



ExtraHop 5.1

Trigger API Reference

© 2018 ExtraHop Networks, Inc. All rights reserved.

This manual in whole or in part, may not be reproduced, translated, or reduced to any machine-readable form without prior written approval from ExtraHop Networks, Inc.

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

Published: 2018-04-20

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

Contents

Overview	5
Keyboard shortcuts	5
ExtraHop data types	5
Global functions	6
Classes and events	8
AAA	12
ActiveMQ	16
AlertRecord	18
Application	19
Buffer	20
CIFS	23
Dataset	26
DB	26
Device	29
DHCP	32
Discover	34
DNS	35
FIX	38
FLOW	40
FTP	57
HL7	62
HTTP	63
IBMMQ	68
ICA	71
ICMP	75
IPAddress	81
Kerberos	82
LDAP	84
LLDP	87
Memcache	88
MetricCycle	91
MetricRecord	91
MongoDB	92
MSMQ	95
Network	97
NFS	98
POP3	100
Record	103
Remote.HTTP	103
Remote.Kafka	106
Remote.MongoDB	108
Remote.Raw	110
Remote.Syslog	111
RemoteSyslog	112
RTCP	112
RTP	119
Sampleset	122
SDP	122
Session	123

SIP	125
SMPP	131
SMTP	132
SSL	135
Telnet	141
Topnset	143
Topnset Keys	144
TroubleGroup	144
Turn	145
VLAN	145
XML	146

Examples **147**

Example: ActiveMQ	148
Example: Send information to Azure table service with Remote.HTTP	148
Example: CIFS trigger	149
Example: Custom trouble group	150
Example: Customer ID header	150
Example: Database trigger	152
Example: Device discovery notification	153
Example: Send data to Elasticsearch with Remote.HTTP	153
Example: HTTP header object	153
Example: Memcache hits and misses	155
Example: Memcache key parsing	155
Example: Use the Metric Cycle store	157
Example: Parse custom POS messages with universal payload analysis	158
Example: Parse syslog over TCP with universal payload analysis	160
Example: Session table	162
Example: SOAP request	163
Example: Topnset key matching	164
Example: Trigger-based application definition	164

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 summary dashboards.
- Generate and sends records to an Explore appliance for long-term storage and retrieval.
- Associate an arbitrary cross-section of monitored traffic with an ExtraHop application container to enable tier-by-tier application-based views as data. Application views augment the device-based views that the ExtraHop system constructs by default.
- 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 criteria. 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.

Keyboard shortcuts

The following keyboard shortcuts are available in the ExtraHop trigger editor.

Control+Space

This shortcut will bring up the class list for you to select your class. After you select your class, press "." to bring up an auto-complete list.

Command+A / Control + A

This shortcut selects the entire contents of the Trigger Editor window.

Shift+Tab

This shortcut will auto-indent the selected contents of the Trigger Editor window based on the use of curly braces ({}). Select the text you want indented, then press Shift+Tab.

ExtraHop data types

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

The ExtraHop system records data using two key metric categories:

Top-level metrics

Time series of simple data types.

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

Time series of data types consisting of key-value pairs, where the key is a string or an IP address and the value is a top-level data type. 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 are available to all Trigger API classes.

commitRecord(id: *String*, {key: *value*, key: *value*}): *boolean*

Commits a record to the ExtraHop Explore appliance. Returns true if the record is successfully committed.

id: *String*

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

fields: *Object*

One or more key-value pairs.



Note: The key property, or field name, in custom records must adhere to the following requirements:

- flowID, client, server, sender, and receiver cannot be used as the name of a field.
- '.', ':', '[', and ']' (period, colon, square brackets) cannot be part of the name of a field.
- 'ex' cannot be used as the name of a field.

.commitRecord will raise an exception if any of these conditions are encountered.

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 fire, 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 or not.

The limit for runtime log entries is 2048 bytes. To log larger entries, use rsyslog.

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

Returns the MD5 sum of a string.

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

Returns the sha1 hash of a string.

uuid(): *String*

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

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

You can use local JavaScript functions when you write triggers. Elements in ECMAScript 5 are supported. Examples are:

```
function example1(a, b, c) { ... }
```

```
var example2 = function(a, b, c) { ... }
```

For triggers that contain multiple events, use the event property to see the event on which the trigger is currently executing. For example:

```
if (event === "HTTP_REQUEST") {
    /* code */
} else if (
event === "
HTTP_RESPONSE") {
    /* other code */
}
```

Deprecated

exit(): *Void*

Deprecated. Use the return statement instead.

getTimestampMSec(): *Number*

Deprecated. User `getTimestamp` instead.

Classes and events

The ExtraHop trigger API supports the following classes and events:

Class name	Events	Description
AAA	AAA_REQUEST AAA_RESPONSE	Retrieves metrics available during the AAA_REQUEST and AAA_RESPONSE events.
ActiveMQ	ACTIVEMQ_MESSAGE	Retrieves metrics available during ACTIVEMQ_MESSAGE events.
AlertRecord	ALERT_RECORD_COMMIT	Provides access to alert information during the ALERT_RECORD_COMMIT event
Application	N/A	Creates new applications and to adds custom metrics at the application level.
Buffer	N/A	Provides access to buffer content.
CIFS	CIFS_REQUEST CIFS_RESPONSE	Retrieves metrics available during the CIFS_REQUEST and CIFS_RESPONSE events.
Dataset	N/A	Represents dataset metrics, provides access to raw dataset values, and provides an interface for computing percentiles.
DB	DB_REQUEST DB_RESPONSE	Allows retrieval of metrics available during the DB_REQUEST and DB_RESPONSE events.
Device	N/A	Retrieves device attributes and adds custom metrics at the device level.
DHCP	DHCP_REQUEST DHCP_RESPONSE	Retrieves metrics available during the DHCP_REQUEST and DHCP_RESPONSE events.
Discover	NEW_APPLICATION NEW_DEVICE NEW_VLAN	Provides access to newly discovered VLANs, devices or applications from the NEW_APPLICATION, NEW_DEVICE, and NEW_VLAN events.
DNS	DNS_REQUEST DNS_RESPONSE	Retrieves metrics available during the DNS_REQUEST and DNS_RESPONSE events.
FIX	FIX_REQUEST FIX_RESPONSE	Retrieves metrics available during the FIX_REQUEST and FIX_RESPONSE events.

Class name	Events	Description
FLOW	FLOW_CLASSIFY FLOW_RECORD FLOW_TICK FLOW_TURN TCP_CLOSE TCP_OPEN TCP_PAYLOAD UDP_PAYLOAD	Provides access to elements of a TCP or UDP connection between two endpoints such as endpoint identities and flow age. This class also contains a flow store designed to pass objects from request to response on the same flow.
FTP	FTP_REQUEST FTP_RESPONSE	Retrieves metrics available during the FTP_REQUEST and FTP_RESPONSE events.
HL7	HL7_REQUEST HL7_RESPONSE	Retrieves metrics available during the HL7_REQUEST and HL7_RESPONSE events.
HTTP	HTTP_REQUEST HTTP_RESPONSE	Retrieves metrics available during the HTTP_REQUEST and HTTP_RESPONSE events.
IBMMQ	IBMMQ_REQUEST IBMMQ_RESPONSE	Retrieves metrics available during the IBMMQ_REQUEST and IBMMQ_RESPONSE events.
ICA	ICA_AUTH ICA_CLOSE ICA_OPEN ICA_TICK	Retrieves metrics available during the ICA_OPEN, ICA_AUTH, ICA_TICK, and ICA_CLOSE events.
ICMP	ICMP_MESSAGE	Retrieves metrics available during the ICMP_MESSAGE event.
IPAddress	N/A	Set and retrieves IP address attributes.
Kerberos	KERBEROS_REQUEST KERBEROS_RESPONSE	Retrieves metrics available during the KERBEROS_REQUEST and KERBEROS_RESPONSE events.
LDAP	LDAP_REQUEST LDAP_RESPONSE	Retrieves metrics available during the LDAP_REQUEST and LDAP_RESPONSE events.
LLDP	LLDP_FRAME	Retrieves metrics available during the LLDP_FRAME event.
Memcache	MEMCACHE_REQUEST MEMCACHE_RESPONSE	Retrieves of metrics available during the MEMCACHE_REQUEST and MEMCACHE_RESPONSE events.

Class name	Events	Description
MetricCycle	METRIC_CYCLE_BEGIN METRIC_CYCLE_END	Represents the interval that stats were published based on the METRIC_CYCLE_BEGIN and METRIC_CYCLE_END events.
MetricRecord	METRIC_RECORD_COMMIT	Provides access to the current set of metrics during the METRIC_RECORD_COMMIT event.
MongoDB	MONGODB_REQUEST MONGODB_RESPONSE	Retrieves metrics available during the MONGODB_REQUEST and MONGODB_RESPONSE events.
MSMQ	MSMQ_MESSAGE	Retrieves metrics available during the MSMQ_MESSAGE event.
Network	N/A	Adds custom metrics at the global level.
NFS	NFS_REQUEST NFS_RESPONSE	Retrieves metrics available during the NFS_REQUEST and NFS_RESPONSE events.
POP3	POP3_REQUEST POP3_RESPONSE	Retrieves metrics available during the POP3_REQUEST and POP3_RESPONSE events.
Record	N/A	Creates a JSON object that sends information to the ExtraHop Explore appliance.
Remote.HTTP	N/A	Submits HTTP REST requests.
Remote.Kafka	N/A	Submits messages to a Kafka server.
Remote.MongoDB	N/A	Insert, removes, and updates documents in collections of MongoDB.
Remote.Raw	N/A	Submits raw data to an Open Data Stream (ODS) target through a TCP or UDP port.
Remote.Syslog	N/A	Creates remote syslog messages with specified content.
RemoteSyslog	N/A	(Deprecated. Use Remote.Syslog instead.)
RTCP	RTCP_MESSAGE	Retrieves metrics available during the RTCP_MESSAGE event.
RTP	RTP_CLOSE RTP_OPEN RTP_TICK	Retrieves metrics available during the RTP_CLOSE, RTP_OPEN, and RTP_TICK events.
Sampleset	N/A	Represents sampleset metrics.

Class name	Events	Description
SDP	SIP_REQUEST SIP_RESPONSE	Retrieves Session Description Protocol (SDP) information during the SIP_REQUEST and SIP_RESPONSE events.
Session	SESSION_EXPIRE	Manages objects in the global session table, which is a construct designed for passing objects across flows.
SIP	SIP_REQUEST SIP_RESPONSE	Retrieves metrics available during the SIP_REQUEST and SIP_RESPONSE events.
SMPP	SMPP_REQUEST SMPP_RESPONSE	Retrieves metrics available during the SMPP_REQUEST and SMPP_RESPONSE events.
SMTP	SMTP_REQUEST SMTP_RESPONSE	Retrieves metrics available during the SMTP_REQUEST and SMTP_RESPONSE events.
SSL	SSL_ALERT SSL_CLOSE SSL_HEARTBEAT SSL_OPEN SSL_PAYLOAD SSL_RECORD SSL_RENEGOTIATE	Retrieves metrics available during the SSL_ALERT, SSL_CLOSE, SSL_HEARTBEAT, SSL_OPEN, SSL_PAYLOAD, SSL_RECORD, and SSL_RENEGOTIATE events.
Telnet	TELNET_MESSAGE	Retrieves metrics available during the TELNET_MESSAGE event.
Topnset	N/A	Represents topnset metrics. A topnset metric contains a list of entries with keys and values. Values may be numbers, datasets, samplesets, or other topnsets.
Topnset Keys	N/A	Objects that represent a property of Topnset.
TroubleGroup	N/A	Provides an interface for creating custom trouble groups.
Turn	N/A	Top-level object available in the FLOW_TURN event.
VLAN	N/A	Represents a VLAN on the network.
XML	N/A	Returns parsed XML data.

AAA

The AAA class allows the retrieval of metrics available during the `AAA_REQUEST` and `AAA_RESPONSE` events.

Events

AAA_REQUEST

Fires on every AAA request processed by the device.

AAA_RESPONSE

Fires on every AAA response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `AAA_REQUEST` and `AAA_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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: Integer

The numeric ID of the attribute.

isGrouped: Boolean

Returns true if this is a grouped AVP (Diameter only).

name: String

A string name for the given AVP.

vendor: String

The vendor name for vendor AVPs (Diameter only).

value: String | Array | Number

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

isDiameter: Boolean

Returns true if the request or response is Diameter.

isError: Boolean

Returns 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).

Applies only to `AAA_RESPONSE`.

isRadius: Boolean

Returns true if the request or response is RADIUS.

isRspAborted: Boolean

Returns true if AAA_RESPONSE is aborted.

Applies only to AAA_RESPONSE.

method: Number

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
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

Command name	Abbr.	Code
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
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

record: Object

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table contains the properties available for the record of each event.

AAA_Request	AAA_Response
authenticator	authenticator
method	isError
reqBytes	isRspAborted
reqL2Bytes	method
reqPkts	roundTripTime
reqRTO	rspBytes
txId	rspL2Bytes
	rspPkts
	rspRTO
	statusCode
	processingTime
	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 RTOs.

Applies only to `AAA_REQUEST`.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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.

Applies only to `AAA_RESPONSE`.

statusCode: *String*

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

Applies only to `AAA_RESPONSE`.

tprocess: *Number*

The server processing time, expressed in milliseconds. Will return NaN if the timing is not valid.

Applies only to `AAA_RESPONSE`.

txId: *Number*

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

Deprecated

error: *String*

Use `isError` instead.

ActiveMQ

The ActiveMQ class allows the retrieval of metrics available during the `ACTIVEMQ_MESSAGE` event. This is an implementation of the Java Messaging Service (JMS).

Events

ACTIVEMQ_MESSAGE

Fires on every JMS message processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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.

totalMsgLength: *Number*

The length of the message, expressed in bytes.

persistent: *Boolean*

Returns 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*

Returns 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.

record: *Object*

Returns an object with all properties appropriately initialized.

The following `ACTIVEMQ_MESSAGE` properties are available:

- correlationId
- expiration
- msgFormat
- msgId
- persistent
- priority
- queue
- receiverBytes
- receiverIsBroker
- receiverL2Bytes
- receiverPkts
- receiverRTO
- redeliveryCount
- replyTo
- roundTripTime
- senderBytes
- senderIsBroker
- senderL2Bytes
- senderPkts
- senderRTO
- 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. Will return NaN if there are no RTT samples.

senderBytes: *Number*

The number of application-level bytes from the sender.

senderIsBroker: *Boolean*

Returns 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.

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.

AlertRecord

The AlertRecord class allows access the current alert information in ALERT_RECORD_COMMIT.

Events

ALERT_RECORD_COMMIT

Fires when an alert fires. Provides access to information about the alert that has fired.

Properties

description: *String*

The description of the alert as it appears in the ExtraHop 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 fired.

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.

Application

The Application class allows you to create new applications and add custom 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 UI. Application names must use ASCII characters only.

Refer to [ExtraHop Trigger API Quick Start Guide](#) for more information about applications.

Methods

commit(): void

Commits metrics on an application. Applications are automatically created the first time they are referenced. For instance, to create an application named "myApp", use the following syntax:

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

The application key (e.g., "myApp") must be a string and not an integer. For example, "1020" is treated the same as 1020. Strings that cannot be used include null and Object Object.

Each unique application key and event pair will be committed only once, even if Application.commit is called multiple times for the same unique pair.

The following table shows which event(s) to use to create applications.

Application component	Event
AAA	AAA_REQUEST -and- AAA_RESPONSE
DB	DB_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
L4	TCP_OPEN -or- FLOW_CLASSIFY
LDAP	LDAP_REQUEST -and- LDAP_RESPONSE
Memcache	MEMCACHE_REQUEST -and- MEMCACHE_RESPONSE
NAS	CIFS_RESPONSE -or- NFS_RESPONSE
RTP	RTP_TICK
RTCP	RTCP_MESSAGE
SIP	SIP_REQUEST -and- SIP_RESPONSE
SSL	SSL_RECORD -and- SSL_CLOSE
SMTP	SMTP_RESPONSE



Note: In TCP_OPEN and FLOW_CLASSIFY, Application.commit is not valid. Instead use Flow.setApplication("Report Pools") to create and assign an L4 application to the flow. Flow.setApplication() can be used from any device event. If you invoke Application.commit in an invalid event, the most common

error returned will be similar to Uncaught Error: Application.commit is not valid on HTTP_REQUEST.

Use the following methods to record custom metrics associated with applications. Refer to [ExtraHop data types](#) for an overview of the data types.

- `metricAddCount(metric_name:String, count:Number, [highPrecision:bool]):void`
- `metricAddDataset(metric_name:String, val:Number, [freq:Number], [highPrecision:bool]):void`
- `metricAddDetailCount(metric_name:String, key:String | IPAddress, count:Number, [highPrecision:bool]):void`
- `metricAddDetailSnap(metric_name:String, key:String | IPAddress, count:Number, [highPrecision:bool]):void`
- `metricAddDetailDataset(metric_name:String, key:String | IPAddress, val:Number, [freq:Number], [highPrecision:bool]):void`
- `metricAddDetailMax(metric_name:String, key:String | IPAddress, val:Number, [highPrecision:bool]):void`
- `metricAddDetailSampleSet(metric_name:String, key:String | IPAddress, val:Number, [highPrecision:bool]):void`
- `metricAddMax(metric_name:String, val:Number, [highPrecision:bool]):void`
- `metricAddSampleSet(metric_name:String, val:Number, [highPrecision:bool]):void`
- `metricAddSnap(metric_name:String, count:Number, [highPrecision:bool]):void`

The optional `highPrecision` flag will enable one second granularity for the metrics when set to true.

The above methods cannot not be called on Application directly, but must be called on specific Application instances. For example:

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



Note: When NaN is passed to a `metricAdd*` function, it is silently discarded. All count parameters for `metricAdd*` functions only accept a non-zero, positive integer between 1 and 2^{64} .

Properties

id: *String*

The application ID.

See also

- [Example: Trigger-based application definition](#)

Buffer

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. It 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 fires one time after the first n bytes of payload.

Methods

Buffer.decode(type: *String*): *String*

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

- utf-8
- ucs2
- hex

Buffer.toString(): *String*

Converts the buffer to a string.

Buffer.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.

Buffer.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. The result is a JavaScript array containing unpacked fields, even if it contains exactly one item.



- 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 following table displays supported format strings:

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

Format	C type	JavaScript type	Standard size
i	int	number	4
l	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	
z	char[]	string	

The following is an example of a UDP_PAYLOAD trigger that parses NTP with Buffer.unpack:

```

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 us to continue parsing where we left off.
values = buf.unpack(fmt, offset);
ntpData.referenceId = values[0];

// We only use the integral parts of the timestamp 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));

```

Properties

Buffer.length: *Number*

The number of bytes in the buffer.

Flow.server.payload: *Buffer*

Returns a buffer containing the server payload. If no payload exists, null is returned.

Flow.client.payload: *Buffer*

Returns a buffer containing the client payload. If no payload exists, null is returned.

CIFS

The CIFS class allows the retrieval of metrics available during the CIFS_REQUEST and CIFS_RESPONSE events.

Events

CIFS_REQUEST

Fires on every CIFS request processed by the device.

CIFS_RESPONSE

Fires on every CIFS response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Applies only to `CIFS_RESPONSE`. See the `record` property below for details.

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

Properties

accessTime: Number

The time it took for 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 payload of its response. Will return NaN if there is no valid measurement, or if the timing is not valid.

Applies only to `CIFS_RESPONSE`.

encryptedBytes: Number

The number of encrypted bytes in the request or response.

error: String

The detailed error message recorded by the ExtraHop system.

Applies only to `CIFS_RESPONSE`.

isCommandFileInfo: Boolean

Returns true if the command is a file info command.

isCommandLock: Boolean

Returns true if the command is a locking command.

isCommandRead: Boolean

Returns true if the command is a read.

isCommandWrite: Boolean

Returns true if the command is a write.

method: String

The CIFS method (appears under Methods in the UI).

record: Object

Returns an object with all properties appropriately initialized.

Applies only to `CIFS_RESPONSE`. The following `CIFS_RESPONSE` properties are available:

- `accessTime`
- `error`
- `isCommandFileInfo`
- `isCommandLock`
- `isCommandRead`
- `isCommandWrite`
- `method`
- `reqBytes`
- `reqL2Bytes`
- `reqPkts`
- `reqRTO`
- `reqSize`

- resource
- roundTripTime
- rspBytes
- rspL2Bytes
- rspPkts
- rspRTO
- rspSize
- share
- statusCode
- user

reqBytes: Number

The number of L4 request bytes.

Applies only to CIFS_RESPONSE.

reqL2Bytes: Number

The number of L2 request bytes.

Applies only to CIFS_RESPONSE.

reqPkts: Number

The number of request packets.

Applies only to CIFS_RESPONSE.

reqRTO: Number

The number of request RTOs.

Applies only to CIFS_RESPONSE.

reqSize: Number

The size of the request, expressed in bytes.

resource: String

The share, path, and filename, concatenated together.

roundTripTime: Number

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

Applies only to CIFS_RESPONSE.

rspBytes: Number

The number of L4 response bytes.

Applies only to CIFS_RESPONSE.

rspL2Bytes: Number

The number of L2 response bytes.

Applies only to CIFS_RESPONSE.

rspPkts: Number

The number of response packets.

Applies only to CIFS_RESPONSE.

rspRTO: Number

The number of response RTOs.

Applies only to CIFS_RESPONSE.

rspSize: Number

The size of the response, expressed in bytes.

Applies only to `CIFS_RESPONSE`.

share: *String*

The name of the share to which the user is connected.

statusCode: *Number*

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

Applies only to `CIFS_RESPONSE`.

user: *String*

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

warning: *String*

The detailed warning message recorded by the ExtraHop system.

Applies only to `CIFS_RESPONSE`.

See also

- [Example: CIFS trigger](#)

Dataset

The dataset class is used to represent dataset metrics. It provides access to the 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.

DB

The DB class allows the retrieval of metrics available during the `DB_REQUEST` and `DB_RESPONSE` events.

Events

DB_REQUEST

Fires on every database request processed by the device.

DB_RESPONSE

Fires on every database response processed by the device.

Method

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

Applies only to `DB_RESPONSE`. See the `record` property below for details.

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

Properties

correlationId: *Number*

Returns the correlation ID for DB2 applications. Returns 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 message(s) recorded by the ExtraHop system in string format. When a database returns multiple errors in one response, error will contain a concatenated string of all errors returned.

Applies only to `DB_RESPONSE`.

errors: *Array of strings*

The detailed error message(s) recorded by the ExtraHop system in array format. When only a single error is returned, errors will contain an array of one string.

Applies only to `DB_RESPONSE`.

isReqAborted: *Boolean*

Returns true if the connection is closed before the DB request is complete.

isRspAborted: *Boolean*

Returns true if the connection is closed before the DB response is complete.

Applies only to `DB_RESPONSE`.

method: *String*

Database method (appears under **Methods** in the user interface).

params: *Array*

List of RPC parameters (only available for Microsoft SQL and DB2 databases).

Applies only to `DB_REQUEST`.

This property is an array of objects and each object has the following properties:

name: *String*

The optional name of the supplied RPC parameter and the value is the value of the parameter.

value: *String | Number*

If the value is not a text, integer, or a time/date field, it will be canonicalized into hex/ASCII form.

procedure: *String*

The stored procedure name (appears under **Methods** in the user interface).

record: *Object*

Returns an object with all properties appropriately initialized.

Applies only to `DB_RESPONSE`. The following `DB_RESPONSE` properties are available:

- correlationId
- database
- error
- isReqAborted
- isRspAborted
- method

- procedure
- reqBytes
- reqL2Bytes
- reqPkts
- reqRTO
- reqSize
- reqTimeToLastByte
- roundTripTime
- rspBytes
- rspL2Bytes
- rspPkts
- rspRTO
- rspSize
- rspTimeToFirstByte
- rspTimeToLastByte
- table
- processingTime
- user

reqBytes: *Number*

The number of L4 request bytes.

Applies only to DB_RESPONSE.

reqL2Bytes: *Number*

The number of L2 request bytes.

Applies only to DB_RESPONSE.

reqPkts: *Number*

The number of request packets.

Applies only to DB_RESPONSE.

reqRTO: *Number*

The number of request RTOs.

Applies only to DB_RESPONSE.

reqSize: *Number*

The size of the request record at L7, expressed in bytes.

reqTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the request, expressed in milliseconds. Will return NaN on malformed and aborted requests, or if the timing is not valid.

roundTripTime: *Number*

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

Applies only to DB_RESPONSE.

rspBytes: *Number*

The number of L4 response bytes.

Applies only to DB_RESPONSE.

rspL2Bytes: *Number*

The number of L2 response bytes.

Applies only to DB_RESPONSE.

rspPkts: *Number*

The number of response packets.

Applies only to DB_RESPONSE.

rspRTO: *Number*

The number of response RTOs.

Applies only to DB_RESPONSE.

rspSize: *Number*

The size of the response record at L7, expressed in bytes.

Applies only to DB_RESPONSE.

rspTimeToFirstByte: *Number*

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to DB_RESPONSE.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to DB_RESPONSE.

statement: *String*

The full SQL statement (may not be available for all DB methods).

Applies only to DB_REQUEST.

table: *String*

The name of the database table specified in the current statement. This field is empty if there is no table name in the request.

Applies only to Sybase IQ database.

tprocess: *Number*

The server processing time, expressed in milliseconds (equivalent to `rspTimeToFirstByte - reqTimeToLastByte`). Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to DB_RESPONSE.

user: *String*

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

See also

- [Example: Database trigger](#)
- [Example: Trigger-based application definition](#)

Device

The Device class allows retrieval of device attributes and adding 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, 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);
```

Use the following methods to record custom metrics associated with devices. Refer to [ExtraHop data types](#) for an overview of the data types.

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

The optional `highPrecision` flag will enable one second granularity for the metrics when set to true.



- Note:**
- Calling `Device.metricAdd*` functions records metrics for both devices on the flow, regardless of how the triggers are assigned.
 - Calling `Flow.client.device.metricAdd*` functions records metrics only for the client device, regardless of whether the trigger is assigned to the client or the server.
 - Calling `Flow.server.device.metricAdd*` functions records metrics only for the server device, regardless of whether the trigger is assigned to the client or the server.
 - Adding a metric to the Network object makes it available to network-level custom pages.
 - Adding a metric to the Device object makes it available to custom pages on that device.
 - The `metricAddMax` and `metricAddDetailMax` functions commit metrics that preserve a maximum. For instance, use the `metricAddMax` function to record maximum values of database server processing times overtime.
 - If the information is unavailable or not applicable, the value of a property will be null where the type is normally a String and NaN where the type is normally a Number.
 - When NaN is passed to a `metricAdd*` function, it is silently discarded.

- All count parameters for `metricAdd*` functions only accept a non-zero, positive integer between 1 and 2^{64} .

equals(): Boolean

Performs an equality test between Device objects.

Instance properties

The following properties allow retrieval of 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

Returns true if the currently executing trigger is configured on the Device object on which the `hasTrigger` property is called. For all trigger events with an associated `FLOW` object, at least one of the Device objects in the flow will have its `hasTrigger` property set to true.

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 UI in the **Device > Device ID** field.

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

Returns true if the device is a gateway.

isL3: Boolean

Returns 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.

See also

- [Example: CIFS trigger](#)
- [Example: Customer ID header](#)
- [Example: Database trigger](#)
- [Example: Device discovery notification](#)

- [Example: HTTP header object](#)
- [Example: Memcache hits and misses](#)
- [Example: Memcache key parsing](#)
- [Example: Parse custom POS messages with universal payload analysis](#)
- [Example: Use the Metric Cycle store](#)

DHCP

The DHCP class allows the retrieval of metrics available during the `DHCP_REQUEST` and `DHCP_RESPONSE` events.

Events

DHCP_REQUEST

Fires on every DHCP request processed by the device.

DHCP_RESPONSE

Fires on every DHCP response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `DHCP_REQUEST` and `DHCP_RESPONSE`. See the `record` property below for details.

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

getOption(optionCode: Integer): Object

Function that takes a DHCP option code as input and returns an object containing the following three fields. If the specified option code is not present in the message, the method return a `NULL`.

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.

Properties

clientAddr: IPAddress

The IP address of the DHCP client.

clientReqDelay: Number

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

Applies only to `DHCP_REQUEST`.

error: String

The error message associated with option code 56. Will return a `NULL` if there is no error.

Applies only to `DHCP_RESPONSE`.

gwaddr: *IPAddress*

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

htype: *Number*

The hardware type code.

hwAddr: *IPAddress*

The DHCP client hardware address.

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.

Applies only to `DHCP_RESPONSE`.

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. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `DHCP_RESPONSE`.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

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.

Applies only to DHCP_REQUEST.

reqL2Bytes: Number

The number of request L2 bytes.

Applies only to DHCP_REQUEST.

reqPkts: Number

The number of request packets.

Applies only to DHCP_REQUEST.

rspBytes: Number

The number of L4 response bytes.

Applies only to DHCP_RESPONSE.

rspL2Bytes: Number

The number of L2 response bytes.

Applies only to DHCP_RESPONSE.

rspPkts: Number

The number of response packets.

Applies only to DHCP_RESPONSE.

serverAddr: IP Address

The IP address of the server to use for the next step in the bootstrap process.

txId: Number

The transaction ID.

Discover

The Discover class provides access to newly discovered VLANs, devices, or applications on the NEW_VLAN, NEW_DEVICE, and NEW_APPLICATION events.

Events

NEW_APPLICATION

Fires when an application is first discovered.

NEW_DEVICE

Fires when a device is first discovered.

NEW_VLAN

Fires when a VLAN is first discovered.

Properties

application: *Application*

A newly discovered application.

Applies only to `NEW_APPLICATION`.

device: *Device*

A newly discovered device.

Applies only to `NEW_DEVICE`.

vlan: *VLAN*

A newly discovered VLAN.

Applies only to `NEW_VLAN`.

See also

- [Example: Device discovery notification](#)

DNS

The DNS class allows retrieval of metrics available during the `DNS_REQUEST` and `DNS_RESPONSE` events.

Events

DNS_REQUEST

Fires on every DNS request processed by the device.

DNS_RESPONSE

Fires on every DNS response processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `DNS_REQUEST` and `DNS_RESPONSE`. See the `record` property below for details.

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

Properties

answers: *Array*

An array of objects corresponding to answer resource records.

Applies only to `DNS_RESPONSE`.

The objects have the following properties:

data: *String*

The value of data depends on the type and will be null for unsupported record types. Supported record types include:

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

name: *String*

Record name.

ttl: *Number*

Time-to-live.

type: *String*

DNS record type.

error: *String*

Detailed error message recorded by the ExtraHop system.

Applies only to DNS_RESPONSE.

isAuthoritative: *Boolean*

Returns true if the authoritative answer is set in the response.

Applies only to DNS_RESPONSE.

isReqTimeout: *Boolean*

Returns true if the request timed out.

Applies only to DNS_REQUEST.

isRspTruncated: *Boolean*

Returns true if the response is truncated.

Applies only to DNS_RESPONSE.

opcode: *String*

DNS opcode. The following codes are valid:

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

qname: *String*

Corresponds to the hostname queried.

qtype: *String*

The DNS request record type.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

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

reqBytes: *Number*

The number of application-level request bytes.

Applies only to DNS_REQUEST.

reqL2Bytes: *Number*

The number of request L2 bytes.

Applies only to DNS_REQUEST.

reqPkts: *Number*

The number of request packets.

Applies only to DNS_REQUEST.

rspBytes: *Number*

The number of response bytes.

Applies only to DNS_RESPONSE.

rspL2Bytes: *Number*

The number of response L2 bytes.

Applies only to DNS_RESPONSE.

rspPkts: *Number*

The number of application-level response bytes.

Applies only to DNS_RESPONSE.

tprocess: *Number*

The server processing time, expressed in bytes. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `DNS_RESPONSE`.

FIX

The `FIX` class allows retrieval of metrics available during the `FIX_REQUEST` and `FIX_RESPONSE` events.

Events

`FIX_REQUEST`

Fires on every `FIX` request processed by the device.

`FIX_RESPONSE`

Fires 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 the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `FIX_REQUEST` and `FIX_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

FIX_REQUEST	FIX_RESPONSE
msgType	msgType
reqBytes	rspBytes
reqL2Bytes	rspL2Bytes
reqPkts	rspPkts
reqRTO	rspRTO
sender	sender
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.

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.

sender: *String*

The value of the SenderCompID key.

target: *String*

The value of the TargetCompID key.

version: *String*

The protocol version.

FLOW

Flow refers to a TCP or UDP connection between two endpoints. The Flow class provides access to elements of these conversations, such as endpoint identities and flow age. It also contains a flow store designed to pass objects from request to response on the same flow. Using a combination of L7 payload analysis, observation of TCP handshakes, and port number-based heuristics, the ExtraHop system identifies the L7 protocol and the client and server roles for the endpoints in a flow.

FLOW_CLASSIFY

Fires once per flow when the L7 protocol of the flow has been determined.

For TCP flows, the `FLOW_CLASSIFY` event fires after the `TCP_OPEN` event.

For certain L7 protocols, the nature of a flow changes over its lifetime (for example, tunneling over HTTP or switching from SMTP to SMTP-TLS). In those cases, `FLOW_CLASSIFY` will fire again after the change.

For flows associated with ExtraHop-monitored protocols (such as HTTP), L7 trigger events (such as `HTTP_REQUEST` and `HTTP_RESPONSE`) will additionally fire for the flow. The Flow object properties and methods discussed in this section are available to every L7 trigger event associated with the flow.

The `FLOW_CLASSIFY` event is useful for initiating an action on a flow when it can be done, based on the earliest knowledge of a flow (for example, client and server IPs, client and server ports, L7 protocol). Actions commonly taken during this trigger event include starting a packet capture via the `captureStart()` method or associating the flow with an application container using `setApplication()`. It is best practice to wait before taking any action until more information becomes known about the flow via a future L7 trigger event (for example, once it's known that the flow is an HTTP request for a specific URI).

Once the `FLOW_CLASSIFY` trigger event fires, two or more of the following data elements are available, depending on the subsequent trigger 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

Event	Client / Server	Sender / Receiver
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
FIX_RESPONSE	yes	yes
FLOW_CLASSIFY	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

Event	Client / Server	Sender / Receiver
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
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

age: *Number*

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

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

Initiates a Precision Packet Capture for the flow and returns a unique identifier of the packet capture (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
- The system creates a separate capture for each flow.
- Capture files with the same name are differentiated by timestamps.

options: *Object*

Omit any of the options to indicate unlimited size for that option. "Lookback buffer" refers to packets captured before the call to `captureStart()`. 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.

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 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 a license with Triggered Packet Capture enabled.
 - The packet capture trigger must have packet capture enabled. When configuring the trigger in the Web UI, select the Packet Capture checkbox under Advanced Options.
 - Captured files are available in the Admin UI.
 - Once the packet capture drive is full, no new captures will be recorded until the user deletes the files manually.
 - 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 lesser of `maxPackets` and `maxBytes`.
 - Size of capture allocated to data in the lookback buffer is the lesser of `maxPacketsLookback` and `maxBytesLookback`.
 - Each passed `max*` parameter will capture up to the next packet boundary.
 - If the packet capture was already started on the current flow, the `Flow.captureStart()` calls result in a warning visible in the debug log, but the trigger will continue to execute.
 - There is a maximum of 128 simultaneous 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.

getApplication(): *String*

Gets the currently associated application.

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.

isExpired: *Boolean*

Returns true if the flow expired at the time of the event.

l7proto: *String*

The L7 protocol associated with the flow. For known protocols, it is a string representing the protocol name (e.g., HTTP, DB, Memcache, or any L7 section heading in this document). For lesser-known protocols, it is a string with the format `l7proto:port` (e.g., `tcp:13724` or `udp:11258`). Not valid during the `TCP_OPEN` event.

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

Associates L2-L4 metrics for a flow with the L4 component of the application specified by name. 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.

`Flow.setApplication(name)` is commonly used in a `FLOW_CLASSIFY` event to associate flows with applications where there is no corresponding L7 trigger event on which to call `Application(name).commit()` (e.g., proprietary protocols for which there is no ExtraHop analysis module). For flows that have L7 trigger events that support the `Application(name).commit()` method, that method collects a larger set of protocol metrics and is generally recommended.

A flow is associated with at most one application at a given instant. It is possible, however, to have a set of triggers call `Flow.setApplication(name)` at different instances and with different name arguments over the course of a given flow. This configuration results in metrics for a flow being dispersed across multiple applications and is generally not recommended.

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.

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 use `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.

The following properties and methods apply to both types of flows.

device1: *Device*

The device object associated with the device with a numerically lower IP address. For instance, to access the MAC address of the server, use `Flow.device1.hwaddr`.

equals: *Boolean*

Performs an equality test between Device objects.

device2: *Device*

The device object associated with the device with a numerically higher IP address. For instance, to access the MAC address of the server, use `Flow.device2.hwaddr`.

equals: *Boolean*

Performs an equality test between Device objects.

ipaddr1: *IPAddress*

The IP address object associated with the device with the numerically lower IP address.

equals: *Boolean*

Performs an equality test between IPAddress objects.

ipaddr2: *IPAddress*

The IP address object associated with the device with the numerically higher IP address.

equals: *Boolean*

Performs an equality test between IPAddress objects.

port1: *Number*

The port number used by the device with the numerically lower IP address.

port2: *Number*

The port number used by the device with the numerically higher IP address.

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.

client

device: *Device*

The device object associated with the client. For instance, to access the MAC address of the client, use `Flow.client.device.hwaddr`.

equals: Boolean

Performs an equality test between Device objects.

ipaddr: IPAddress

The IP address object associated with the client.

equals: Boolean

Performs an equality test between IPAddress objects.

isAborted: Boolean

Returns true if the client has aborted a TCP flow by issuing a TCP reset (RST). 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

Returns true if the client initiated the shutdown of the TCP Connection.

port: Number

The port number used by the client in the flow.

receiver

device: Device

The device object associated with the receiver. For instance, to access the MAC address of the receiver, use `Flow.receiver.device.hwaddr`.

equals: Boolean

Performs an equality test between Device objects.

ipaddr: IPAddress

The IP address object associated with the receiver.

equals: Boolean

Performs an equality test between IPAddress objects.

isAborted: Boolean

Returns true if the receiver has aborted a TCP flow by issuing a TCP reset (RST). 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

Returns true if the receiver initiated the shutdown of the TCP Connection.

port: Number

The port number used by the receiver in the flow.

sender

device: *Device*

The device object associated with the sender. For instance, to access the MAC address of the sender, use `Flow.sender.device.hwaddr`.

equals: *Boolean*

Performs an equality test between Device objects.

ipaddr: *IPAddress*

The IP address object associated with the sender.

equals: *Boolean*

Performs an equality test between IPAddress objects.

isAborted: *Boolean*

Returns true if the sender has aborted a TCP flow by issuing a TCP reset (RST). 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*

Returns true if the sender initiated the shutdown of the TCP Connection.

port: *Number*

The port number used by the sender in the flow.

server

device: *Device*

The device object associated with the server. For instance, to access the MAC address of the server, use `Flow.server.device.hwaddr`.

equals: *Boolean*

Performs an equality test between Device objects.

ipaddr: *IPAddress*

The IP address object associated with the server.

equals: *Boolean*

Performs an equality test between IPAddress objects.

isAborted: *Boolean*

Returns true if the server has aborted a TCP flow by issuing a TCP reset (RST). 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*

Returns true if the server initiated the shutdown of the TCP Connection.

port: *Number*

The port number used by the server in the flow.

FLOW_RECORD

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.

In additions to the properties and methods gives for the `FLOW_CLASSIFY` event, the following properties become available during `FLOW_RECORD` events and the related `FLOW_TICK` and `FLOW_TURN` events. These properties are not available during the `FLOW_CLASSIFY` event.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

The following record properties are supported:

- bytes
- first
- last
- pkts
- proto
- receiverAddr
- receiverPort
- senderAddr
- senderPort
- tcpFlags
- tos

Properties

Flow.client.record: *Object*

Returns an object with all properties appropriately initialized for the client in the flow

Flow.record1: *Object*

Returns an object with all properties appropriately initialized for the device with the numerically higher IP address in the flow.

Flow.record2: *Object*

Returns an object with all properties appropriately initialized for the device with the numerically lower IP address in the flow.

Flow.server.record: *Object*

Returns an object with all properties appropriately initialized for the server on the flow.

FLOW_TICK

Once `FLOW_CLASSIFY` has fired, the `FLOW_TICK` event will fire on every subsequent turn. A turn represents one full cycle of a client transferring request data followed by a server transferring a response. For turns larger than 128 packets, `FLOW_TICK` will fire every 128 packets. You can configure this threshold in the Admin UI.

`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.

In addition to the properties and methods given for the `FLOW_CLASSIFY` event, the following properties become available during `FLOW_TICK` events and during the related `FLOW_TURN` event. These properties are not available during the `FLOW_CLASSIFY` event.

Flow.bytes1: *Number*

The number of L4 payload bytes transmitted by the device with the numerically lower IP address.

Flow.bytes2: *Number*

The number of L4 payload bytes transmitted by the device with the numerically higher IP address.

Flow.dscp1: *Number*

The last Differentiated Services Code Point (DSCP) value transmitted by the device with the numerically lower IP address.

Flow.dscp2: *Number*

The last Differentiated Services Code Point (DSCP) value transmitted by the device with the numerically higher IP address.

Flow.l2Bytes1: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by the device with the numerically lower IP address.

Flow.l2Bytes2: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by the device with the numerically higher IP address.

Flow.client.customDevices: *Array*

A list of custom devices that are acting as the clients in the flow.

Flow.client.dscp: *Number*

The last Differentiated Services Code Point (DSCP) value transmitted by the client in the flow.

Flow.client.dscpBytes: *Array*

An array of counts of the number of L2 bytes for a given Differentiated Services Code Point (DSCP) value transmitted by the client in the flow. A zero is returned for each entry which had no bytes of that DSCP value since the last `FLOW_TICK`.

Flow.client.dscpPkts: *Array*

An array of counts of the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by the client in the flow. A zero is returned for each entry which had no packets of that DSCP value since the last `FLOW_TICK`.

Flow.client.l2Bytes: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by the client in the flow.

Flow.client.nagleDelay: *Number*

The number of Nagle delays associated with the client in the flow.

Flow.client.payload: *Buffer*

The payload associated with the client in the flow.

Flow.client.pkts: *Number*

The number of packets transmitted by the client in the flow.

Flow.client.rcvWndThrottle: *Number*

The number of receive window throttles associated with the client in the flow.

Flow.client.rto: *Number*

The number of "RTO Out" when the device is acting as the client in the flow.

Flow.client.zeroWnd: *Number*

The number of zero windows associated with the client in the flow.

Flow.customDevices1: *Number*

A list of custom devices associated with the device with the numerically lower IP address.

Flow.customDevices2: *Number*

A list of custom devices associated with the device with the numerically higher IP address.

Flow.dscpBytes1: *Array*

An array of counts of the number of L2 bytes for a given Differentiated Services Code Point (DSCP) value transmitted by the device with the numerically lower IP address in the flow.

A zero is returned for each entry which had no bytes of that DSCP value since the last FLOW_TICK.

Flow.dscpBytes2: *Array*

An array of counts of the number of L2 bytes for a given Differentiated Services Code Point (DSCP) value transmitted by the device with the numerically higher IP address in the flow.

A zero is returned for each entry which had no bytes of that DSCP value since the last FLOW_TICK.

Flow.dscpPkts1: *Array*

An array of counts of the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by the device with the numerically lower IP address in the flow.

A zero is returned for each entry which had no packets of that DSCP value since the last FLOW_TICK.

Flow.dscpPkts2: *Array*

An array of counts of the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by the device with the numerically higher IP address in the flow.

A zero is returned for each entry which had no packets of that DSCP value since the last FLOW_TICK.

Flow.id: *String*

Retrieves the unique identifier of a Flow record.

Flow.nagleDelay1: *Number*

The number of nagle delays associated with the device with numerically lower IP address.

Flow.nagleDelay2: *Number*

The number of nagle delays associated with the device with numerically higher IP address.

Flow.payload1: *Buffer*

The payload associated with the device with the numerically lower IP address.

Flow.payload2: *Buffer*

The payload associated with the device with the numerically higher IP address.

Flow.pkts1: *Number*

The number of packets transmitted by the device with the numerically lower IP address.

Flow.pkts2: *Number*

The number of packets transmitted by the device with the numerically higher IP address.

Flow.rcvWndThrottle1: *Number*

The number of times the advertised receive window of the device with the numerically lower IP address limits the throughput of the connection.

Flow.rcvWndThrottle2: *Number*

The number of times the advertised receive window of the device with the numerically higher IP address limits the throughput of the connection.

Flow.receiver.bytes: *Number*

The number of bytes transmitted by the receiver in the flow.

Flow.receiver.customDevices: *Number*

A list of custom devices that are acting as the receivers in the flow.

Flow.receiver.dscp: *Number*

The last Differentiated Services Code Point (DSCP) value transmitted by the receiver in the flow.

Flow.receiver.dscpBytes: *Array*

An array of counts of the number of L2 bytes for a given Differentiated Services Code Point (DSCP) value transmitted by the receiver in the flow. A zero is returned for each entry which had no bytes of that DSCP value since the last `FLOW_TICK`.

Flow.receiver.dscpPkts: *Array*

An array of counts of the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by the receiver in the flow. A zero is returned for each entry which had no packets of that DSCP value since the last `FLOW_TICK`.

Flow.receiver.l2Bytes: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by the receiver in the flow.

Flow.receiver.nagleDelay: *Number*

The number of nagle delays associated with the receiver in the flow.

Flow.receiver.payload: *Buffer*

The payload associated with the receiver in the flow. This is always null.

Flow.receiver.pkts: *Number*

The number of packets transmitted by the receiver in the flow.

Flow.receiver.rcvWndThrottle: *Number*

The number of receive window throttles associated with the receiver in the flow.

Flow.receiver.rto: *Number*

The number of "RTO Out" when the device is acting as the receiver in the flow.

Flow.receiver.zeroWnd: *Number*

The number of zero windows associated with the receiver in the flow.

Flow.roundTripTime: *Number*

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

Flow.rto1: *Number*

The number of RTOs associated with the device with numerically lower IP address.

Flow.rto2: *Number*

The number of RTOs associated with the device with numerically higher IP address.

Flow.sender.bytes: *Number*

The number of L4 payload bytes transmitted by the sender in the flow.

Flow.sender.customDevices: *Number*

A list of custom devices that are acting as the senders in the flow.

Flow.sender.dscp: *Number*

The last Differentiated Services Code Point (DSCP) value transmitted by the sender in the flow.

Flow.sender.dscpBytes: *Array*

An array of counts of the number of L2 bytes for a given Differentiated Services Code Point (DSCP) value transmitted by the sender in the flow. A zero is returned for each entry which had no bytes of that DSCP value since the last `FLOW_TICK`.

Flow.sender.dscpPkts: *Array*

An array of counts of the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by the sender in the flow. A zero is returned for each entry which had no packets of that DSCP value since the last `FLOW_TICK`.

Flow.sender.l2Bytes: *Number*

The number of L2 bytes, including the ethernet headers, transmitted by the sender in the flow.

Flow.sender.nagleDelay: *Number*

The number of nagle delays associated with the sender in the flow.

Flow.sender.payload: *Buffer*

The payload associated with the sender in the flow.

Flow.sender.pkts: *Number*

The number of packets transmitted by the sender in the flow.

Flow.sender.rcvWndThrottle: *Number*

The number of receive window throttles associated with the sender in the flow.

Flow.sender.rto: *Number*

The number of "RTO Out" when the device is acting as the sender in the flow.

Flow.sender.zeroWnd: *Number*

The number of zero windows associated with the sender in the flow.

Flow.server.bytes: *Number*

The number of L4 payload bytes transmitted by the server in the flow.

Flow.server.customDevices: *Number*

A list of custom devices that are acting as the servers in the flow.

Flow.server.dscp: *Number*

The last Differentiated Services Code Point (DSCP) value transmitted by the server in the flow.

Flow.server.dscpBytes: *Array*

An array of counts of the number of L2 bytes for a given Differentiated Services Code Point (DSCP) value transmitted by the server in the flow. A zero is returned for each entry which had no bytes of that DSCP value since the last `FLOW_TICK`.

Flow.server.dscpPkts: *Array*

An array of counts of the number of L2 packets for a given Differentiated Services Code Point (DSCP) value transmitted by the server in the flow. A zero is returned for each entry which had no packets of that DSCP value since the last `FLOW_TICK`.

Flow.server.l2Bytes: *Number*

The number of L2 bytes transmitted by the server in the flow.

Flow.server.nagleDelay: *Number*

The number of nagle delays associated with the server in the flow.

Flow.server.payload: *Buffer*

The payload associated with the server in the flow.

Flow.server.pkts: *Number*

The number of packets transmitted by the server in the flow.

Flow.server.rcvWndThrottle: *Number*

The number of receive window throttles associated with the server in the flow.

Flow.server.rto: *Number*

The number of "RTO Out" when the device is acting as the server in the flow.

Flow.server.zeroWnd: *Number*

The number of zero windows associated with the server in the flow.

Flow.zeroWnd1: *Number*

The number of zero windows associated with the device with numerically lower IP address.

Flow.zeroWnd2: *Number*

The number of zero windows associated with the device with numerically higher IP address.

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

Fires 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.

In addition to the properties of `FLOW_TICK`, the event also exposes a Turn object.

TCP_CLOSE

Fires when the TCP connection is shut down by being closed, expired or aborted.

TCP.client.isReset(): *Boolean*

Returns true if a TCP reset (RST) was seen from the client while the connection was in the process of being shut down.

TCP.isReset(): *Boolean*

Returns true if a TCP reset (RST) was seen while the connection was in the process of being shut down.

TCP.server.isReset(): *Boolean*

Returns true if a TCP reset (RST) was seen from the server while the connection was in the process of being shut down.

TCP_OPEN

Fires when the TCP connection is first fully established. Provides a hook for recording metrics using the following properties:

TCP.client.getOption(): *Array*

Returns an array of all TCP options on the client that have a kind number matching the passed in value.

TCP.client.hasECNEcho: *Boolean*

Returns true if the ECN flag is set on the client during the three-way handshake.

TCP.client.initSeqNum: *Number*

The initial sequence number sent from the client during the three-way handshake.

TCP.client.options: *Array*

An array of objects representing the TCP options of the client with a numerically lower IP address in the initial handshake packets. For more information, refer to TCP Options below.

TCP.client.wndSize: *Number*

The size of the TCP sliding window on the client negotiated during the three-way handshake.

TCP.hasECNEcho1: *Boolean*

Returns true if the ECN flag is set on the device with a numerically lower IP address during the three-way handshake.

TCP.hasECNEcho2: *Boolean*

Returns true if the ECN flag is set on the device with a numerically higher IP address during the three-way handshake.

TCP.initSeqNum1: *Number*

The initial sequence number of the device with a numerically lower IP address sent during the three-way handshake.

TCP.initSeqNum2: *Number*

The initial sequence number of the device with a numerically higher IP address sent during the three-way handshake.

TCP.options1: *Array*

An array of options representing the TCP options of the device with a numerically lower IP address in the initial handshake packets. For more information, refer to TCP Options below.

TCP.options2: *Array*

An array of options representing the TCP options of the device with a numerically higher IP address in the initial handshake packets. For more information, refer to TCP Options below.

TCP.server.getOption(): *Array*

Returns an array of all TCP options on the server that have a kind number matching the passed in value.

TCP.server.hasECNEcho: *Boolean*

Returns true if the ECN flag is set on the server during the three-way handshake.

TCP.server.initSeqNum: *Number*

The initial sequence sent from the server during the three-way handshake.

TCP.server.options: *Array*

An array of objects representing the TCP options of the server with a numerically higher IP address in the initial handshake packets. For more information, refer to TCP Options below.

TCP.wndSize1: *Number*

The size of the TCP sliding window of the device with a numerically lower IP address negotiated during the three-way handshake.

TCP.wndSize2: *Number*

The size of the TCP sliding window of the device with a numerically higher IP address negotiated during the three-way handshake.

TCP options

All TCP Options objects have the following properties:

kind: *Number*

The TCP option kind number.

Kind No.	Meaning
0	End of Option List
1	No-Operation
2	Maximum Segment Size
3	Window Scale

Kind No.	Meaning
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)
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)

Kind No.	Meaning
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 using 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);
    }
}
```

TCP_PAYLOAD

Fires when the payload matches the criteria configured in the associated trigger.

UDP_PAYLOAD

Fires when the payload matches the criteria configured in the associated trigger.

Deprecated

Flow.isClientAborted: *Boolean*

Deprecated. Use `Flow.client.isAborted` instead.

Flow.isServerAborted: *Boolean*

Deprecated. Use `Flow.server.isAborted` instead.

Flow.turnInfo: *String*

Deprecated. Use the top-level Turn object with attributes for the turn. See [Turn](#).

See also

- [Example: CIFS trigger](#)
- [Example: Customer ID header](#)
- [Example: Database trigger](#)
- [Example: Parse custom POS messages with universal payload analysis](#)
- [Example: Parse syslog over TCP with universal payload analysis](#)
- [Example: SOAP request](#)

FTP

The FTP class allows retrieval of metrics available during the `FTP_REQUEST` and `FTP_RESPONSE` events.

Events

FTP_REQUEST

Fires on every FTP request processed by the device.

FTP_RESPONSE

Fires on every FTP response processed by the device.

Method

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Applies only to `FTP_RESPONSE`. See the `record` property below for details.

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

Properties

args: String

The arguments to the command.

Applies only to `FTP_RESPONSE`.

cwd: String

String (`FTP_RESPONSE` only) 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.

Applies only to `FTP_RESPONSE`.

error: string

The detailed error message recorded by the ExtraHop system.

Applies only to `FTP_RESPONSE`.

isReqAborted: Boolean

Returns true if the connection is closed before the FTP request was complete.

isRspAborted: Boolean

Returns true if the connection is closed before the FTP response was complete.

Applies only to `FTP_RESPONSE`.

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.

Applies only to `FTP_RESPONSE`.

record: Object

Returns an object with all properties appropriately initialized.

Applies only to `FTP_RESPONSE`. The following `FTP_RESPONSE` properties are available:

- `args`

- `cwd`
- `error`
- `isReqAborted`
- `isRspAborted`
- `method`
- `path`
- `reqBytes`
- `reqL2Bytes`
- `reqPkts`
- `reqRTO`
- `roundTripTime`
- `rspBytes`
- `rspL2Bytes`
- `rspPkts`
- `rspRTO`
- `statusCode`
- `processingTime`
- `user`

reqBytes: *Number*

The number of L4 request bytes.

Applies only to `FTP_RESPONSE`.

reqL2Bytes: *Number*

The number of L2 request bytes.

Applies only to `FTP_RESPONSE`.

reqPkts: *Number*

The number of request packets.

Applies only to `FTP_RESPONSE`.

reqRTO: *Number*

The number of request RTOs.

Applies only to `FTP_RESPONSE`.

roundTripTime: *Number*

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

Applies only to `FTP_RESPONSE`.

rspBytes: *Number*

The number of L4 response bytes.

Applies only to `FTP_RESPONSE`.

rspL2Bytes: *Number*

The number of L2 response bytes.

Applies only to `FTP_RESPONSE`.

rspPkts: *Number*

The number of response packets.

Applies only to `FTP_RESPONSE`.

rspRTO: *Number*

The number of response RTOs.

Applies only to `FTP_RESPONSE`.

statusCode: *Number*

The FTP status code of the response.

Applies only to `FTP_RESPONSE`.

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.

Code	Description
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.
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.

tprocess: *Number*

The server processing time, expressed in milliseconds (equivalent to `rspTimeToFirstPayload - reqTimeToLastByte`). Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `FTP_RESPONSE`.

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 allows retrieval of metrics available during the `HL7_REQUEST` and `HL7_RESPONSE` events.

Events

HL7_REQUEST

Fires on every HL7 request processed by the device.

HL7_RESPONSE

Fires on every HL7 response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Applies only to `HL7_RESPONSE`. See the `record` property below for details.

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

Properties

ackCode: String

The two character acknowledgment code.

Applies only to `HL7_RESPONSE`.

ackId: String

The identifier for the message being acknowledged.

Applies only to `HL7_RESPONSE`.

msgId: String

The unique identifier for this message.

msgType: String

The entire message type field, including the `msgId` subfield.

record: Object

Returns an object with all properties appropriately initialized.

Applies only to `HL7_RESPONSE`. The following `HL7_RESPONSE` properties are available:

- `ackCode`
- `ackId`
- `msgId`
- `msgType`
- `roundTripTime`
- `processingTime`
- `version`

roundTripTime: Number

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

Applies only to `HL7_RESPONSE`.

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.

tprocess: *Number*

The server processing time, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to HL7_RESPONSE.

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 allows retrieval of metrics available during the HTTP_REQUEST and HTTP_RESPONSE events.

Events

HTTP_REQUEST

Fires on every HTTP request processed by the device.

HTTP_RESPONSE

Fires on every HTTP response processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

Applies only to HTTP_RESPONSE. See the `record` property below for details.

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

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

Allows access to HTTP 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`. Refer to [Example: HTTP header object](#) for more information.

parseQuery(String): *Object*

Function that accepts a query string and returns an object with names and values corresponding to those in the query string. 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. Returns a valid value on malformed and aborted requests. Returns NaN on expired requests and responses, or if the timing is invalid.

contentType: *String*

The value in the HTTP content-type header.

cookies: *Array*

An array of objects representing cookies, containing properties corresponding to the content of each cookie. For 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. 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 = 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. 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 `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*

Returns true if the protocol parser became desynchronized due to missing packets.

isEncrypted: *Boolean*

Returns true if the transaction is over secure HTTP.

isPipelined: *Boolean*

Returns true if the request is pipelined.

isReqAborted: *Boolean*

Returns true if the connection is closed before the HTTP request was complete.

isRspAborted: *Boolean*

Returns true if the connection is closed before the HTTP response was complete.

Applies only to `HTTP_RESPONSE`.

isRspChunked: *Boolean*

Returns true if the response is chunked.

Applies only to `HTTP_RESPONSE`.

isRspCompressed: *Boolean*

Returns true if the response is compressed.

method: *String*

The HTTP method such as POST and GET.

origin: *IPAddress | String*

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

path: *String*

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

payload: *Buffer*

The `n` first bytes of HTTP request or response payload (data past the headers), where `n` is the number specified in the trigger. When configuring the trigger in the Web UI, select the `HTTP_RESPONSE` or `HTTP_REQUEST` event, click **Show advanced options**, and enter the number of payload bytes to buffer. If the payload was compressed, the decompressed content is returned.

Example of how to use 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) {
        Flow.store.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.

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.

Applies only to `HTTP_REQUEST`.

record: *Object*

Returns an object with all properties appropriately initialized.

Applies only to `HTTP_RESPONSE`. The following `HTTP_RESPONSE` properties are available:

- `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
- statusCode
- thinkTime
- title
- processingTime
- uri
- userAgent

referer: *String*

The value in the HTTP referrer header.

reqBytes: *Number*

The number of L4 request bytes.

Applies only to HTTP_RESPONSE.

reqL2Bytes: *Number*

The number of request L2 bytes.

Applies only to HTTP_RESPONSE.

reqPkts: *Number*

The number of request packets.

Applies only to HTTP_RESPONSE.

reqRTO: *Number*

The number of request RTOs.

Applies only to HTTP_RESPONSE.

reqSize: *Number*

The size of the request payload, expressed in bytes. Does not count the HTTP header and only counts the number of bytes in the body of the request.

reqTimeToLastByte: *Number*

Time from the first byte of the request until the last byte of the request, expressed in milliseconds. Will return NaN on expired requests and responses, or if the timing is not valid.

roundTripTime: *Number*

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

Applies only to HTTP_RESPONSE.

rspBytes: *Number*

The number of L4 response bytes.

Applies only to `HTTP_RESPONSE`.

rspL2Bytes: *Number*

The number of response L2 bytes.

Applies only to `HTTP_RESPONSE`.

rspPkts: *Number*

The number of response packets.

Applies only to `HTTP_RESPONSE`.

rspRTO: *Number*

The number of response RTOs.

Applies only to `HTTP_RESPONSE`.

rspSize: *Number*

The size of the response payload, expressed in bytes. Does not count the HTTP header and only counts the number of bytes in the body of the response.

Applies only to `HTTP_RESPONSE`.

rspTimeToFirstHeader: *Number*

The time from the first byte of the request until the status line that precedes the response headers, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `HTTP_RESPONSE`.

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. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `HTTP_RESPONSE`.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `HTTP_RESPONSE`.

rspVersion: *String*


The HTTP version.

Applies only to `HTTP_RESPONSE`.

statusCode: *Number*

The HTTP status code of the response.

Applies only to `HTTP_RESPONSE`.

 **Note:** A status code of 0 is returned when there is not a valid `HTTP_RESPONSE` returned.

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. Will return NaN if there is no valid measurement.

tprocess: *Number*

The server processing time, expressed in milliseconds (equivalent to `rspTimeToFirstPayload - reqTimeToLastByte`). Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `HTTP_RESPONSE`.

uri: *String*

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

userAgent: *String*

The value in the HTTP user-agent header.

Applies only to `HTTP_REQUEST`.

Deprecated**payloadText: *String***

Deprecated. Use `payload` instead.

See also

- [Example: Customer ID header](#)
- [Example: SOAP request](#)
- [Example: HTTP header object](#)
- [Example: Session table](#)
- [Example: Trigger-based application definition](#)

IBMMQ

The IBMMQ class allows retrieval of metrics available during the `IBMMQ_REQUEST` and `IBMMQ_RESPONSE` events.



Note: The IBMMQ protocol supports EBCDIC encoding.

Events**IBMMQ_REQUEST**

Fires on every IBMMQ request processed by the device.

IBMMQ_RESPONSE

Fires on every IBMMQ response processed by the device.

Methods**commitRecord(): void**

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `IBMMQ_REQUEST` and `IBMMQ_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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 corresponding to the error code on the wire.

messageld: *String*

The IBMMQ message ID.

method: *String*

The wire protocol request/response method name.



Note: Wireshark users may notice some method names used by ExtraHop differ from those used by Wireshark. These are:

ExtraHop	Wireshark
ASYNC_MSG_V7	ASYNC_MESSAGE
MQCLOSEv7	SOCKET_ACTION
MQGETv7	REQUEST_MSGS
MQGETv7_REPLY	NOTIFICATION

msgFormat: *String*

The message format.

msgSize: *Number*

The size of the IBMMQ message, expressed in bytes.

payload: *Buffer*

For MQPUT, MQPUT1, MQGET_REPLY, ASYNC_MSG_V7, and MESSAGE_DATA messages, the payload is set to an instance of the Buffer class and could be converted to string using `toString()` or formatted using unpack commands. Large queue messages (those greater than approximately 32K) may be broken into more than one segment and, in those cases, a trigger will fire for each segment but only the first segment will have a non-null payload.

For all other messages, it is null.

pcfError: *String*

The error string corresponding to the error code on the wire for the PCF channel.

pcfMethod: *String*

The wire protocol request/response method name for the PCF channel.

pcfWarning: *String*

The warning string corresponding to the warning string on the wire for the PCF channel.

queue: *String*

String The local queue name or null if there was no MQOPEN, MQOPEN_REPLY, MQSP1 (Open), or MQSP1_REPLY seen.

queueMgr: *String*

The local queue manager or null if there was no INITIAL_DATA seen at the start of the connection.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

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

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.

resolvedQueue: *String*

The resolved queue name from MQGET_REPLY, MQPUT_REPLY, or MQPUT1_REPLY. For a remote queue, this will be different from IBMMQ.queue.

resolvedQueueMgr: *String*

The resolved queue manager from MQGET_REPLY, MQPUT_REPLY, or MQPUT1_REPLY. For a remote queue, this will be different from IBMMQ.queueMgr.

rfh: *Array of Strings*

The strings found in the optional RFH header. If there is no RFH header or no strings, the array will be empty.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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 request packets.

rspRTO: *Number*

The number of response RTOs.

warning: *String*

The warning string corresponding to the warning string on the wire.

Deprecated

objectHandle: *String*

Deprecated. Do not use this property.

ICA

The ICA class allows retrieval of metrics available during the `ICA_OPEN`, `ICA_AUTH`, `ICA_TICK`, and `ICA_CLOSE` events.

Events

ICA_AUTH

Fires when the ICA authentication is complete.

ICA_CLOSE

Fires when the ICA session is torn down.

ICA_OPEN

Fires right after the ICA application first loads.

ICA_TICK

Runs periodically while the user interacts with the ICA application.

After the `ICA_LOAD` event has run at least once, the `ICA_TICK` event is run anytime network or client latency is reported.

Network latency is reported when a specific ICA packet from the client contains latency data. The latency in that packet is returned by the `networkLatency` property.

Client latency is reported when a packet from the client on the EUEM channel reports the result of a single ICA round-trip measurement. The value of that measurement is returned by the `clientLatency` property.



Note: The `clientLatency` property is only available through the Trigger API. It is not supported in the UI.

Methods

commitRecord(): *void*

Applies only to `ICA_CLOSE`, `ICA_OPEN`, and `ICA_TICK`.

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `ICA_OPEN`, `ICA_TICK`, and `ICA_CLOSE`. See the `record` property below for details.

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

Properties

application: *String*

The name of the application that is being launched.

authDomain: *String*

The Windows authentication domain to which the user belongs.

channels: *Array*

An array of objects containing information about virtual channels seen since the last tick event. Each object has the following properties:

name: *String*

The name of the virtual channel.

description: *String*

The friendly description of the channel name.

clientBytes: *Integer*

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

serverBytes: *Integer*

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

Applies only to `ICA_TICK`.

client: *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, returns 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.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

clientCGPMsgCount: *Number*

The number of client CGP messages since the last tick event.

Applies only to `ICA_TICK`.

clientLatency: *Number*

The network latency, expressed in milliseconds, as reported by EUEM beacon.



Note: The UI may see some network latency even if the EUEM beacon is disabled.

Applies only to `ICA_TICK` events.

clientL2Bytes: *Number*

Upon an `ICA_CLOSE` event, returns 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.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

clientMsgCount: *Number*

The number of client messages since the last tick event.

Applies only to `ICA_TICK`.

clientPkts: *Number*

Upon an `ICA_CLOSE` event, returns the incremental number of client packets observed since the last `ICA_TICK` event. Does not specify the total number of packets for the session.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

clientRTO: *Number*

Upon an `ICA_CLOSE` event, returns 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.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

clientType: *String*

The user-agent equivalent to ICA. This is the type of the ICA client.

frameCutDuration: *Number*

Frame cut duration, as reported by EUEM beacon.

Applies only to `ICA_TICK`.

frameSendDuration: *Number*

The frame send duration, as reported by EUEM beacon.

Applies only to `ICA_TICK`.

host: *String*

The host name of the Citrix server.

isAborted: *Boolean*

Returns true if the application failed to launch successfully.

Applies only to `ICA_CLOSE`.

isCleanShutdown: *Boolean*

Returns true if the application shut down cleanly.

Applies only to `ICA_CLOSE`.

isEncrypted: *Boolean*

Returns true if the application is encrypted using RC5 encryption

isSharedSession: *Boolean*


Returns true if the application is launched over an existing connection.

launchParams: *String*

Returns a 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. Use `ICA.isSharedSession` to distinguish between initial and subsequent application loads.

loginTime: *Number*

The user login time, expressed in milliseconds.

Applies only to `ICA_OPEN`, `ICA_CLOSE`, and `ICA_TICK`.

 **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. Use `ICA.isSharedSession` to distinguish between initial and subsequent application loads.

networkLatency: *Number*

Returns the current client-advertised latency, expressed in milliseconds.

Applies only to `ICA_TICK` events.

record: Object

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

Applies only to ICA_OPEN, ICA_CLOSE, and ICA_TICK events. The following table shows the properties available for the record of each event.

ICA_CLOSE	ICA_OPEN	ICA_TICK
application	application	application
authDomain	authDomain	authDomain
client	client	client
clientBytes	clientType	clientBytes
clientL2Bytes	host	clientCGPMsgCount
clientPkts	isEncrypted	clientL2Bytes
clientRTO	isSharedSession	clientLatency
clientType	launchParams	clientMsgCount
host	loadTime	clientPkts
isAborted	loginTime	clientRTO
isCleanShutdown	user	clientType
isEncrypted		frameCutDuration
isSharedSession		frameSendDuration
launchParams		host
loadTime		isEncrypted
loginTime		isSharedSession
roundTripTime		launchParams
serverBytes		loadTime
serverL2Bytes		loginTime
serverPkts		networkLatency
serverRTO		roundTripTime
user		serverBytes
		serverCGPMsgCount
		serverL2Bytes
		serverMsgCount
		serverPkts
		serverRTO
		user

roundTripTime: *Number*

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

Applies only to `ICA_CLOSE` and `ICA_TICK`.

serverBytes: *Number*

Upon an `ICA_CLOSE` event, returns 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.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

serverCGPMsgCount: *Number*

The number of CGP server messages since the last tick event.

Applies only to `ICA_TICK`.

serverL2Bytes: *Number*

Upon an `ICA_CLOSE` event, returns 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.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

serverMsgCount: *Number*

The number of server messages since the last tick event.

Applies only to `ICA_TICK`.

serverPkts: *Number*

Upon an `ICA_CLOSE` event, returns the incremental number of server packets observed since the last `ICA_TICK` event. Does not specify the total number of packets for the session.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

serverRTO: *Number*

Upon an `ICA_CLOSE` event, returns 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.

Applies only to `ICA_CLOSE` and `ICA_TICK` events.

user: *String*

The name of the user, if available.

Deprecated

authTicket: *String*

Use `user` instead.

ICMP

The ICMP class allows retrieval of metrics available during the `ICMP_MESSAGE` event.

Events

ICMP_MESSAGE

Fires on every ICMP message processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance. See the `record` property below for details.

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

Properties

gwAddr: *IPAddress*

For a redirect message, 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*

Returns true for messages 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*

Returns true for messages 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
Home Agent Address Discovery Solicitation	n/a	144
Mobile Prefix Solicitation	n/a	146
Certification Path Solicitation	n/a	148

isReply: *Boolean*

Returns 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 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. Returns null for all other message types.

The following table displays v4 and v6 type IDs for the ICMP messages:

Message	ICMPv4 Type	ICMPv6 Type
Echo Request	8	128
Echo Reply	0	129
Timestamp Request	13	n/a

Message	ICMPv4 Type	ICMPv6 Type
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 shows the available ICMPv4 message types:

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)
31	Datagram Conversion Error (deprecated)
32	Mobile Host Redirect (deprecated)
33	Where Are You (deprecated)

Type	Message
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 shows the available ICMPv6 message types:

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
141	Inverse Neighbor Discovery Solicitation Message
142	Inverse Neighbor Discovery Advertisement Message
143	Multicast Listener Discovery (MLDv2) reports

Type	Message
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

For an ICMPv4 Destination Unreachable or an ICMPv6 Packet Too Big message, the maximum transmission unit of the next-hop link. Returns null for all other messages.

Message	ICMPv4 Type	ICMPv6 Type
Destination Unreachable	3	n/a
Packet Too Big	n/a	2

pointer: Number

For a Parameter Problem message, the octet of the original datagram's header where the error was detected. Returns null for all other messages.

Message	ICMPv4 Type	ICMPv6 Type
Parameter Problem	12	4

record: Object

Returns an object with all properties appropriately initialized.

The following `ICMP_MESSAGE` properties are available:

- 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. Null is returned for all other messages.

version: *Number*

The ICMP version. Can be either 4 or 6.

IPAddress

The IPAddress class allows setting and retrieval of IP address attributes. IPAddress is the type of IP address properties available on 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(): *Boolean*

Performs an equality test between IPAddress objects

mask(mask: *Number*): *IPAddress*

Sets the subnet mask of the IPAddress object. Takes one parameter:

mask: *Number*

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*

Returns true if the IP address is a broadcast address.

isLinkLocal: *Boolean*

Returns true if the IP address is a link local address (169.254.0.0/16).

isMulticast: *Boolean*

Returns true if the IP address is a multicast address.

isRFC1918: *Boolean*

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

isV4: Boolean

Returns true if the IP address is an IPv4 address.

isV6: Boolean

Returns true if the IP address is an IPv6 address.

Kerberos

The Kerberos class allows the retrieval of metrics available during the `KERBEROS_REQUEST` and `KERBEROS_RESPONSE` events.

Events

KERBEROS_REQUEST

Fires on every Kerberos AS-REQ and TGS-REQ message type processed by the device.

KERBEROS_RESPONSE

Fires on every Kerberos AS-REP and TGS-REP message type processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of `KERBEROS_REQUEST` and `KERBEROS_RESPONSE`. See the `record` property below for details.

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

Properties

addresses: Array of Objects

The addresses from which the requested ticket is valid.

Applies only to `KERBEROS_REQUEST`.

cNames: Array of Strings

The name portions of the principal identifier.

cNameType: String

The type for the `cNames` field.

cRealm: String

The client's realm.

error: String

The error returned.

Applies only to `KERBEROS_RESPONSE`.

eType: Array of Numbers

An array of the preferred encryption methods.

Applies only to `KERBEROS_REQUEST`.

from: String

In `AS_REQ` and `TGS_REQ` message types, the time when the requested ticket is to be postdated to.

Applies only to `KERBEROS_REQUEST`.

kdcOptions: Object

An object containing booleans for each option flag in AS_REQ and TGS_REQ messages.

Applies only to KERBEROS_REQUEST.

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.

Applies only to KERBEROS_RESPONSE.

realm: String

The server's realm. In an AS_REQ message type, this is the client's realm.

record: Object

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

KERBEROS_REQUEST	KERBEROS_RESPONSE
cNames	cNames
cNameType	cNameType
cRealm	cRealm
eType	error
from	msgType
msgType	processingTime
realm	realm
reqBytes	roundTripTime
reqL2Bytes	rspBytes

KERBEROS_REQUEST	KERBEROS_RESPONSE
reqPkts	rspL2Bytes
reqRTO	rspPkts
sNames	rspRTO
sNameType	sNames
till	sNameType

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.

Applies only to KERBEROS_REQUEST.

till: String

The expiration date requested by the client in a ticket request.

Applies only to KERBEROS_REQUEST.

LDAP

The LDAP class allows retrieval of metrics available during the LDAP_REQUEST and LDAP_RESPONSE events.

Events

LDAP_REQUEST

Fires on every LDAP request processed by the device.

LDAP_RESPONSE

Fires on every LDAP response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Different properties are returned for the record of LDAP_REQUEST and LDAP_RESPONSE. See the record property below for details.

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

Properties

bindDN: String

The bind DN of the LDAP request.

Applies only to LDAP_REQUEST.

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).

Applies only to LDAP_RESPONSE.

Result Code	Result String
1	operationsError
2	protocolError
3	timeLimitExceeded
4	sizeLimitExceeded
7	authMethodNotSupported
8	strongerAuthRequired
11	adminLimitExceeded
12	unavailableCriticalExtension
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

Result Code	Result String
69	objectClassModsProhibited
71	affectsMultipleDSAs
80	other

errorDetail: *String*

The LDAP error detail, when available for that error type (e.g., protocolError : historical protocol version requested, use LDAPv3 instead).

Applies only to LDAP_RESPONSE.

method: *String*

The LDAP method.

msgSize: *Number*

The size of the LDAP message, expressed in bytes.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

LDAP_REQUEST	LDAP_RESPONSE
bindDN	dn
dn	error
method	errorDetail
msgSize	method
reqBytes	msgSize
reqL2Bytes	roundTripTime
reqPkts	rspBytes
reqRTO	rspL2Bytes
saslMechanism	rspPkts
searchFilter	rspRTO
searchScope	saslMechanism
	processingTime

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.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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.

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.

Applies only to LDAP_REQUEST.

searchFilter: *String*

The mechanism to allow certain entries in the subtree and exclude others.

Applies only to LDAP_REQUEST.

searchScope: *String*

The depth of a search within the search base.

Applies only to LDAP_REQUEST.

tprocess: *Number*

The server processing time, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid or is not available. Available for the following:

- BindRequest
- SearchRequest
- ModifyRequest
- AddRequest
- DelRequest
- ModifyDNRequest
- CompareRequest
- ExtendedRequest

Applies only to LDAP_RESPONSE.

LLDP

The LLDP class allows retrieval of metrics available during the LLDP_FRAME events.

Events

LLDP_FRAME

Fires 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: *Integer*

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 allows retrieval of metrics available during the MEMCACHE_REQUEST and MEMCACHE_RESPONSE events.

Events

MEMCACHE_REQUEST

Fires on every memcache request processed by the device.

MEMCACHE_RESPONSE

Fires on every memcache response processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

The different properties are returned for the record of MEMCACHE_REQUEST and MEMCACHE_RESPONSE. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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.

Applies only to `MEMCACHE_RESPONSE`.

error: *String*

The detailed error message recorded by the ExtraHop system.

Applies only to `MEMCACHE_RESPONSE`.

hits: *Array*

An array of objects with the following properties:

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.

Applies only to `MEMCACHE_RESPONSE`.

isBinaryProtocol: *Boolean*

Returns true if the request/response corresponds to the binary version of the memcache protocol.

isNoReply: *Boolean*

Returns true if the request has the "noreply" keyword and therefore should never receive a response (text protocol only).

Applies only to `MEMCACHE_REQUEST`.

isRspImplicit: *Boolean*

Returns true if the response was implied by a subsequent response from the server (binary protocol only).

Applies only to `MEMCACHE_RESPONSE`.

method: *String*

The Memcache method (as recorded in ExtraHop metrics).

misses: *Array*

An array of objects with the following property:

key: *String* | *Null*

The Memcache key for which this was a miss, if available.

Applies only to `MEMCACHE_RESPONSE`.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

MEMCACHE_REQUEST	MEMCACHE_RESPONSE
isBinaryProtocol	accessTime
isNoReply	error
method	hits

MEMCACHE_REQUEST	MEMCACHE_RESPONSE
reqBytes	isBinaryProtocol
reqL2Bytes	isRspImplicit
reqPkts	method
reqRTO	misses
reqSize	roundTripTime
vbucket	rspBytes
	rspL2Bytes
	rspPkts
	rspRTO
	statusCode
	vbucket

reqBytes: *Number*

The number of application-level request bytes.

reqKeys: *Array*

An array containing the Memcache key strings sent with the request.

reqL2Bytes: *Number*

The number of request L2 bytes.

reqPkts: *Number*

The number of request packets.

reqRTO: *Number*

The number of request RTOs.

Applies only to MEMCACHE_REQUEST.

reqSize: *Number*

The size of the value in the request (if any), expressed in bytes. Note that methods such as set include values, while get and delete do not.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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.

Applies only to MEMCACHE_RESPONSE.

statusCode: *String*

The Memcache status code. For the binary protocol, ExtraHop 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 ExtraHop metrics.

Applies only to `MEMCACHE_RESPONSE`.

vbucket: *Number*

The Memcache vbucket, if available (binary protocol only).

See also

- [Example: Memcache hits and misses](#)
- [Example: Memcache key parsing](#)

MetricCycle

The `MetricCycle` class represents an interval where stats were published. It is valid on the following events:

- `METRIC_CYCLE_BEGIN`
- `METRIC_CYCLE_END`
- `METRIC_RECORD_COMMIT`

Events

METRIC_CYCLE_BEGIN

Fires when a metric interval begins.

METRIC_CYCLE_END

Fires when a metric interval ends.

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` in `capture`. `MetricCycle.store` is shared among triggers for `METRIC_*` events. It is cleared at the end of a metric cycle.

See also

- [Example: Use the Metric Cycle store](#)

MetricRecord

The `MetricRecord` class allows access to the current set of metrics in `METRIC_RECORD_COMMIT`.

Events

METRIC_RECORD_COMMIT

Fires when a metric record is committed to the datastore and provides access to various metric properties.

Properties

fields: *Object*

An object containing metric values. The properties are the field names and the values can be numbers, Topset, 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.

See also

- [Example: Custom trouble group](#)
- [Example: Topset key matching](#)
- [Example: Use the Metric Cycle store](#)

MongoDB

The MongoDB class allows retrieval of metrics available during the `MONGODB_REQUEST` and `MONGODB_RESPONSE` events.

Events

MONGODB_REQUEST

Fires on every MongoDB request processed by the device.

MONGODB_RESPONSE

Fires on every MongoDB response processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

The different properties are returned for the record of `MONGODB_REQUEST` and `MONGODB_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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.

Applies only to MONGODB_RESPONSE.

isReqAborted: *Boolean*

Returns true if the connection is closed before the MongoDB request was complete.

isReqTruncated: *Boolean*

Returns true if the request document(s) size is greater than the maximum payload document size.

isRspAborted: *Boolean*

Returns true if the connection is closed before the MongoDB response was complete.

Applies only to MONGODB_RESPONSE.

method: *String*

The MongoDB database method (appears under **Methods** in the user interface).

opcode: *String*

The MongoDB operational code on the wire protocol, which may differ from the MongoDB method used.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

MONGODB_REQUEST	MONGODB_RESPONSE
collection	collection
database	database
isReqAborted	error
isReqTruncated	isRspAborted
method	method
opcode	opcode
reqBytes	roundTripTime
reqL2Bytes	rspBytes
reqPkts	rspL2Bytes
reqRTO	rspPkts
reqSize	rspRTO
reqTimeToLastByte	rspSize
user	rspTimeToFirstByte
	rspTimeToLastByte
	processingTime
	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 record at L7, expressed in bytes.

reqTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the request, expressed in milliseconds.

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.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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 record at L7, expressed in bytes.

Applies only to `MONGODB_RESPONSE`.

rspTimeToFirstByte: *Number*

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `MONGODB_RESPONSE`.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last by of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `MONGODB_RESPONSE`.

tprocess: *Number*

The time to process the request, expressed in milliseconds (equivalent to `rspTimeToFirstByte - reqTimeToLastByte`). Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `MONGODB_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 allows for retrieval of metrics available during the `MSMQ_MESSAGE` event.

Events

MSMQ_MESSAGE

Fires on every MSMQ user message processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop appliance. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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*

Returns true if the payload is encrypted.

label: *String*

The label or description of the message.

msgClass: *String*

The message class of the message. Possible values are:

- `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.

record: *Object*

Returns an object with all properties appropriately initialized.

The following `MSMQ_MESSAGE` properties are available:

- adminQueue
- dstQueueMgr
- isEncrypted
- label
- msgClass
- msgId
- priority
- queue
- receiverBytes
- receiverL2Bytes
- receiverPkts
- receiverRTO
- responseQueue
- roundTripTime
- senderBytes
- senderL2Bytes
- senderPkts
- senderRTO

- `srcQueueMgr`

responseQueue: *String*

The name of the response queue of the message.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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.

srcQueueMgr: *String*

The source message broker of the message.


Network

The network class allows adding custom metrics at the global level.

Use the following functions to record custom metrics associated with networks. Refer to [ExtraHop data types](#) for an overview of the data types.

- `metricAddCount(metric_name:String, count:Number, [highPrecision:bool]):void`
- `metricAddDataset(metric_name:String, val:Number, [freq:Number], [highPrecision:bool]):void`
- `metricAddDetailCount(metric_name:String, key:String | IPAddress, count:Number, [highPrecision:bool]):void`
- `metricAddDetailSnap(metric_name:String, key:String | IPAddress, count:Number, [highPrecision:bool]):void`
- `metricAddDetailDataset(metric_name:String, key:String | IPAddress, val:Number, [freq:Number], [highPrecision:bool]):void`
- `metricAddDetailMax(metric_name:String, key:String | IPAddress, val:Number, [highPrecision:bool]):void`
- `metricAddDetailSampleSet(metric_name:String, key:String | IPAddress, val:Number, [highPrecision:bool]):void`
- `metricAddMax(metric_name:String, val:Number, [highPrecision:bool]):void`
- `metricAddSampleSet(metric_name:String, val:Number, [highPrecision:bool]):void`
- `metricAddSnap(metric_name:String, count:Number, [highPrecision:bool]):void`

The optional `highPrecision` flag will enable one second granularity for the metrics when set to true.

-  **Note:**
- Freq is the number of occurrences of the value being passed in. If the value is not passed in, the value of freq is 1. The freq argument is useful in cases when you want to simultaneously record multiply occurrences of particular values in a dataset.
 - When NaN is passed to a `metricAdd*` function, it is silently discarded.
 - All count parameters for `metricAdd*` functions only accept a non-zero, positive integer between 1 and 2^{64} .

See also

- [Example: Database trigger](#)
- [Example: Parse syslog over TCP with universal payload analysis](#)
- [Example: Session table](#)
- [Example: SOAP request](#)

NFS

The NFS class allows retrieval of metrics available during the `NFS_REQUEST` and `NFS_RESPONSE` events.

Events

NFS_REQUEST

Fires on every NFS request processed by the device.

NFS_RESPONSE

Fires on every NFS response processed by the device

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Applies only to `NFS_RESPONSE`. See the `record` property below for details.

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

Propertis

accessTime: Number

The time it took for 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. Will return NaN on malformed and aborted responses, or if the timing is not valid or is not applicable.

Applies only to `NFS_RESPONSE`.

authMethod: String

The method for authenticating users.

error: String

The detailed error message recorded by the ExtraHop system.

Applies only to `NFS_RESPONSE`.

fileHandle: Buffer

The file handle returned by the server on LOOKUP, CREATE, SYMLINK, MKNOD, LINK, or REaddirPLUS operations.

isCommandFileInfo: Boolean

Returns true if the command is a file info.

isCommandRead: Boolean

Returns true if the command is a read.

isCommandWrite: Boolean

Returns true if the command is a write.

method: String

The NFS method (appears under Methods in the UI).

offset: *Number*

The file offset associated with READ and WRITE NFS commands.

Applies only to `NFS_REQUEST`.

record: *Object*

Returns an object with all properties appropriately initialized.

Applies only to `NFS_RESPONSE`. The following `NFS_RESPONSE` properties are available:

- `accessTime`
- `authMethod`
- `error`
- `isCommandFileInfo`
- `isCommandRead`
- `isCommandWrite`
- `method`
- `offset`
- `renameDirChanged`
- `reqBytes`
- `reqL2Bytes`
- `reqPkts`
- `reqRTO`
- `reqSize`
- `resource`
- `roundTripTime`
- `rspBytes`
- `rspL2Bytes`
- `rspPkts`
- `rspRTO`
- `rspSize`
- `statusCode`
- `user`
- `version`

renameDirChanged: *Boolean*

Returns true if a resource rename request includes a directory move.

Applies only to `NFS_REQUEST`.

reqBytes: *Number*

The number of L4 request bytes.

Applies only to `NFS_RESPONSE`.

reqL2Bytes: *Number*

The number of L2 request bytes.

Applies only to `NFS_RESPONSE`.

reqPkts: *Number*

The number of request packets.

Applies only to `NFS_RESPONSE`.

reqRTO: *Number*

The number of request RTOs.

Applies only to `NFS_REQUEST`.

reqSize: *Number*

The size of the request record at L7, expressed in bytes.

resource: *String*

The path and filename, concatenated together.

roundTripTime: *Number*

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

Applies only to NFS_RESPONSE.

rspBytes: *Number*

The number of L4 response bytes.

Applies only to NFS_RESPONSE.

rspL2Bytes: *Number*

The number of L2 response bytes.

Applies only to NFS_RESPONSE.

rspPkts: *Number*

The number of response packets.

Applies only to NFS_RESPONSE.

rspRTO: *Number*

Number of response RTOs.

Applies only to NFS_RESPONSE.

rspSize: *Number*

The size of the response record at L7, expressed in bytes.

Applies only to NFS_RESPONSE.

statusCode: *String*

The NFS status code of the request or response.

user: *String*

The Linux user ID on the system. Uses the format `uid:xxxx@ip_address`.

version: *Number*

The NFS version.

POP3

The POP3 class enables you to retrieve 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 the record to the ExtraHop Explore appliance.

Properties are returned for the record only on `POP3_RESPONSE` events. See the `record` property below for details.

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

Properties

dataSize: *Number*

Returns the size of the message, expressed in bytes.

Applies only to `POP3_RESPONSE` events.

error: *String*

Returns the detailed error message recorded by the ExtraHop system.

Applies only to `POP3_RESPONSE` events.

isEncrypted: *Boolean*

Returns `TRUE` if the transaction is over a secure POP3 server.

isReqAborted: *Boolean*

Returns `TRUE` if the connection is closed before the POP3 request was complete.

isRspAborted: *Boolean*

Returns `TRUE` if the connection is closed before the POP3 response was complete.

Applies only to `POP3_RESPONSE` events.

method: *String*

Returns the POP3 method such as `RETR` or `DELE`.

processingTime: *Number*

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

Applies only to `POP3_RESPONSE` events.

recipientList: *Array*

Returns an array that contains a list of recipient addresses.

record: *Object*

Returns an object with all record properties appropriately initialized. The content of the object is determined by the event called. This property applies only to `POP3_RESPONSE` events. The following record object properties are available on `POP3_RESPONSE` events:

- `dataSize`
- `error`
- `isEncrypted`
- `isReqAborted`
- `isRspAborted`
- `method`
- `processingTime`
- `recipientList`
- `reqSize`
- `reqTimeToLastByte`
- `rspSize`
- `rspTimeToFirstByte`
- `rspTimeToLastByte`
- `sender`
- `statusCode`

reqBytes: Number

Returns the number of L4 request bytes.

reqL2Bytes: Number

Returns the number of L2 request bytes.

reqPkts: Number

Returns the number of request packets.

reqRTO: Number

Returns the number of request retransmission timeouts (RTOs).

reqSize: Number

Returns the size of the request payload, expressed in bytes. Returns only the number of bytes in the body of the response; the header is not factored into the size.

reqTimeToLastByte: Number

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

roundTripTime: Number

Returns the median TCP round-trip time (RTT), expressed in milliseconds. Will return NaN if there are no RTT samples.

rspBytes: Number

Returns the number of L4 response bytes.

rspL2Bytes: Number

Returns the number of response L2 bytes.

rspPkts: Number

The number of response packets.

rspRTO: Number

Returns the number of response retransmission timeouts (RTOs).

rspSize: Number

Returns the size of the response payload, expressed in bytes. Returns only the number of bytes in the body of the response; the header is not factored into the size.

Applies only to `POP3_RESPONSE` events.

rspTimeToFirstByte: Number

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

Applies only to `POP3_RESPONSE` events.

rspTimeToLastByte: Number

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

Applies only to `POP3_RESPONSE` events.

sender: String

Returns the address of the sender of the message.

status: String

Returns the POP3 status message of the response. Returns NaN if the response is invalid.

Applies only to `POP3_RESPONSE` events.

Record

A record is a JSON object used to send information to the ExtraHop Explore appliance.

Built-in protocols provide access to a default record on events that represent the completion of transactions (for example, `HTTP.record` on `HTTP_RESPONSE`) or on partial transaction updates. (`Flow.record` on `FLOW_RECORD`). Not all events will result in a record.

Properties

type: *String*

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

fields: *Object*

One or more key-value pairs.

Examples

For built-in protocols, there is an accessor to retrieve a prepopulated record. This will return an object with all the fields initialized appropriately for the protocol and trigger event.

```
record = HTTP.record;
```

A record is a javascript object.

```
record = {
  field1: 'myfield1',
  field2: 'myfield2'
};
```

To commit a custom record, use `commitRecord`.

```
commitRecord('custom_record', record);
```

A built-in record can be the basis for a custom record.

```
record = HTTP.record;
record.session = HTTP.headers["X-Session-Id"]
commitRecord('custom_record', record);
```

You can also export a record using one of the Remote classes like `Remote.MongoDB`.

```
Remote.MongoDB.insert('collection', record); // record.toJSON()
```

Remote.HTTP

The `Remote.HTTP` class allows submission of HTTP requests to an HTTP Open Data Stream Configuration previously configured in the ExtraHop Admin UI. This includes the ability to use HTTP REST APIs. Consult with your ExtraHop appliance administrator for the possible values to use for HTTP Data Stream Configuration names.



Note: To use `Remote.HTTP`, you must have previously configured an HTTP data stream in the Open Data Streams pane of the ExtraHop Admin UI.

Methods

request

Submits an HTTP REST request to an HTTP Data Stream Configuration previously configured in the ExtraHop Admin UI.

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 specifying the HTTP method to be used.

- 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.

It is possible to 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) } );
```

See also

- [Example: Send data to Elasticsearch with Remote.HTTP](#)
- [Example: Send information to Azure table service with Remote.HTTP](#)

Remote.Kafka

The Remote.Kafka class allows the submission of messages to a Kafka server.



Note: To use Remote.Kafka, you must have previously configured a Kafka data stream in the Open Data Streams pane of the ExtraHop Admin UI.

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.

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 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 for the target in the Open Data Stream configuration 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.

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 allows insertion, removal, and updating of documents in collections in MongoDB.



Note: To use Remote.MongoDB, you must have previously configured a MongoDB data stream in the Open Data Streams pane of the ExtraHop Admin UI.

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 as it appears in the ExtraHop Open Data Streams UI. If no host is specified, the default host will be used.

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 used to limit 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 as it appears in the ExtraHop Open Data Streams UI. If no host is specified, the default host will be used.

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 as it appears in the ExtraHop Open Data Streams UI. If no host is specified, the default host will be used.

options:

Optional flags that indicate the following additional update options:

upsert: *Boolean*

An optional boolean parameter to create 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 to update 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:

Returns TRUE if the request is queued, otherwise returns FALSE.

Examples:

```
var x = Remote.MongoDB.update('test.tbc', { _id: 1 }, { $set:
  { 'foobar': 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.

See also

- [Example: Parse syslog over TCP with universal payload analysis](#)

Remote.Raw

The Remote.Raw class allows the submission of raw data to an Open Data Stream (ODS) target through a TCP or UDP port.



Note: To use Remote.Raw, you must have previously configured a Raw Data ODS in the Open Data Streams pane of the ExtraHop Admin UI. If the Gzip feature is enabled for the raw data stream in the ExtraHop Admin UI, Remote.Raw 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 `Remote.Raw`, 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 allows creation of remote syslog messages with specified content.



Note: To use `Remote.Syslog`, you must have previously configured a Syslog data stream in the Open Data Streams pane of the ExtraHop Admin UI.

Each of these methods sends a message to the configured remote syslog server with a severity corresponding to the method name using the "user" facility. You can specify the specific host using the 'name' field and using the name as it appears in the ExtraHop Open Data Streams UI. If no host is specified, the default host will be used.

- `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 + " tprocess=" + HTTP.tprocess);
```

To send an rsyslog message to the host 'name' 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 + " tprocess=" + HTTP.tprocess);
```

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.

To increase the default message size, go to the Admin UI, click **Running Config**, and then click **Edit**. Go to the "capture" section, and under "rsyslog", add "message_length_max". The "message_length_max" setting applies only to the message passed to the Remote.Syslog APIs, the message header does not count against the max. Sample configuration:

```
"remote": {
  "rsyslog": {
    "host": "splunkium",
    "port": 54322,
    "ipproto": "tcp",
    "message_length_max": 4000
  }
}
```

Timestamp

The timestamp format for rsyslog messages in versions 5.0.0 defaults to UTC but can be changed to local time with an offset in the Data Stream Configuration setup.

The timestamp format for rsyslog messages is expressed in UTC in versions before 4.1.24484 and after 4.1.24504. For example:

```
2015-07-08T23:42:35.075Z
```

For versions 4.1.24484 through 4.1.24504, the timestamp format for rsyslog messages is expressed in local time with a offset. For example:

```
2015-07-08T13:47:32.724-10:00
```

See also

- [Example: Device discovery notification](#)
- [Example: Parse syslog over TCP with universal payload analysis](#)
- [Example: Topnset key matching](#)

RemoteSyslog

Deprecated.

Use [Remote.Syslog](#) instead.

RTCP

The RTCP class allows for retrieval of metrics available during the `RTCP_MESSAGE` event.

Events

RTCP_MESSAGE

Fires on every RTCP UDP packet processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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_Stamp 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

Block Type	Name
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
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*

Returns an object with all properties appropriately initialized.

The following `RTCP_MESSAGE` properties are available:

- `callId`
- `cName`

RTP

The RTP class allows for retrieval of metrics available during the `RTP_OPEN`, `RTP_CLOSE`, and `RTP_TICK` events.

Events

RTP_CLOSE

Fires when an RTP connection is closed.

RTP_OPEN

Fires when a new RTP connection is opened.

RTP_TICK

Fires periodically on RTP flows.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

Applies only to `RTP_TICK`. See the `record` property below for details.

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

Properties

bytes: *Number*

The number of bytes sent.

Applies only to RTP_TICK.

callId: *String*

The call ID for associating with SIP flow.

drops: *Number*

The number of dropped packets detected.

Applies only to RTP_TICK.

dups: *Number*

The number of duplicate packets detected.

Applies only to RTP_TICK.

jitter: *Number*

An estimate of the statistical variance of the data packet interarrival time.

Applies only to RTP_TICK.

l2Bytes: *Number*

The number of L2 bytes.

Applies only to RTP_TICK.

mos: *Number*

The estimated mean opinion score for quality.

Applies only to RTP_TICK.

outOfOrder: *Number*

The number of out-of-order messages detected.

Applies only to RTP_TICK.

payloadType: *String*

The type of RTP payload.

Applies only to RTP_TICK.

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

payloadTypeld	payloadType
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

payloadTypeld: *Number*

The numeric value of the payload type. See table under `payloadType`.

Applies only to `RTP_TICK`.

pkts: *Number*

The number of packets sent.

Applies only to `RTP_TICK`.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

Applies only to `RTP_TICK`. The following `RTP_TICK` properties are available for the record of each event:

- bytes
- callId
- drops
- dups
- jitter
- l2Bytes
- mos
- outOfOrder
- payloadType
- payloadTypeld
- pkts
- rFactor
- ssrc
- version

rFactor: *Number*

The R factor quality metric.

Applies only to RTP_TICK.

ssrc: *Number*

The SSRC of sender.

version: *Number*

The RTP version number.

Sampleset

The Sampleset class is used to represent sampleset 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.

SDP

The SDP class allows for retrieval of Session Description Protocol (SDP) information during the SIP_REQUEST and SIP_RESPONSE events.

For more information, see [SIP](#).

Properties

mediaDescriptions: *Array*

An array of objects applicable to SIP_REQUEST and SIP_RESPONSE events 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 address. 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.

sessionDescription: Object

An object applicable to `SIP_ REQUEST` and `SIP_ RESPONSE` events 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.

timeDescriptions: Array

An array of objects applicable to `SIP_ REQUEST` and `SIP_ RESPONSE` events 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.

Session

The Session object is in-memory and global per ExtraHop appliance. It is designed to support coordination across multiple independently executing triggers. The ExtraHop Open Data Context API exposes the Session object via the management network, enabling coordination with external processes. The Session object's global state means any changes by a trigger or external process becomes visible to all other entities on the same ExtraHop appliance using the Session object. Because the Session object is in-memory, changes are not saved when you restart the ExtraHop appliance or the capture process.

 **Note:** Discover nodes that are part of a Command cluster do not share their global state. The Command appliance does not execute triggers; it only manages them.

Events

SESSION_EXPIRE

Fires periodically (in approximately 30 second increments) as long as the session table is in use. It does not fire once for every expired entry, and it does not fire immediately after an entry has expired.

Methods

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

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

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

Returns the Options object for the entry corresponding to the one passed in through `Session.add` or `Session.replace`.

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

Atomic lookup and increment. The default count value is 1. On success, the new value is returned. If lookup fails, `null` is returned. If the value is not a number, an exception is thrown.

lookup(key: *String*): *

Look up the given key in the session table and return the corresponding value. Returns `null` if the key is not present.

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

Modify 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, no new entry is created. If options are provided, the options of the entry are updated and old options are merged with new ones. If the "expire" option is provided, 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*]): *

Update 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 options are provided, the options of the entry are updated and old options are merged with new ones. If the "expire" option is provided, the expiration timer is reset.

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: *Integer*

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.

name: *String*

The key of the expired object.

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

The value of the entry in the session table.

Applies only to `SESSION_EXPIRE`.

Session table options are as follows:

expire: *Number*

The duration after which eviction will occur, expressed in seconds. If null or undefined, the entry will be evicted only when the session table grows too large. This is the default.

notify: *Boolean*

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

priority: *String*

A constant, `Session.PRIORITY_{LOW, NORMAL, HIGH}`, used to determine which entries to evict if the session table grows too large. The default value is `PRIORITY_NORMAL`.

The `SESSION_EXPIRE` event is not associated with any particular flow, so triggers on `SESSION_EXPIRE` events cannot commit device metrics in the usual way (e.g., `Device.metricAdd*` or `Flow.client.device.metricAdd*`). To commit device metrics on this event, you have to add any needed Device objects to the session table using the `Device()` constructor.

Deprecated

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

Deprecated. Use `Session.replace` instead.

See also

- [Example: Session table](#)

SIP

The SIP class allows retrieval of metrics available during the `SIP_REQUEST` and `SIP_RESPONSE` events.

Events

SIP_REQUEST

Fires on every SIP request processed by the device.

SIP_RESPONSE

Fires on every SIP response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

The different properties are returned for the record of `SIP_REQUEST` and `SIP_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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*

Returns 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.

Method Name	Description
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. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `SIP_RESPONSE`.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following table shows the properties available for the record of each event.

SIP_REQUEST	SIP_RESPONSE
callId	callId
from	from
hasSDP	hasSDP
method	processingTime
reqBytes	roundTripTime
reqL2Bytes	rspBytes
reqPkts	rspL2Bytes
reqRTO	rspPkts
reqSize	rspRTO
to	rspSize
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. Does not include headers.

Applies only to `SIP_REQUEST`.

roundTripTime: *Number*

The median round-trip time (RTT), expressed in milliseconds. Will return 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. Does not include headers.

Applies only to `SIP_RESPONSE`.

statusCode: *Number*

The SIP response status code.

Applies only to `SIP_RESPONSE`. 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
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

Number	Response
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
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 allows retrieval of metrics available during the `SMPP_REQUEST` and `SMPP_RESPONSE` events.



Note: The `mdn`, `shortcode`, and `error` properties may be null, depending on availability and relevance.

Events

SMPP_REQUEST

Fires on every SMPP request processed by the device.

SMPP_RESPONSE

Fires on every SMPP response processed by the device.

Methods

commitRecord(): void

Commits the record to the ExtraHop Explore appliance.

Applies only to `SMPP_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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`. Will be 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 of error will be null.

Applies only to `SMPP_RESPONSE`.

message: Buffer

The contents of the `short_message` field on `DELIVER_SM` and `SUBMIT_SM` messages. Will be null if unavailable or not applicable.

Applies only to `SMPP_REQUEST`.

record: Object

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

Applies only to `SMPP_RESPONSE`. The following `SMPP_RESPONSE` properties are available for the record of each event:

- `command`
- `destination`
- `error`
- `reqSize`
- `reqTimeToLastByte`
- `rspSize`
- `rspTimeToFirstByte`

- `rspTimeToLastByte`
- `source`
- `processingTime`

reqSize: *Number*

The size of the request, 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 not valid.

rspSize: *Number*

The size of the response, expressed in bytes.

Applies only to `SMPP_RESPONSE`.

rspTimeToFirstByte: *Number*

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `SMPP_RESPONSE`.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `SMPP_RESPONSE`.

source: *String*

The source address as specified in the `SMPP_REQUEST`. Will be null if this is not available for the current command type.

tprocess: *Number*

The server processing time, expressed in milliseconds. Equivalent to `rspTimeToFirstByte - reqTimeToLastByte`. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to `SMPP_RESPONSE`.

SMTP

The SMTP class allows retrieval of metrics available during the `SMTP_REQUEST` and `SMTP_RESPONSE` events.

Events

SMTP_REQUEST

Fires on every SMTP request processed by the device.

SMTP_RESPONSE

Fires on every SMTP response processed by the device.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

Applies only to `SMTP_RESPONSE`. See the `record` property below for details.

For built-in records, each unique record will be committed only once, even if `.commitRecord` 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.

Applies only to `SMTP_RESPONSE`.

headers: *Object*

An object that allows access to SMTP header names and values.

isEncrypted: *Boolean*

Returns true if the application is encrypted using STARTTLS encryption.

isReqAborted: *Boolean*

Returns true if the connection is closed before the SMTP request is complete.

isRspAborted: *Boolean*

Returns true if the connection is closed before the SMTP response is complete.

Applies only to `SMTP_RESPONSE`.

method: *String*

The SMTP method.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

Applies only to `SMTP_RESPONSE`. The following `SMTP_RESPONSE` properties are available for the record of each event:

- dataSize
- domain
- error
- isEncrypted
- isReqAborted
- isRspAborted
- method
- recipient
- recipientList
- reqBytes
- reqL2Bytes
- reqPkts
- reqRTO
- reqSize
- reqTimeToLastByte
- roundTripTime
- rspBytes
- rspL2Bytes
- rspPkts
- rspRTO
- rspSize
- rspTimeToFirstByte
- rspTimeToLastByte

- sender
- statusCode
- statusText
- processingTime

recipient: *String*

The address the message should be sent to.

recipientList: *Array of Strings*

A list of recipient addresses.

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, 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 not valid.

roundTripTime: *Number*

The median TCP round-trip time (RTT), expressed in milliseconds. Will return 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, expressed in bytes.

Applies only to SMTP_RESPONSE.

rspTimeToFirstByte: *Number*

The time from the first byte of the request until the first byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to SMTP_RESPONSE.

rspTimeToLastByte: *Number*

The time from the first byte of the request until the last byte of the response, expressed in milliseconds. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to SMTP_RESPONSE.

sender: *String*

The sender of the message.

statusCode: *Number*

The SMTP status code of the response.

Applies only to SMTP_RESPONSE.

statusText: *String*

The multi-line response string.

Applies only to SMTP_RESPONSE.

tprocess: *Number*

The server processing time, expressed in milliseconds. Equivalent to `rspTimeToFirstByte - reqTimeToLastByte`. Will return NaN on malformed and aborted responses, or if the timing is not valid.

Applies only to SMTP_RESPONSE.

SSL

The SSL class allows retrieval of metrics available during the SSL_OPEN, SSL_CLOSE, SSL_ALERT, SSL_RECORD, SSL_HEARTBEAT, and SSL_RENEGOTIATE events.

Events

SSL_ALERT

Fires when an SSL alert record is exchanged.

SSL_CLOSE

Fires when the SSL connection is shut down.

SSL_HEARTBEAT

Fires when an SSL heartbeat record is exchanged.

SSL_OPEN

Fires when the SSL connection is first established.

SSL_PAYLOAD

Fires 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`

SSL_RECORD

Fires when an SSL record is exchanged.

SSL_RENEGOTIATE

Fires on SSL renegotiation.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance.

Applies only to SSL_ALERT, SSL_CLOSE, SSL_HEARTBEAT, and SSL_OPEN.

The different properties are returned for the record of each event. See the `record` property below for details.

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

getClientExtensionData(extension_name | extension_id): *Buffer*

Returns the extension data if the extension was passed as part of the hello message from the client and had data. Returns null otherwise.

getServerExtensionData(extension_name | extension_id): *Buffer*

Returns the extension data if the extension was passed as part of the hello message from the server and had data. Returns null otherwise.

hasClientExtension(extension_name | extension_id): *boolean*

Returns true if the extension was passed as part of the hello message from the client.

hasServerExtension(extension_name | extension_id): *boolean*

Returns true if the extension was passed as part of the hello message from the server.

setApplication(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.setApplication` 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.

Applies only to `SSL_OPEN`.

Properties

alertCode: *Number*

If the session is opaque, the value is `SSL.ALERT_LEVEL_UNKNOWN (255)`. Otherwise, it corresponds to the numeric part of the `AlertDescription` data structure specified in RFC2246.

Applies only to `SSL_ALERT`.

alertLevel: *Number*

If the session is opaque, the value is `SSL.ALERT_LEVEL_UNKNOWN (255)`. Otherwise, it corresponds to the numeric part of the `AlertLevel` data structure as specified in RFC2246.

Applies only to `SSL_ALERT`.

certificate: *SSLCert*

The SSL certificate object associated with the communication. Each object has 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' certificate information dialog boxes, but without spaces. For example:

```
"55F30E6D49E19145CF680E8B7E3DC8FC7041DC81"
```

keySize: *Number*

The certificate key size.

notAfter: *Number*

The certificate expiration time in UTC.

publicKeyExponent: *String*

A string hex representation of the public key's 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's modulus. This is the same string shown in most browser's certificate information dialog boxes, but without spaces. For example:
"010001"

signatureAlgorithm: *String*

The algorithm used to sign the certificate. Some possible values are:

- From RFC 3279:
 - md2WithRSAEncryption
 - md5WithRSAEncryption
 - sha1WithRSAEncryption
- From RFC 4055:
 - sha224WithRSAEncryption
 - sha256WithRSAEncryption
 - sha384WithRSAEncryption
 - sha512WithRSAEncryption
- From RFC 4491:
 - id-GostR3411-94-with-Gost3410-94
 - id-GostR3411-94-with-Gost3410-2001

subject: *String*

The certificate subject CN string.

cipherSuite: *String*

String representing the cryptographic cipher suite negotiated between the server and the client.

clientExtensions: *Array*

An array of extension objects.

Applies only to `SSL_OPEN` and `SSL_RENEGOTIATE`.

Each object has the following properties:

id: *Number*

The ID number of the SSL extension

name: *String*

The name of the SSL extension, if known. Otherwise "unknown" will be used.

The following table displays 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

ID	Name
9	cert_type
10	elliptic_curves
11	ec_point_formats
12	srp
13	signature_algorithms
14	use_srtp
15	heartbeat
16	application_layer_protocol_negotiation
17	status_request_v2
18	signed_certificate_timestamp
19	client_certificate_type
20	server_certificate_type
35	SessionTicket TLS
65281	renegotiation_info

clientSessionId: *String*

The client session ID, byte array encoded as a string.

contentType: *String*

The content type for the current record.

Applies only to `SSL_RECORD`.

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.

Applies only to `SSL_OPEN` and `SSL_RENEGOTIATE`.

heartbeatPayloadLength: *Number*

The value of the `payload_length` field of the HeartbeatMessage data structure as specified in RFC 6520.

Applies only to `SSL_HEARTBEAT`.

heartbeatType: *Number*

Returns the numeric representation of the HeartbeatMessageType field of the HeartbeatMessage 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)`.

Applies only to `SSL_HEARTBEAT` events.

host: *string*

The value of the SSL Server Name Indication (SNI), if present.

Applies only to `SSL_OPEN` and `SSL_RENEGOTIATE`.

isAborted: *Boolean*

Returns true if the SSL session is aborted.

Applies only to `SSL_CLOSE`.

isCompressed: *Boolean*

Returns true if the SSL record is compressed.

isV2ClientHello: *Boolean*

Returns true if the Hello record corresponds to SSLv2.

privateKeyId: *String*

Returns null if the ExtraHop appliance is not decrypting the SSL traffic. Returns a string ID associated with the private key if the ExtraHop appliance is decrypting the SSL traffic.

To find the private key ID in the ExtraHop Admin UI, go to the **Configuration** section, click **Capture**, click **SSL Decryption**, and then click a certificate. The pop-up window displays all identifiers for the certificate.

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

Applies only to `SSL_OPEN` and `SSL_CLOSE`, `SSL_ALERT`, AND `SSL_HEARTBEAT`.

The following table shows the properties available for the record of each event.

Event	Available properties
SSL_ALERT	<ul style="list-style-type: none"> • alertCode • alertLevel • certificate.fingerprint • certificate.keySize • certificate.notAfter • certificate.signatureAlgorithm • certificate.subject • cipherSuite • isCompressed • version
SSL_CLOSE	<ul style="list-style-type: none"> • certificate.fingerprint • certificate.keySize • certificate.notAfter • certificate.signatureAlgorithm • certificate.subject • cipherSuite • isAborted • isCompressed • version
SSL_HEARTBEAT	<ul style="list-style-type: none"> • certificate.fingerprint • certificate.keySize • certificate.notAfter • certificate.signatureAlgorithm • certificate.subject • cipherSuite • heartbeatPayloadLength • heartbeatType • isCompressed • version

Event	Available properties
SSL_OPEN	<ul style="list-style-type: none"> certificate.fingerprint certificate.keySize certificate.notAfter certificate.signatureAlgorithm certificate.subject cipherSuite handshakeTime host isCompressed version

recordLength: *Number*

Returns the value of the length field of the TLSPlaintext, TLSCompressed, and TLSCiphertext data structures as specified in RFC 5246.

Applies only to SSL_RECORD, SSL_ALERT, and SSL_HEARTBEAT events.

recordType: *Number*

Returns the numeric representation of the type field of the TLSPlaintext, TLSCompressed, and TLSCiphertext data structures as specified in RFC 5246.

Applies only to SSL_RECORD, SSL_ALERT, and SSL_HEARTBEAT events.

reqBytes: *Number*

The number of request bytes.

Applies only to SSL_RECORD and SSL_CLOSE.

reqL2Bytes: *Number*

The number of L2 request bytes.

Applies only to SSL_RECORD and SSL_CLOSE.

reqPkts: *Number*

The number of request packets.

Applies only to SSL_RECORD and SSL_CLOSE.

rspBytes: *Number*

The number of response bytes.

Applies only to SSL_RECORD and SSL_CLOSE.

rspL2Bytes: *Number*

The number of L2 response bytes.

Applies only to SSL_RECORD and SSL_CLOSE.

rspPkts: *Number*

The number of response packets.

Applies only to SSL_RECORD and SSL_CLOSE.

roundTripTime: *Number*

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

Applies only to SSL_RECORD and SSL_CLOSE.

serverExtensions: *Array*

An array of extension objects. Each object has the following properties:

Applies only to `SSL_OPEN` and `SSL_RENEGOTIATE`.

id: *Number*

The ID number of the SSL extension.

name: *String*

The name of the SSL extension, if known. Otherwise "unknown" will be used. See `clientExtensions` above for a list of known extension names.

serverSessionId: *String*

The server session ID, byte array encoded as a string.

setApplication(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.setApplication` 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.

Applies only to `SSL_OPEN`.

version: *Number*

The SSL protocol version with the RFC hexadecimal version number expressed as a decimal.

Version	Hex	Decimal
SSLv3	0x300	768
TLS 1.0	0x301	769
TLS 1.1	0x302	770
TLS 1.2	0x303	771

Telnet

The Telnet class allows retrieval of metrics available during the `TELNET_MESSAGE` event.

Events

TELNET_MESSAGE

Fires on a telnet command or line of data from the telnet client or server.

Methods

commitRecord(): *void*

Commits the record to the ExtraHop Explore appliance. See the `record` property below for details.

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

Properties

command: *String*

The command type. Will be null if the event fired due to a line of data being sent.

The possible values returned are:

- 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. Things like cursor movement/line editing are not currently simulated (with the exception of backspace characters).

option: *String*

The option being negotiated. Will be null if the command is not an option command.

The possible values are:

- 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
- NAOCR D
- 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. Will be null if the command is not "SB".

record: *Object*

Returns an object with all properties appropriately initialized. The event that this is called from determines the content of object returned.

The following `TELNET_MESSAGE` properties are available for the record of each event:

- command
- option

Topnset

The Topnset class is used to represent topnset metrics. A topnset metric contains a list of entries with keys and values. Values may be numbers, Dataset, Sampleset, or even other Topnsets. Topnset Keys are objects that represent a property of Topnset.

Methods

findEntries(keyPattern: *Object*): *Arr*

Returns all entries with matching keys.

findKeys(keyPattern: *Object*): *Array*

Returns all keys matching the specified pattern.

lookup(keyPattern: *Object*): *

Look up an item in the topnset, returning the first that matches the pattern.

Properties

entries: *Array*

An array of the topnset entries. The array contains N objects with key and value properties, with N currently being set to 1000.

Topnset Keys

Topnset Keys are objects that represent a property of Topnset.

Properties

type: *String*

The type of the topnset key. Possibilities include:

- string
- int
- ipaddr
- ether
- device_id

value: *

Varies depending on the type of key.

- For string keys, the value is a string.
- For int keys, the value is a number.
- For ipaddr keys, the value is an object containing:
 - addr
 - proto
 - port
 - device_oid
 - origin
 - custom_devices
- For ether keys, the value is an object containing:
 - ethertype
 - hwaddr

TroubleGroup

It is possible to create your own trouble group. A trouble group is a set of devices for which a potential problem has been identified. This class may be used in conjunction with metric and discovery events (NEW_DEVICE, METRIC_CYCLE_BEGIN, METRIC_RECORD_COMMIT) to identify potential performances, security or configuration problems.

Methods

addDevice(troubleGroupName: *String*, device: *Device*): *void*

Adds a device to the specified trouble group.

See also

- [Example: Custom trouble group](#)

Turn

Turn is top-level object available in the `FLOW_TURN` event.

Properties

clientBytes: *Number*

The size of the request that the client transferred, expressed in bytes.

clientTransferTime: *Number*

The request 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.

serverBytes: *Number*

The size of the response that the server transferred, expressed in bytes.

serverTransferTime: *Number*

The response transfer time, expressed in milliseconds. Corresponds to the Network Out metric in the ExtraHop Web UI.

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. Will return NaN if there is no valid measurement.

Deprecated

reqSize: *Number*

Deprecated. Use `clientBytes` instead.

reqXfer: *Number*

Deprecated. Use `clientTransferTime` instead.

rspSize: *Number*

Deprecated. Use `serverBytes` instead.

rspXfer: *Number*

Deprecated. Use `serverTransferTime` instead.

tprocess: *Number*

Deprecated. Use `processingTime` instead.

VLAN

The VLAN class represents a VLAN on the network.

Properties

id: *Number*

The numerical ID of the VLAN.

XML

Returns parsed XML data.

For example:

```
var payload = "<Header><storeid>my_payload</storeid></Header>"
var xml = new XML(payload);
debug("storeid: " + xml.storeid); // The value is "my_payload."
```

Examples

The following examples are available:

Example: HTTP header object

Shows how to use the HTTP Headers object.

Example: SOAP request

Tracks SOAP requests via the SOAPAction header and stashes them into the flow store.

Example: Customer ID header

Records 500 errors by customer ID and URI.

Example: Database trigger

Records responses and processing time by database query.

Example: CIFS trigger

Records total read and written bytes as well as bytes written by users not authorized to access a sensitive resource.

Example: Memcache hits and misses

Records keys for each hit or miss, and hit access time.

Example: Memcache key parsing

Parses the memcache keys to extract detailed breakdowns by module and class name, as well as by ID.

Example: Trigger-based application definition

Creates an ExtraHop application container based on traffic associated with a two-tier application.

Example: Session table

Records specific transactions to the session table with operating system strings in HTTP.userAgent.

Example: Custom trouble group

Creates a custom trouble group.

Example: Topnset key matching

Matches Topnset Keys.

Example: Use the Metric Cycle store

Illustrates the use of the Metric Cycle store.

Example: Device discovery notification

Creates remote syslog messages when new devices are discovered.

Example: Parse custom POS messages with universal payload analysis

Parses a POS system custom TCP messages, looking for specific values in the 4th to 7th bytes of both response and request messages.

Example: Parse syslog over TCP with universal payload analysis

Parses the syslog over TCP and count the syslog activity over time, both network-wide and per device.

Example: Send data to Elasticsearch with Remote.HTTP

Specify the JSON text suitable for Elasticsearch and send the information using Remote.HTTP

Example: Send information to Azure table service with Remote.HTTP

Sends information to the Microsoft Azure Table Service via Remote.HTTP.

Example: ActiveMQ

Records JMS destination information.

Example: ActiveMQ

Records JMS destination information.

```
{code:javascript}
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);
}
{code}
```

See also

- [ActiveMQ](#)

Example: Send information to Azure table service with Remote.HTTP

Sends information to the Microsoft Azure Table Service via Remote.HTTP.



Note: This example requires the configuration of Open Datastream for HTTP to use Microsoft Azure request signing.

```
/*
 * Name: Send Information to Azure Table Service with Remote.HTTP
 */

// 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 Table Storage expects to
receive XML;
 * however, it is easier for a trigger to send JSON.
 * The odata information allows you to specify the datatype of fields; in
this case
 * TS 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.tprocess,
        "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) } );

```

See also

- [Remote.HTTP](#)

Example: CIFS trigger

Records total read and written bytes as well as bytes written by users not authorized to access a sensitive resource.

```

/*
 * Name: CIFS Audit Support
 * Event: 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\bob" : {read: true, write: true}
};

/* Check if this is an action on our 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);
    }
}
}

```

See also

- [CIFS](#)
- [Device](#)
- [FLOW](#)

Example: Custom trouble group

Creates a custom trouble group.

```

/*
 * Name: Create a Custom Trouble Group
 * Event: METRIC_RECORD_COMMIT
 * Advanced options: 30sec cycle, extrahop.device.http_server
 */
var fields = MetricRecord.fields,
    tprocess = fields.tprocess,
    slowWebServerThreshold = 200,
    pct1;

pct1 = tprocess.percentile(50);

if (pct1 > slowWebServerThreshold) {
    TroubleGroup.addDevice('Slow Web Servers', MetricRecord.object);
}

```

See also

- [MetricRecord](#)
- [TroubleGroup](#)

Example: Customer ID header

Records 500 errors by customer ID and URI.

```

/*
 * Name: Customer ID

```

```

* Event: HTTP_REQUEST
*/
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 record it on
        * 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")
{
    /*
    * Name: HTTP Cust-ID 500 URI Tracking
    * Comment: Record 500 errors by Customer ID and URI.
    * 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);
    }
}
}

```

See also

- [Device](#)
- [FLOW](#)
- [HTTP](#)

Example: Database trigger

Records responses and processing times by database query.

```

/*
 * Name: DB Full Statement
 * Description: Record responses and processing times by database query
 * Events: DB_REQUEST, DB_RESPONSE
 *
 * Note: Assign this trigger to all the devices in a group and then
 * create a custom page for the Capture that displays the Network metric
 * to show the counts across the device group.
 */
var stmt = Flow.store.stmt;

if (event === "DB_REQUEST")
{
    /* Store the statement for the response. */
    Flow.store.stmt = DB.statement || DB.procedure;
    return;
} if ((typeof stmt === "undefined") || (stmt === null))
{
    /*
     * Restrict statement length to 1024 bytes, since statements larger than
     * this are unlikely to be useful.
     */
    if (stmt.length > 1024) {
        stmt = stmt.substr(0, 1023);
    }

    /* Remove common blank line from front of DB procedures */
    if ((stmt.length > 0) && (stmt[0] === "\n")) {
        stmt = stmt.slice(1);
    }
    /* 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.tprocess);
    Device.metricAddDetailSampleset("db_proc_time_detail",
        stmt, DB.tprocess);
    /* Record these metrics at the network level as well */
    Network.metricAddCount("db_rsp_count", 1);
    Network.metricAddDetailCount("db_rsp_count_detail", stmt, 1);
    Network.metricAddSampleset("db_proc_time", DB.tprocess);
    Network.metricAddDetailSampleset("db_proc_time_detail",
        stmt, DB.tprocess);
}

```

See also

- [DB](#)
- [Device](#)
- [FLOW](#)
- [Network](#)

Example: Device discovery notification

Creates remote syslog messages when new devices are discovered.

```
/*
 * Name: Device Discovery Notification
 * Event: NEW_DEVICE
 */
var dev = Discover.device;
Remote.Syslog.info('Discovered device ' + dev.id + ' (hwaddr: ' + dev.hwaddr
+ ' )
');
```

See also

- [Device](#)
- [Discover](#)
- [Remote.Syslog](#)

Example: Send data to Elasticsearch with Remote.HTTP

Specify the JSON text suitable for Elasticsearch and send the information using Remote.HTTP.

```
/*
 * Name: Send Data to Elasticsearch with Remote.HTTP
 */

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);
```

See also

- [Remote.HTTP](#)

Example: HTTP header object

Shows how to use the HTTP Headers object.

```
/*
 * Trigger: HTTP Headers Example
 * Event: 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 was not present). header values are always strings,
 * even for headers that typically have numbers as values.
 */

/* This syntax works too, 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
 *
 * "length" is a special property (case-sensitive) that is not
 * treated as a header lookup, but instead 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-");

/*
 * result 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);
}

```

See also

- [Device](#)

- [HTTP](#)

Example: Memcache hits and misses

Records keys for each hit or miss, and hit access time.

```

/*
 * Name: Memcache_hits_misses
 * Event: 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);
    }
}

```

See also

- [Device](#)
- [Memcache](#)

Example: Memcache key parsing

Parses the memcache keys to extract detailed breakdowns.

```

/*
 * Name: Memcache_Example_Trigger
 * Description: Parses the memcache keys to extract detailed breakdowns.
Keys
 * look like:
 * "com.extrahop.<module>.<class>_<id>"
 * (for example: "com.extrahop.widgets.sprocket_12345")
 * and we want breakdowns by module and class name as well as ID.

```

```

* Event: 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);
            }
        }
    }
}

/*
 * Users have reported slowness accessing sprockets. 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 key(s) 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);
}
}
}

```

See also

- [Device](#)
- [Memcache](#)

Example: Use the Metric Cycle store

Illustrates the use of the Metric Cycle store.

```

/*
 * Name: Using the Metric Cycle Store
 * Event: METRIC_CYCLE_BEGIN, METRIC_RECORD_COMMIT, METRIC_CYCLE_END
 * Advanced options: 30sec cycle, 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;
}

```

See also

- [Device](#)
- [MetricCycle](#)
- [MetricRecord](#)

Example: Parse custom POS messages with universal payload analysis

Parses a POS system custom TCP messages, looking for specific values in the 4th to 7th bytes of both response and request messages.

```

/* Name: Parse Custom POS Messages with Universal Payload Analysis
 * Event: TCP_PAYLOAD
 */

//
// Define variables, store client or server payload into Buffer
//

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's other protocol of no interest or there is no payload
//
if (protocol !== 'tcp:4015' || (buf_client === null && buf_server === null))
{
    //debug('Not interested protocol: ' + 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;
}

```

See also

- [Device](#)
- [FLOW](#)

Example: Parse syslog over TCP with universal payload analysis

Parses the syslog over TCP and count the syslog activity over time, both network-wide and per device.



Note: The complete solution that includes this trigger code is available on the ExtraHop forums.

```

/* Name: Parse Syslog over TCP via Universal Payload Analysis
 * Event: TCP_PAYLOAD, UDP_PAYLOAD
 *
 */

//
// Variables we need throughout (aka global)
//

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
    // we are just treating 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));"
}
```

See also

- [FLOW](#)
- [Network](#)
- [Remote.MongoDB](#)
- [Remote.Syslog](#)

Example: Session table

Records specific transactions to the session table with operating system strings in HTTP.userAgent.

```
/*
 * Name: Session Table
 * 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;
}
/* Use matched string as key for session table entry */
os = os[0];

var opts =
{
    /* Expire added entries after 30 seconds */
    expire: 30,
    /* Retain entries with normal priority if the 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);
    }
  }
}

```

See also

- [HTTP](#)
- [Network](#)
- [Session](#)

Example: SOAP request

Tracks SOAP requests via the SOAPAction header and stashes them into the flow store.

```

/*
 * Name: SOAP Action
 * Description: Tracks SOAP requests via the SOAPAction header and stashes
 * them
 * into the flow store. The big requirement is that the SOAP implementation
 * actually passes this information. Not all do, so be sure and confirm that
 * part.
 * 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.tprocess);
    Network.metricAddDetailSampleset("soap_proc_detail", method,
                                     HTTP.tprocess);
    }
}

```

See also

- [FLOW](#)
- [HTTP](#)
- [Network](#)

Example: Topnset key matching

Matches Topnset Keys.

```

/*
 * Name: Topnset Key Matching
 * Event: METRIC_RECORD_COMMIT
 * Advanced options: 30sec cycle, 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));
}

```

See also

- [MetricRecord](#)
- [Remote.Syslog](#)

Example: Trigger-based application definition

Trigger to create an ExtraHop application container based on traffic associated with a two-tier application.

```

/*
 * Name: Application Builder - My App

```

```

* Events: HTTP_RESPONSE, DB_RESPONSE
* Assignments: All HTTP servers that process application HTTP traffic
* All DB servers that process application DB traffic
* Firmware: 3.9+ required, 3.10+ recommended
*/

// Initialize the Application object against which we'll
// commit specific HTTP and DB transactions. After traffic is
// committed, an application container "My App" will appear in the
// Applications tab in the WebUI.

var myApp = Application("My App");

// These configurable properties describe features that define
// our application traffic.

var myAppHTTPHost = "myapp.internal.example.com";
var myAppDatabaseName = "myappdb";
if (event == "HTTP_RESPONSE") {

    // HTTP transactions can be committed to an Application on
    // HTTP_RESPONSE events.

    // Commit this HTTP transaction only if the HTTP Host header for
    // this response is defined and matches our 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 an 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();
    }
}
}

```

See also

- [Application](#)

- [DB](#)
- [HTTP](#)