessed on either the `MONGODB_REQUEST` or the `MONGODB_RESPONSE` event.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: ***Number***

The number of application-level response bytes.

rspL2Bytes: ***Number***

The number of response L2 bytes.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response RTOs.

rspSize: ***Number***

The size of the response payload, expressed in bytes.

Access only on `MONGODB_RESPONSE` events or an error will occur.

rspTimeToFirstByte: *Number*

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `MONGODB_RESPONSE` events or an error will occur.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last by of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `MONGODB_RESPONSE` events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

user: *String*

The user name, if available. In some cases, such as when login events are encrypted, the user name is not available.

MSMQ

The MSMQ class enables you to access properties and record metrics from `MSMQ_MESSAGE` event.

Events

MSMQ_MESSAGE

Runs on every MSMQ user message processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `MSMQ_MESSAGE` event.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

adminQueue: *String*

The name of the administration queue of the message.

correlationId: *Buffer*

The application-generated correlation ID of the message.

dstQueueMgr: *String*

The destination message broker of the message.

isEncrypted: *Boolean*

The value is `true` if the payload is encrypted.

label: *String*

The label or description of the message.

msgClass: *String*

The message class of the message. The following values are valid:

- MQMSG_CLASS_NORMAL
- MQMSG_CLASS_ACK_REACH_QUEUE
- MQMSG_CLASS_NACK_ACCESS_DENIED
- MQMSG_CLASS_NACK_BAD_DST_Q

- MQMSG_CLASS_NACK_BAD_ENCRYPTION
- MQMSG_CLASS_NACK_BAD_SIGNATURE
- MQMSG_CLASS_NACK_COULD_NOT_ENCRYPT
- MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED
- MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG
- MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q
- MQMSG_CLASS_NACK_PURGED
- MQMSG_CLASS_NACK_Q_EXCEEDED_QUOTA
- MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT
- MQMSG_CLASS_NACK_SOURCE_COMPUTER_GUID_CHANGED
- MQMSG_CLASS_NACK_UNSUPPORTED_CRYPTO_PROVIDER
- MQMSG_CLASS_ACK_RECEIVE
- MQMSG_CLASS_NACK_Q_DELETED
- MQMSG_CLASS_NACK_Q_PURGED
- MQMSG_CLASS_NACK_RECEIVE_TIMEOUT
- MQMSG_CLASS_NACK_RECEIVE_TIMEOUT_AT_SENDER
- MQMSG_CLASS_REPORT

msgId: ***Number***

The MSMQ message id of the message.

payload: ***Buffer***

The body of the MSMQ message.

priority: ***Number***

The priority of the message. This can be a number between 0 and 7.

queue: ***String***

The name of the destination queue of the message.

receiverBytes: ***Number***

The number of L4 receiver bytes.

receiverL2Bytes: ***Number***

The number of L2 receiver bytes.

receiverPkts: ***Number***

The number of receiver packets.

receiverRTO: ***Number***

The number of receiver RTOs.

receiverZeroWnd: ***Number***

The number of zero windows sent by the receiver.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `MSMQ.commitRecord` on an `MSMQ_MESSAGE` event.

The record object contains the following default properties:

- adminQueue
- dstQueueMgr
- isEncrypted
- label
- msgClass
- msgId
- priority
- queue

- receiverBytes
- receiverL2Bytes
- receiverPkts
- receiverRTO
- receiverZeroWnd
- responseQueue
- roundTripTime
- senderBytes
- senderL2Bytes
- senderPkts
- senderRTO
- serverZeroWnd
- srcQueueMgr

responseQueue: *String*

The name of the response queue of the message.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

senderBytes: *Number*

The number of sender L4 bytes.

senderL2Bytes: *Number*

The number of sender L2 Bytes.

senderPkts: *Number*

The number of sender packets.

senderRTO: *Number*

The number of sender RTOs.

senderZeroWnd: *Number*

The number of zero windows sent by the sender.

srcQueueMgr: *String*

The source message broker of the message.

NetFlow

The NetFlow class object enables you to access properties and record metrics from `NETFLOW_RECORD` events.

The ExtraHop Discover appliance can be licensed for the NetFlow module, which supports the following flow types:

- NetFlow version 5 (Cisco)
- NetFlow version 9 (Cisco)
- IPFIX (open standard based on RFC 5101)

Events

NETFLOW_RECORD

Runs upon receipt of a flow record from a flow network.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `NETFLOW_RECORD` event.

Note: The record is not committed to the Explore appliance if the record object contains one or more enterprise fields.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

findField(field: *Number* [enterpriseId: *Number*]): *String* | *Number* | *IPAddress* | *Buffer* | *Boolean*

Searches the NetFlow record and returns the specified field. Returns a null value if the field is not in the record. If the optional `enterpriseId` argument is included, the specified field is returned only if the enterprise ID is a match, otherwise the method returns a null value.

hasField(field: *Number*): *Boolean*

Determines whether the specified field is in the NetFlow record.

Properties

age: *Number*

The amount of time elapsed, expressed in seconds, between the `first` and `last` property values reported in the NetFlow record.

deltaBytes: *Number*

The number of L3 bytes in the flow since the last `NETFLOW_RECORD` event.

deltaPkts: *Number*

The number of packets in the flow since the last `NETFLOW_RECORD` event.

dscp: *Number*

The number representing the last differentiated services code point (DSCP) value of the flow packet.

dscpName: *String*

The name associated with the DSCP value of the flow packet. The following table displays well-known DSCP names:

Number	Name
8	CS1
10	AF11
12	AF12
14	AF13
16	CS2
18	AF21
20	AF22
22	AF23
24	CS3
26	AF31
28	AF32
30	AF33
32	CS4

Number	Name
34	AF41
36	AF42
38	AF43
40	CS5
44	VA
46	EF
48	CS6
56	CS7

egressInterface: *FlowInterface*

The FlowInterface object that identifies the output device.

fields: *Array*

An array of objects that contain information fields found in the flow packets. Each object can contain the following properties:

fieldID: *Number*

The ID number that represents the field type.

enterpriseID: *Number*

The ID number that represents enterprise-specific information.

first: *Number*

The amount of time elapsed, expressed in milliseconds, since the epoch of the first packet in the flow.

format: *String*

The format of the NetFlow record. Valid values are `NetFlow v5`, `NetFlow v9`, and `IPFIX`.

ingressInterface: *FlowInterface*

The FlowInterface object that identifies the input device.

ipPrecedence: *Number*

The value of the IP precedence field associated with the DSCP of the flow packet.

ipproto: *String*

The IP protocol associated with the flow, such as TCP or UDP.

last: *Number*

The amount of time elapsed, expressed in milliseconds, since the epoch of the last packet in the flow.

network: *FlowNetwork*

An object that identifies the FlowNetwork and contains the following properties:

id: *String*

The identifier of the FlowNetwork.

ipaddr: *IPAddress*

The IP address of the FlowNetwork.

nextHop: *IPAddress*

The IP address of the next hop router.

receiver: *Object*

An object that identifies the receiver and contains the following properties:

asn: ***Number***

The autonomous system number (ASN) of the destination device.

ipaddr: ***IPAddress***

The IP address of the destination device.

prefixLength: ***Number***

The number of bits in the prefix of the destination address.

port: ***Number***

The TCP or UDP port number of the destination device.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `NetFlow.commitRecord` on a `NETFLOW_RECORD` event.

Note: A null value is returned if the record contains one or more enterprise fields.

The record object contains the following default properties:

- age
- dscpName
- deltaBytes
- deltaPkts
- egressInterface
- first
- format
- ingressInterface
- last
- network
- networkAddr
- nextHop
- proto
- receiverAddr
- receiverAsn
- receiverPort
- receiverPrefixLength
- senderAddr
- senderAsn
- senderPort
- senderPrefixLength
- tcpFlagName
- tcpFlags

sender: ***Object***

An object that identifies the sender and contains the following properties:

asn: ***Number***

The autonomous system number (ASN) of the source device.

ipaddr: ***IPAddress***

The IP address of the source device.

prefixLength: ***Number***

The number of bits in the prefix of the source address.

port: ***Number***

The TCP or UDP port number of the source device.

tcpFlagNames: *Array*

A string array of TCP flag names, such as SYN or ACK, found in the flow packets.

tcpFlags: *Number*

The bitwise OR of all TCP flags set on the flow.

templateID: *Number*

The ID of the template that is referred to by the record. Template IDs are applicable only to IPFIX and NetFlow v9 records.

tos: *Number*

The type of service (ToS) number defined in the IP header.

NFS

The NFS class enables you to access properties and record metrics from `NFS_REQUEST` and `NFS_RESPONSE` events.

Events

NFS_REQUEST

Runs on every NFS request processed by the device.

NFS_RESPONSE

Runs on every NFS response processed by the device

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `NFS_RESPONSE` event. Record commits on `NFS_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

accessTime: *Number*

The amount of time taken by the server to access a file on disk, expressed in milliseconds. For NFS, it is the time from every non-pipelined READ and WRITE command in an NFS flow until the payload containing the response is recorded by the ExtraHop system. The value is `NaN` on malformed and aborted responses, or if the timing is invalid or is not applicable.

Access only on `NFS_RESPONSE` events or an error will occur.

authMethod: *String*

The method for authenticating users.

error:*String*

The detailed error message recorded by the ExtraHop system.

Access only on `NFS_RESPONSE` events or an error will occur.

fileHandle: *Buffer*

The file handle returned by the server on LOOKUP, CREATE, SYMLINK, MKNOD, LINK, or READDIRPLUS operations.

isCommandFileInfo: *Boolean*

The value is `true` for file info commands.

isCommandRead: ***Boolean***

The value is `true` for READ commands.

isCommandWrite: ***Boolean***

The value is `true` for WRITE commands.

method: ***String***

The NFS method. Valid methods are listed under the NFS metric in the ExtraHop Web UI.

offset: ***Number***

The file offset associated with NFS READ and WRITE commands.

Access only on `NFS_REQUEST` events or an error will occur.

processingTime: ***Number***

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `NFS_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `NFS.commitRecord` on a `NFS_RESPONSE` event.

The record object contains the following default properties:

- accessTime
- authMethod
- clientZeroWnd
- error
- isCommandFileInfo
- isCommandRead
- isCommandWrite
- isRspAborted
- method
- offset
- processingTime
- renameDirChanged
- reqSize
- reqXfer
- resource
- rspSize
- rspXfer
- serverZeroWnd
- statusCode
- txID
- user
- version

Access the record object only on `NFS_RESPONSE` events or an error will occur.

renameDirChanged: ***Boolean***

The value is `true` if a resource rename request includes a directory move.

Access only on `NFS_REQUEST` events or an error will occur.

reqBytes: ***Number***

The number of L4 request bytes.

Access only on `NFS_RESPONSE` events or an error will occur.

reqL2Bytes: ***Number***

The number of L2 request bytes.

Access only on `NFS_RESPONSE` events or an error will occur.

reqPkts: ***Number***

The number of request packets.

Access only on `NFS_RESPONSE` events or an error will occur.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

Access only on `NFS_REQUEST` events or an error will occur.

reqSize: ***Number***

The size of the request payload, expressed in bytes.

reqTransferTime: ***Number***

The request transfer time, expressed in milliseconds. If the request is contained in a single packet, the transfer time is zero. If the request spans multiple packets, the value is the amount of time between detection of the first NFS request packet and detection of the last packet by the ExtraHop system. A high value might indicate a large NFS request or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

Access only on `NFS_REQUEST` events or an error will occur.

reqZeroWnd: ***Number***

The number of zero windows in the request.

resource: ***String***

The path and filename, concatenated together.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `NFS_RESPONSE` events or an error will occur.

rspBytes: ***Number***

The number of L4 response bytes.

Access only on `NFS_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of L2 response bytes.

Access only on `NFS_RESPONSE` events or an error will occur.

rspPkts: ***Number***

The number of response packets.

Access only on `NFS_RESPONSE` events or an error will occur.

rspRTO: ***Number***

The number of request retransmission timeouts (RTOs).

Access only on `NFS_RESPONSE` events or an error will occur.

rspSize: ***Number***

The size of the response payload, expressed in bytes.

Access only on `NFS_RESPONSE` events or an error will occur.

rspTransferTime: ***Number***

The response transfer time, expressed in milliseconds. If the response is contained in a single packet, the transfer time is zero. If the response spans multiple packets, the value is the amount

of time between detection of the first NFS response packet and detection of the last packet by the ExtraHop system. A high value might indicate a large NFS response or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

Access only on `NFS_RESPONSE` events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

statusCode: *String*

The NFS status code of the request or response.

txId: *Number*

The transaction ID.

user: *String*

The ID of the Linux user, formatted as `uid:xxxx@ip_address`.

version: *Number*

The NFS version.

POP3

The POP3 class enables you to access properties and record metrics from `POP3_REQUEST` and `POP3_RESPONSE` events.

Events

POP3_REQUEST

Runs on every POP3 request processed by the device.

POP3_RESPONSE

Runs on every POP3 response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `POP3_RESPONSE` event. Record commits on `POP3_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

dataSize: *Number*

The size of the message, expressed in bytes.

Access only on `POP3_RESPONSE` events or an error will occur.

error: *String*

The detailed error message recorded by the ExtraHop system.

Access only on `POP3_RESPONSE` events or an error will occur.

isEncrypted: *Boolean*

The value is `true` if the transaction is over a secure POP3 server.

isReqAborted: *Boolean*

The value is `true` if the connection is closed before the POP3 request was complete.

isRspAborted: ***Boolean***

The value is `true` if the connection is closed before the POP3 response was complete.

Access only on `POP3_RESPONSE` events or an error will occur.

method: ***String***

The POP3 method such as RETR or DELE.

processingTime: ***Number***

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `POP3_RESPONSE` events or an error will occur.

recipientList: ***Array***

An array that contains a list of recipient addresses.

Access only on `POP3_RESPONSE` events or an error will occur.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `POP3.commitRecord` on a `POP3_RESPONSE` event.

The record object contains the following default properties:

- clientZeroWnd
- dataSize
- error
- isEncrypted
- isReqAborted
- isRspAborted
- method
- processingTime
- recipientList
- reqSize
- reqTimeToLastByte
- rspSize
- rspTimeToFirstByte
- rspTimeToLastByte
- sender
- serverZeroWnd
- statusCode

Access the record object only on `POP3_RESPONSE` events or an error will occur.

reqBytes: ***Number***

The number of L4 request bytes.

reqL2Bytes: ***Number***

The number of L2 request bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

reqSize: ***Number***

The size of the request payload, expressed in bytes. The size does not include headers.

reqTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the request, expressed in milliseconds. The value is `NaN` on expired requests and responses, or if the timing is invalid.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median TCP round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `POP3_RESPONSE` events or an error will occur.

rspBytes: ***Number***

The number of L4 response bytes.

Access only on `POP3_RESPONSE` events or an error will occur.

rspL2Bytes: ***Number***

The number of response L2 bytes.

Access only on `POP3_RESPONSE` events or an error will occur.

rspPkts: ***Number***

The number of response packets.

Access only on `POP3_RESPONSE` events or an error will occur.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `POP3_RESPONSE` events or an error will occur.

rspSize: ***Number***

The size of the response payload, expressed in bytes. The size does not include headers.

Access only on `POP3_RESPONSE` events or an error will occur.

rspTimeToFirstByte: ***Number***

The time from the first byte of the request until the furst byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `POP3_RESPONSE` events or an error will occur.

rspTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `POP3_RESPONSE` events or an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

sender: ***String***

The address of the sender of the message.

Access only on `POP3_RESPONSE` events or an error will occur.

status: ***String***

The POP3 status message of the response which can be `OK`, `ERR` or `NULL`.

Access only on `POP3_RESPONSE` events or an error will occur.

Redis

Remote Dictionary Server (Redis) is an open-source, in-memory data structure server. The Redis class enables you to access properties and record metrics from `REDIS_REQUEST` and `REDIS_RESPONSE` events.

Events

REDIS_REQUEST

Runs on every Redis request processed by the device.

REDIS_RESPONSE

Runs on every Redis response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either a `REDIS_REQUEST` or `REDIS_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

errors: *Array*

An array of detailed error messages recorded by the ExtraHop system.

Access only on `REDIS_RESPONSE` events or an error will occur.

isReqAborted: *Boolean*

The value is `true` if the connection is closed before the Redis request was complete.

isRspAborted: *Boolean*

The value is `true` if the connection is closed before the Redis response was complete.

Access only on `REDIS_RESPONSE` events or an error will occur.

method: *String*

The Redis method such as GET or KEYS.

payload: *Buffer*

The body of the response or request.

processingTime: *Number*

The server processing time, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `REDIS_RESPONSE` events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `Redis.commitRecord` on either an `REDIS_REQUEST` or `REDIS_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

REDIS_REQUEST	REDIS_RESPONSE
clientZeroWnd	clientZeroWnd

REDIS_REQUEST	REDIS_RESPONSE
method	error
reqKey	method
reqSize	processingTime
reqTransferTime	reqKey
isReqAborted	rspSize
serverZeroWnd	rspTransferTime
	isRspAborted
	rspTimeToFirstByte
	rspTimeToLastByte
	serverZeroWnd

reqKey: ***Array***

An array containing the Redis key strings sent with the request.

reqBytes: ***Number***

The number of L4 request bytes.

reqL2Bytes: ***Number***

The number of L2 request bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

reqSize: ***Number***

The size of the request payload, expressed in bytes. The size does not include headers.

reqTransferTime: `Number`

The request transfer time, expressed in milliseconds. If the request is contained in a single packet, the transfer time is zero. If the request spans multiple packets, the value is the amount of time between detection of the first Redis request packet and detection of the last packet by the ExtraHop system. A high value might indicate a large Redis request or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median TCP round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: ***Number***

The number of L4 response bytes.

rspL2Bytes: ***Number***

The number of response L2 bytes.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

rspTransferTime: `Number`

The response transfer time, expressed in milliseconds. If the response is contained in a single packet, the transfer time is zero. If the response spans multiple packets, the value is the amount of time between detection of the first Redis response packet and detection of the last packet by the ExtraHop system. A high value might indicate a large Redis response or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

Access only on `REDIS_RESPONSE` events or an error will occur.

rspSize: ***Number***

The size of the response payload, expressed in bytes. The size does not include headers.

Access only on `REDIS_RESPONSE` events or an error will occur.

rspTimeToFirstByte: ***Number***

The time from the first byte of the request until the furst byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `REDIS_RESPONSE` events or an error will occur.

rspTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `REDIS_RESPONSE` events or an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

RTCP

The RTCP class enables you to access properties and record metrics from `RTCP_MESSAGE` events.

Events

RTCP_MESSAGE

Runs on every RTCP UDP packet processed by the device.

Methods

commitRecord(): ***void***

Commits a record object to the ExtraHop Explore appliance on an `RTCP_MESSAGE` event.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

callId: ***String***

The Call ID for associating with a SIP flow.

packets: ***Array***

An array of RTCP packet objects where each object represents a packet and contains a packetType field. Each object has different fields based on the message type, as described below.

packetType: ***String***

The type of packet. If the packet type is not recognizable, then the `packetType` will be "Unknown N" where N is the RTP control packet type value.

Value	Type	Name
194	SMPTETC	SMPTE time-code mapping
195	IJ	Extended inter-arrival jitter report
200	SR	sender report
201	RR	receiver report
202	SDES	source description
203	BYE	goodbye
204	APP	application-defined
205	RTPFB	Generic RTP Feedback
206	PSFB	Payload-specific
207	XR	extended report
208	AVB	AVB RTCP packet
209	RSI	Receiver Summary Information
210	TOKEN	Port Mapping
211	IDMS	IDMS Settings

APP packet objects have the following fields:

name: *String*

The name chosen by the person defining the set of APP packets to be unique. Interpreted as four case-sensitive ASCII characters.

ssrc: *Number*

The SSRC of the sender.

value: *Buffer*

The optional application-dependent data.

BYE packet objects have the following fields:

packetType: *Number*

Contains the number 203 to identify this as an RTCP BYE packet.

SR packet objects have the following fields:

ntpTimestamp: *Number*

The NTP timestamp, converted to milliseconds since the epoch (January 1, 1970).

reportBlocks: *Array*

An array of report objects which contain:

fractionLost: *Number*

The 8-bit number indicating the number of packets lost divided by the number of packets expected.

jitter: *Number*

An estimate of the statistical variance of the RTP data packet interarrival time, expressed in milliseconds.

lastSR: *Number*

The middle 32 bits of the ntp_Timestamp received as part of the most recent RTCP sender report (SR) packet from the source SSRC. If no SR has been received yet, this field is set to zero.

lastSRDelay: ***Number***

The delay between receiving the last SR packet from the source SSRC and sending this reception block, expressed in units of 1/65536 seconds. If no SR packet has been received yet, this field is set to zero.

packetsLost: ***Number***

The total number of RTP data packets from the source SSRC that have been lost since the beginning of reception.

seqNum: ***Number***

The highest sequence number received from the source SSRC.

ssrc: ***Number***

The SSRC of the sender.

rtpTimestamp: ***Number***

The RTP timestamp, converted to milliseconds since the epoch (January 1, 1970).

senderOctets: ***Number***

The sender octet count.

senderPkts: ***Number***

The sender packet count.

RR packet objects have the following fields:

reportBlocks: ***Array***

An array of report objects which contain:

fractionLost: ***Number***

The 8-bit number indicating the number of packets last divided by the number of packets expected.

jitter: ***Number***

An estimate of the statistical variance of the RTP data packet interarrival, expressed in milliseconds.

lastSR: ***Number***

The middle 32 bits of the ntp_Timestamp received as part of the most recent RTCP sender report (SR) packet from the source SSRC. If no SR has been received yet, this field is set to zero.

lastSRDelay: ***Number***

The delay between receiving the last SR packet from the source SSRC and sending this reception report block, expressed in units of 1/65536 seconds. If no SR packet has been received yet, this field is set to zero.

packetsLost: ***Number***

The total number of RTP data packets from the source SSRC that have been lost since the beginning of reception.

seqNum: ***Number***

The highest sequence number received from the source SSRC.

ssrc: ***Number***

The SSRC of the sender.

ssrc: ***Number***

The SSRC of the sender.

SDES packet objects have the following fields:

descriptionBlocks: ***Array***

An array of objects that contain:

type: ***Number***

The SDES type.

SDES Type	Abbrev.	Name
0	END	end of SDES list
1	CNAME	canonical name
2	NAME	user name
3	EMAIL	user's electronic mail address
4	PHONE	user's phone number
5	LOC	geographic user location
6	TOOL	name of application or tool
7	NOTE	notice about the source
8	PRIV	private extensions
9	H323-C ADDR	H.323 callable address
10	APSI	Application Specific Identifier

value: ***Buffer***

A buffer containing the text portion of the SDES packet.

ssrc: ***Number***

The SSRC of the sender.

XR packet objects have the following fields:

ssrc: ***Number***

The SSRC of the sender.

xrBlocks: ***Array***

An array of report blocks which contain:

statSummary: ***Object***

Type 6 only. The `statSummary` object contains the following properties:

beginSeq: ***Number***

The beginning sequence number for the interval.

devJitter: ***Number***

The standard deviation of the relative transit time between each two packet series in the sequence interval.

devTTLOrHL: ***Number***

The standard deviation of TTL or Hop Limit values of data packets in the sequence number range.

dupPackets: ***Number***

The number of duplicate packets in the sequence number interval.

endSeq: ***Number***

The ending sequence number for the interval.

lostPackets: ***Number***

The number of lost packets in the sequence number interval.

maxJitter: ***Number***

The maximum relative transmit time between two packets in the sequence interval, expressed in milliseconds.

maxTTLOrHL: ***Number***

The maximum TTL or Hop Limit value of data packets in the sequence number range.

meanJitter: ***Number***

The mean relative transit time between two packet series in the sequence interval, rounded to the nearest value expressible as an RTP timestamp, expressed in milliseconds.

meanTTLOrHL: ***Number***

The mean TTL or Hop Limit value of data packets in the sequence number range.

minJitter: ***Number***

The minimum relative transmit time between two packets in the sequence interval, expressed in milliseconds.

minTTLOrHL: ***Number***

The minimum TTL or Hop Limit value of data packets in the sequence number range.

ssrc: ***Number***

The SSRC of the sender.

type: ***Number***

The XR block type.

Block Type	Name
1	Loss RTE Report Block
2	Duplicate RLE Report Block
3	Packet Receipt Times Report Block
4	Receiver Reference Time Report Block
5	DLRR Report Block
6	Statistics Summary Report Block
7	VoIP Metrics Report Block
8	RTCP XP
9	Texas Instruments Extended VoIP Quality Block
10	Post-repair Loss RLE Report Block
11	Multicast Acquisition Report Block
12	IBMS Report Block
13	ECN Summary Report
14	Measurement Information Block
15	Packet Delay Variation Metrics Block
16	Delay Metrics Block
17	Burst/Gap Loss Summary Statistics Block

Block Type	Name
18	Burst/Gap Discard Summary Statistics Block
19	Frame Impairment Statistics Summary
20	Burst/Gap Loss Metrics Block
21	Burst/Gap Discard Metrics Block
22	MPEG2 Transport Stream PSI-Independent Decodability Statistics Metrics Block
23	De-Jitter Buffer Metrics Block
24	Discard Count Metrics Block
25	DRLE (Discard RLE Report)
26	BDR (Bytes Discarded Report)
27	RFISD (RTP Flows Initial Synchronization Delay)
28	RFSO (RTP Flows Synchronization Offset Metrics Block)
29	MOS Metrics Block
30	LCB (Loss Concealment Metrics Block)
31	CSB (Concealed Seconds Metrics Block)
32	MPEG2 Transport Stream PSI Decodability Statistics Block

typeSpecific: *Number*

The contents of this field depend on the block type.

value: *Buffer*

The contents of this field depend on the block type.

voipMetrics: *Object*

Type 7 only. The `voipMetrics` object contains the following properties:

burstDensity: *Number*

The fraction of RTP data packets within burst periods since the beginning of reception that were either lost or discarded.

burstDuration: *Number*

The mean duration, expressed in milliseconds, of the burst periods that have occurred since the beginning of reception.

discardRate: *Number*

The fraction of RTP data packets from the source that have been discarded since the beginning of reception, due to late or early arrival, under-run or overflow at the receiving jitter buffer.

endSystemDelay: *Number*

The most recently estimated end system delay, expressed in milliseconds.

extRFactor: *Number*

The external R factor quality metric. A value of 127 indicates this parameter is unavailable.

gapDensity: ***Number***

The fraction of RTP data packets within inter-burst gaps since the beginning of reception that were either lost or discarded.

gapDuration: ***Number***

The mean duration of the gap periods that have occurred since the beginning of reception, expressed in milliseconds.

gmin: ***Number***

The gap threshold.

jbAbsMax: ***Number***

The absolute maximum delay, expressed in milliseconds, that the adaptive jitter buffer can reach under worst case conditions.

jbMaximum: ***Number***

The current maximum jitter buffer delay, which corresponds to the earliest arriving packet that would not be discarded, expressed in milliseconds.

jbNominal: ***Number***

The current nominal jitter buffer delay, which corresponds to the nominal jitter buffer delay for packets that arrive exactly on time, expressed in milliseconds.

lossRate: ***Number***

The fraction of RTP data packets from the source lost since the beginning of reception.

mosCQ: ***Number***

The estimated mean opinion score for conversational quality (MOS-CQ). A value of 127 indicates this parameter is unavailable.

mosLQ: ***Number***

The estimated mean opinion score for listening quality (MOS-LQ). A value of 127 indicates this parameter is unavailable.

noiseLevel: ***Number***

The noise level, expressed in decibels.

rerl: ***Number***

The residual echo return loss value, expressed in decibels.

rFactor: ***Number***

The R factor quality metric. A value of 127 indicates this parameter is unavailable.

roundTripDelay: ***Number***

The most recently calculated round-trip time (RTT) between RTP interfaces, expressed in milliseconds.

rxConfig: ***Number***

The receiver configuration byte.

signalLevel: ***Number***

The voice signal relative level, expressed in decibels.

ssrc: ***Number***

The SSRC of the sender.

record: ***Object***

The record object committed to the ExtraHop Explore appliance through a call to `RTCP.commitRecord` on an `RTCP_MESSAGE` event.

The record object contains the following default properties:

- callId
- cName

RTP

The RTP class enables you to access properties and record metrics from `RTP_OPEN`, `RTP_CLOSE`, and `RTP_TICK` events.

Events

RTP_CLOSE

Runs when an RTP connection is closed.

RTP_OPEN

Runs when a new RTP connection is opened.

RTP_TICK

Runs periodically on RTP flows.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `RTP_TICK` event. Record commits on `RTP_OPEN` and `RTP_CLOSE` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

bytes: *Number*

The number of bytes sent.

Access only on `RTP_TICK` events or an error will occur.

callId: *String*

The call ID for associating with SIP flow.

drops: *Number*

The number of dropped packets detected.

Access only on `RTP_TICK` events or an error will occur.

dups: *Number*

The number of duplicate packets detected.

Access only on `RTP_TICK` events or an error will occur.

jitter: *Number*

An estimate of the statistical variance of the data packet interarrival time.

Access only on `RTP_TICK` events or an error will occur.

l2Bytes: *Number*

The number of L2 bytes.

Access only on `RTP_TICK` events or an error will occur.

mos: *Number*

The estimated mean opinion score for quality.

Access only on `RTP_TICK` events or an error will occur.

outOfOrder: *Number*

The number of out-of-order messaged detected.

Access only on `RTP_TICK` events or an error will occur.

payloadType: *String*

The type of RTP payload.

Access only on `RTP_TICK` events or an error will occur.

payloadTypeId	payloadType
0	ITU-T G.711 PCMU Audio
3	GSM 6.10 Audio
4	ITU-T G.723.1 Audio
5	IMA ADPCM 32kbit Audio
6	IMA ADPCM 64kbit Audio
7	LPC Audio
8	ITU-T G.711 PCMA Audio
9	ITU-T G.722 Audio
10	Linear PCM Stereo Audio
11	Linear PCM Audio
12	QCELP
13	Comfort Noise
14	MPEG Audio
15	ITU-T G.728 Audio
16	IMA ADPCM 44kbit Audio
17	IMA ADPCM 88kbit Audio
18	ITU-T G.729 Audio
25	Sun CellB Video
26	JPEG Video
28	Xerox PARC Network Video
31	ITU-T H.261 Video
32	MPEG Video
33	MPEG-2 Transport Stream
34	ITU-T H.263-1996 Video

payloadTypeId: *Number*

The numeric value of the payload type. See table under `payloadType`.

Access only on `RTP_TICK` events or an error will occur.

pkts: *Number*

The number of packets sent.

Access only on RTP_TICK events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to RTP.commitRecord on an RTP_TICK event.

The record object contains the following default properties:

- bytes
- callId
- drops
- dups
- jitter
- l2Bytes
- mos
- outOfOrder
- payloadType
- payloadTypeId
- pkts
- rFactor
- ssrc
- version

Access record objects only on RTP_TICK events or an error will occur.

rFactor: *Number*

The R factor quality metric.

Access only on RTP_TICK events or an error will occur.

ssrc: *Number*

The SSRC of sender.

version: *Number*

The RTP version number.

SDP

The SDP class enables you to access SDP properties from SIP_REQUEST and SIP_RESPONSE events.

The SIP_ REQUEST and SIP_RESPONSE events are defined in the SIP section.

Properties

mediaDescriptions: *Array*

An array of objects that contain the following fields:

attributes: *Array of Strings*

The optional session attributes.

bandwidth: *Array of Strings*

The optional proposed bandwidth type and bandwidth to be used by the session or media.

connectionInfo: *String*

The connection data, including network type, address type and connection adddress. May also contain optional sub-fields, depending on the address type.

description: *String*

The session description which may contain one or more media descriptions. Each media description consists of media, port and transport protocol fields.

encryptionKey: ***String***

The optional encryption method and key for the session.

mediaTitle: ***String***

The title of the media stream.

Access only on `SIP_ REQUEST` and `SIP_RESPONSE` events or an error will occur.

sessionDescription: ***Object***

An object that contains the following fields:

attributes: ***Array of Strings***

The optional session attributes.

bandwidth: ***Array of Strings***

The optional proposed bandwidth type and bandwidth to be used by the session or media.

connectionInfo: ***String***

The connection data, including network type, address type and connection address. May also contain optional sub-fields, depending on the address type.

email: ***String***

The optional email address. If present, this can contain multiple email addresses.

encryptionKey: ***String***

The optional encryption method and key for the session.

origin: ***String***

The originator of the session, including username, address of the user's host, a session identifier, and a version number.

phoneNumber: ***String***

The optional phone number. If present, this can contain multiple phone numbers.

sessionInfo: ***String***

The session description.

sessionName: ***String***

The session name.

timezoneAdjustments: ***String***

The adjustment time and offset for a scheduled session.

uri: ***String***

The optional URI intended to provide more information about the session.

version: ***String***

The version number. This should be 0.

Access only on `SIP_ REQUEST` and `SIP_RESPONSE` events or an error will occur.

timeDescriptions: ***Array***

An array of objects that contain the following fields:

repeatTime: ***String***

The session repeat time, including interval, active duration, and offsets from start time.

time: ***String***

The start time and stop times for a session.

Access only on `SIP_ REQUEST` and `SIP_RESPONSE` events or an error will occur.

SIP

The SIP class enables you to access properties and record metrics from `SIP_REQUEST` and `SIP_RESPONSE` events.

Events

SIP_REQUEST

Runs on every SIP request processed by the device.

SIP_RESPONSE

Runs on every SIP response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either an `SIP_REQUEST` or `SIP_RESPONSE` event.

The event determines which properties are committed to the record object. To view the default properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

findHeaders(name: *String*): *Array*

Allows access to SIP header values. The result is an array of header objects (with name and value properties) where the names match the prefix of the string passed to `findHeaders`.

Properties

callId: *String*

The call ID for this message.

from: *String*

The contents of the From header.

hasSDP: *Boolean*

The value is `true` if this event includes SDP information.

headers: *Object*

An array-like object that allows access to SIP header names and values. Access a specific header using one of these methods:

string property:

The name of the header, accessible in a dictionary-like fashion. For example:

```
var headers = SIP.headers;
session = headers["X-Session-Id"];
accept = headers.accept;
```

numeric property:

The order in which headers appear on the wire. The returned object has a name and a value property. Numeric properties are useful for iterating over all the headers and disambiguating headers with duplicate names. For example:

```
for (i = 0; i < headers.length; i++) {
   hdr = headers[i];
   debug("headers[" + i + "].name: " + hdr.name);
   debug("headers[" + i + "].value: " + hdr.value);
```

```
}
```

Note: Saving SIP.headers to the Flow store does not save all of the individual header values. It is best practice to save the individual header values to the Flow store.

method: *String*

The SIP method.

Method Name	Description
ACK	Confirms the client has received a final response to an INVITE request.
BYE	Terminates a call. Can be sent by either the caller or the callee.
CANCEL	Cancels any pending request
INFO	Sends mid-session information that doesn't change the session state.
INVITE	Invites a client to participate in a call session.
MESSAGE	Transports instant messages using SIP.
NOTIFY	Notify the subscriber of a new event.
OPTIONS	Queries the capabilities of servers.
PRACK	Provisional Acknowledgement.
PUBLISH	Publish an event to the server.
REFER	Ask recipient to issue a SIP request (call transfer).
REGISTER	Registers the address listed in the To header field with a SIP server.
SUBSCRIBE	Subscribes for an event of Notification from the Notifier.
UPDATE	Modifies the state of a session without changing the state of the dialog.

processingTime: *Number*

The time between the request and the first response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `SIP_RESPONSE` events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `SIP.commitRecord` on either an `SIP_REQUEST` or `SIP_RESPONSE` event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

SIP_REQUEST	SIP_RESPONSE
callId	callId
clientZeroWnd	clientZeroWnd
from	from

SIP_REQUEST	SIP_RESPONSE
hasSDP	hasSDP
method	processingTime
reqBytes	roundTripTime
reqL2Bytes	rspBytes
reqPkts	rspL2Bytes
reqRTO	rspPkts
reqSize	rspRTO
serverZeroWnd	rspSize
to	serverZeroWnd
uri	statusCode
	to

reqBytes: ***Number***

The number of L4 request bytes.

reqL2Bytes: ***Number***

The number of L2 request bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request RTOs.

reqSize: ***Number***

The size of the request payload, expressed in bytes. The size does not include headers.

Access only on `SIP_REQUEST` events or an error will occur.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

rspBytes: ***Number***

The number of L4 response bytes.

rspL2Bytes: ***Number***

The number of L2 response bytes.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response RTOs.

rspSize: ***Number***

The size of the response payload, expressed in bytes. The size does not include headers.

Access only on `SIP_RESPONSE` events or an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

statusCode: ***Number***

The SIP response status code.

Access only on `SIP_RESPONSE` events or an error will occur.

The following table displays provisional responses:

Number	Response
100	Trying
180	Ringing
181	Call is Being Forwarded
182	Queued
183	Session In Progress
199	Early Dialog Terminated

The following table displays successful responses:

Number	Response
200	OK
202	Accepted
204	No Notification

The following table displays redirection responses:

Number	Response
300	Multiple Choice
301	Moved Permanently
302	Moved Temporarily
305	Use Proxy
380	Alternative Service

The following table displays client failure responses:

Number	Response
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

Number	Response
409	Conflict
410	Gone
411	Length Required
412	Conditional Request Failed
413	Request Entity Too Large
414	Request URI Too Long
415	Unsupported Media Type
416	Unsupported URI Scheme
417	Unknown Resource Priority
420	Bad Extension
421	Extension Required
422	Session Interval Too Small
423	Interval Too Brief
424	Bad Location Information
428	Use Identity Header
429	Provide Referrer Identity
430	Flow Failed
433	Anonymity Disallowed
436	Bad Identity Info
437	Unsupported Certificate
438	Invalid Identity Header
439	First Hop Lacks Outbound Support
470	Consent Needed
480	Temporarily Unavailable
481	Call/Transaction Does Not Exist
482	Loop Detected
483	Too Many Hops
484	Address Incomplete
485	Ambiguous
486	Busy Here
487	Request Terminated
488	Not Acceptable Here
489	Bad Event
491	Request Pending

Number	Response
493	Undecipherable
494	Security Agreement Required

The following table displays server failure responses:

Number	Response
500	Server Internal Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Server Timeout
505	Version Not Supported
513	Message Too Large
580	Precondition Failure

The following table displays global failure responses:

Name	Response
600	Busy Everywhere
603	Decline
604	Does Not Exist Anywhere
606	Not Acceptable

to: ***String***

The contents of the To header.

uri: ***String***

The URI for SIP request or response.

SMPP

The SMPP class enables you to access properties and record metrics from `SMPP_REQUEST` and `SMPP_RESPONSE` events.

Note: The `mdn`, `shortcode`, and `error` properties may be `null`, depending on availability and relevance.

Events

SMPP_REQUEST

Runs on every SMPP request processed by the device.

SMPP_RESPONSE

Runs on every SMPP response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `SMPP_RESPONSE` event. Record commits on `SMPP_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

command: *String*

The SMPP command ID.

destination: *String*

The destination address as specified in the `SMPP_REQUEST`. The value is `null` if this is not available for the current command type.

error: *String*

The error code corresponding to command_status. If the command status is ROK, the value is `null`.

Access only on `SMPP_RESPONSE` events or an error will occur.

message: *Buffer*

The contents of the short_message field on DELIVER_SM and SUBMIT_SM messages. The value is `null` if unavailable or not applicable.

Access only on `SMPP_REQUEST` events or an error will occur.

processingTime: *Number*

The server processing time, expressed in milliseconds. Equivalent to `rspTimeToFirstByte - reqTimeToLastByte`. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `SMPP_RESPONSE` events or an error will occur.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `SMPP.commitRecord` on a `SMPP_RESPONSE` event.

The record object contains the following default properties:

- clientZeroWnd
- command
- destination
- error
- reqSize
- reqTimeToLastByte
- rspSize
- rspTimeToFirstByte
- rspTimeToLastByte
- serverZeroWnd
- source
- processingTime

reqSize: *Number*

The size of the request payload, expressed in bytes.

reqTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the request, expressed in milliseconds. The value is `NaN` on malformed and aborted requests, or if the timing is invalid.

rspSize: ***Number***

The size of the response payload, expressed in bytes.

Access only on `SMPP_RESPONSE` events or an error will occur.

rspTimeToFirstByte: ***Number***

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `SMPP_RESPONSE` events or an error will occur.

rspTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `SMPP_RESPONSE` events or an error will occur.

source: ***String***

The source address as specified in the `SMPP_REQUEST`. The value is `null` if this is not available for the current command type.

SMTP

The SMTP class enables you to access properties and record metrics from `SMTP_REQUEST` and `SMTP_RESPONSE` events.

Events

SMTP_REQUEST

Runs on every SMTP request processed by the device.

SMTP_RESPONSE

Runs on every SMTP response processed by the device.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on a `SMTP_RESPONSE` event. Record commits on `SMTP_REQUEST` events are not supported.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

dataSize: ***Number***

The size of the attachment, expressed in bytes.

domain: ***String***

The domain of the address the message is coming from.

error: ***String***

The error code corresponding to status code.

Access only on `SMTP_RESPONSE` events or an error will occur.

headers: *Object*

An object that allows access to SMTP header names and values.

The value of the `headers` property is the same when accessed on either the `SMTP_REQUEST` or the `SMTP_RESPONSE` event.

isEncrypted: *Boolean*

The value is `true` if the application is encrypted using STARTTLS encryption.

isReqAborted: *Boolean*

The value is `true` if the connection is closed before the SMTP request is complete.

isRspAborted: *Boolean*

The value is `true` if the connection is closed before the SMTP response is complete.

Access only on `SMTP_RESPONSE` events or an error will occur.

method: *String*

The SMTP method.

processingTime: *Number*

The server processing time, expressed in milliseconds. Equivalent to `rspTimeToFirstByte - reqTimeToLastByte`. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `SMTP_RESPONSE` events or an error will occur.

recipient: *String*

The address the message should be sent to.

recipientList: *Array of Strings*

A list of recipient addresses.

The value of the `recipientList` property is the same when accessed on either the `SMTP_REQUEST` or the `SMTP_RESPONSE` event.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `SMTP.commitRecord` on a `SMTP_RESPONSE` event.

The record object contains the following default properties:

- clientZeroWnd
- dataSize
- domain
- error
- isEncrypted
- isReqAborted
- isRspAborted
- method
- processingTime
- recipient
- recipientList
- reqBytes
- reqL2Bytes
- reqPkts
- reqRTO
- reqSize
- reqTimeToLastByte
- roundTripTime

- rspBytes
- rspL2Bytes
- rspPkts
- rspRTO
- rspSize
- rspTimeToFirstByte
- rspTimeToLastByte
- sender
- serverZeroWnd
- statusCode
- statusText

Access the record object only on SMTP_RESPONSE events or an error will occur.

reqBytes: ***Number***

The number of L4 request bytes.

reqL2Bytes: ***Number***

The number of request L2 bytes.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request RTOs.

reqSize: ***Number***

The size of the request payload, expressed in bytes.

reqTimeToLastByte: ***Number***

The time from the first byte of the request until the last byte of the request, expressed in milliseconds. The value is NaN on malformed and aborted requests, or if the timing is invalid.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median TCP round-trip time (RTT), expressed in milliseconds. The value is NaN if there are no RTT samples.

rspSize: ***Number***

The size of the response, expressed in bytes.

rspL2Bytes: ***Number***

The number of response L2 bytes.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response RTOs.

rspSize: ***Number***

The size of the response payload, expressed in bytes.

Access only on SMTP_RESPONSE events or an error will occur.

rspTimeToFirstByte: ***Number***

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. The value is NaN on malformed and aborted responses, or if the timing is invalid.

Access only on SMTP_RESPONSE events or an error will occur.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. The value is `NaN` on malformed and aborted responses, or if the timing is invalid.

Access only on `SMTP_RESPONSE` events or an error will occur.

rspZeroWnd: *Number*

The number of zero windows in the response.

sender: *String*

The sender of the message.

statusCode: *Number*

The SMTP status code of the response.

Access only on `SMTP_RESPONSE` events or an error will occur.

statusText: *String*

The multi-line response string.

Access only on `SMTP_RESPONSE` events or an error will occur.

SSH

Secure Socket Shell (SSH) is a network protocol that provides a secure method for remote login and other network services over an unsecured network. The SSH class object enables you to access properties and record metrics from `SSH_CLOSE`, `SSH_OPEN` and `SSH_TICK` events.

Events

SSH_CLOSE

Runs when the SSH connection is shut down by being closed, expired, or aborted.

SSH_OPEN

Runs when the SSH connection is first fully established after session information is negotiated.

If a connection closes before `SSH_OPEN` has run, `SSH_OPEN`, `SSH_TICK`, and `SSH_CLOSE` will run in immediate succession.

SSH_TICK

Runs periodically on SSH flows.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on either an `SSH_OPEN`, `SSH_CLOSE`, or `SSH_TICK` event.

The event determines which properties are committed to the record object. To view the properties committed for each event, see the `record` property below.

For built-in records, each unique record is committed only once, even if `.commitRecord` is called multiple times for the same unique record.

Properties

clientBytes: *Number*

Upon an `SSH_CLOSE` event, the incremental number of application-level client bytes observed since the last `SSH_TICK` event. Does not specify the total number of bytes for the session.

clientCipherAlgorithm: *String*

The encryption cipher algorithm on the SSH client.

clientCompressionAlgorithm: *String*

The compression algorithm applied to data transferred over the connection by the SSH client.

clientImplementation: *String*

The SSH implementation installed on the client, such as OpenSSH or PUTTY.

clientL2Bytes: *Number*

The incremental number of L2 client bytes observed since the last SSH_TICK event. Does not specify the total number of bytes for the session.

Access only on SSH_CLOSE and SSH_TICK events or an error will occur.

clientMacAlgorithm: *String*

The Method Authentication Code (MAC) algorithm on the SSH client.

clientPkts: *Number*

The incremental number of client packets observed since the last SSH_TICK event. Does not specify the total number of packets for the session.

Access only on SSH_CLOSE and SSH_TICK events or an error will occur.

clientRTO: *Number*

The incremental number of client retransmission timeouts (RTOs) observed since the last SSH_TICK event. Does not specify the total number of RTOs for the session.

Access only on SSH_CLOSE and SSH_TICK events or an error will occur.

clientVersion: *String*

The version of SSH on the client.

clientZeroWnd: *Number*

The number of zero windows sent by the client.

Access only on SSH_OPEN, SSH_CLOSE, or SSH_TICK events or an error will occur.

duration: *Number*

The duration, expressed in milliseconds, of the SSH connection.

Access only on SSH_CLOSE events or an error will occur.

kexAlgorithm: *String*

The Key Exchange (Kex) algorithm on the SSH connection.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to SSH.commitRecord on either an SSH_OPEN, SSH_CLOSE, or SSH_TICK event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

SSH_TICK	SSH_OPEN	SSH_CLOSE
clientCipherAlgorithm	clientCipherAlgorithm	clientCipherAlgorithm
clientCompressionAlgorithm	clientCompressionAlgorithm	clientCompressionAlgorithm
clientImplementation	clientImplementation	clientImplementation
clientMacAlgorithm	clientMacAlgorithm	clientMacAlgorithm
clientVersion	clientVersion	clientVersion
clientZeroWnd	clientZeroWnd	clientZeroWnd
kexAlgorithm	kexAlgorithm	kexAlgorithm
serverCipherAlgorithm	serverCipherAlgorithm	serverCipherAlgorithm

SSH_TICK	SSH_OPEN	SSH_CLOSE
serverCompressionAlgorithm	serverCompressionAlgorithm	serverCompressionAlgorithm
serverImplementation	serverImplementation	serverImplementation
serverMacAlgorithm	serverMacAlgorithm	serverMacAlgorithm
serverVersion	serverVersion	serverVersion
serverZeroWnd	serverZeroWnd	serverZeroWnd
		duration

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

serverBytes: ***Number***

The incremental number of application-level server bytes observed since the last `SSH_TICK` event. Does not specify the total number of bytes for the session.

Access only on `SSH_CLOSE` and `SSH_TICK` events or an error will occur.

serverCipherAlgorithm: ***String***

The encryption cipher algorithm on the SSH server.

serverCompressionAlgorithm: ***String***

Returns the type of compression applied to data transferred over the connection by the SSH server.

serverImplementation: ***String***

The SSH implementation installed on the server, such as OpenSSH or PUTTY.

serverL2Bytes: ***Number***

The incremental number of L2 server bytes observed since the last `SSH_TICK` event. Does not specify the total number of bytes for the session.

Access only on `SSH_CLOSE` and `SSH_TICK` events or an error will occur.

serverMacAlgorithm: ***String***

The Method Authentication Code (MAC) algorithm on the SSH server.

serverPkts: ***Number***

The incremental number of server packets observed since the last `SSH_TICK` event. Does not specify the total number of packets for the session.

Access only on `SSH_CLOSE` and `SSH_TICK` events or an error will occur.

serverRTO: ***Number***

The incremental number of server retransmission timeouts (RTOs) observed since the last `SSH_TICK` event. Does not specify the total number of RTOs for the session.

Access only on `SSH_CLOSE` and `SSH_TICK` events or an error will occur.

serverVersion: ***String***

The version of SSH on the server.

serverZeroWnd: ***Number***

The number of zero windows sent by the server.

Access only on `SSH_OPEN`, `SSH_CLOSE`, or `SSH_TICK` events or an error will occur.

SSL

The SSL class enables you to access properties and record metrics from `SSL_OPEN`, `SSL_CLOSE`, `SSL_ALERT`, `SSL_ RECORD`, `SSL_HEARTBEAT`, and `SSL_RENEGOTIATE` events.

Events

SSL_ALERT

Runs when an SSL alert record is exchanged.

SSL_CLOSE

Runs when the SSL connection is shut down.

SSL_HEARTBEAT

Runs when an SSL heartbeat record is exchanged.

SSL_OPEN

Runs when the SSL connection is first established.

SSL_PAYLOAD

Runs when the decrypted SSL payload matches the criteria configured in the associated trigger.

Depending on the flow, the payload can be found in the following:

- `Flow.client.payload`
- `Flow.payload1`
- `Flow.payload2`
- `Flow.receiver.payload`
- `Flow.sender.payload`
- `Flow.server.payload`

Additional payload options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

SSL_RECORD

Runs when an SSL record is exchanged.

SSL_RENEGOTIATE

Runs on SSL renegotiation.

Methods

addApplication(name: *String*): *void*

Associates an SSL session with the named application to collect SSL metric data about the session. For example, you might use `SSL.addApplication` to associate SSL certificate data in an application.

An SSL session is associated with at most one application at a given instant. After an SSL session is associated with an application, that pairing is permanent for the lifetime of the session.

Call only on `SSL_OPEN` events.

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance only on `SSL_ALERT`, `SSL_CLOSE`, `SSL_HEARTBEAT`, `SSL_OPEN`, or `SSL_RENEGOTIATE` events. Record commits on `SSL_PAYLOAD` and `SSL_RECORD` events are not supported through this method.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

getClientExtensionData(extension_name | extension_id): *Buffer*

Returns the data for the specified extension if the extension was passed as part of the Hello message from the client and had data, otherwise returns `null`.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events.

getServerExtensionData(extension_name | extension_id): *Buffer*

Returns data for the specified extension if the extension was passed as part of the Hello message from the server and had data, otherwise returns `null`.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events.

hasClientExtension(extension_name | extension_id): *boolean*

Returns `true` for the specified extension if the extension was passed as part of the Hello message from the client.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events.

hasServerExtension(extension_name | extension_id): *boolean*

Returns `true` for the specified extension if the extension was passed as part of the Hello message from the server.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events.

The following table provides a list of known SSL extensions.

ID	Name
0	server_name
1	max_fragment_length
2	client_certificate_url
3	trusted_ca_keys
4	truncated_hmac
5	status_request
6	user_mapping
7	client_authz
8	server_authz
9	cert_type
10	elliptic_curves
11	ec_point_formats
12	srp
13	signature_algorithms
14	use_srtp
15	heartbeat
16	application_layer_protocal_negotiation
17	status_request_v2
18	signed_certificate_timestamp
19	client_certificate_type

ID	Name
20	server_certificate_type
35	SessionTicket TLS
65281	renegotiation_info

Properties

alertCode: *Number*

The numeric representation of the SSL alert. The following table displays the possible SSL alerts which are defined in the AlertDescription data structure in RFC 2246:

Alert	Number
close_notify	0
unexpected_message	10
bad_record_mac	20
decryption_failed	21
record_overflow	22
decompression_failure	30
handshake_failure	40
bad_certificate	42
unsupported_certificate	43
certificate_revoked	44
certificate_expired	45
certificate_unknown	46
illegal_parameter	47
unknown_ca	48
access_denied	49
decode_error	50
decrypt_error	51
export_restriction	60
protocol_version	70
insufficient_security	71
internal_error	80
user_canceled	90
no_renegotiation	100

If the session is opaque, the value is `SSL.ALERT_CODE_UNKNOWN (null)`.

Access only on `SSL_ALERT` events or an error will occur.

alertCodeName: ***String***

The name of the SSL alert associated with the alert code. See the `alertCode` property for alert names associated with alert codes. The value is `null` if no name is available for the associated alert code.

Access only on SSL_ALERT events or an error will occur.

alertLevel: ***Number***

The numeric representation of the SSL alert level. The following possible alert levels are defined in the AlertLevel data structure in RFC 2246:

- warning (1)
- fatal (2)

If the session is opaque, the value is `SSL.ALERT_LEVEL_UNKNOWN (null)`.

Access only on SSL_ALERT events or an error will occur.

certificate: ***SSLCert***

The SSL server certificate object associated with the communication. Each object contains the following properties:

fingerprint: ***String***

The string hex representation of the SHA-1 hash of the certificate. This is the same string shown in most browsers' client certificate information dialog boxes, but without spaces, such as the following example:

```
55F30E6D49E19145CF680E8B7E3DC8FC7041DC81
```

keySize: ***Number***

The key size of the server certificate.

issuer: ***String***

The common name of the server certificate issuer. The value is `null` if the issuer is not available.

notAfter: ***Number***

The expiration time of the server certificate, expressed in UTC.

notBefore: ***Number***

The start time of the server certificate, expressed in UTC. The server certificate is not valid before this time.

publicKeyExponent: ***String***

A string hex representation of the public key exponent. This is the same string shown in most browsers' certificate information dialog boxes, bit without spaces.

publicKeyModulus: ***String***

A string hex representation of the public key modulus. This is the same string shown in most browser's certificate information dialog boxes, but without spaces, such as `010001`

signatureAlgorithm: ***String***

The algorithm applied to sign the server certificate. The following table displays some of the possible values:

RFC	Algorithm
RFC 3279	• md2WithRSAEncryption • md5WithRSAEncryption • sha1WithRSAEncryption
RFC 4055	• sha224WithRSAEncryption

RFC	Algorithm
	• sha256WithRSAEncryption • sha384WithRSAEncryption • sha512WithRSAEncryption
4491	• id-GostR3411-94-with-Gost3410-94 • id-GostR3411-94-with-Gost3410-2001

subject: *String*

The subject common name (CN) of the server certificate.

cipherSuite: *String*

A string representing the cryptographic cipher suite negotiated between the server and the client.

cipherSuiteType: *Number*

The numeric value that represents the cryptographic cipher suite negotiated between the server and the client. Possible values are defined by the IANA TLS Cipher Suite Registry.

clientBytes: *Number*

The number of bytes sent by the client since the last `SSL_RECORD` event.

Access only on `SSL_RECORD` or `SSL_CLOSE` events or an error will occur.

clientCertificate: *SSLCert*

The SSL client certificate object associated with the communication. Each object contains the following properties:

fingerprint: *String*

The string hex representation of the SHA-1 hash of the client certificate, such as the following example:

```
55F30E6D49E19145CF680E8B7E3DC8FC7041DC81
```

keySize: *Number*

The key size of the client certificate.

issuer: *String*

The common name of the server certificate issuer. The value is `null` if the issuer is not available.

notAfter: *Number*

The expiration time of the client certificate, expressed in UTC.

notBefore: *Number*

The start time of the client certificate, expressed in UTC. The client certificate is not valid before this time.

publicKeyExponent: *String*

A string hex representation of the public key exponent.

publicKeyModulus: *String*

A string hex representation of the public key modulus, such as`010001`.

signatureAlgorithm: *String*

The algorithm applied to sign the client certificate. The following table displays some of the possible values:

RFC	Algorithm
RFC 3279	• md2WithRSAEncryption • md5WithRSAEncryption

RFC	Algorithm
	• sha1WithRSAEncryption
RFC 4055	• sha224WithRSAEncryption • sha256WithRSAEncryption • sha384WithRSAEncryption • sha512WithRSAEncryption
4491	• id-GostR3411-94-with-Gost3410-94 • id-GostR3411-94-with-Gost3410-2001

subject: ***String***

The subject common name (CN) of the client certificate.

clientExtensions: ***Array***

An array of client extension objects that contain the following properties:

id: ***Number***

The ID number of the SSL client extension

length: ***Number***

The full length of the SSL client extension, expressed in bytes.

Note: An extension might be truncated if the length exceeds the maximum size. The default is 512 bytes. Truncation has occurred if the value of this property is smaller than the buffer returned by the `getClientExtensionData` method.

name: ***String***

The name of the SSL client extension, if known. Otherwise "unknown" will be specified. See the table of known SSL extensions in the Methods section.

Access only on `SSL_OPEN` or `SSL_RENEGOTIATE` events or an error will occur.

clientL2Bytes: ***Number***

The number of L2 bytes sent by the client since the last `SSL_RECORD` event.

Access only on `SSL_RECORD` or `SSL_CLOSE` events or an error will occur.

clientPkts: ***Number***

The number of packets sent by the client since the last `SSL_RECORD` event.

Access only on `SSL_RECORD` or `SSL_CLOSE` events or an error will occur.

clientSessionId: ***String***

The client session ID as a byte array encoded as a string.

clientZeroWnd: ***Number***

The number of zero windows sent by the client since the last `SSL_RECORD` event.

Access only on `SSL_RECORD` or `SSL_CLOSE` events or an error will occur.

contentType: ***String***

The content type for the current record.

Access only on `SSL_RECORD` events or an error will occur.

handshakeTime: ***Number***

The amount of time required to negotiate the SSL connection, expressed in milliseconds. This is the amount of time between the client sending ClientHello and the server sending ChangeCipherSpec values as specified in RFC 2246.

Access only on SSL_OPEN or SSL_RENEGOTIATE events or an error will occur.

heartbeatPayloadLength: *Number*

The value of the payload_length field of the HeartbeatMessage data structure as specified in RFC 6520.

Access only on SSL_HEARTBEAT events or an error will occur.

heartbeatType: *Number*

The numeric representation of the HeartbeatMessageType field of the HeartbeartMessage data structure as specified in RFC 6520. Valid values are SSL.HEARTBEAT_TYPE_REQUEST (1), SSL.HEARTBEAT_TYPE_RESPONSE (2), or SSL.HEARTBEAT_TYPE_UNKNOWN (255).

Access only on SSL_HEARTBEAT events or an error will occur.

host: *string*

The SSL Server Name Indication (SNI), if present.

Access only on SSL_OPEN or SSL_RENEGOTIATE events or an error will occur.

isAborted: *Boolean*

The value is `true` if the SSL session is aborted.

Access only on SSL_CLOSE events or an error will occur.

isCompressed: *Boolean*

The value is `true` if the SSL record is compressed.

is StartTLS: *Boolean*

The value is `true` if negotiation of the SSL session was initiated by the STARTTLS mechanism of the protocol.

Access only on SSL_OPEN, SSL_CLOSE, SSL_ALERT, SSL_HEARTBEAT, or SSL_RENEGOTIATE events or an error will occur.

isV2ClientHello: *Boolean*

The value is `true` if the Hello record corresponds to SSLv2.

isWeakCipherSuite: *Boolean*

The value is `true` if the cipher suite encrypting the SSL session is considered weak. NULL, anonymous, and EXPORT cipher suites are considered weak, as are suites that encrypt with DES, 3DES, or RC4.

Access only on SSL_OPEN, SSL_CLOSE, SSL_ALERT, SSL_HEARTBEAT, or SSL_RENEGOTIATE events or an error will occur.

privateKeyId: *String*

The string ID associated with the private key if the ExtraHop appliance is decrypting SSL traffic. The value is `null` if the ExtraHop appliance is not decrypting SSL traffic.

To find the private key ID in the ExtraHop Admin UI, click **Capture** from the System Configuration section, click **SSL Decryption**, and then click a certificate. The pop-up window displays all identifiers for the certificate.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to SSL.commitRecord on either an SSL_OPEN, SSL_CLOSE, SSL_ALERT, SSL_HEARTBEAT, or SSL_RENEGOTIATE event.

The event on which the method was called determines which default properties the record object contains as displayed in the following table:

Event	Available properties
SSL_ALERT	• alertCode

Event	Available properties
	• alertCodeName • alertLevel • certificateFingerprint • certificateIssuer • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • isCompressed • version
SSL_CLOSE	• certificateFingerprint • certificateIssuer • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • clientZeroWnd • isAborted • isCompressed • serverZeroWnd • version
SSL_HEARTBEAT	• certificateFingerprint • certificateIssuer • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • clientZeroWnd • heartbeatPayloadLength • heartbeatType • isCompressed • serverZeroWnd • version
SSL_OPEN	• certificateFingerprint • certificateIssuer • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject

Event	Available properties
	• cipherSuite • clientZeroWnd • handshakeTime • host • isCompressed • clientZeroWnd • version
SSL_RENEGOTIATE	• certificateFingerprint • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • handshakeTime • host • isCompressed • version

recordLength: ***Number***

The value of the length field of the TLSPlaintext, TLSCompressed, and TLSCiphertext data structures as specified in RFC 5246.

Access only on `SSL_RECORD`, `SSL_ALERT`, or `SSL_HEARTBEAT` events or an error will occur.

recordType: ***Number***

The numeric representation of the type field of the TLSPlaintext, TLSCompressed, and TLSCiphertext data structures as specified in RFC 5246.

Access only on `SSL_RECORD`, `SSL_ALERT`, and `SSL_HEARTBEAT` events or an error will occur.

roundTripTime: ***Number***

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

Access only on `SSL_RECORD` or `SSL_CLOSE` events or an error will occur.

serverExtensions: ***Array***

An array of server extension objects that contain the following properties:

id: ***Number***

The ID number of the SSL server extension.

length: ***Number***

The full length of the SSL server extension, expressed in bytes.

Note: An extension might be truncated if the length exceeds the maximum size. The default is 512 bytes. Truncation has occurred if the value of this property is smaller than the buffer returned by the `getClientExtensionData` method.

name: ***String***

The name of the SSL server extension, if known. Otherwise "unknown" will be specified. See the table of known SSL extensions in the Methods section.

Access only on `SSL_OPEN` or `SSL_RENEGOTIATE` events or an error will occur.

serverBytes: ***Number***

The number of bytes sent by the server since the last SSL_RECORD event.

Access only on SSL_RECORD or SSL_CLOSE events or an error will occur.

serverL2Bytes: ***Number***

The number of L2 bytes sent by the server since the last SSL_RECORD event.

Access only on SSL_RECORD or SSL_CLOSE events or an error will occur.

serverPkts: ***Number***

The number of packets sent by the server since the last SSL_RECORD event.

Access only on SSL_RECORD or SSL_CLOSE events or an error will occur.

serverSessionId: ***String***

The server session ID, byte array encoded as a string.

serverZeroWnd: ***Number***

The number of zero windows sent by the server since the last SSL_RECORD event.

Access only on SSL_RECORD or SSL_CLOSE events or an error will occur.

version: ***Number***

The SSL protocol version with the RFC hexidecimal version number, expressed as a decimal.

Version	Hex	Decimal
SSLv2	0x200	2
SSLv3	0x300	768
TLS 1.0	0x301	769
TLS 1.1	0x302	770
TLS 1.2	0x303	771

TCP

The TCP class enables you to access properties and retrieve metrics from TCP events and from FLOW_TICK and FLOW_TURN events.

The FLOW_TICK and FLOW_TURN events are defined in the Flow section.

Events

TCP_CLOSE

Runs when the TCP connection is shut down by being closed, expired or aborted.

TCP_DESYNC

Runs when packet drops that will interrupt the processing of the TCP connection are detected.

TCP_OPEN

Runs when the TCP connection is first fully established.

The FLOW_CLASSIFY event runs after the TCP_OPEN event to determine the L7 protocol of the TCP flow.

TCP_PAYLOAD

Runs when the payload matches the criteria configured in the associated trigger.

Depending on the Flow, the TCP payload can be found in the following properties:

- `Flow.client.payload`
- `Flow.payload1`
- `Flow.payload2`
- `Flow.receiver.payload`
- `Flow.sender.payload`
- `Flow.server.payload`

Additional payload options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

Methods

getOption(): *Array*

Returns an array of all TCP options on the devices that have a kind number matching the passed in value. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.getOption()` or `TCP.server.getOption()`.

Applies only to `TCP_OPEN` events.

Properties

handshakeTime: *Number*

The amount of time required to negotiate the TCP connection, expressed in milliseconds.

Access only on `TCP_OPEN` events or an error will occur.

hasECNEcho: *Boolean*

The value is `true` if the ECN flag is set on a device during the three-way handshake. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.hasECNEcho` or `TCP.server.hasECNEcho`.

Access only on `TCP_OPEN` events or an error will occur.

hasECNEcho1: *Boolean*

The value is `true` if the ECN flag is set during the three-way handshake associated with one of two devices in the connection; the other device is represented by `hasECNEcho2`. The device represented by `hasECNEcho1` remains consistent for the connection.

Access only on `TCP_OPEN` events or an error will occur.

hasECNEcho2: *Boolean*

The value is `true` if the ECN flag is set during the three-way handshake associated with one of two devices in the connection; the other device is represented by `hasECNEcho1`. The device represented by `hasECNEcho2` remains consistent for the connection.

Access only on `TCP_OPEN` events or an error will occur.

initSeqNum: *Number*

The initial sequence number sent from a device during the three-way handshake. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.initSeqNum` or `TCP.server.initSeqNum`.

Access only on `TCP_OPEN` events or an error will occur.

initSeqNum1: *Number*

The initial sequence number during the three-way handshake associated with one of two devices in the connection; the other device is represented by `initSeqNum2`. The device represented by `initSeqNum1` remains consistent for the connection.

Access only on `TCP_OPEN` events or an error will occur.

initSeqNum2: ***Number***

The initial sequence number during the three-way handshake associated with one of two devices in the connection; the other device is represented by `initSeqNum1`. The device represented by `initSeqNum2` remains consistent for the connection.

Access only on `TCP_OPEN` events or an error will occur.

isAborted: ***Boolean***

The value is `true` if a TCP flow has been aborted through a TCP reset (RST) before the connection is shut down. The flow can be aborted by a device. If applicable, specify the device role in the syntax—for example, `TCP.client.isAborted` or `TCP.server.isAborted`.

This condition may be detected in any TCP event and in any impacted L7 events (for example, `HTTP_REQUEST` or `DB_RESPONSE`).

Note:
- An L4 abort occurs when a TCP connection is closed with a RST instead of a graceful shutdown.
- An L7 response abort occurs when a connection closes while in the middle of a response. This can be due to a RST, a graceful FIN shutdown, or an expiration.
- An L7 request abort occurs when a connection closes in the middle of a request. This can also be due to a RST, a graceful FIN shutdown, or an expiration.

isExpired: ***Boolean***

The value is `true` if the TCP connection expired at the time of the event. If applicable, specify TCP client or the TCP server in the syntax—for example, `TCP.client.isExpired` or `TCP.server.isExpired`.

Access only on `TCP_OPEN` events or an error will occur.

isReset: ***Boolean***

The value is `true` if a TCP reset (RST) was seen while the connection was in the process of being shut down.

nagleDelay: ***Number***

The number of Nagle delays associated with a device in the flow. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.nagleDelay` or `TCP.server.nagleDelay`.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

nagleDelay1: ***Number***

The number of Nagle delays associated with one of two devices in the flow; the other device is represented by `nagleDelay1`. The device represented by `nagleDelay2` remains consistent for the connection.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

nagleDelay1: ***Number***

The number of Nagle delays associated with one of two devices in the flow; the other device is represented by `nagleDelay2`. The device represented by `nagleDelay1` remains consistent for the connection.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

options: ***Array***

An array of objects representing the TCP options of a device in the initial handshake packets. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.options` or `TCP.server.options`. For more information, see the TCP options section below.

Access only on `TCP_OPEN` events or an error will occur.

options1: ***Array***

An array of options representing the TCP options in the initial handshake packets associated with one of two devices in the connection; the other device is represented by `options2`. The device represented by `options1` remains consistent for the connection. For more information, For more information, see the TCP options section below.

Access only on `TCP_OPEN` events or an error will occur.

options2: ***Array***

An array of options representing the TCP options in the initial handshake packets associated with one of two devices in the connection; the other device is represented by `options1`. The device represented by `options2` remains consistent for the connection. For more information, For more information, see the TCP options section below.

Access only on `TCP_OPEN` events or an error will occur.

rcvWndThrottle: ***Number***

The number of receive window throttles sent from a device in the flow. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.rcvWndThrottle` or `TCP.server.rcvWndThrottle`.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

rcvWndThrottle1: ***Number***

The number of receive window throttles sent from one of two devices in the flow; the other device is represented by `rcvWndThrottle2`. The device represented by `rcvWndThrottle1` remains consistent for the connection.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

rcvWndThrottle2: ***Number***

The number of receive window throttles sent from one of two devices in the flow; the other device is represented by `rcvWndThrottle1`. The device represented by `rcvWndThrottle2` remains consistent for the connection.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

retransBytes: ***Number***

The number of bytes retransmitted over TCP by a client or server device in the flow. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.retransBytes` or `TCP.server.retransBytes`.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

retransBytes1: ***Number***

The number of bytes retransmitted over TCP by one of two devices in the flow; the other device is represented by `retransBytes2`. The device represented by `retransBytes1` remains consistent for the connection.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

retransBytes2: ***Number***

The number of bytes retransmitted over TCP by one of two devices in the flow; the other device is represented by `retransBytes1`. The device represented by `retransBytes2` remains consistent for the connection.

Access only on `FLOW_TICK` or `FLOW_TURN` events or an error will occur.

wndSize: ***Number***

The size of the TCP sliding window on a device which is negotiated during the three-way handshake. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.wndSize` or `TCP.server.wndSize`.

Access only on `TCP_OPEN` events or an error will occur.

wndSize1: ***Number***

The size of the TCP sliding window negotiated during the three-way handshake associated with one of two devices in the connection; the other device is represented by `wndSize2`. The device represented by `wndSize1` remains consistent for the connection.

Access only on `TCP_OPEN` events or an error will occur.

wndSize2: ***Number***

The size of the TCP sliding window negotiated during the three-way handshake associated with one of two devices in the connection; the other device is represented by `wndSize1`. The device represented by `wndSize2` remains consistent for the connection.

Access only on `TCP_OPEN` events or an error will occur.

zeroWnd: ***Number***

The number of zero windows sent from a device in the flow. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.zeroWnd` or `TCP.server.zeroWnd`.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

zeroWnd1: ***Number***

The number of zero windows sent from one of two devices in the flow; the other device is represented by `zeroWnd2`. The device represented by `zeroWnd1` remains consistent for the connection.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

zeroWnd2: ***Number***

The number of zero windows sent from one of two devices in the flow; the other device is represented by `zeroWnd1`. The device represented by `zeroWnd2` remains consistent for the connection.

Access only on `FLOW_TICK` and `FLOW_TURN` events or an error will occur.

TCP options

All TCP Options objects have the following properties:

kind:*Number*

The TCP option kind number.

Kind Number	Meaning
0	End of Option List
1	No-Operation
2	Maximum Segment Size
3	Window Scale
4	SACK Permitted
5	SACK
6	Echo (obsoleted by option 8)
7	Echo Reply (obsoleted by option 8)
8	Timestamps
9	Partial Order Connection Permitted (obsolete)
10	Partial Order Service Profile (obsolete)

Kind Number	Meaning
11	CC (obsolete)
12	CC.NEW (obsolete)
13	CC.ECHO (obsolete)
14	TCP Alternate Checksum Request (obsolete)
15	TCP Alternate Checksum Data (obsolete)
16	Skeeter
17	Bubba
18	Trailer Checksum Option
19	MD5 Signature Option (obsoleted by option 29)
20	SCPS Capabilities
21	Selective Negative Acknowledgements
22	Record Boundaries
23	Corruption experienced
24	SNAP
25	Unassigned (released 2000-12-18)
26	TCP Compression Filter
27	Quick-Start Response
28	User Timeout Option (also, other known authorized use)
29	TCP Authentication Option (TCP-AO)
30	Multipath TCP (MPTCP)
31	Reserved (known authorized used without proper IANA assignment)
32	Reserved (known authorized used without proper IANA assignment)
33	Reserved (known authorized used without proper IANA assignment)
34	TCP Fast Open Cookie
35-75	Reserved
76	Reserved (known authorized used without proper IANA assignment)
77	Reserved (known authorized used without proper IANA assignment)
78	Reserved (known authorized used without proper IANA assignment)
79-252	Reserved
253	RFC3692-style Experiment 1 (also improperly used for shipping products)
254	RFC3692-style Experiment 2 (also improperly used for shipping products)

name: ***String***

The name of the TCP option.

The following list contains the names of common TCP options and their specific properties:

Maximum Segment Size (name 'mss', option kind 2)

value: *Number*

The maximum segment size.

Window Scale (name 'wscale', kind 3)

value: *Number*

The window scale factor.

Selective Acknowledgement Permitted (name 'sack-permitted', kind 4)

No additional properties. Its presence indicates that the selective acknowledgment option was included in the SYN.

Timestamp (name 'timestamp', kind 8)

tsval: *Number*

The TSVal field for the option.

tsecr: *Number*

The TSecr field for the option.

Quickstart Response (name 'quickstart-rsp', kind 27)

rate-request: *Number*

The requested rate for transport, expressed in bytes per second.

ttl-diff: *Number*

The TTLDif.

qs-nonce: *Number*

The QS Nonce.

Akamai Address (name 'akamai-addr', kind 28)

value: *IPAddr*

The IP Address of the Akamai server.

User Timeout (name 'user-timeout', kind 28)

value: *Number*

The user timeout.

Authentication (name 'tcp-ao', kind 29)

keyId property: *Number*

The key id for the key in use.

rNextKeyId: *Number*

The key id for the "receive next" key id.

mac: *Buffer*

The message authentication code.

Multipath (name 'mptcp', kind 30)

value: *Buffer*

The multipath value.

Note: The Akamai address and user timeout options are differentiated by the length of the option.

The following is an example of TCP options:

```
if (TCP.client.options != null) {
```

```
    var optMSS = TCP.client.getOption(2)

    if (optMSS && (optMSS.value > 1460)) {
        Network.metricAddCount('large_mss', 1);
        Network.metricAddDetailCount('large_mss_by_client_ip',
                                     Flow.client.ipaddr + " " + optMSS.value,
 1);
    }

}
```

Telnet

The Telnet class enables you to access properties and record metrics from `TELNET_MESSAGE` events.

Events

TELNET_MESSAGE

Runs on a telnet command or line of data from the telnet client or server.

Methods

commitRecord(): *void*

Commits a record object to the ExtraHop Explore appliance on an `TELNET_MESSAGE` event.

To view the default properties committed to the record object, see the `record` property below.

For built-in records, each unique record is committed only once, even if the `commitRecord()` method is called multiple times for the same unique record.

Properties

command: *String*

The command type. The value is `null` if the event was run due to a line of data being sent.

The following values are valid:

- Abort
- Abort Output
- Are You There
- Break
- Data Mark
- DO
- DON'T
- End of File
- End of Record
- Erase Character
- Erase Line
- Go Ahead
- Interrupt Process
- NOP
- SB
- SE
- Suspend
- WILL
- WON'T

line: ***String***

A line of the data sent by the client or server. Terminal escape sequences and special characters are filtered out. Cursor movement and line editing are not simulated except for backspace characters.

option: ***String***

The option being negotiated. The value is `null` if the option is invalid. The following values are valid:

- 3270-REGIME
- AARD
- ATCP
- AUTHENTICATION
- BM
- CHARSET
- COM-PORT-OPTION
- DET
- ECHO
- ENCRYPT
- END-OF-RECORD
- ENVIRON
- EXPOPL
- EXTEND-ASCII
- FORWARD-X
- GMCP
- KERMIT
- LINEMODE
- LOGOUT
- NAOCRD
- NAOFFD
- NAOHTD
- NAOHTS
- NAOL
- NAOLFD
- NAOP
- NAOVTD
- NAOVTS
- NAWS
- NEW-ENVIRON
- OUTMRK
- PRAGMA-HEARTBEAT
- PRAGMA-LOGON
- RCTE
- RECONNECT
- REMOTE-SERIAL-PORT
- SEND-LOCATION
- SEND-URL
- SSPI-LOGON
- STATUS
- SUPDUP
- SUPDUP-OUTPUT

- SUPPRESS-GO-AHEAD
- TERMINAL-SPEED
- TERMINAL-TYPE
- TIMING-MARK
- TN3270E
- TOGGLE-FLOW-CONTROL
- TRANSMIT-BINARY
- TTYLOC
- TUID
- X-DISPLAY-LOCATION
- X.3-PAD
- XAUTH

optionData: *Buffer*

For option subnegotiations (the SB command), the raw, option-specific data sent. The value is `null` if the command is not SB.

record: *Object*

The record object committed to the ExtraHop Explore appliance through a call to `Telnet.commitRecord` on an `TELNET_MESSAGE` event.

The record object contains the following default properties:

- command
- option
- receiverBytes
- receiverL2Bytes
- recieverPkts
- receiverRTO
- receiverZeroWnd
- roundTripTime
- senderBytes
- senderL2Bytes
- senderPkts
- senderRTO
- senderZeroWnd

receiverBytes: *Number*

The number of application-level bytes from the receiver.

receiverL2Bytes: *Number*

The number of L2 bytes from the receiver.

receiverPkts: *Number*

The number of packets from the receiver.

receiverRTO: *Number*

The number of RTOs from the receiver.

receiverZeroWnd: *Number*

The number of zero windows sent by the receiver.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. The value is `NaN` if there are no RTT samples.

senderBytes: *Number*

The number of application-level bytes from the sender.

senderL2Bytes: ***Number***

The number of L2 bytes from the sender.

senderPkts: ***Number***

The number of packets from the sender.

senderRTO: ***Number***

The number of RTOs from the sender.

senderZeroWnd: ***Number***

The number of zero windows sent by the sender.

Turn

Turn is a class that enables you to access properties and record metrics available on `FLOW_TURN` events.

The `FLOW_TURN` event is defined in the Flow section.

Properties

clientBytes: ***Number***

The size of the request that the client transferred, expressed in bytes.

clientTransferTime: ***Number***

The client transfer time, expressed in milliseconds. Corresponds to the Network In metric in the ExtraHop Web UI.

processingTime: ***Number***

The time elapsed between the client having transferred the request to the server and the server beginning to transfer the response back to the client, expressed in milliseconds. Corresponds to the Processing Time metric in the Turn Timing sections of the ExtraHop Web UI.

reqSize: ***Number***

The size of the request payload, expressed in bytes.

reqTransferTime: ***Number***

The request transfer time, expressed in milliseconds. If the request is contained in a single packet, the transfer time is zero. If the request spans multiple packets, the value is the amount of time between detection of the first request packet and detection of the last packet by the ExtraHop system. A high value might indicate a large request or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

rspSize: ***Number***

The size of the response payload, expressed in bytes.

rspTransferTime: ***Number***

The response transfer time, expressed in milliseconds. If the response is contained in a single packet, the transfer time is zero. If the response spans multiple packets, the value is the amount of time between detection of the first response packet and detection of the last packet by the ExtraHop system. A high value might indicate a large response or a network delay. The value is `NaN` if there is no valid measurement, or if the timing is invalid.

serverBytes: ***Number***

The size of the response that the server transferred, expressed in bytes.

serverTransferTime: ***Number***

The server transfer time, expressed in milliseconds. Corresponds to the Network Out metric in the ExtraHop Web UI.

sourceDevice: ***Device***

The source device object. See the Device class for more information.

thinkTime: *Number*

The time elapsed between the server having transferred the response to the client and the client transferring a new request to the server, expressed in milliseconds. The value is `NaN` if there is no valid measurement.

UDP

The UDP class enables you to access properties and retrieve metrics from UDP events and from `FLOW_TICK` and `FLOW_TURN` events.

The `FLOW_TICK` and `FLOW_TURN` events are defined in the Flow section.

Events

UDP_PAYLOAD

Runs when the payload matches the criteria configured in the associated trigger.

Depending on the Flow, the UDP payload can be found in the following properties:

- `Flow.client.payload`
- `Flow.payload1`
- `Flow.payload2`
- `Flow.receiver.payload`
- `Flow.sender.payload`
- `Flow.server.payload`

Additional payload options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

WebSocket

The WebSocket class enables you to access metrics from WebSocket activity.

Events

WEBSOCKET_OPEN

Runs when a successful handshake has been observed.

WEBSOCKET_CLOSE

Runs when both close frames are observed, or when the underlying TCP connection is closed.

WEBSOCKET_MESSAGE

Runs when all frames of a text or binary message have been observed.

Properties

clientBytes: *Number*

The total number of bytes sent by the client during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

clientL2Bytes: *Number*

The total number of L2 bytes sent by the client during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

clientPkts: *Number*

The total number of packets sent by the client during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

clientRTO: ***Number***

The total number of client retransmission timeouts (RTOs) observed during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

closeReason: ***String***

The text message included in the first observed close frame that describes the reason the connection was closed. The value is `null` if the frame does not contain this information.

Access only on `WEBSOCKET_CLOSE` events or an error will occur.

host: ***String***

The host provided in the handshake request from the client. The value is `null` if no host is provided.

Access only on `WEBSOCKET_OPEN` events or an error will occur.

isClientClose: ***Boolean***

The value is `true` if the initial close frame was sent by the client.

Access only on `WEBSOCKET_CLOSE` events or an error will occur.

isEncrypted: ***Boolean***

The value is `true` if the WebSocket connection is SSL-encrypted.

isServerClose: ***Boolean***

The value is `true` if the initial close frame was sent by the server. The value is `false` if the connection was terminated abnormally.

Access only on `WEBSOCKET_CLOSE` events or an error will occur.

msg: ***Buffer***

The Buffer contents of the WebSocket message. The buffer is `null` if the contents exceeded that maximum length.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

msgType: ***String***

The type of WebSocket message frame. Valid values are `TEXT` or `BINARY`.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

origin: ***String***

The origin URL provided by in the handshake request initiated by the client.

Access only on `WEBSOCKET_OPEN` events or an error will occur.

serverBytes: ***Number***

The total number of bytes returned by the server during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

serverL2Bytes: ***Number***

The total number of L2 bytes returned by the server during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

serverPkts: ***Number***

The total number of packets returned by the server during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

serverRTO: ***Number***

The total number of server retransmission timeouts (RTOs) observed during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events or an error will occur.

statusCode: ***Number***

The status code that represents the reason the connection was closed, as defined in RFC 6455.

The value is `NO_STATUS_RECVD` (`1005`) if the initial close frame does not include a status code. The value is `NaN` if connection was terminated abnormally.

Access only on `WEBSOCKET_CLOSE` events or an error will occur.

uri: ***String***

The URI provided in the handshake request initiated by the client.

Access only on `WEBSOCKET_OPEN` events or an error will occur.

Open data stream classes

The Trigger API classes in this section enable you to send data to a third-party syslog, database, or server through an open data stream (ODS) you have configured in the ExtraHop Admin UI.

Class	Description
Remote.HTTP	Enables you to submit HTTP request data to a remote server thorugh REST API endpoints.
Remote.Kafka	Enables you to submit message data to remote a Kafka server.
Remote.MongoDB	Enables you to insert, remove, and update document collections to a remote MongoDB database.
Remote.Raw	Enables you to submit raw data to a remote server through a TCP or UDP port.
Remote.Syslog	Enables you to send syslog data to a remote server.

Remote.HTTP

The Remote.HTTP class enables you to submit HTTP request data to an HTTP open data stream (ODS) target and provides access to HTTP REST API endpoints.

You must first configure an HTTP ODS target from the ExtraHop Admin UI, which requires full system privileges. For configuration information, see the Open Data Streams section in the ExtraHop Admin UI Guide.

Methods

request

Submits an HTTP REST request to a configured HTTP ODS.

Syntax:

```
Remote.HTTP("name").request("method", {path: "path", [headers:
 headers],
[payload: "payload"]})
```

```
Remote.HTTP.request("method", {path: "path", [headers: headers],
[payload: "payload"]})
```

Parameters:

method: ***String***

String that specifies the HTTP method.

- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE

- OPTIONS
- CONNECT
- PATCH

options: *Object*

The options object has the following properties:

path: *String*

The string specifying the request path.

headers: *Object*

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- Connection
- Authorization
- Proxy-Connection
- Content-Length
- X-Forwarded-For
- Transfer-Encoding

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Admin UI.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Admin UI. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests by using the Content- Encoding header.

```
'Content-Encoding': 'gzip'
```

The following values are supported for this compression header:

- gzip
- deflate

payload: *String* | *Buffer*

The optional string or Buffer specifying the request payload.

name: *String*

The name of the HTTP Data Stream Configuration previously configured in the ExtraHop Admin UI. If no name is specified, the request will go to the first (default) Data Stream Configuration.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

Helper methods

The following helper methods allow you to more easily make use of the most common HTTP methods.

- `Remote.HTTP.delete`
- `Remote.HTTP.get`
- `Remote.HTTP.patch`

- `Remote.HTTP.post`
- `Remote.HTTP.put`

Syntax:

```
Remote.HTTP("name").delete({path: "path", [headers: headers],
[payload: "payload"]})
```

```
Remote.HTTP.delete({path: "path", [headers: headers], [payload:
 "payload"]})
```

```
Remote.HTTP("name").get({path: "path", [headers: headers],
[payload: "payload"]})
```

```
Remote.HTTP.get({path: "path", [headers: headers], [payload:
 "payload"]})
```

```
Remote.HTTP("name").patch({path: "path", [headers: headers],
[payload: "payload"]})
```

```
Remote.HTTP.patch({path: "path", [headers: headers], [payload:
 "payload"]})
```

```
Remote.HTTP("name").post({path: "path", [headers: headers],
[payload: "payload"]})
```

```
Remote.HTTP.post({path: "path", [headers: headers], [payload:
 "payload"]})
```

```
Remote.HTTP("name").put({path: "path", [headers: headers],
[payload: "payload"]})
```

```
Remote.HTTP.put({path: "path", [headers: headers], [payload:
 "payload"]})
```

Parameters:

All of these helper methods take the following parameters:

options: *Object*

The options object has the following properties:

path: *String*

The string specifying the request path.

headers: *Object*

The optional object specifying the request headers.

payload: *String*

The optional string specifying the request payload.

name: *String*

The name of the HTTP Data Stream Configuration previously configured in the ExtraHop Admin UI. If no name is specified, the request will go to the first (default) Data Stream Configuration.

Return values:

Returns `true` if the request is queued, otherwise returns `false`.

Examples

HTTP GET

The following example will issue an HTTP GET request to the HTTP configuration called "my_destination" and a path that is the URI, including query string variables, that you want the request to be sent to.

```
Remote.HTTP("my_destination").get( { path: "/?
example=example1&example2=my_data" } );
```

HTTP POST

The following example will issue an HTTP POST request to the HTTP configuration called "my_destination", the path that is the URI you want the request to be sent to and a payload. The payload can be data similar to what an HTTP client would send, a JSON blob, XML, or whatever else you want to send.

```
Remote.HTTP("my_destination").post( { path: "/", payload: "data I want
 to
send" } );
```

Custom HTTP Headers

The following example defines a Javascript object with keys to represent the header names and their corresponding values and provide that in a call as the value for the headers key.

```
var my_json = { example: "my_data", example1: 42, example2: false };
var headers = { "Content-Type": "application/json" };
Remote.HTTP("my_destination").post( { path: "/", headers: headers,
 payload:
JSON.stringify(my_json) });
```

Trigger Examples

- Example: Send data to Elasticsearch with Remote.HTTP
- Example: Send data to Azure with Remote.HTTP

Remote.Kafka

The Remote.Kafka class enables you to submit message data to a Kafka server through a Kafka open data stream (ODS).

You must first configure a Kafka ODS target from the ExtraHop Admin UI, which requires full system privileges. For configuration information, see the Open Data Streams section in the ExtraHop Admin UI Guide.

Methods

send

Sends an array of messages to a single topic with an option to indicate which Kafka partition the messages will be sent to.

Syntax:

```
Remote.Kafka.send({"topic": "topic", "messages":[messages],
["partition": partition]})
```

```
Remote.Kafka("name").send({"topic": "topic", "messages":
[messages],
["partition": partition]})
```

Parameters:

If `Remote.Kafka.send` is called with one argument, that argument must be a JavaScript object that contains the following fields:

topic: *String*

A string corresponding to the topic associated with the Kafka `send` method. The topic string has the following restrictions:

- The string length must be between 1 and 249 characters.
- The string supports only alphanumeric characters and the following symbols: "-", "_", or ".".
- The string cannot be "." or "..".

messages: *Array*

An array of messages to be sent. An element in this array cannot be an array itself.

partition: *Number*

An optional non-negative integer corresponding to the Kafka partition the messages will be sent to. The `send` action will fail silently if the number provided exceeds the number of partitions on the Kafka cluster associated with the given target. This value is ignored unless Manual Partitioning is selected as the partitioning strategy when you configured the open data stream in the ExtraHop Admin UI.

Return values:

None

Examples:

```
Remote.Kafka.send({"topic": "my_topic", "messages": ["hello
 world", 42,
DHCP.msgType], "partition": 2});
```

```
Remote.Kafka("my-target").send({"topic": "my_topic", "messages":
 [HTTP.query,
HTTP.uri]});
```

send

Sends messages to a single topic.

Syntax:

```
Remote.Kafka.send("topic", message1, message2, etc...)
```

```
Remote.Kafka("my-target").send("topic", message1, message2,
 etc...)
```

Parameters:

If `Remote.Kafka.send` is called with multiple arguments, the following fields are required:

topic: *String*

A string corresponding to the topic associated with the Kafka `send` method. The topic string has the following restrictions:

- The string length must be between 1 and 249 characters.
- The string supports only alphanumeric characters and the following symbols: "-", "_", or ".".
- The string cannot be "." or "..".

messages: *String* | *Number*

The messages to be sent. This cannot be an array.

Return values:

None.

Examples:

```
Remote.Kafka.send("my_topic", HTTP.query, HTTP.uri);
```

```
Remote.Kafka("my-target").send("my_topic", HTTP.query, HTTP.uri);
```

Remote.MongoDB

The Remote.MongoDB class enables you to insert, remove, and update MongoDB document collections through a MongoDB open data stream (ODS).

You must first configure a MongoDB ODS target from the ExtraHop Admin UI, which requires full system privileges. For configuration information, see the Open Data Streams section in the ExtraHop Admin UI Guide.

Methods

Insert

Inserts a document or array of documents into a collection, and handles both add and modify operations.

Syntax:

```
Remote.MongoDB.insert("db.collection", document);
```

```
Remote.MongoDB("name").insert("db.collection", document);
```

Parameters:

collection: ***String***

The name of a group of MongoDB documents.

document: ***Object***

The JSON-formatted document to insert into the collection.

name: ***String***

The name of the host specified when you configured the open data stream in the ExtraHop Admin UI. If no host was specified, the value is the default host.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

Examples:

```
Remote.MongoDB.insert('sessions.sess_www',
   {
   'session_id': "100",
   'path': "/index.html",
   'host': "www.extrahop.com",
   'status': "500",
   'src_ip': "10.10.1.120",
   'dst_ip': "10.10.1.100"
   }
);
var x = Remote.MongoDB.insert('test.tbc', {example: 1});
if (x) {
   Network.metricAddCount('perf_trigger_success', 1);
}
```

```
else {
   Network.metricAddCount('perf_trigger_error', 1);
}
```

Refer to http://docs.mongodb.org/manual/reference/method/db.collection.insert/#db.collection.insert for more information.

Remove

Removes documents from a collection.

Syntax:

```
Remote.MongoDB.remove("db.collection", document, [justOnce]);
```

```
Remote.MongoDB("name").remove("db.collection", document,
 [justOnce]);
```

Parameters:

collection: ***String***

The name of a group of MongoDB documents.

document: ***Object***

The JSON-formatted document to remove from the collection.

justOnce: ***Boolean***

An optional boolean parameter that limits the removal to just one document. Set to `true` to limit the deletion. The default value is `false`.

name: ***String***

The name of the host specified when you configured the open data stream in the ExtraHop Admin UI. If no host was specified, the value is the default host.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

Examples:

```
var x = Remote.MongoDB.remove('test.tbc', {qty: 100000}, false);
if (x) {
   Network.metricAddCount('perf_trigger_success', 1);
}
else {
   Network.metricAddCount('perf_trigger_error', 1);
}
```

Refer to http://docs.mongodb.org/manual/reference/method/db.collection.remove/#db.collection.remove for more information.

Update

Modifies an existing document or documents in a collection.

Syntax:

```
Remote.MongoDB.update("db.collection", document, update,
 [{"upsert":true,
"multi":true}]);
```

```
Remote.MongoDB("name").update("db.collection", document, update,
[{"upsert":true, "multi":true}]);
```

Parameters:

collection: *String*

The name of a group of MongoDB documents.

document: *Object*

The JSON-formatted document that specifies which documents to update or insert, if upsert option is set to true.

update: *Object*

The JSON-formatted document that specifies how to update the specified documents.

name: *String*

The name of the host specified when you configured the open data stream in the ExtraHop Admin UI. If no host was specified, the value is the default host.

options:

Optional flags that indicate the following additional update options:

upsert: *Boolean*

An optional boolean parameter that creates a new document when no document matches the query data. Set to `true` to create a new document. The default value is `false`.

multi: *Boolean*

An optional boolean parameter that updates all documents that match the query data. Set to `true` to update multiple documents. The default value is `false`, which updates only the first document returned.

Return Values:

The value is `true` if the request is queued, otherwise returns FALSE.

Examples:

```
var x = Remote.MongoDB.update('test.tbc', {_id: 1}, {$set:
 {example:2}},
{'upsert':true, 'multi':false} );
if (x) {
   Network.metricAddCount('perf_trigger_success', 1);
}
else {
   Network.metricAddCount('perf_trigger_error', 1);
}
```

Refer to http://docs.mongodb.org/manual/reference/method/db.collection.update/#db.collection.update for more information.

Trigger Examples

- Example: Parse syslog over TCP with universal payload analysis

Remote.Raw

The Remote.Raw class enables you to submit raw data to a Raw open data stream (ODS) target through a TCP or UDP port.

You must first configure a raw ODS target from the ExtraHop Admin UI, which requires full system privileges. For configuration information, see the Open Data Streams section in the ExtraHop Admin UI Guide.

Note: If the Gzip feature is enabled for the raw data stream in the ExtraHop Admin UI, the Remote.Raw class will automatically compress the data with Gzip.

Methods

Send

Sends raw bytes to a raw data ODS target. If a name is specified as an optional argument to the Remote.Raw class, then the data is sent to the named ODS target, which was configured in the ExtraHop Admin UI. If a name is not specified, the data is sent to the default target.

Syntax:

```
Remote.Raw.send("my data")
```

```
Remote.Raw("name").send("my data")
```

Parameters:

`Remote.Raw.send` accepts only one argument which is the JavaScript string representing the bytes to send.

Return Values:

None

Examples

```
Remote.Raw.send("data over the wire");
```

```
Remote.Raw("my-target").send("extra data for my-target");
```

Remote.Syslog

The Remote.Syslog class enables you to create remote syslog messages and send message data to a Syslog open data stream (ODS).

You must first configure a syslog ODS target from the ExtraHop Admin UI, which requires full system privileges. For configuration information, see the Open Data Streams section in the ExtraHop Admin UI Guide.

Each of the following methods sends a message to the remote syslog server with a severity corresponding to the method name.

- `emerg(message:String):void`
- `alert(message:String):void`
- `crit(message:String):void`
- `error(message:String):void`
- `warn(message:String):void`
- `notice(message:String):void`
- `info(message:String):void`
- `debug(message:String):void`

For instance, to send an rsyslog message to the default host for every HTTP response that includes the URI, request and response sizes, and server processing time, add the following trigger on the `HTTP_RESPONSE` event:

```
Remote.Syslog.info("eh_event=web uri=" + HTTP.uri + " req_size=" +
 HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" + HTTP.processingTime);
```

A host name was specified when you configured the open data stream in the ExtraHop Admin UI. Add the `name` parameter to the trigger code to send rsyslog messages to the host name specified in the configuration. If no host was specified, the parameter value is the default host. To send a rsyslog message

to the named host for every HTTP response that includes the URI, request and response sizes, and server processing time, add the following trigger on the HTTP_RESPONSE event:

```
Remote.Syslog("name").info("eh_event=web uri=" + HTTP.uri + " req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

If submitting an rsyslog message succeeds, the APIs will return true. In the case of either success or failure, the trigger will continue to execute as a failure to submit an rsyslog message is a "soft" failure. Incorrect usage of the APIs, i.e. calling them with the wrong number or type of arguments, will still result in trigger execution stopping.

Message size

By default, the message sent to the remote server is limited to 1024 bytes, including the message header and trailer (if necessary). The message header always includes the priority and timestamp, which together are up to 30 bytes.

If you have full system privileges, you can increase the default message size in the ExtraHop Admin UI. Click **Running Config** from the Appliance Settings section, and then click **Edit config**. Go to the "remote" section, and under the ODS target name, such as "rsyslog", add "message_length_max" as shown in the example below. The "message_length_max" setting applies only to the message passed to the Remote.Syslog APIs; the message header does not count against the maximum.

```
"remote": {
   "rsyslog": {
   "host": "hostname",
   "port": 54322,
   "ipproto": "tcp",
   "message_length_max": 4000
   }
}
```

Timestamp

The default timestamp format for rsyslog messages is UTC. You can change the timestamp to local time when you configure the open data stream in the ExtraHop Admin UI.

Trigger Examples

- Example: Send discovered device data to a remote syslog server
- Example: Parse syslog over TCP with universal payload analysis
- Example: Matching topnset keys

Datastore classes

The Trigger API classes in this section enable you to access datastore, or bridge, metrics.

Class	Description
AlertRecord	Enables you to access alert information from ALERT_RECORD_COMMIT events.
Dataset	Enables you to access raw dataset values and provides an interface for computing percentiles.
Discover	Enables you to access newly discovered devices or applications on NEW_DEVICE and NEW_APPLICATION events.
MetricCycle	Enables you to retrieve metrics published during a metric cycle interval represented by the METRIC_CYCLE_BEGIN, METRIC_CYCLE_END, and METRIC_RECORD_COMMIT events.
MetricRecord	Enables access to the current set of metrics in METRIC_RECORD_COMMIT events.
Sampleset	Enables you to retrieve summary data about metrics.
Topnset	Enables you to access data from a collection of metrics grouped by a key such as a URI or a client IP address.

AlertRecord

The AlertRecord class enables you to access alert information from ALERT_RECORD_COMMIT events.

Events

ALERT_RECORD_COMMIT

Runs when an alert occurs. Provides access to information about the alert.

Additional datastore options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

Properties

description: *String*

The description of the alert as it appears in the ExtraHop Web UI.

id: *String*

The ID of the alert record. For example, extrahop.device.alert. A list of IDs can be supplied as a hint to the ALERT_RECORD_COMMIT event.

name: *String*

The name of the alert that occurred.

object: *Object*

The object the alert applies to. For device, application, or VLAN alerts, this property will contain a Device, Application, or VLAN instance, respectively. For capture alerts (e.g., extrahop.- capture.net), the property will contain the global Network class.

time: *Number*

The time that the alert record will be published with.

Dataset

The dataset class enables you to access raw dataset values and provides an interface for computing percentiles.

Instance Methods

percentile(...): *Array* | *Number*

Accepts a list of percentiles (either as an array or as multiple arguments) to compute and returns the computed percentile values for the dataset. If passed a single numeric argument, a number is returned. Otherwise an array is returned. The arguments must be in ascending order with no duplicates. Floating point values are allowed (e.g., 99.99).

Instance Properties

entries: *Array*

An array of objects with frequency and value attributes. This is analogous to a frequency table where there is a set of values and the number of times each value was observed.

Discover

The Discover class enables you to access newly discovered devices and applications on `NEW_DEVICE` and `NEW_APPLICATION` events.

Events

NEW_APPLICATION

Runs when an application is first discovered.

NEW_DEVICE

Runs when a device is first discovered.

Properties

application: *Application*

A newly discovered application.

Applies only to `NEW_APPLICATION` events.

device: *Device*

A newly discovered device.

Applies only to `NEW_DEVICE` events.

Trigger Examples

- Example: Send discovered device data to a remote syslog server

MetricCycle

The MetricCycle class represents an interval during which metrics are published. The MetricCycle class is valid on `METRIC_CYCLE_BEGIN`, `METRIC_CYCLE_END`, and `METRIC_RECORD_COMMIT` events.

The `METRIC_RECORD_COMMIT` event is defined in the MetricRecord section.

Events

METRIC_CYCLE_BEGIN

Runs when a metric interval begins.

METRIC_CYCLE_END

Runs when a metric interval ends.

Additional datastore options are available when you create a trigger that runs on either of these events. See Advanced trigger options for more information.

Properties

id: *String*

A string representing the metric cycle. Possible values are:

- 30sec
- 5min
- 1hr
- 24hr

interval: *Object*

An object containing from and until properties, expressed in milliseconds since the epoch.

store: *Object*

An object that retains information across all the `METRIC_RECORD_COMMIT` events that occur during a metric cycle, that is, from the `METRIC_CYCLE_BEGIN` event to the `METRIC_CYCLE_END` event. This object is analogous to `Flow.store`. The `store` object is shared among triggers for `METRIC_* events`. It is cleared at the end of a metric cycle.

Trigger Examples

- Example: Add metrics to the metric cycle store

MetricRecord

The MetricRecord class enables access to the current set of metrics in `METRIC_RECORD_COMMIT` events.

Events

METRIC_RECORD_COMMIT

Runs when a metric record is committed to the datastore and provides access to various metric properties.

Additional datastore options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

Properties

fields: *Object*

An object containing metric values. The properties are the field names and the values can be numbers, Topnset, Dataset or Sampleset.

id: *String*

The metric type. For example, `extrahop.device.http_server`.

object: *Object*

The object the metric applies to. For device, application, or VLAN metrics, this property will contain a Device, Application, or VLAN instance, respectively. For capture metrics, such as `extrahop.capture.net`, the property will contain the global Network class.

time: *Number*

The time that the metric record will be published with.

Trigger Examples

- Example: Matching topnset keys
- Example: Add metrics to the metric cycle store

Sampleset

The Sampleset class enables you to retrieve summary data about metrics.

Properties

count: *Number*

The number of samples in the sampleset.

mean: *Number*

The average value of the samples.

sigma: *Number*

The standard deviation.

sum: *Number*

The sum of the samples.

sum2: *Number*

The sum of the squares of the samples.

Topnset

The Topnset class represents a collection of metrics grouped by a key such as a URI or a client IP address.

For custom metrics, keys in the topnset corresponds to the keys passed into `metricAddDetail*` methods. Key values can be a number, string, Dataset, Sampleset, or another topnset.

Methods

findEntries(key: *IPAddress* | *String* | *Object*): *Array*

Returns all entries with matching keys.

findKeys(key: *IPAddress* | *String* | *Object*): *Array*

Returns all matching keys.

lookup(key: *IPAddress* | *String* | *Object*): *

Look up an item in the topnset and retrieves the first matching entry.

Properties

entries: *Array*

An array of the topnset entries. The array contains at most N objects with key and value properties where N is currently set to 1000.

Keys in the `entries` array adhere to the following structure, or key pattern:

type: *String*

The type of the topnset key. The following key types are supported:

- int
- string
- device_id
- ipaddr
- addr_pair
- ether

value: *

The key value, which varies depending on the key type.

- For `int`, `string`, and `device_id` keys, the value is a number, string, and device ID, respectively.
- For `ipaddr` keys, the value is an object containing the following properties:
  - addr
  - proto
  - port
  - device_id
  - origin
- For `addr_pair` keys, the value is an object containing the following properties:
  - addr1
  - addr2
  - port1
  - port2
  - proto
- For `ether` keys, the value is an object containing the following properties:
  - ethertype
  - hwaddr

Deprecated API elements

The API elements listed in this section are deprecated. Each element includes a replacement and the version in which the element was deprecated. Trigger scripts with deprecated elements must be updated with replacement elements.

Deprecated global functions

Function	Replacement	Version
exit(): *Void*	The return statement	4.0
getTimestampMSec(): *Number*	getTimestamp(): *Number*	4.0

Deprecated events

Events	Replacement	Version
NEW_VLAN	No replacement	6.1

Deprecated classes

Class	Replacement	Version
RemoteSyslog	Remote.Syslog	4.0
XML	Regular expressions	6.0
TroubleGroup	No replacement	6.0

Deprecated methods by class

Class	Method	Replacement	Version
Flow	getApplication(): *String*	getApplications(): *String*	5.3
	setApplication(name: *String*, [turnTiming: *Boolean*]): *void*	addApplication(name: *String*, [turnTiming: *Boolean*]): *void*	5.3
Session	update(key: *String*, value: *, [options: *Object*])*	replace(key: *String*, value: *, [options: *Object*]): *	3.9
SSL	setApplication(name: *String*): *void*	addApplication(name: *String*): *void*	5.3

Deprecated properties by class

Class	Property	Replacement	Version
AAA	error: *String*	isError: *Boolean*	5.0
	tprocess: *Number*	processingTime: *Number*	5.2
DB	tprocess: *Number*	processingTime: *Number*	5.2
Discover	vlan: *VLAN*	No replacement	6.1
DNS	tprocess: *Number*	processingTime: *Number*	5.2
Flow	isClientAborted: *Boolean*	isAborted: *Boolean*	3.10

Class	Property	Replacement	Version
	isServerAborted: *Boolean*	isAborted: *Boolean*	3.10
	turnInfo: *String*	Top-level Turn object with attributes for the turn	3.9
FTP	tprocess: *Number*	processingTime: *Number*	5.2
HL7	tprocess: *Number*	processingTime: *Number*	5.2
HTTP	payloadText: *String*	payload: *Buffer*	4.0
	tprocess: *Number*	processingTime: *Number*	5.2
IBMMQ	messageID: *String*	msgID: *Buffer*	5.2
	msgSize: *Number*	totalMsgLength: *Number*	5.2
	objectHandle: *String*	No replacement	5.0
	payload: *Buffer*	msg: *Buffer*	5.2
ICA	authTicket: *String*	user: *String*	3.7
	application: *String*	program: *String*	5.2
	client: *String*	clientMachine: *String*	6.0
LDAP	tprocess: *Number*	processingTime: *Number*	5.2
MongoDB	tprocess: *Number*	processingTime: *Number*	5.2
NetFlow	tos: *Number*	dscp: *Number* dscp: *String*	6.1
SMPP	tprocess: *Number*	processingTime: *Number*	5.2
SMTP	tprocess: *Number*	processingTime: *Number*	5.2
SSL	reqBytes: *Number*	clientBytes	6.1
	reqL2Bytes: *Number*	clientL2Bytes: *Number*	6.1
	reqPkts: *Number*	clientPkts: *Number*	6.1
	rspBytes: *Number*	serverBytes: *Number*	6.1
	rspL2Bytes: *Number*	serverL2Bytes: *Number*	6.1
	rspPkts: *Number*	serverPkts: *Number*	6.1
Turn	reqSize: *Number*	clientBytes: *Number*	4.0
	reqXfer: *Number*	clientTransferTime: *Number*	4.0
	respSize: *Number*	serverBytes: *Number*	4.0
	rspXfer: *Number*	serverTransferTime: *Number*	4.0
	tprocess: *Number*	processingTime: *Number*	4.0

Advanced trigger options

You can configure advanced options for some events when you create a trigger.

The following table describes available advanced options and applicable events.

Option	Description	Applicable events
Bytes per packet to capture	Specifies the number of bytes to capture per packet. The capture starts with the first byte in the packet. Specify this option only if the trigger script performs packet capture. A value of 0 specifies that the capture should collect all bytes in each packet.	All events except: • ALERT_RECORD_COMMIT • METRIC_CYCLE_BEGIN • METRIC_CYCLE_END • FLOW_REPORT • NEW_APPLICATION • NEW_DEVICE • SESSION_EXPIRE
Bytes to Buffer	Specifies the number of captured payload bytes to buffer.	• CIFS_REQUEST • CIFS_RESPONSE • HTTP_REQUEST • HTTP_RESPONSE • ICA_TICK
Clipboard Bytes to Buffer	Specifies the number of bytes to buffer on a Citrix clipboard transfer.	• ICA_TICK
Metric Cycle	Specifies the length of the metric cycle, expressed in seconds. The following values are valid: • 30sec • 5min • 1hr • 24hr	• METRIC_CYCLE_BEGIN • METRIC_CYCLE_END • METRIC_RECORD_COMMIT
Metric Types	Specifies the metric type by the raw metric name, such as extrahop.device.http_server. Specify multiple metric types in a comma-delimited list.	• ALERT_RECORD_COMMIT • METRIC_RECORD_COMMIT
Per Turn	Enables packet capture on each flow turn. Per-turn analysis continuously analyzes communication between two endpoints to extract a single payload data point from the flow. If this option is enabled, any values specified for the **Client matching string** and **Server matching string** options are ignored.	• SSL_PAYLOAD • TCP_PAYLOAD

Option	Description	Applicable events
Client port min	Specifies the minimum port number of the client port range. Valid values are 0 to 65535. A value of 0 specifies matching of any port.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Client port max	Specifies the maximum port number of the client port range. Valid values are 0 to 65535. Any value specified for this option is ignored if the value of the **Client port min** option is 0.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Client bytes to buffer	Specifies the number of client bytes to buffer. The value of this option cannot be set to 0 if the value of the **Server bytes to buffer** option is also set to 0.	• SSL_PAYLOAD • TCP_PAYLOAD
Client matching string	Specifies the format string that indicates when to begin buffering client data. Any value specified for this option is ignored if the **Per Turn** option is enabled.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Server port min	Specifies the minimum port number of the server port range. Valid values are 0 to 65535. A value of 0 specifies matching of any port.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Server port max	Specifies the maximum port number of the server port range. Valid values are 0 to 65535. Any value specified for this option is ignored if the value of the **Server port min** option is 0.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Server bytes buffer	Specifies the number of server bytes to buffer. The value of this option cannot be set to 0 if the value of the **Client bytes to buffer** option is also set to 0.	• SSL_PAYLOAD • TCP_PAYLOAD
Server matching string	Specifies the format string that indicates when to begin buffering	• SSL_PAYLOAD • TCP_PAYLOAD

Option	Description	Applicable events
	data. Returns the entire packet upon a string match. Any value specified for this option is ignored if the **Per Turn** option is enabled.	• UDP_PAYLOAD
All UDP Datagrams	Enables capture of all UDP datagrams.	• UDP_PAYLOAD
Run FLOW_CLASSIFY on expired flows	Enables running the event upon expiration to accumulate metrics for flows that were not classified before expiring.	• FLOW_CLASSIFY

Performance optimization tips

Triggers consume system resources and can affect system performance. A poorly-written trigger can cause unnecessary system load.

The Performance tab in the Trigger Configuration window provides a graphical representation of the performance cost of a trigger by tracking the number of cycles used by the trigger in a given time interval. Based on the performance impact, you might want to re-evaluate and optimize your trigger.

ExtraHop recommends the following practices to improve trigger performance:

- Re-evaluate what you want the trigger to accomplish. The trigger might be performing more actions than you really need.
- Assign triggers only to devices that require the collection of metrics. Assigning triggers to all devices causes unnecessary trigger executions that might degrade system performance.
- Monitor how frequently your triggers are run by checking the Performance tab.
- Remove or comment out all debug() statements after you test your trigger and verify that the expected debug output is being logged. A large number of debug statements can cause both performance issues and excessive debug statements in the runtime log.
- Avoid unnecessarily complex code, such as loops with potentially high number of iterations, nested while and for loops, and inefficient regular expressions.
- Use exclusion logic so your trigger does the minimum amount of processing and object creation. For example, you can optimize the following trigger code:

  Original code:

  ```
  var example1 = HTTP.cookies;
  var example2 = HTTP.method;
  if (example2 === 'POST') {
         // process cookies
  }
  ```

  Optimized code:

  ```
  var example1;
  var example2 = HTTP.method;
  if (example2 === 'POST') {
  example1 = HTTP.cookies;
     // process cookies
  }
  ```

- Use `var` to declare variables, unless you only need the value once. For example, use `var example1 = HTTP.uri;` instead of `uri = HTTP.uri;`.
- Declare variables as you need them to minimize performance impact. For example, if you write a conditional statement that calls variables in both statements, declaring the variables up front causes the system to process all of the variables even if some will not get used.
- Cache array lengths before iterating. Optimize code to evaluate the length of the list once, as opposed to evaluating the list length once per loop iteration.

  Original code:

  ```
  for (i = 0; i < list.length; i++) { do something; }
  ```

  Optimized code:

  ```
  var len = list.lengthfor (i = 0; i < len; i++) { do something; }
  ```

- Use strict equality (triple equals) whenever possible.

```
if (some_thing !== null) { do something; }
```

- Do not put large objects into `Flow.store`. For example, instead of storing all of `DNS.answers`, keep only the information that will be used for future events.

```
Flow.store.ttl = DNS.answers[i].ttl;
```

- Clear `Flow.store` values when you no longer need them by setting the property to `null`. This also helps to prevent errors in which the trigger reads previously recorded data from `Flow.store`.

```
if (event === 'DB_REQUEST') {
   Flow.store.stmt = DB.statement;
} else if (event === 'DB_RESPONSE') {
   Device.metricAddCount('count', 1);
   Device.metricAddDetailCount('count', Flow.store.stmt, 1);
   Flow.store.stmt = null;
}
```

Note: There are separate flow stores for every flow, so how much you can store on each flow store depends on the number and the size of the values being stored in each flow store.

- Conserve CPU cycles by caching created applications that are used multiple times.

```
var myapp = Application(‘example’);
// now commit like
myapp.commit();
myapp.metricAddCount(‘custom’, 1);
```

- Avoid property lookups whenever possible, especially when using a property lookup more than once.

```
if (HTTP.uri.indexOf('example1') > -1) // content
else if (HTTP.uri.indexOf('example2') > -1) // content
```

- Cache information once and save the property lookups.

```
var uri = HTTP.uri;
if (uri.indexOf('example1') > -1) // content
else if (uri.indexOf('example2') > -1) // content
```

- Use `return` instead of `exit()`. The `exit()` global function is supported but it might cause the system to run slowly. For example, use `(!HTTP.uri) return;` instead of `HTTP.uri || exit();`.
- Choose `match` instead of `indexOf` when searching with regular expressions.

Examples

The following examples are available:

- Example: Collect ActiveMQ metrics
- Example: Send data to Azure with Remote.HTTP
- Example: Monitor CIFS actions on devices
- Example: Track 500-level HTTP responses by customer ID and URI
- Example: Collect response metrics on database queries
- Example: Send discovered device data to a remote syslog server
- Example: Send data to Elasticsearch with Remote.HTTP
- Example: Access HTTP header attributes
- Example: Collect IBMMQ metrics
- Example: Record Memcache hits and misses
- Example: Parse memcache keys
- Example: Add metrics to the metric cycle store
- Example: Parse NTP with universal payload analysis
- Example: Parse custom PoS messages with universal payload analysis
- Example: Parse syslog over TCP with universal payload analysis
- Example: Record data to a session table
- Example: Track SOAP requests
- Example: Matching topnset keys
- Example: Create an application container

Example: Collect ActiveMQ metrics

The trigger in this example records destination information from the Java Messaging Service (JMS). The trigger creates an application and collects custom metrics that include the whether the broker of an event is the sender or receiver and the JMS destination field specified on that event.

Run the trigger on the following events: `ACTIVEMQ_MESSAGE`

```
var app = Application("ActiveMQ Sample");
    if (ActiveMQ.senderIsBroker) {
       if (ActiveMQ.receiverIsBroker) {
          app.metricAddCount("amq_broker", 1);
          app.metricAddDetailCount("amq_broker", ActiveMQ.queue, 1);
       }
       else {
          app.metricAddCount("amq_msg_out", 1);
          app.metricAddDetailCount("amq_msg_out", ActiveMQ.queue, 1);
       }
}
else {
    app.metricAddCount("amq_msg_in", 1);
    app.metricAddDetailCount("amq_msg_in", ActiveMQ.queue, 1);
}
```

Related classes

- ActiveMQ
- Application

Example: Send data to Azure with Remote.HTTP

The trigger in this example sends data to the Microsoft Azure Table storage service through an HTTP open data stream (ODS).

You must first configure an HTTP open data stream from the ExtraHop Admin UI before you create the trigger. The ODS configuration contains the authentication information required to sign in to your Microsoft Azure service. For configuration information, see Configure Open Data Stream for HTTP in the ExtraHop Admin UI Guide.

Run the trigger on the following events: HTTP_RESPONSE

```
// The name of the HTTP destination defined in the ODS config
var REST_DEST = "my_table_storage";

// The name of the table within Azure Table storage
var TABLE_NAME = "TestTable";

/* If the header is not set to this value, Azure expects to receive XML;
 * however, it is easier for a trigger to send JSON.
 * The ODS config enables you to specify the datatype of fields; in this
 case
 * the timestamp (TS) field is a datetime even though it is serialized from
 a
 * Date to a String.
 */

var headers = { "Content-Type": "application/json;odata=minimalmetadata" };

var now = new Date(getTimestamp());
var msg = {
    "RowKey":        now.getTime().toString(), // must be a string
    "PartitionKey":  "my_key", // must be a string
    "HTTPMethod":    HTTP.method,
    "DestAddr":      Flow.server.ipaddr,
    "SrcAddr":       Flow.client.ipaddr,
    "SrcPort":       Flow.client.port,
    "DestPort":      Flow.server.port,
    "TS@odata.type": "Edm.DateTime", // metadata to describe format of TS
 field
    "TS":            now.toISOString(),
    "ServerTime":    HTTP.processingTime,
    "RspTTLB":       HTTP.rspTimeToLastByte,
    "RspCode":       HTTP.statusCode.toString(),
    "URI":           "http://" + HTTP.host + HTTP.path,
};

// debug(JSON.stringify(msg));
Remote.HTTP(REST_DEST).post( { path: "/" + TABLE_NAME, headers: headers,
 payload:
JSON.stringify(msg) } );
```

Related classes

- Remote.HTTP
- Flow
- HTTP

Example: Monitor CIFS actions on devices

The trigger in this example monitors the CIFS actions performed on devices, and then creates custom device metrics that collect the total number of bytes read and written, and the number of bytes written by CIFS users that are not authorized to access a sensitive resource.

Run the trigger on the following events: `CIFS_RESPONSE`

```
var client = Flow.client.device,
   server = Flow.server.device,
   clientAddress = Flow.client.ipaddr,
   serverAddress = Flow.server.ipaddr,
   file = CIFS.resource,
   user = CIFS.user,
   resource,
   permissions,
   writeBytes,
   readBytes;

// Resource to monitor
resource = "\\Clients\\Confidential\\";
// Users of interest and their permissions
permissions = {
   "\\\\EXTRAHOP\\tom" : {read: false, write: false},
   "\\\\Anonymous" : {read: true, write: false},
   "\\\\WORKGROUP\\maria" : {read: true, write: true}
};

// Check if this is an action on your monitored resource
if ((file !== null) && (file.indexOf(resource) !== -1)) {
   if (CIFS.isCommandWrite) {
      writeBytes = CIFS.reqSize;
      // Record bytes written
      Device.metricAddCount("cifs_write_bytes", writeBytes);
      Device.metricAddDetailCount("cifs_write_bytes", user, writeBytes);
      // Record number of writes
      Device.metricAddCount("cifs_writes", 1);
      Device.metricAddDetailCount("cifs_writes", user, 1);
      // Record number of unauthorized writes
      if (!permissions[user] || !permissions[user].write) {
         Device.metricAddCount("cifs_unauth_writes", 1);
         Device.metricAddDetailCount("cifs_unauth_writes", user, 1);
      }
   }

   if (CIFS.isCommandRead) {
      readBytes = CIFS.reqSize;
      // Record bytes read
      Device.metricAddCount("cifs_read_bytes", readBytes);
      Device.metricAddDetailCount("cifs_read_bytes", user, readBytes);
      // Record number of reads
      Device.metricAddCount("cifs_reads", 1);
      Device.metricAddDetailCount("cifs_reads", user, 1);
      // Record number of unauthorized reads
      if (!permissions[user] || !permissions[user].read) {
         Device.metricAddCount("cifs_unauth_reads", 1);
         Device.metricAddDetailCount("cifs_unauth_reads", user, 1);
      }
   }
}
```

Related classes

- CIFS
- Device
- Flow

Example: Track 500-level HTTP responses by customer ID and URI

The trigger in this example tracks HTTP server responses that result in an error code of 500. The trigger also creates custom device metrics that collect the customer ID and URI in the header of each 500 response.

Run the trigger on the following events: `HTTP_REQUEST` and `HTTP_RESPONSE`

```
var custId,
   query,
   uri,
   key;

if (event === "HTTP_REQUEST") {
   custId = HTTP.headers["Cust-ID"];
   // Only keep the URI if there is a customer id
   if (custId !== null) {
      Flow.store.custId = custId;

      query = HTTP.query;

      /* Pull the complete URI (URI plus query string) and save it to
       * the Flow store for a subsequent response event.
       *
       * The query string data is only available on the request.
       */
      uri = HTTP.uri;
      if ((uri !== null) && (query !== null)) {
         uri = uri + "?" + query;
      }

      // Keep URIs for handling by HTTP_RESPONSE triggers
      Flow.store.uri = uri;
   }
}
else if (event === "HTTP_RESPONSE") {
   custId = Flow.store.custId;

   // Count total requests by customer ID
   Device.metricAddCount("custid_rsp_count", 1);
   Device.metricAddDetailCount("custid_rsp_count_detail", custId, 1);

   // If the status code is 500 or 503, record the URI and customer ID
   if ((HTTP.statusCode === 500) || (HTTP.statusCode === 503))
{

      // Combine URI and customer ID to create the detail key
      key = custId;
      if (Flow.store.uri != null) {
      key += ", " + Flow.store.uri;
      }
      Device.metricAddCount("custid_error_count", 1);
      Device.metricAddDetailCount("custid_error_count_detail", key, 1);
   }
}
```

Related classes

- HTTP
- Flow
- Device

Example: Collect response metrics on database queries

The trigger in this example creates custom device metrics that collect the number of responses and the processing times on database queries.

Run the trigger on the following events: `DB_RESPONSE`

```
let stmt = DB.statement;
if (stmt === null) {
    return;
}

// Remove leading whitespace and truncate
stmt = stmt.trimLeft().substr(0, 1023);

// Record counts by statement
Device.metricAddCount("db_rsp_count", 1);
Device.metricAddDetailCount("db_rsp_count_detail", stmt, 1);

// Record processing times by statement
Device.metricAddSampleset("db_proc_time", DB.processingTime);
Device.metricAddDetailSampleset("db_proc_time_detail",
                                stmt, DB.processingTime);
```

Related classes

- DB
- Device

Example: Send discovered device data to a remote syslog server

The trigger in this example discovers when a new device is detected on the ExtraHop system and creates remote syslog messages that contain device attributes.

You must first configure a remote open data stream from the ExtraHop Admin UI before you create the trigger. The ODS configuration specifies the location of the remote syslog server. For configuration information, see Configure Open Data Stream for Syslog in the ExtraHop Admin UI Guide.

Run the trigger on the following events: `NEW_DEVICE`

```
var dev = Discover.device;
Remote.Syslog.info('Discovered device ' + Device.id + ' (hwaddr: ' +
 Device.hwaddr + ')
');
```

Related classes

- Remote.Syslog
- Discover
- Device

Example: Send data to Elasticsearch with Remote.HTTP

The trigger in this example sends data to an Elasticsearch server through an HTTP open data stream (ODS).

You must first configure an HTTP open data stream from the ExtraHop Admin UI before you create the trigger. The ODS configuration specifies the Elasticsearch target and any required authentication credentials. For configuration information, see Configure Open Data Stream for HTTP in the ExtraHop Admin UI Guide.

Run the trigger on the following events: `HTTP_REQUEST` and `HTTP_RESPONSE`

```
var date = new Date();
var payload = {
    'ts' : date.toISOString(), // Timestamp recognized by Elasticsearch
    'eh_event' : 'http',
    'my_path' : HTTP.path};
var obj = {
    'path' : '/extrahop/http', // Add to Extrahop index
    'headers' : {},
    'payload' : JSON.stringify(payload)} ;
Remote.HTTP('elasticsearch').request('POST', obj);
```

Related classes

- Remote.HTTP

Example: Access HTTP header attributes

The trigger in this example accesses HTTP event attributes from the header object, and creates custom device metrics that count header requests and attributes.

Run the trigger on the following events: `HTTP_RESPONSE`

```
var hdr,
   session,
   accept,
   results,
   headers = HTTP.headers,
   i;

// Header lookups are case-insensitive properties
session = headers["X-Session-Id"];

/* Session is a string representing the value of the header (or null
 * if the header is not present). Header values are always strings.
 */

// This syntax also works if the header is a legal property name
accept = headers.accept;

/*
 * In the event that there are multiple instances of a header,
 * accessing the header in the above manner (as a property)
 * will always return the value for the first appearance of the
 * header.
 */

if (session !== null)
{
```

```
   // Count requests per session ID
   Device.metricAddCount("req_count", 1);
   Device.metricAddDetailCount("req_count", session, 1);
}

/* Looping over all headers
 *
 * The "length" property is case-sensitive and is not
 * treated as a header lookup. Instead, it returns the number of
 * headers (as if HTTP.headers were an array). In the unlikely
 * event that there is a header called "Length," it would still be
 * accessible with HTTP.headers["Length"] (or HTTP.headers.Length).
 */

for (i = 0; i < headers.length; i++) {
   hdr = headers[i];
   debug("headers[" + i + "].name: " + hdr.name);
   debug("headers[" + i + "].value: " + hdr.value);
   Device.metricAddCount("hdr_count", 1);
   /* Count instances of each header */
   Device.metricAddDetailCount("hdr_count", hdr.name, 1);
}

// Searching for headers by prefix
results = HTTP.findHeaders("Content-");

/* The "results" property is an array (a real javascript array, as opposed
 * to an array-like object) of header objects (with name and value
 * properties) where the names match the prefix of the string passed
 * to findHeaders.
 */
for (i = 0; i < results.length; i++) {
   hdr = results[i];
   debug("results[" + i + "].name: " + hdr.name);
   debug("results[" + i + "].value: " + hdr.value);
}
```

Related classes

- HTTP
- Device

Example: Collect IBMMQ metrics

The triggers in this example work together to give a view of the flow of queue level messages through the IBMMQ protocol. The triggers create custom application metrics that count the number of messages in, out, and exchanged between brokers by different message queues.

Run the following trigger on the `IBMMQ_REQUEST` event.

```
if (IBMMQ.method == "MESSAGE_DATA") {
    var app = Application("IBMMQ Sample");
    app.metricAddCount("broker", 1);
    if (IBMMQ.queue !== null) {
        var ret = IBMMQ.queue.split(":");
        var queue = ret.length > 1 ? ret[1] : ret[0];
        app.metricAddDetailCount("broker", queue, 1);
    }
    else {
        app.metricAddCount("queueless_broker", 1);
    }
```

```
    if (IBMMQ.queue !== null && IBMMQ.queue.indexOf("QUEUE2") > -1) {
        app.metricAddCount("queue2_broker", 1);
    }
    app.commit();
}
elseif (IBMMQ.method == "MQPUT" || IBMMQ.method == "MQPUT1") {
    var app = Application("IBMMQ Sample");
    app.metricAddCount("msg_in", 1);
    if (IBMMQ.queue !== null) {
        var ret = IBMMQ.queue.split(":");
        var queue = ret.length > 1 ? ret[1] : ret[0];
        app.metricAddDetailCount("msg_in", queue, 1);
    }
    else {
        app.metricAddCount("queueless_msg_in", 1);
    }
    if (IBMMQ.queue !== null && IBMMQ.queue.indexOf("QUEUE2") > -1) {
        app.metricAddCount("queue2_msg_in", 1);
    }
    app.commit();
}
```

Run the following trigger on the IBMMQ_RESPONSE event.

```
if (IBMMQ.method == "ASYNC_MSG_V7" || IBMMQ.method == "MQGET_REPLY") {
    var app = Application("IBMMQ Sample");
    if (IBMMQ.payload === null) {
        app.metricAddCount("payloadless_msg_out", 1);
    }
    else {
        app.metricAddCount("msg_out", 1);
        if (IBMMQ.queue !== null) {
            var ret = IBMMQ.queue.split(":");
            var queue = ret.length > 1 ? ret[1] : ret[0];
            app.metricAddDetailCount("msg_out", queue, 1);
        }
        else {
            app.metricAddCount("queueless_msg_out", 1);
        }
        if (IBMMQ.queue !== null && IBMMQ.queue.indexOf("QUEUE2") > -1) {
            app.metricAddCount("queue2_msg_out", 1);
        }
    }
    app.commit();
}
```

Related classes

- IBMMQ
- Application

Example: Record Memcache hits and misses

The trigger in this example creates custom device metrics that record each memcache hit or miss and the access time of each hit.

Run the trigger on the following events: MEMCACHE_RESPONSE

```
var hits = Memcache.hits;
var misses = Memcache.misses;
var accessTime = Memcache.accessTime;
```

```
var i;

Device.metricAddCount('memcache_key_hit', hits.length);

for (i = 0; i < hits.length; i++) {
   var hit = hits[i];
   if (hit.key != null) {
      Device.metricAddDetailCount('memcache_key_hit_detail', hit.key, 1);
   }
}

if (!isNaN(accessTime)) {
   Device.metricAddSampleset('memcache_key_hit', accessTime);
   if ((hits.length > 0) && (hits[0].key != null)) {
      Device.metricAddDetailSampleset('memcache_key_hit_detail',
 hits[0].key,
                                      accessTime);
   }
}

Device.metricAddCount('memcache_key_miss', misses.length);

for (i = 0; i < misses.length; i++) {
   var miss = misses[i];
   if (miss.key != null) {
      Device.metricAddDetailCount('memcache_key_miss_detail', miss.key, 1);
   }
}
```

Related classes

- Memcache
- Device

Example: Parse memcache keys

Parses the memcache keys to extract detailed breakdowns, such as by ID module and class name, and creates custom device metrics to collect key details.

Keys are formatted as "com.extrahop.<module>.<class>_<id>"—for example: "com.extrahop.widgets.sprocket_12345".

Run the trigger on the following events: MEMCACHE_RESPONSE

```
var method = Memcache.method;
var statusCode = Memcache.statusCode;
var reqKeys = Memcache.reqKeys;
var hits = Memcache.hits;
var misses = Memcache.misses;
var error = Memcache.error;
var hit;
var miss;
var key;
var size;
var reqKey;
var i;

// Record breakdown of hit count and value size by module and class
for (i = 0; i < hits.length; i++) {
   hit = hits[i];
   key = hit.key;
```

```
   size = hit.size;

   Device.metricAddCount("hit", 1);
   if (key != null) {
      var parts = key.split(".");

      if ((parts.length == 4) && (parts[0] == "com") &&
         (parts[1] == "extrahop")) {
         var module = parts[2];
         var subparts = parts[3].split("_");

         Device.metricAddDetailCount("hit_module", module, 1);
         Device.metricAddDetailSampleset("hit_module_size", module, size);

         if (subparts.length == 2) {
            var hitClass = module + "." + subparts[0];

            Device.metricAddDetailCount("hit_class", hitClass, 1);
            Device.metricAddDetailSampleset("hit_class_size", hitClass,
                                            size);
         }
      }
   }
}

// Record misses by ID to help identify caching issues
for (i = 0; i < misses.length; i++) {
   miss = misses[i];
   key = miss.key;
   if (key != null) {
      var parts = key.split(".");

      if ((parts.length == 4) && (parts[0] == "com") &&
         (parts[1] == "extrahop") && (parts[2] == "widgets")) {
         var subparts = parts[3].split("_");

         if ((subparts.length == 2) && (subparts[0] == "sprocket")) {
            Device.metricAddDetailCount("sprocket_miss_id", subparts[1], 1);
         }
      }
   }
}

// Record the keys that produced any errors
if (error != null && method != null) {
   for (i = 0; i < reqKeys.length; i++) {
      reqKey = reqKeys[i];
      if (reqKey != null) {
         var errDetail = method + " " + reqKey + " / " + statusCode + ": " +
                         error;
         Device.metricAddDetailCount("error_key", errDetail, 1);
      }
   }
}

// Record the status code, matching built-in metrics
if (Memcache.isBinaryProtocol && statusCode != "NO_ERROR") {
   Device.metricAddDetailCount("status_code",
                               method + "/" + statusCode, 1);
}
else {
   Device.metricAddDetailCount("status_code", statusCode, 1);
}
```

Related classes

- Memcache
- Device

Example: Add metrics to the metric cycle store

The trigger in this example illustrates how to temporarily store data from all metric record commits that occur during a metric cycle.

Run the trigger on the following events: `METRIC_CYCLE_BEGIN`, `METRIC_CYCLE_END`, `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.http_server, extrahop.device.tcp

```
var store = MetricCycle.store;

function processMetric() {
    var id = MetricRecord.id,
        deviceId = MetricRecord.object.id,
        fields = MetricRecord.fields;

    if (!store.metrics[deviceId]) {
        store.metrics[deviceId] = {};
    }
    if (id === 'extrahop.device.http_server') {
        store.metrics[deviceId].httpRspAborted= fields['rsp_abort'];
    }
    else if (id === 'extrahop.device.tcp') {
        store.metrics[deviceId].tcpAborted = fields['aborted_out'];
    }
}

function commitSyntheticMetrics() {
    var dev,
        metrics,
        abortPct,
        deviceId;
    for (deviceId in store.metrics) {
        metrics = store.metrics[deviceId];
        abortPct = (metrics.httpRspAborted / metrics.tcpAborted) * 100;
        dev = new Device(deviceId);
        dev.metricAddSnap('http-tcp-abort-pct', abortPct);
    }
}

switch (event) {
case 'METRIC_CYCLE_BEGIN':
    store.metrics = {};
    break;

case 'METRIC_RECORD_COMMIT':
    processMetric();
    break;
```

```
case 'METRIC_CYCLE_END':
    commitSyntheticMetrics();
    break;
}
```

Related classes

- MetricCycle
- MetricRecord
- Device

Example: Parse custom PoS messages with universal payload analysis

The trigger in this example parses TCP messages from a point-of-sale (PoS) system and creates custom device metrics that collect specific values in the 4th to 7th bytes of both response and request messages.

Run the trigger on the following events: TCP_PAYLOAD

```
// Define variables; store client or server payload into a Buffer object

var buf_client = Flow.client.payload,
    buf_server = Flow.server.payload,
    protocol = Flow.l7proto,

// PoS Message Type Structure Definition
    pos_message_type = {
        "0100" : "0100_Authorization_Request",
        "0101" : "0101_Authorization_Request_Repeat",
        "0110" : "0110_Authorization_Response",
        "0200" : "0200_Financial_Request",
        "0201" : "0201_Financial_Request_Repeat",
        "0210" : "0210_Financial_Response",
        "0220" : "0220_Financial_Transaction_Advice_Request",
        "0221" : "0221_Financial_Transaction_Advice_Request_Repeat",
        "0230" : "0230_Financial_Transaction_Advice_Response",
        "0420" : "0420_Reversal_Advice_Request",
        "0421" : "0421_Reversal_Advice_Request_Repeat",
        "0430" : "0430_Reversal_Advice_Response",
        "0600" : "0600_Administration_Request",
        "0601" : "0601_Administration_Request_Repeat",
        "0610" : "0610_Administration_Response",
        "0620" : "0620_Administration_Advice_Request",
        "0621" : "0621_Administration_Advice_Request_Repeat",
        "0630" : "0630_Administration_Advice_Response",
        "0800" : "0800_Administration_Request",
        "0801" : "0801_Administration_Request_Repeat",
        "0810" : "0810_Administration_Response"
    };

// Skip parsing if it is a protocol of no interest or there is no payload
if (protocol !== 'tcp:4015' || (buf_client === null && buf_server === null))
 {
    // debug('Protocol of no interest: ' + protocol);
    return;
} else {
    /* Store the data into variables for future access since there is some
 payload
     * to parse
     */
    var client_ip = Flow.client.ipaddr,
```

```
        server_ip = Flow.server.ipaddr,
        client_port = Flow.client.port,
        server_port = Flow.server.port;
        // client = new Device(Flow.client.device.id),
        // server = new Device(Flow.server.device.id);
}

if (buf_client !== null && buf_client.length >= 7) {

    // This is a client payload
    var cli_msg_type = buf_client.slice(3,7).decode('utf-8');
    debug('Client: ' + client_ip + ":" + client_port + " Type: " +
 pos_message_type[cli_msg_type]);
    Device.metricAddCount('UPA_Request', 1);
    Device.metricAddDetailCount('UPA_Request_by_Message',
 pos_message_type[cli_msg_type], 1);
    Device.metricAddDetailCount('UPA_Request_by_Client',
 client_ip.toString(), 1);

} else if (buf_server !== null && buf_server.length >= 7) {

    // This is a server payload
    var srv_msg_type = buf_server.slice(3,7).decode('utf-8');
    debug('Server: ' + server_ip + " Client: " + client_ip + ":" +
 client_port +"
Type: " + pos_message_type[srv_msg_type]);
    Device.metricAddCount('UPA_Response', 1);
    Device.metricAddDetailCount('UPA_Response_by_Message',
 pos_message_type[srv_msg_type], 1);
    Device.metricAddDetailCount('UPA_Response_by_Client',
 client_ip.toString(), 1);

} else {

    // No buffer captured situation
    //debug('Null or not enough buffer data');
    return;
}
```

Related classes

- Buffer
- Device
- Flow

Example: Parse syslog over TCP with universal payload analysis

The trigger in this example parses the syslog over TCP and counts the syslog activity over time, both network-wide and per device.

Note: You might need to edit the trigger example to make sure the network ports for your syslog server match the ports in your environment.

This trigger example is available for download through a solutions bundle from the ExtraHop community.

Run the trigger on the following events: `TCP_PAYLOAD`, `UDP_PAYLOAD`

```
// Global variables
var buffer         = Flow.client.payload,
    buffer_size    = Flow.client.payload.length + 1,
```

```
    client         = new Device(Flow.client.device.id),
    data_as_json   = { client_ip       : Flow.client.ipaddr.toString(),
                       client_port     : Flow.client.port.toString(),
                       server_ip       : Flow.server.ipaddr.toString(),
                       server_port     : Flow.server.port.toString(),
                       protocol        : 'syslog',
                       protocol_fields : {} },
    protocol        = Flow.l7proto,
    server          = new Device(Flow.server.device.id),
    syslog          = {},
    syslog_facility = {
        "0": "kern",
        "1": "user",
        "2": "mail",
        "3": "daemon",
        "4": "auth",
        "5": "syslog",
        "6": "lpr",
        "7": "news",
        "8": "uucp",
        "9": "clock_daemon",
        "10": "authpriv",
        "11": "ftp",
        "12": "ntp",
        "13": "log_audit",
        "14": "log_alert",
        "15": "cron",
        "16": "local0",
        "17": "local1",
        "18": "local2",
        "19": "local3",
        "20": "local4",
        "21": "local5",
        "22": "local6",
        "23": "local7",
    },
    syslog_priority = {
         "0": "emerg",
         "1": "alert",
         "2": "crit",
         "3": "err",
         "4": "warn",
         "5": "notice",
         "6": "info",
         "7": "debug",
    };

// Exit out early if not classified properly or no payload

if ( ( protocol != 'tcp:5141' ) || ( buffer === null ) ) {
    debug('Invalid protocol ' + protocol +
          ' or null buffer (' + buffer.unpack('z').join(' ') + ')');
    return;
}

// Get started parsing Syslog

var data = buffer.unpack('z');

// Separate the PRIO field from the rest of the message
var msg_part  = data[0].split('>')[1].split(' ');
var prio_part = data[0].split('>')[0].split('<')[1];

// Decode the PRIO field into Syslog facility and priority
```

```
var raw_facility = parseInt(prio_part) >> 3;
var raw_priority = parseInt(prio_part) & 7;

syslog.facility = syslog_facility[raw_facility];
syslog.priority = syslog_priority[raw_priority];

/* Timestamp and hostname are technically part of the HEADER field, but
 * treat the rest of the message as a <space> delimited
 * string, which it is (the syslog protocol is very basic)
 */
syslog.timestamp = msg_part.slice(0,3).join(' ');
syslog.hostname  = msg_part[3];
syslog.message   = msg_part.slice(4).join(' ');

/* At the network level, keep counts of who is sending messages by
 * both facility and priority
 */
Network.metricAddCount('syslog:priority_' + syslog.priority, 1);
Network.metricAddDetailCount('syslog:priority_' +
                             syslog.priority + '_detail',
                             Flow.client.ipaddr, 1);
Network.metricAddCount('syslog:facility_' + syslog.facility, 1);
Network.metricAddDetailCount('syslog:facility_' +
                             syslog.facility + '_detail',
                             Flow.client.ipaddr, 1);

/* Devices receiving messages keep a count of who sent those messages
 * by facility and priority
 */
server.metricAddCount('syslog:priority_' + syslog.priority, 1);
server.metricAddDetailCount('syslog:priority_' +
                            syslog.priority + '_detail',
                            Flow.client.ipaddr, 1);
server.metricAddCount('syslog:facility_' + syslog.facility, 1);
server.metricAddDetailCount('syslog:facility_' +
                            syslog.facility + '_detail',
                            Flow.client.ipaddr, 1);

/* Devices sending messages keep a count of who they sent those messages
 * to by facility and priority
 */
client.metricAddCount('syslog:priority_' + syslog.priority, 1);
client.metricAddDetailCount('syslog:priority_' +
                            syslog.priority + '_detail',
                            Flow.server.ipaddr, 1);
client.metricAddCount('syslog:facility_' + syslog.facility, 1);
client.metricAddDetailCount('syslog:facility_' +
                            syslog.facility + '_detail',
                            Flow.server.ipaddr, 1);

data_as_json.protocol_fields = syslog;
data_as_json.ts              = new Date();

//try {
//    Remote.MongoDB.insert('payload.syslog', data_as_json);
//}
//catch ( err ) {
//    Remote.Syslog.debug(JSON.stringify(data_as_json));
//}
debug('Syslog data: ' + JSON.stringify(data_as_json, null, 4));
```

Related classes

- Flow
- Network
- Buffer
- Remote.MongoDB
- Remote.Syslog

Example: Parse NTP with universal payload analysis

The trigger in the following example parses the network time protocol through universal payload analysis (UDP).

Run the trigger on the following events: `UDP_PAYLOAD`

```
var buf = Flow.server.payload,
    flags,
    values,
    fmt,
    offset = 0,
    ntpData = {},
    proto = Flow.l7proto;
if ((proto !== 'NTP') || (buf === null)) {
    return;
}
// Parse individual flag values from flags byte
function parseFlags(flags) {
    return {
        'LI': flags >> 6,
        'VN': (flags & 0x3f) >> 3,
        'mode': flags & 0x7
    };
}

// Convert from NTP short format
function ntpShort(n) {
    return n / 65536.0;
}

// Convert integral part of NTP timestamp format to Date
function ntpTimestamp(n) {
    /* NTP dates start at 1900, subtract the difference
     * and convert to milliseconds */
    var ms = (n - 0x83aa7e80) * 1000;
    return new Date(ms);
}

// First part of NTP header
fmt = ('B' + // Flags (LI, VN, mode)
       'B' + // Stratum
       'b' + // Polling interval (signed)
       'b' + // Precision (signed)
       'I' + // Root delay
       'I'); // Root dispersion

values = buf.unpack(fmt);

offset = values.bytes;

flags = parseFlags(values[0]);
if (flags.VN !== 4) {
```

```
    // Expecting NTPv4
    return;
}

ntpData.flags = flags;
ntpData.stratum = values[1];
ntpData.poll = values[2];
ntpData.precision = values[3];
ntpData.rootDelay = ntpShort(values[4]);
ntpData.rootDispersion = ntpShort(values[5]);

// The next field, the reference ID, depends upon the stratum field
switch (ntpData.stratum)
{
case 0:
case 1:
    // Identifier string (4 bytes), and 4 NTP timestamps in two parts
    fmt = '4s8I';
    break;
default:
    // Unsigned int (based on IP), and 4 NTP timestamps in two parts
    fmt = 'I8I';
    break;
}
// Passing in offset allows you to continue parsing where you left off
values = buf.unpack(fmt, offset);
ntpData.referenceId = values[0];

// Only the integral parts of the timestamp are referenced here
ntpData.referenceTimestamp = ntpTimestamp(values[1]);
ntpData.originTimestamp = ntpTimestamp(values[3]);
ntpData.receiveTimestamp = ntpTimestamp(values[5]);
ntpData.transmitTimestamp = ntpTimestamp(values[7]);

debug('NTP data:' + JSON.stringify(ntpData, null, 4));
```

Related classes

- Buffer
- Flow
- UDP

Example: Record data to a session table

The trigger in this example records specific HTTP transactions to the session table and creates custom network metrics that collect session expiration data.

Run the trigger on the following events: `HTTP_REQUEST`, `SESSION_EXPIRE`

```
// HTTP_REQUEST
   if (HTTP.userAgent === null) {
      return;
   }

   // Look for the OS name
   var re = /(Windows|Mac|Linux)/;
   var os = HTTP.userAgent.match(re);
   if (os === null) {
      return;
   }
   // Specify the matched string as the key for session table entry
```

```
   os = os[0];

   var opts =
{
      // Expire added entries after 30 seconds
      expire: 30,
      // Retain entries with normal priority if session table grows too
 large
      priority: Session.PRIORITY_NORMAL,
      // Make expired entries available on SESSION_EXPIRE events
      notify: true
   };
   // Ensure an entry for this key is present; an existing entry will not be
 replaced
   Session.add(os, 0, opts);
   // Increase the count for this entry
   var count = Session.increment(os);
   debug(os + ": " + count);

/* After 30 seconds, the accumulated per-OS counts appear in the
 Session.expiredKeys
 * list, accessible in the SESSION_EXPIRE event:
 */
   //SESSION_EXPIRE
   var keys = Session.expiredKeys;
   for (var i = 0; i < keys.length; i++) {
      debug("count of " + keys[i].name + ": " + keys[i].value);
      if (keys[i].value > 500) {
         Network.metricAddCount("os-high-request-count", 1);
         Network.metricAddDetailCount("os-high-request-count",
                                      keys[i].name, 1);
      }
   }
```

Related classes

- HTTP
- Network
- Session

Example: Track SOAP requests

The trigger in this example tracks SOAP requests through the SOAPAction header, saves them into the flow store, and creates custom network metrics that collect data about the transactions.

Note: Before you begin, confirm your SOAP implementation passes the necessary information through the header.

Run the trigger on the following events: `HTTP_REQUEST`, `HTTP_RESPONSE`

```
var soapAction,
   headers = HTTP.headers,
   method,
   detailMethod,
   parts;

if (event === "HTTP_REQUEST") {
   soapAction = headers["SOAPAction"]
   if (soapAction != null) {
       Flow.store.soapAction = soapAction;
```

```
   }
}
else if (event === "HTTP_RESPONSE") {
   soapAction = Flow.store.soapAction;
   if (soapAction != null) {
      parts = soapAction.split("/");
      if (parts.length > 0) {
         method = soapAction.split("/")[1];
      }
      else {
         method = soapAction;
      }
      detailMethod = method + "_detail";
      Network.metricAddCount(method, 1);
      Network.metricAddDetailCount(detailMethod, Flow.client.ipaddr, 1);
      Network.metricAddSampleset("soap_proc", HTTP.processingTime);
      Network.metricAddDetailSampleset("soap_proc_detail", method,
                                       HTTP.processingTime);
   }
}
```

Related classes

- Flow
- HTTP
- Network

Example: Matching topnset keys

The triggers in this example illustrate topnset key matching by string and IPAddress, and includes advanced key mapping.

Topnset key matching by string

Run the trigger on the following events: `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.app

```
var stat = MetricRecord.fields['bytes_out'],
    id = MetricRecord.object.id,
    proto = 'HTTP2-TLS',
    entry;

entry = stat.lookup(proto);
if (entry !==null) {
    debug('Device ' + id + ' sent ' + entry.value + ' bytes over ' + proto);
}
```

Topnset key matching by IPAddress

Run the trigger on the following events: `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.net_detail

```
var stat = MetricRecord.fields['bytes_out'],
    total = 0,
    entry,
    entries,
    i,
    ip = new IPAddress('192.168.112.1');

entries = stat.findEntries(ip);
for (i = 0; i < entries.length; i++) {
    entry = entries[i];
    total += entry.value;
}
Remote.Syslog.alert('IP ' + ip + ' sent ' + total + ' bytes.');
```

Advanced topnset key matching

Run the trigger on the following events: `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.net_detail

```
var stat = MetricRecord.fields['bytes_out'],
    entry,
    entries,
    key,
    i;

entries = stat.findEntries({addr: /192.168.112.1*/, proto: 17});

debug('matched ' + entries.length + '/' + stat.entries.length + '
 entries')};

for (i = 0; i < entries.length; i++) {
    entry = entries[i];
    key = entry.key;
    Remote.Syslog.alert('unexpected outbound UDP traffic from: ' +
                        JSON.stringify(key));
}
```

Related classes

- MetricRecord
- IPAddress
- Remote.Syslog

Example: Create an application container

The trigger in this example creates an application container based on traffic associated with a two-tier application, and creates custom application metrics collected on HTTP and database events.

Run the trigger on the following events: `HTTP_RESPONSE` and `DB_RESPONSE`

```
/* Initialize the application object against which you will
 * commit specific HTTP and DB transactions. After traffic is
 * committed, an application container called "My App" will appear
 * in the Applications tab in the Web UI.
 */

var myApp = Application("My App");

/* These configurable properties describe features that define
 * your application traffic.
 */

var myAppHTTPHost = "myapp.internal.example.com";
var myAppDatabaseName = "myappdb";
if (event == "HTTP_RESPONSE") {

   /* HTTP transactions can be committed to the application on
    * HTTP_RESPONSE events.
    */

   /* Commit this HTTP transaction only if the HTTP host header for
    * this response is defined and matches your application's HTTP host.
    */

   if (HTTP.host && (HTTP.host == myAppHTTPHost)) {
      myApp.commit();

      /* Capture custom metrics about user agents that experience
       * HTTP 40x or 50x responses.
       */

      if (HTTP.statusCode && (HTTP.statusCode >= 400))
{

         // Increment the overall count of 40x or 50x responses

         myApp.metricAddCount('myapp_40x_50x', 1);

         // Collect additional detail on referer, if any

         if (HTTP.referer) {
            myApp.metricAddDetailCount('myapp_40x_50x_refer_detail',
                                       HTTP.referer, 1);
         }
      }
   }

} else if (event == "DB_RESPONSE") {
   /* Database transactions can be committed to the application on
    * DB_RESPONSE events.
    *
    * Commit this database transaction only if the database name for
    * this response matches the name of our application database.
    */
   if (DB.database && (DB.database == myAppDatabaseName)) {
```

```
        myApp.commit();
    }
}
```

Related classes

- Application
- DB
- HTTP