ExtraHop 25.2
Trigger API Reference

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

Published: 2025-04-29

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

Contents

Overview

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

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

Triggers enable you to:

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

The purpose of this guide is to provide reference material when writing the blocks of JavaScript code that run when trigger conditions are met. The Trigger API resources section contains a list of topics that provide a comprehensive overview of trigger concepts and procedures.

Trigger API resources

This section contains a list of topics that will help familiarize you with trigger concepts, building a trigger, and best practices.

- Triggers
- Build a trigger
  - Configure trigger settings
  - Write a trigger script
- Monitor trigger performance
- Triggers Best Practices Guide
- Triggers FAQ
- Walkthrough: Build a trigger to collect custom metrics for HTTP 404 errors
- Walkthrough: Initiate precision packet captures to analyze zero window conditions
- Walkthrough: Build a trigger to monitor responses to NTP monlist requests

Data types for custom metrics

The ExtraHop Trigger API enables you to create custom metrics that collect data about your environment, beyond what is provided by built-in protocol metrics.

You can create custom metrics of the following data types:

count

The number of metric events that occurred over a specific time range. For example, to record information about the number of HTTP requests over time, select a top-level count metric. You could also select a detail count metric to record information about the number of times clients accessed a server, with the IPAddress key and an integer representing the number of accesses as a value.

snapshot

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

distinct

The estimated number of unique items observed over time, such as the number of unique ports that received SYN packets, where a high number might indicate port scanning.

dataset

A statistical summary of timing information, such as 5-number summary: min, 25th-percentile, median, 75th-percentile, max. For example, to record information about HTTP processing time over time, select a top-level dataset metric.

sampleset

A statistical summary of timing information, such as mean and standard deviation. For example, to record information about the length of time it took the server to process each URI, select a detail sampleset with the URI string key and an integer representing processing time as a value.

max

A special type of count metric that preserves the maximum. For example, to record the slowest HTTP statements over time without relying on a session table, select a top-level and a detail max metric.

Custom metrics are supported for the following source types:

- Application
- Device
- Network
- FlowInterface
- FlowNetwork

For more information about the differences between top-level and detail metrics, see the Metrics FAQ.

Global functions

Global functions can be called on any event.

`cache(key: String, valueFn: () => Any): Any`

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

`key: String`

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

`valueFn: () => Any`

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

In the following example, the `cache` method is called with large amounts of data hard-coded into the trigger script:

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

var storeCity;
var query = HTTP.parseQuery(HTTP.query);

if (query.storeCode) {
   storeCity = storeLookup[parseInt(query.storeCode)];
}
```

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

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

var badUserAgents = cache("badUserAgents", jbossUserAgents);
```

`commitDetection(type: String, options: Object)`

Generates a detection on the ExtraHop system.

`type: String`

A user-defined type for the definition, such as `brute_force_attack`. You can tune detections to hide multiple detections with the same type. The string can only contain letters, numbers, and underscores.

`options: Object`

An object that specifies the following properties for the detection:

`title:` ***String***

A user-defined title that identifies the detection.

`description:` ***String***

A description of the detection.

`riskScore:` ***Number*** | ***null***

An optional number between 1 and 99 that represents the risk score of the detection.

`participants:` ***Array of Objects***

An optional array of participant objects associated with the detection. Specify each participant as the offender or victim property of a Flow endpoint object. For example, the following code specifies the client in the flow as the offender and the server as the victim:

```
commitDetection('exampledetection', {
     participants: [Flow.client.offender,
 Flow.server.victim],
```

The following participant objects are valid:

- `Flow.client.offender`
- `Flow.client.victim`
- `Flow.server.offender`
- `Flow.server.victim`
- `Flow.sender.offender`
- `Flow.sender.victim`
- `Flow.receiver.offender`
- `Flow.receiver.victim`

`identityKey:` ***String*** | ***null***

A unique identifier that enables ongoing detections. If multiple detections with the same identity key and detection type are generated within the time period specified by the `identityTtl` property, the detections are consolidated into a single ongoing detection.

Note: If the ExtraHop system is generating a large number of detections with unique identity keys, the system might fail to consolidate some ongoing detections. However, the system will not generate more than 250 individual detections for a trigger in a day.

`identityTtl:` ***String***

The amount of time after a detection is generated that duplicate detections are consolidated into an ongoing detection.

After a detection is generated, if another detection with the same identity key and detection type is generated within the specified time period, the two detections are consolidated into a single ongoing detection. Each time a detection is consolidated into an ongoing detection, the time period is reset, and the detection does not end until the time period expires. For example, if `identityTtl` is set to `day`, and four duplicate detections are each generated 12 hours apart, the ongoing detection spans three days. The following time periods are valid:

- `hour`
- `day`
- `week`

The default time period is `hour`.

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

Sends a custom record object to the configured recordstore.

`id: String`

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

`record: Object`

An object containing a list of property and value pairs to be sent to the configured recordstore as a custom record.

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

- `ex`
- `flowID`
- `client`
- `clientAddr`
- `clientPort`
- `receiver`
- `receiverAddr`
- `receiverPort`
- `sender`
- `senderAddr`
- `senderPort`
- `server`
- `serverAddr`
- `serverPort`
- `timestamp`
- `vlan`

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

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

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

`.`
: Period

`:`
: Colon

`[`
: Square bracket

`]`
: Square bracket

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

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

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

On most events, you can commit a built-in record that contains default properties. For example, a built-in record such as the `HTTP.record` object can be the basis for a custom record.

The following example code commits a custom record that includes all of the built-in metrics from the `HTTP.record` object and an additional metric from the `HTTP.headers` property:

```
var record = Object.assign(
   {'server': HTTP.headers.server},
   HTTP.record
);
commitRecord('custom-http-record', record);
```

You can access a built-in record object on the following events:

Class	Events
AAA	AAA_REQUEST AAA_RESPONSE
ActiveMQ	ACTIVEMQ_MESSAGE
AJP	AJP_RESPONSE
CIFS	CIFS_RESPONSE
DB	DB_RESPONSE
DHCP	DHCP_REQUEST DHCP_RESPONSE
DICOM	DICOM_REQUEST DICOM_RESPONSE
DNS	DNS_REQUEST DNS_RESPONSE
FIX	FIX_REQUEST FIX_RESPONSE
Flow	FLOW_RECORD
FTP	FTP_RESPONSE
HL7	HL7_RESPONSE
HTTP	HTTP_RESPONSE
IBMMQ	IBMMQ_REQUEST IBMMQ_RESPONSE
ICA	ICA_OPEN ICA_CLOSE ICA_TICK
ICMP	ICMP_MESSAGE
Kerberos	KERBEROS_REQUEST

Class	Events
	KERBEROS_RESPONSE
LDAP	LDAP_REQUEST LDAP_RESPONSE
Memcache	MEMCACHE_REQUEST MEMCACHE_RESPONSE
Modbus	MODBUS_RESPONSE
MongoDB	MONGODB_REQUEST MONGODB_RESPONSE
MSMQ	MSMQ_MESSAGE
NetFlow	NETFLOW_RECORD
NFS	NFS_RESPONSE
NTLM	NTLM_MESSAGE
POP3	POP3_RESPONSE
RDP	RDP_OPEN RDP_CLOSE RDP_TICK
Redis	REDIS_REQUEST REDIS_RESPONSE
RTCP	RTCP_MESSAGE
RTP	RTP_TICK
SCCP	SCCP_MESSAGE
SFlow	SFLOW_RECORD
SIP	SIP_REQUEST SIP_RESPONSE
SMPP	SMPP_RESPONSE
SMTP	SMTP_RESPONSE
SSH	SSH_OPEN SSH_CLOSE SSH_TICK
SSL	SSL_ALERT SSL_OPEN SSL_CLOSE SSL_HEARTBEAT SSL_RENEGOTIATE

Class	Events
Telnet	TELNET_MESSAGE

`debug(message: String): void`

Writes to the debug log if debugging is enabled. The maximum message size is 2048 bytes. Messages longer than 2048 bytes are truncated.

`getTimestamp(): Number`

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

`log(message: String): void`

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

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

The limit for debug log entries is 2048 bytes. To log larger entries, see Remote.Syslog.

`md5(message: String|Buffer): String`

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

`sha1(message: String|Buffer): String`

Hashes the UTF-8 representation of the specified message Buffer object or string and returns the SHA-1 sum of the string.

`sha256(message: String|Buffer): String`

Hashes the UTF-8 representation of the specified message Buffer object or string and returns the SHA-256 sum of the string.

`sha512(message: String|Buffer): String`

Hashes the UTF-8 representation of the specified message Buffer object or string and returns the SHA-512 sum of the string.

`uuid(): String`

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

General purpose classes

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

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

Application

The Application class enables you collect metrics across multiple types of network traffic to capture information with cross-tier impact. For example, if you want a unified view of all the network traffic associated with a website—from web transactions to DNS requests and responses to database transactions—you can write a trigger to create a custom application that contains all of these related metrics. The Application class also enables you to create custom metrics and commit the metric data to applications. Applications can only be created and defined through triggers.

Instance methods

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

```
Application("sampleApp").metricAddCount("responses", 1);
```

However, the following statement is invalid:

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

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

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

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

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

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

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

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

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

Metric types	Event
AAA	AAA_REQUEST -and- AAA_RESPONSE
AJP	AJP_RESPONSE
CIFS	CIFS_RESPONSE
DB	DB_RESPONSE
DHCP	DHCP_REQUEST -and- DHCP_RESPONSE
DNS	DNS_REQUEST -and- DNS_RESPONSE
FIX	FIX_REQUEST -and- FIX_RESPONSE
FTP	FTP_RESPONSE
HTTP	HTTP_RESPONSE
IBMMQ	IBMMQ_REQUEST -and- IBMMQ_RESPONSE

Metric types	Event
ICA	ICA_TICK -and- ICA_CLOSE
Kerberos	KERBEROS_REQUEST -and- KERBEROS_RESPONSE
LDAP	LDAP_REQUEST -and- LDAP_RESPONSE
Memcache	MEMCACHE_REQUEST -and- MEMCACHE_RESPONSE
Modbus	MODBUS_RESPONSE
MongoDB	MONGODB_REQUEST -and- MONGODB_RESPONSE
NAS	CIFS_RESPONSE -and/or- NFS_RESPONSE
NetFlow	NETFLOW_RECORD Note that the commit will not occur if enterprise IDs are present in the NetFlow record.
NFS	NFS_RESPONSE
RDP	RDP_TICK
Redis	REDIS_REQUEST -and- REDIS_RESPONSE
RPC	RPC_REQUEST -and- RPC_RESPONSE
RTP	RTP_TICK
RTCP	RTCP_MESSAGE
SCCP	SCCP_MESSAGE
SIP	SIP_REQUEST -and- SIP_RESPONSE
SFlow	SFLOW_RECORD
SMTP	SMTP_RESPONSE
SSH	SSH_CLOSE -and- SSH_TICK
SSL	SSL_RECORD -and- SSL_CLOSE
WebSocket	WEBSOCKET_OPEN, WEBSOCKET_CLOSE, and WEBSOCKET_MESSAGE

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

Creates a custom top-level count metric. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the top-level count metric.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDetailCount(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, count:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail count metric by which you can drill down. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDataset(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level dataset metric. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the top-level dataset metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`freq:` ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

`highPrecision:` ***Boolean***

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

`metricAddDetailDataset(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail dataset metric by which you can drill down. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`freq:` ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

highPrecision: ***Boolean***

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

`metricAddDistinct(metric_name:` ***String***`, item:` ***Number*** | ***String*** | ***IPAddress***`:void`

Creates a custom top-level distinct count metric. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the top-level distinct count metric.

`item:` ***Number*** | ***String*** | ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddDetailDistinct(metric_name:` ***String***`, key:` ***String*** | ***IPAddress***`, item:` ***Number*** | ***String*** | ***IPAddress***`:void`

Creates a custom detail distinct count metric by which you can drill down. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the detail distinct count metric.

`key:` ***String*** | ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`item:` ***Number*** | ***String*** | ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddMax(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level maximum metric. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the top-level maximum metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailMax(metric_name:` ***String***`, key:` ***String*** | ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail maximum metric by which you can drill down. Commits the metric data to the specified application.

`metric_name:` ***String***

The name of the detail maximum metric.

`key:` ***String | IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddSampleset(metric_name: String, val: Number, options: Object):void`

Creates a custom top-level sampleset metric. Commits the metric data to the specified application.

`metric_name: String`

The name of the top-level sampleset metric.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddDetailSampleset(metric_name: String, key: String | IPAddress, val: Number, options: Object):void`

Creates a custom detail sampleset metric by which you can drill down. Commits the metric data to the specified application.

`metric_name: String`

The name of the detail sampleset metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddSnap(metric_name: String, count: Number, options: Object):void`

Creates a custom top-level snapshot metric. Commits the metric data to the specified application.

`metric_name: String`

The name of the top-level snapshot metric.

`count: Number`

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddDetailSnap(metric_name: String, key: String | IPAddress, count: Number, options: Object):void`

Creates a custom detail snapshot metric by which you can drill down. Commits the metric data to the specified application.

`metric_name: String`

The name of the detail sampleset metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`toString():` ***String***

Returns the Application object as a string in the following format:

```
[object Application <application_id>]
```

Instance properties

`id:` ***String***

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

Trigger examples

- Example: Create an application container

Buffer

The `Buffer` class provides access to binary data.

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

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

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

Methods

`Buffer(string:` ***String*** `|` `format:` ***String***`)`

Constructor for the Buffer class that decodes an encoded string into a Buffer object. The following parameters are required:

`string:` ***String***

The encoded string.

`format:` ***String***

The format that the string argument is encoded with. The following encoding formats are valid:

- `base64`
- `base64url`

Instance methods

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

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

- `utf-8`
- `utf-16`

- ucs2
- hex

`equals(buffer:` ***Buffer***`):` ***Boolean***

Performs an equality test between Buffer objects, where `buffer` is the object to be compared against.

`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. Specify 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. Specify negative numbers to select from the end of a buffer. This is zero-based.

`toString(format:` ***String***`):` ***String***

Converts the buffer to a string. The following parameter is optional:

`format:` ***String***

The format to encode the string with. If no encoding is specified, the string is unencoded. The following encoding formats are valid:

- base64
- base64url
- hex

`unpack(format:` ***String***`, offset:` ***Number***`):` ***Array***

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

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

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

The table below displays supported buffer string formats:

Format	C type	JavaScript type	Standard size
x	pad type	no value	
A	struct in6_addr	IPAddress	16
a	struct in_addr	IPAddress	4

Format	C type	JavaScript type	Standard size
b	signed char	string of length 1	1
B	unsigned char	number	1
?	_Bool	boolean	1
H	unsigned short	number	2
h	short	number	2
i	int	number	4
I	unsigned int	number	4
l	long	number	4
L	unsigned long	number	4
q	long long	number	8
Q	unsigned long long	number	8
f	number	number	4
d	double	number	4
s	char[]	string	
z	char[]	string	

Instance Properties

`length:` ***Number***

The number of bytes in the buffer.

Trigger Examples

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

`Detection`

The `Detection` class enables you to retrieve information about detections on the ExtraHop system.

Note: Machine learning detections require a connection to ExtraHop Cloud Services.

Events

`DETECTION_UPDATE`

Runs when a detection is created or updated on the ExtraHop system.

Tip: Instead of writing a trigger to export detection data, we recommend that you create a detection notification rule. You can configure these rules to send JSON payloads with a webhook and avoid the complexity of writing a trigger.

Important: This event runs for all detections, regardless of the module access granted to the user who creates the trigger. For example, triggers created by users with

NPM module access run on `DETECTION_UPDATE` events for both security and performance detections.

Note: This event does not run when a detection ticket status is updated. For example, changing a detection assignee will not cause the DETECTION_UPDATE event to run. This event also does not run for hidden detections.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

Properties

`applianceId:` ***Number***

If called on a console, returns the ID of the connected sensor that the detection occurred on. If called on a sensor, returns `0`.

`assignee:` ***String***

The assignee of the ticket associated with the detection.

`categories:` ***Array of Strings***

The list of categories the detection belongs to.

The following values are valid:

Value	Category
`sec`	Security
`sec.action`	Actions on Objective
`sec.botnet`	Botnet
`sec.caution`	Caution
`sec.command`	Command & Control
`sec.cryptomining`	Cryptomining
`sec.dos`	Denial of Service
`sec.exfil`	Exfiltration
`sec.exploit`	Exploitation
`sec.hardening`	Hardening
`sec.lateral`	Lateral Movement
`sec.ransomware`	Ransomware
`sec.recon`	Reconnaissance
`perf`	Performance
`perf.auth`	Authorization & Access Control
`perf.db`	Database
`perf.network`	Network Infrastructure
`perf.service`	Service Degradation
`perf.storage`	Storage
`perf.virtual`	Desktop & App Virtualization
`perf.web`	Web Application

`description:` ***String***

The description of the detection.

Tip: It is often easier to extract information about a detection from the `Detection.properties` property than parsing the `Detection.description` text. For more information, see the `Detection.properties` description.

The following table shows common Markdown formats that you can include in the description:

Format	Description	Example
Headings	Place a number sign (#) and a space before your text to format headings. The level of heading is determined by the amount of number signs.	`#### Example H4 heading`
Unordered lists	Place a single asterisk (*) before your text. If possible, put each list item on a separate line.	`* First example` `* Second example`
Ordered lists	Place a the number 1 and period (1.) before your text for each line item; Markdown will automatically increment the list number. If possible, put each list item on a separate line.	`1. First example` `1. Second example`
Bold	Place double asterisks before and after your text.	`**bold text**`
Italics	Place an underscore before and after your text.	`_italicized text_`
Hyperlinks	Place link text in brackets before the URL in parentheses. Or type your URL. Links to external websites open in a new browser tab. Links within the ExtraHop system, such as dashboards, open in the current browser tab.	`[Visit our home page](https://www.extrahop.com)` `https://www.extrahop.com`
Blockquotes	Place a right angle bracket and a space before your text.	`On the ExtraHop website:` `> Access the live demo and review case studies.`
Emojis	Copy and paste an emoji image into the text box. See the Unicode Emoji Chart website for images. Markdown syntax does not support emoji shortcodes.	

`endTime:` ***Number***

The time that the detection ended, expressed in milliseconds since the epoch.

`id:` ***Number***

The unique identifier for the detection.

`isCustom:` ***Boolean***

The value is `true` if the detection is a custom detection generated by a trigger.

`isEventCreate:` ***Boolean***

If the value is true, the `DETECTION_UPDATE` event ran when the detection was created. If the value is false, the `DETECTION_UPDATE` event ran when the detection was updated.

`mitreCategories:` ***Array of Objects***

An array of objects that contains the MITRE techniques and tactics associated with the detection. Each object contains the following properties:

`id`

The ID of the MITRE technique or tactic.

`name`

The name of the MITRE technique or tactic.

`url`

The web address of the technique or tactic on the MITRE website.

`participants:` ***Array of Objects***

An array of participant objects associated with the detection. A participant object contains the following properties:

`object:` ***Object***

The Device, Application, or IP address object associated with the participant.

`id:` ***Number***

The ID of the participant.

`role:` ***String***

The role of the participant in the detection. The following values are valid:

- `offender`
- `victim`

`properties:` ***Object***

An object that contains the properties of the detection. Only built-in detection types include detection properties. The detection type determines which properties are available.

The field names of the object are the names of the detection properties. For example, the Anonymous FTP Auth Enabled detection type includes the `client_port` property, which you can access with the following code:

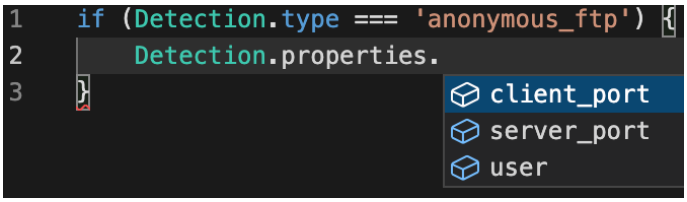
```
Detection.properties.client_port
```

To view detection property names, view detection types with the `GET /detections/formats` operation in the ExtraHop REST API.

Tip: In the trigger editor, you can view valid detection properties with the autocomplete functionality if you include logic that determines the detection type. For example, if the trigger contains the following code, and you type a period after "properties", the trigger editor displays the valid properties for the Anonymous FTP Auth Enabled detection:

```
if (Detection.type === 'anonymous_ftp') {
    Detection.properties
```

```
}
```



`resolution:` ***String***

The resolution of the ticket associated with the detection. Valid values are `action_taken` and `no_action_taken`.

`riskScore:` ***number*** | ***null***

The risk score of the detection.

`startTime:` ***Number***

The time that the detection started, expressed in milliseconds since the epoch.

`status:` ***String*** | ***null***

The status of the ticket associated with the detection. Valid string values are `acknowledged`, `new`, `in_progress`, and `closed`. The value is `null` if no status has been specified for the detection. On the Detections page, null statuses appear as `Open`.

`ticketId:` ***String***

The ID of the ticket associated with the detection.

`title:` ***String***

The title of the detection.

`type:` ***String***

The type of detection. For custom detections, "custom" is prepended to the user-defined string. For example, if you specify `brute_force_attack` in the `commitDetection` function, the detection type is `custom.brute_force_attack`.

`updateTime:` ***Number***

The last time that the detection was updated, expressed in milliseconds since the epoch.

`Device`

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

Methods

`Device(id:` ***String****`)`

Constructor for the Device object that accepts one parameter, which is a unique 16-character string ID.

If supplied with an ID from an existing Device object, the constructor creates a copy of that object with all of the object properties, as shown in the following example:

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

Metrics committed to a Device object through a `metricAdd*` function are persisted to the datastore

`lookupByIP(addr:` ***IPAddress*** | ***String****`, vlan:` ***Number****`):` ***Device***

Returns the L3 device that matches the specified IP address and VLAN ID. Returns `null` if no match is found.

`addr:` ***IPAddress*** | ***String***

The IP address for the device. The IP address can be specified as an IPAddress object or as a string.

`vlan:` ***number***

The VLAN ID for the device. Returns a default value of `0` if a VLAN ID is not provided or if the value of the `devices_across_vlans` settings is set to `true` in the running configuration file.

`lookupByMAC(addr:` ***String***`, vlan:` ***Number***`):` ***Device***

Returns the L2 device that matches the specified MAC address and VLAN ID. Returns `null` if no match is found.

`addr:` ***String***

The MAC address for the device.

`vlan:` ***Number***

The VLAN ID for the device. Returns a default value of `0` if a VLAN ID is not provided or if the value of the `devices_across_vlans` settings is set to `true` in the running configuration file.

`toString():` ***String***

Returns the Device object as a string in the following format:

```
[object Device <discovery_id>]
```

Instance methods

The methods described in this section are present only on instances of the Device class. The majority of the methods enable you to create device-level custom metrics, as shown in the following example:

```
Flow.server.device.metricAddCount("slow_rsp", 1);
```

Note: A device might sometimes act as a client and sometimes as a server on a flow.

- Call a method as `Device.metricAdd*` to collect data for both device roles.
- Call a method as `Flow.client.device.metricAdd*` to collect data only for the client role, regardless of whether the trigger is assigned to the client or the server.
- Call a method as `Flow.server.device.metricAdd*` to collect data only for the server role, regardless of whether the trigger is assigned to the client or the server.

`equals(device:` ***Device***`):` ***Boolean***

Performs an equality test between Device objects, where `device` is the object to be compared against.

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

Creates a custom top-level count metric. Commits the metric data to the specified device.

`metric_name:` ***String***

The name of the top-level count metric.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDetailCount(metric_name: String, key: String | IPAddress, count: Number, options: Object):void`

Creates a custom detail count metric by which you can drill down. Commits the metric data to the specified device.

`metric_name: String`

The name of the detail count metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`count: Number`

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following property:

`highPrecision: Boolean`

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

`metricAddDataset(metric_name: String, val: Number, options: Object):void`

Creates a custom top-level dataset metric. Commits the metric data to the specified device.

`metric_name: String`

The name of the top-level dataset metric.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`freq: Number`

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

`highPrecision: Boolean`

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

`metricAddDetailDataset(metric_name: String, key: String | IPAddress, val: Number, options: Object):void`

Creates a custom detail dataset metric by which you can drill down. Commits the metric data to the specified device.

`metric_name: String`

The name of the detail count metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`freq: Number`

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

highPrecision: ***Boolean***

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

`metricAddDistinct(metric_name:` ***String***`, item:` ***Number*** | ***String*** | ***IPAddress***`:void`

Creates a custom top-level distinct count metric. Commits the metric data to the specified device.

`metric_name:` ***String***

The name of the top-level distinct count metric.

`item:` ***Number*** | ***String*** | ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddDetailDistinct(metric_name:` ***String***`, key:` ***String*** | ***IPAddress***`, item:` ***Number*** | ***String*** | ***IPAddress***`:void`

Creates a custom detail distinct count metric by which you can drill down. Commits the metric data to the specified device.

`metric_name:` ***String***

The name of the detail distinct count metric.

`key:` ***String*** | ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`item:` ***Number*** | ***String*** | ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddMax(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level maximum metric. Commits the metric data to the specified device.

`metric_name:` ***String***

The name of the top-level maximum metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

highPrecision: ***Boolean***

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

`metricAddDetailMax(metric_name:` ***String***`, key:` ***String*** | ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail maximum metric by which you can drill down. Commits the metric data to the specified device.

`metric_name:` ***String***

The name of the detail maximum metric.

`key:` ***String*** | ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

highPrecision: ***Boolean***

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

`metricAddSampleset(metric_name: String, val: Number, options: Object):void`

Creates a custom top-level sampleset metric. Commits the metric data to the specified device.

`metric_name: String`

The name of the top-level sampleset metric.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddDetailSampleset(metric_name: String, key: String | IPAddress, val: Number, options: Object):void`

Creates a custom detail sampleset metric by which you can drill down. Commits the metric data to the specified device.

`metric_name: String`

The name of the detail sampleset metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddSnap(metric_name: String, count: Number, options: Object):void`

Creates a custom top-level snapshot metric. Commits the metric data to the specified device.

`metric_name: String`

The name of the top-level snapshot metric.

`count: Number`

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddDetailSnap(metric_name: String, key: String | IPAddress, count: Number, options: Object):void`

Creates a custom detail snapshot metric by which you can drill down. Commits the metric data to the specified device.

`metric_name: String`

The name of the detail sampleset metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

Instance properties

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

`cdpName:` ***String***

The CDP name associated with the device, if present.

`dhcpName:` ***String***

The DHCP name associated with the device, if present.

`discoverTime:` ***Number***

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

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

`dnsNames:` ***Array***

An array of strings listing the DNS names associated with the device, if present.

`hasTrigger:` ***Boolean***

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

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

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

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

`hwaddr:` ***String***

The MAC address of the device, if present.

`id:` ***String***

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

`ipaddrs:` ***Array***

An array of IPAddress objects representing the device's known IP addresses. For L3 devices, the array always contains one IPAddress.

`isGateway:` ***Boolean***

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

`isL3:` ***Boolean***

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

Important: If you have not enabled the ExtraHop system to discover devices by IP address, the isL3 property is always set to False because the system does not make a distinction between L3 child and L2 parent devices.

`netbiosName:` ***String***

The NetBIOS name associated with the device, if present.

`vlanId:` ***Number***

The VLAN ID for the device.

Trigger Examples

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

`Discover`

The `Discover` class enables you to retrieve information about newly discovered devices and applications.

Events

`NEW_APPLICATION`

Runs when an application is first discovered. This event consumes capture resources.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

`NEW_DEVICE`

Runs when activity is first observed on a device. This event consumes capture resources.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

Properties

`application:` ***Application***

A newly discovered application.

Applies only to `NEW_APPLICATION` events.

`device:` ***Device***

A newly discovered device.

Applies only to `NEW_DEVICE` events.

Note: You cannot specify this property as a participant in the `commitDetection` function.

Trigger Examples

- Example: Send discovered device data to a remote syslog server

ExternalData

The ExternalData class enables you to retrieve data sent from external sources to the Trigger API through the ExtraHop REST API.

Events

EXTERNAL_DATA

Runs every time data is sent to the ExtraHop system through the POST triggers/externaldata operation.

Properties

body: ***String***

The external data sent to the trigger.

type: ***String***

An identifier that describes the data sent to the trigger. The type is defined when the data is sent to the ExtraHop REST API.

Flow

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

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

Events

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

FLOW_CLASSIFY

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

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

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

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

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

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

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

FLOW_DETACH

Runs when the parser has encountered an unexpected error or has run out of memory and stops following the flow. In addition, a low quality data feed with missing packets can cause the parser to detach.

The FLOW_DETACH event is useful for detecting malicious content sent by clients and servers. The following is an example of how a trigger can detect bad DNS responses upon FLOW_DETACH events:

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

FLOW_RECORD

Enables you to record information about a flow at timed intervals. After 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 in the Administration settings.

FLOW_TICK

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

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

The following is an example of FLOW_TICK:

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

FLOW_TURN

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

FLOW_TURN also exposes a Turn object.

Endpoints

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

- client

- server
- sender
- receiver

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

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

Event	Client / Server	Sender / Receiver
AAA_REQUEST	yes	yes
AAA_RESPONSE	yes	yes
AJP_REQUEST	yes	yes
AJP_RESPONSE	yes	yes
ACTIVEMQ_MESSAGE	no	yes
CIFS_REQUEST	yes	yes
CIFS_RESPONSE	yes	yes
DB_REQUEST	yes	yes
DB_RESPONSE	yes	yes
DHCP_REQUEST	yes	yes
DHCP_RESPONSE	yes	yes
DICOM_REQUEST	yes	yes
DICOM_RESPONSE	yes	yes
DNS_REQUEST	yes	yes
DNS_RESPONSE	yes	yes
FIX_REQUEST	yes	yes
FIX_RESPONSE	yes	yes
FLOW_CLASSIFY	yes	no
FLOW_DETACH	yes	no
FLOW_RECORD	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
HTTP_REQUEST	yes	yes

Event	Client / Server	Sender / Receiver
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
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
MOBUS_REQUEST	yes	yes
MODBUS_RESPONSE	yes	yes
MONGODB_REQUEST	yes	yes
MONGODB_RESPONSE	yes	yes
MSMQ_MESSAGE	no	yes
NFS_REQUEST	yes	yes
NFS_RESPONSE	yes	yes
POP3_REQUEST	yes	yes
POP3_RESPONSE	yes	yes
REDIS_REQUEST	yes	yes
REDIS_RESPONSE	yes	yes
RDP_CLOSE	yes	no
RDP_OPEN	yes	no
RDP_TICK	yes	no
RTCP_MESSAGE	no	yes
RTP_CLOSE	no	yes
RTP_OPEN	no	yes
RTP_TICK	no	yes
SCCP_MESSAGE	no	yes
SIP_REQUEST	yes	yes

Event	Client / Server	Sender / Receiver
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
WEBSOCKET_OPEN	yes	no
WEBSOCKET_CLOSE	yes	no
WEBSOCKET_MESSAGE	yes	yes

Endpoint methods

`commitRecord(): void`

Sends a record to the configured recordstore on a FLOW_RECORD event. Record commits are not supported on FLOW_CLASSIFY, FLOW_DETACH, FLOW_TICK, or FLOW_TURN events.

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

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

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

Endpoint properties

`bytes: Number`

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

Access only on FLOW_TICK, FLOW_TURN, or FLOW_RECORD events; otherwise, an error will occur.

`customDevices:` ***Array***

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

`device:` ***Device***

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

`equals:` ***Boolean***

Performs an equality test between Device objects.

`dscp:` ***Number***

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

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

`dscpBytes:` ***Array***

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

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

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

`dscpName1:` ***String***

The name associated with the DSCP value transmitted by device1 in the flow. The following table displays well-known DSCP names:

Number	Name
8	CS1
10	AF11
12	AF12
14	AF13
16	CS2
18	AF21
20	AF22
22	AF23
24	CS3
26	AF31
28	AF32
30	AF33
32	CS4
34	AF41
36	AF42

Number	Name
38	AF43
40	CS5
44	VA
46	EF
48	CS6
56	CS7

`dscpName2:` ***String***

The name associated with the DSCP value transmitted by device2 in the flow. The following table displays well-known DSCP names:

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

`dscpPkts:` ***Array***

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

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

Applies only to FLOW_TICK or FLOW_TURN events.

`fragPkts:` ***Number***

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

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

`ipaddr1:` ***IPAddress***

The IPAddress object associated with device1 in the flow.

`equals:` ***Boolean***

Performs an equality test between IPAddress objects.

`ipaddr2:` ***IPAddress***

The IPAddress object associated with device2 in the flow.

`equals:` ***Boolean***

Performs an equality test between IPAddress objects.

`isAborted:` ***Boolean***

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

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

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

`isShutdown:` ***Boolean***

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

`l2Bytes:` ***Number***

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

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

`nagleDelay:` ***Number***

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

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

`offender:` ***Object***

Returns an offender participant object for a device in the flow. Specify this property in the `commitDetection()` function to identify the device in the flow as the offender in a detection, as shown in the following code example:

```
commitDetection('exampledetection', {
    participants: [Flow.client.offender, Flow.server.victim],
```

`overlapFragPkts:` ***Number***

The number of non-identical IP fragment packets with overlapping data transmitted by a device in the flow. Specify the device role in the syntax—for example, `Flow.client.overlapFragPkts` or `Flow.server.overlapFragPkts`.

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

`overlapSegments:` ***Number***

The number of non-identical TCP segments, transmitted by a device in the flow, where two or more TCP segments contain data for the same part of the flow. Specify the device role in the syntax—for example, `Flow.client.overlapSegments` or `Flow.server.overlapSegments`.

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

`payload:` ***Buffer***

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

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

`pkts:` ***Number***

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

Access only on `FLOW_TICK`, `FLOW_TURN`, or `FLOW_RECORD` events; otherwise, an error will occur.

`port:` ***Number***

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

`rcvWndThrottle:` ***Number***

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

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

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Flow.commitRecord()` on a `FLOW_RECORD` event. The record object represents data from a single direction on the flow.

The default record object can contain the following properties:

- `age`
- `bytes (L3)`

  Note: This property represents the total number of bytes that were transmitted by the flow at the time that the FLOW_RECORD event ran. The FLOW_RECORD event runs several times over the course of each flow, so the value will increase every time the event runs.
- `clientIsExternal`

- dscpName
- first
- firstPayloadBytes

  A hexadecimal representation of the first 16 payload bytes in the flow.
- last
- pkts
- proto
- receiverAddr
- receiverIsExternal
- receiverPort
- roundTripTime

  The most recent round trip time (RTT) in this flow. An RTT is the time it took for a device to send a packet and receive an immediate acknowledgment (ACK).
- senderAddr
- senderIsExternal
- senderPort
- serverIsExternal
- tcpFlags

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

Access the record object only on `FLOW_RECORD` events; otherwise, an error will occur.

`rto:` ***Number***

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

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

`victim:` ***Object***

Returns an victim participant object for a device in the flow. Specify this property in the `commitDetection()` function to identify the device in the flow as the victim in a detection, as shown in the following code example:

```
commitDetection('exampledetection', {
    participants: [Flow.client.offender, Flow.server.victim],
```

`totalL2Bytes`

The number of L2 bytes sent by a device during the flow. Specify the device role in the syntax—for example, `Flow.client.totalL2Bytes` or `Flow.server.totalL2Bytes`.

`totalL2Bytes1:` ***Number***

The number of L2 bytes sent during the flow by device1.

`totalL2Bytes2:` ***Number***

The number of L2 bytes sent during the flow by device2.

`zeroWnd:` ***Number***

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

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

Methods

`addApplication(name: String, turnTiming: Boolean): void`

Creates an application with the specified name and collects L2-L4 metrics from the flow. The application can be viewed in the ExtraHop system and the metrics are displayed on an L4 page in the application. A flow can be associated with one or more applications at a given instant; the L2-L4 metrics collected by each application will be the same.

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

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

`captureStart(name: String, options: Object): String`

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

`name: String`

The name of the packet capture file.

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

`options: Object`

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

`maxBytes: Number`

The total maximum number of bytes.

`maxBytesLookback: Number`

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

`maxDurationMSec: Number`

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

`maxPackets: Number`

The total maximum number of packets. The maximum value might be exceeded if the trigger load is heavy.

`maxPacketsLookback: Number`

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

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

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

```
    var opts = {
        maxPackets: 20,
        maxPacketsLookback: 10
    };
    Flow.captureStart(name, opts);
}
```

Note:

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

`captureStop()`: ***Boolean***

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

`commitRecord1()`: ***void***

Sends a record to the configured recordstore that represents data sent from `device1` in a single direction on the flow.

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

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

`commitRecord2()`: ***void***

Sends a record to the configured recordstore that represents data sent from `device2` in a single direction on the flow.

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

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

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

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

`getApplications()`: ***String***

Retrieves all applications associated with the flow.

Properties

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

By default, the ExtraHop system uses loosely-initiated protocol classification, so it will try to classify flows even after the connection was initiated. Loose initiation can be turned off for ports that do not always carry the protocol traffic (for example, the wildcard port 0). For such flows, `device1`, `port1`, and `ipaddr1` represent the device with the numerically lower IP address and `device2`, `port2`, and `ipaddr2` represent the device with the numerically higher IP address.

`age`: ***Number***

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

`bytes1`: ***Number***

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

Access only on `FLOW_TICK`, `FLOW_TURN`, or `FLOW_RECORD` events; otherwise, an error will occur.

`bytes2`: ***Number***

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

Access only on `FLOW_TICK`, `FLOW_TURN`, or `FLOW_RECORD` events; otherwise, an error will occur.

`customDevices1`: ***Array***

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

`customDevices2`: ***Array***

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

`device1`: ***Device***

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

`equals`: ***Boolean***

Performs an equality test between Device objects.

`device2`: ***Device***

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

`equals`: ***Boolean***

Performs an equality test between Device objects.

`dscp1`: ***Number***

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

`dscp2:` ***Number***

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

`dscpBytes1:` ***Array***

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

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

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

`dscpBytes2:` ***Array***

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

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

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

`dscpName1:` ***String***

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

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

`dscpName2:` ***String***

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

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

`dscpPkts1:` ***Array***

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

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

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

`dscpPkts2:` ***Array***

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

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

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

`fragPkts1:` ***Number***

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

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

`fragPkts2:` ***Number***

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

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

`id:` ***String***

The unique identifier of a Flow record.

`ipaddr:` ***IPAddress***

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

`equals:` ***Boolean***

Performs an equality test between IPAddress objects.

`ipproto:` ***String***

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

`ipver:` ***String***

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

`isAborted:` ***Boolean***

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

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

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

`isExpired:` ***Boolean***

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

`isShutdown:` ***Boolean***

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

`l2Bytes1:` ***Number***

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

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

`l2Bytes2:` ***Number***

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

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

`l7proto:` ***String***

The L7 protocol associated with the flow. For known protocols, the property returns a string representing the protocol name, such as HTTP, DHCP, Memcache. For lesser-known protocols, the

property returns a string formatted as `ipproto:port—tcp:13724` or `udp:11258` For custom protocol names, the property returns a string representing the name set through the Protocol Classification section in the Administration settings.

This property is not valid during `TCP_OPEN` events.

`nagleDelay1:` ***Number***

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

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

`nagleDelay2:` ***Number***

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

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

`overlapFragPkts1:` ***Number***

The number of non-identical IP fragment packets transmitted by one of two devices in the flow; the other device is represented by `overlapFragPkts2`. The device represented by `overlapFragPkts1` remains consistent for the flow.

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

`overlapFragPkts2:` ***Number***

The number of non-identical IP fragment packets transmitted by one of two devices in the flow; the other device is represented by `overlapFragPkts1`. The device represented by `overlapFragPkts2` remains consistent for the flow.

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

`overlapSegments1:` ***Number***

The number of non-identical TCP segments where two or more segments contain data for the same part of the flow. The TCP segments are transmitted by one of two devices in the flow; the other device is represented by `overlapSegments2`. The device represented by `overlapSegments1` remains consistent for the flow.

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

`overlapSegments2:` ***Number***

The number of non-identical TCP segments where two or more segments contain data for the same part of the flow. The TCP segments are transmitted by one of two devices in the flow; the other device is represented by `overlapSegments1`. The device represented by `overlapSegments2` remains consistent for the flow.

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

`payload1:` ***Buffer***

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

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

`payload2:` ***Buffer***

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

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

`pkts1:` ***Number***

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

Access only on `FLOW_TICK`, `FLOW_TURN`, or `FLOW_RECORD` events; otherwise, an error will occur.

`pkts2:` ***Number***

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

Access only on `FLOW_TICK`, `FLOW_TURN`, or `FLOW_RECORD` events; otherwise, an error will occur.

`port1:` ***Number***

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

`port2:` ***Number***

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

`rcvWndThrottle1:` ***Number***

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

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

`rcvWndThrottle2:` ***Number***

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

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

`record1:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Flow.commitRecord1()` on a `FLOW_RECORD` event.

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

Access the record object only on `FLOW_RECORD` events; otherwise, an error will occur.

The default record object can contain the following properties:

- `age`
- `bytes (L3)`
- `clientIsExternal`
- `dscpName`
- `first`
- `last`
- `pkts`
- `proto`
- `receiverAddr`
- `receiverIsExternal`
- `receiverPort`
- `roundTripTime`

  The most recent round trip time (RTT) observed in the flow. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet.

- `senderAddr`
- `senderIsExternal`
- `senderPort`
- `serverIsExternal`
- `tcpOrigin`

  This record field is included only if the record represents traffic sent from a client or sender device.
- `tcpFlags`

`record2:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Flow.commitRecord2()` on a `FLOW_RECORD` event.

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

Access the record object only on `FLOW_RECORD` events; otherwise, an error will occur.

The default record object can contain the following properties:

- `age`
- `bytes (L3)`
- `clientIsExternal`
- `dscpName`
- `first`
- `last`
- `pkts`
- `proto`
- `receiverAddr`
- `receiverIsExternal`
- `receiverPort`
- `roundTripTime`

  The most recent round trip time (RTT) observed in the flow. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet.
- `senderAddr`
- `senderIsExternal`
- `senderPort`
- `serverIsExternal`
- `tcpOrigin`

  This record field is included only if the record represents traffic sent from a client or sender device.
- `tcpFlags`

`roundTripTime:` ***Number***

The median round trip time (RTT) observed since the last `FLOW_TICK` event ran, expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The value is `NaN` if there are no RTT samples.

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

`rto1:` ***Number***

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

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

`rto2:` ***Number***

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

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

`store:` ***Object***

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

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

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

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

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

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

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

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

Note: Because DHCP requests often occur on different flows than corresponding DHCP responses, we recommend that you combine DHCP request and response information by storing DHCP transaction IDs in the session table. For example, the following trigger code creates a metric that tracks how many DHCP discover messages received a corresponding DHCP offer message:

```
if (event === 'DHCP_REQUEST'){
```

```
    var opts = {
        expire: 30
    };
    Session.add(DHCP.txId.toString(), DHCP.msgType, opts);
}
else if (event === 'DHCP_RESPONSE'){
    var reqMsgType = Session.lookup(DHCP.txId.toString());
    if (reqMsgType && DHCP.msgType === 'DHCPOFFER') {
        Device.metricAddCount('dhcp-discover-offer', 1);
    }
}
```

`tcpOrigin:` ***IPAddress | Null***

The original IP address of the client or sender if specified by a network proxy in TCP option 28.

`vlan:` ***Number***

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

`vxlanVNI:` ***Number***

The VXLAN Network Identifier number associated with the flow. If no VXLAN tag is present, this value is set to `NaN`.

`zeroWnd1:` ***Number***

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

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

`zeroWnd2:` ***Number***

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

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

Trigger Examples

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

`FlowInterface`

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

Methods

`FlowInterface(id:` ***string***`)`

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

Instance methods

The methods in this section enable you to create custom metrics on a flow interface. The methods are present only on instances of the NetFlow class. For example, the following statement collects metrics from NetFlow traffic on the ingress interface:

```
NetFlow.ingressInterface.metricAddCount("slow_rsp", 1);
```

However, you can call the FlowInterface method as a static method on NETFLOW_RECORD events. For example, the following statement collects metrics from NetFlow traffic on both the ingress and egress interfaces:

```
FlowInterface.metricAddCount("slow_rsp", 1);
```

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

Creates a custom top-level count metric. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the top-level count metric.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDetailCount(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, count:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail count metric by which you can drill down. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDataset(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level dataset metric. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the top-level dataset metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

freq: ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

highPrecision: ***Boolean***

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

`metricAddDetailDataset(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail dataset metric by which you can drill down. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

freq: ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

highPrecision: ***Boolean***

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

`metricAddDistinct(metric_name:` ***String***`, item:` ***Number*** `|` ***String*** `|` ***IPAddress***`:void`

Creates a custom top-level distinct count metric. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the top-level distinct count metric.

`item:` ***Number*** `|` ***String*** `|` ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddDetailDistinct(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, item:` ***Number*** `|` ***String*** `|` ***IPAddress***`:void`

Creates a custom detail distinct count metric by which you can drill down. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the detail distinct count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`item:` ***Number*** `|` ***String*** `|` ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddMax(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level maximum metric. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the top-level maximum metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailMax(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail maximum metric by which you can drill down. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the detail maximum metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddSampleset(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level sampleset metric. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the top-level sampleset metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailSampleset(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail sampleset metric by which you can drill down. Commits the metric data to the specified flow interface.

`metric_name:` ***String***

The name of the detail sampleset metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

options: ***Object***

An optional object that can contain the following properties:

highPrecision: ***Boolean***

A flag that enables one-second granularity for the custom metric when set to true.

metricAddSnap(metric_name: ***String***, count: ***Number***, options: ***Object***):void

Creates a custom top-level snapshot metric. Commits the metric data to the specified flow interface.

metric_name: ***String***

The name of the top-level snapshot metric.

count: ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A NaN value is silently discarded.

options: ***Object***

An optional object that can contain the following properties:

highPrecision: ***Boolean***

A flag that enables one-second granularity for the custom metric when set to true.

metricAddDetailSnap(metric_name: ***String***, key: ***String*** | ***IPAddress***, count: ***Number***, options: ***Object***):void

Creates a custom detail snapshot metric by which you can drill down. Commits the metric data to the specified flow interface.

metric_name: ***String***

The name of the detail sampleset metric.

key: ***String*** | ***IPAddress***

The key specified for the detail metric. A null value is silently discarded.

count: ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A NaN value is silently discarded.

options: ***Object***

An optional object that can contain the following properties:

highPrecision: ***Boolean***

A flag that enables one-second granularity for the custom metric when set to true.

Instance properties

id: ***String***

A string that uniquely identifies the flow interface.

number: ***Number***

The flow interface number reported by the NetFlow record.

FlowNetwork

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

Methods

FlowNetwork(id: ***string***)

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

Instance methods

The methods in this section enable you to create custom metrics on a flow network. The methods are present only on instances of the NetFlow class. For example, the following statement collects metrics from NetFlow traffic on an individual network:

```
NetFlow.network.metricAddCount("slow_rsp", 1);
```

However, you can call the FlowNetwork method as a static method on NETFLOW_RECORD events. For example, the following statement collects metrics from NetFlow traffic on both devices on the flow network:

```
FlowNetwork.metricAddCount("slow_rsp", 1);
```

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

Creates a custom top-level count metric. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the top-level count metric.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDetailCount(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, count:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail count metric by which you can drill down. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String | IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The increment value. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following property:

`highPrecision:` ***Boolean***

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

`metricAddDataset(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level dataset metric. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the top-level dataset metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

freq: ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

highPrecision: ***Boolean***

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

`metricAddDetailDataset(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail dataset metric by which you can drill down. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String | IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`freq:` ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

`highPrecision:` ***Boolean***

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

`metricAddDistinct(metric_name:` ***String***`, item:` ***Number*** `|` ***String*** `|` ***IPAddress***`:void`

Creates a custom top-level distinct count metric. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the top-level distinct count metric.

`item:` ***Number*** `|` ***String*** `|` ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddDetailDistinct(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, item:` ***Number*** `|` ***String*** `|` ***IPAddress***`:void`

Creates a custom detail distinct count metric by which you can drill down. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the detail distinct count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`item:` ***Number*** `|` ***String*** `|` ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddMax(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level maximum metric. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the top-level maximum metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailMax(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail maximum metric by which you can drill down. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the detail maximum metric.

`key:` ***String | IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddSampleset(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level sampleset metric. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the top-level sampleset metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailSampleset(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail sampleset metric by which you can drill down. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the detail sampleset metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddSnap(metric_name:` ***String***`, count:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level snapshot metric. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the top-level snapshot metric.

`count:` ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailSnap(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, count:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail snapshot metric by which you can drill down. Commits the metric data to the specified flow network.

`metric_name:` ***String***

The name of the detail sampleset metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

Instance properties

`id:` ***String***

A string that uniquely identifies the flow network.

`ipaddr:` ***IPAddress***

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

`GeoIP`

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

Methods

Values returned by GeoIP methods are obtained from the MaxMind GeoLite2 country or city databases unless configured otherwise by the Geomap Data Source settings in the Administration settings.

From the Geomap Data Source settings, you can upload custom databases and specify which database to reference by default for city or country lookups.

We recommend uploading only a custom city-level database if you intend to call both `GeoIP.getCountry()` and `GeoIP.getPreciseLocation()` methods in triggers. If both types of custom databases are uploaded, the ExtraHop system retrieves values for both methods from the city-level database and ignores the country-level database, which is considered to be a subset of the city-level database.

`getCountry(ipaddr:` ***IPAddress****):* ***Object***

Returns country-level detail for the specified IPAddress in an object that contains the following fields:

`continentName:` ***String***

The name of the continent, such as `Europe`, that is associated with the country from which the specified IP address originates. The value is the same as the `continentName` field returned by the `getPreciseLocation()` method.

`continentCode:` ***Number***

The code of the continent, such as EU, that is associated with the value of the `countryCode` field, according to ISO 3166. The value is the same as the `continentCode` field returned by the `getPreciseLocation()` method.

`countryName:` ***String***

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

`countryCode:` ***String***

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

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

Note: The `getCountry()` method requires 20 MB of total RAM on the ExtraHop system, which might affect system performance. The first time this method is called in any trigger, the ExtraHop system reserves the required amount of RAM unless the `getPreciseLocation()` method has already been called. The `getPreciseLocation()` method requires 100 MB of RAM, so adequate RAM will already be available to call the `getCountry()` method. The required amount of RAM is not per trigger or per method call; the ExtraHop system only reserves the required amount of RAM one time.

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

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

`getPreciseLocation(ipaddr:` ***IPAddress****):* ***Object***

Returns city-level detail for the specified IPAddress in an object that contains the following fields:

`continentName:` ***String***

The name of the continent, such as `Europe`, that is associated with the country from which the specified IP address originates. The value is the same as the `continentName` field returned by the `getCountry()` method.

`continentCode:` ***Number***

The code of the continent, such as `EU`, that is associated with the value of the `countryCode` field, according to ISO 3166. The value is the same as the `continentCode` field returned by the `getCountry()` method.

`countryName:` ***String***

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

`countryCode:` ***String***

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

`region:` ***String***

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

`city:` ***String***

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

`latitude:` ***Number***

The latitude of the IP address location.

`longitude:` ***Number***

The longitude of the of the IP address location.

`radius:` ***Number***

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

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

Note: The `getPreciseLocation()` method requires 100 MB of total RAM on the ExtraHop system, which might affect system performance. The first time this method is called in any trigger, the ExtraHop system reserves the required amount of RAM unless the `getCountry()` method has already been called. The `getCountry()` method requires 20 MB of RAM, so the ExtraHop system reserves an additional 80 MB of RAM. The required amount of RAM is not per trigger or per method call; the ExtraHop system only reserves the required amount of RAM one time.

`IPAddress`

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

Methods

`IPAddress(ip:` ***String*** `|` ***Number***`, mask:` ***Number***`)`

Constructor for the IPAddress class that takes two parameters:

`ip:` ***String***

The IP address string in CIDR format.

`mask:` ***Number***

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

Instance methods

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

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

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

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

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

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

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

`toJSON():` ***String***

Converts the IPAddress object to JSON format.

`toString():` ***String***

Converts the IPAddress object to a printable string.

Properties

`hostNames:` ***Array of Strings***

An array of hostnames associated with the IPAddress.

`isBroadcast:` ***Boolean***

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

`isExternal:` ***Boolean***

The value is `true` if the IP address is external to your network.

`isLinkLocal:` ***Boolean***

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

`isMulticast:` ***Boolean***

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

`isRFC1918:` ***Boolean***

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

`isV4:` ***Boolean***

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

`isV6:` ***Boolean***

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

`localityName:` ***String*** | ***null***

The name of the network locality that the IP address is in. If the IP address is not in any network locality, the value is null.

Network

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

Methods

metricAddCount(metric_name: *String*, count: *Number*, options: *Object*):void

Creates a custom top-level count metric. Commits the metric data to the specified network.

metric_name: *String*

The name of the top-level count metric.

count: *Number*

The increment value. Must be a non-zero, positive signed 64-bit integer. A NaN value is silently discarded.

options: *Object*

An optional object that can contain the following property:

highPrecision: *Boolean*

A flag that enables one-second granularity for the custom metric when set to true.

metricAddDetailCount(metric_name: *String*, key: *String* | *IPAddress*, count: *Number*, options: *Object*):void

Creates a custom detail count metric by which you can drill down. Commits the metric data to the specified network.

metric_name: *String*

The name of the detail count metric.

key: *String* | *IPAddress*

The key specified for the detail metric. A null value is silently discarded.

count: *Number*

The increment value. Must be a non-zero, positive signed 64-bit integer. A NaN value is silently discarded.

options: *Object*

An optional object that can contain the following property:

highPrecision: *Boolean*

A flag that enables one-second granularity for the custom metric when set to true.

metricAddDataset(metric_name: *String*, val: *Number*, options: *Object*):void

Creates a custom top-level dataset metric. Commits the metric data to the specified network.

metric_name: *String*

The name of the top-level dataset metric.

val: *Number*

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A NaN value is silently discarded.

options: *Object*

An optional object that can contain the following properties:

freq: *Number*

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the val parameter. If no value is specified, the default value is 1.

highPrecision: *Boolean*

A flag that enables one-second granularity for the custom metric when set to true.

`metricAddDetailDataset(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail dataset metric by which you can drill down. Commits the metric data to the specified network.

`metric_name:` ***String***

The name of the detail count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`freq:` ***Number***

An option that enables you to simultaneously record multiple occurrences of particular values in the dataset when set to the number of occurrences specified by the `val` parameter. If no value is specified, the default value is 1.

`highPrecision:` ***Boolean***

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

`metricAddDistinct(metric_name:` ***String***`, item:` ***Number*** `|` ***String*** `|` ***IPAddress***`:void`

Creates a custom top-level distinct count metric. Commits the metric data to the specified network.

`metric_name:` ***String***

The name of the top-level distinct count metric.

`item:` ***Number*** `|` ***String*** `|` ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

metricAddDetailDistinct (metric_name: *String*, key: *String* | *IPAddress*, item: *Number* | *String* | *IPAddress*:void

Creates a custom detail distinct count metric by which you can drill down. Commits the metric data to the specified network.

`metric_name:` ***String***

The name of the detail distinct count metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`item:` ***Number*** `|` ***String*** `|` ***IPAddress***

The value to be placed into the set. The value is converted to a string before it is placed in the set.

`metricAddMax(metric_name:` ***String***`, val:` ***Number***`, options:` ***Object***`):void`

Creates a custom top-level maximum metric. Commits the metric data to the specified network.

`metric_name:` ***String***

The name of the top-level maximum metric.

`val:` ***Number***

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailMax(metric_name: String, key: String | IPAddress, val: Number, options: Object):void`

Creates a custom detail maximum metric by which you can drill down. Commits the metric data to the specified network.

`metric_name: String`

The name of the detail maximum metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddSampleset(metric_name: String, val: Number, options: Object):void`

Creates a custom top-level sampleset metric. Commits the metric data to the specified network.

`metric_name: String`

The name of the top-level sampleset metric.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddDetailSampleset(metric_name: String, key: String | IPAddress, val: Number, options: Object):void`

Creates a custom detail sampleset metric by which you can drill down. Commits the metric data to the specified network.

`metric_name: String`

The name of the detail sampleset metric.

`key: String | IPAddress`

The key specified for the detail metric. A `null` value is silently discarded.

`val: Number`

The observed value, such as a processing time. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options: Object`

An optional object that can contain the following properties:

`highPrecision: Boolean`

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

`metricAddSnap(metric_name: String, count: Number, options: Object):void`

Creates a custom top-level snapshot metric. Commits the metric data to the specified network.

`metric_name: String`

The name of the top-level snapshot metric.

`count:` ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

`metricAddDetailSnap(metric_name:` ***String***`, key:` ***String*** `|` ***IPAddress***`, count:` ***Number***`, options:` ***Object***`):void`

Creates a custom detail snapshot metric by which you can drill down. Commits the metric data to the specified network.

`metric_name:` ***String***

The name of the detail sampleset metric.

`key:` ***String*** `|` ***IPAddress***

The key specified for the detail metric. A `null` value is silently discarded.

`count:` ***Number***

The observed value, such as current established connections. Must be a non-zero, positive signed 64-bit integer. A `NaN` value is silently discarded.

`options:` ***Object***

An optional object that can contain the following properties:

`highPrecision:` ***Boolean***

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

Trigger Examples

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

`Session`

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

Here are some important things to know about session tables:

- The session table supports ordinary JavaScript values, enabling you to add JS objects to the table.
- Session table entries can be evicted when the table grows too large or when the configured expiration is reached.
- Because the session table on a sensor is not shared with the console, the values in the session table are not shared with other connected sensors.
- The ExtraHop Open Data Context API exposes the session table via the management network, enabling coordination with external processes through the memcache protocol.

Events

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

`SESSION_EXPIRE`

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

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

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

`TIMER_30SEC`

Runs exactly every 30 seconds. This event enables you to perform periodic processing, such as regularly accessing session table entries added through the Open Data Context API.

Note: You can apply any trigger class to the TIMER_30SEC event.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

Methods

`add(key: String, value*, options: Object): *`

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

You can configure an optional Options object for the specified key.

`getOptions(key: String): Object`

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

`increment(key: String, count: Number): Number | null`

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

`lookup(key: String): *`

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

`modify(key: String, value: *, options: Object): *`

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

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

`remove(key: String): *`

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

`replace(key: String, value: *, options: Object): *`

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

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

Options

`expire:` ***Number***

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

`notify:` ***Boolean***

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

`priority:` ***String***

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

Constants

`PRIORITY_LOW:` ***Number***

The numeric representation of the lowest priority level. The value is 0. Priority levels determine the order that entries are removed from the session table if the table grows too large.

`PRIORITY_NORMAL:` ***Number***

The numeric representation of the default priority level. The value is 1. Priority levels determine the order that entries are removed from the session table if the table grows too large.

`PRIORITY_HIGH:` ***Number***

The numeric representation of the highest priority level. The value is 2. Priority levels determine the order that entries are removed from the session table if the table grows too large.

Properties

`expiredKeys:` ***Array***

An array of objects with the following properties:

`age:` ***Number***

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

`name:` ***String***

The key of the expired object.

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

The value of the entry in the session table.

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

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

Trigger Examples

- Example: Record data to a session table

`System`

The `System` class enables you to retrieve information about the sensor or console on which a trigger is running. This information in useful in environments with multiple sensors.

Properties

`uuid:` ***String***

The universally unique identifier (UUID) of the sensor or console.

`ipaddr:` ***IPAddress***

The IPAddress object of the primary management interface (Interface 1) on the sensor.

`hostname:` ***String***

The hostname for the sensor or console configured in the Administration settings.

`version:` ***String***

The firmware version running on the sensor or console.

`ThreatIntel`

The `ThreatIntel` class enables you to see whether threats have been found for IP addresses, hostnames, or URIs. (ExtraHop RevealX Premium and Ultra only)

Methods

`hasIP(address:` ***IPAddress***`):` ***boolean***

The value is `true` if the threats have been found for the specified IP address. If no intelligence information is available on the ExtraHop system, the value is `null`.

`hasDomain(domain:` ***String***`):` ***boolean***

The value is `true` if the threats have been found for the specified domain. If no intelligence information is available on the ExtraHop system, the value is `null`.

`hasURI(uri:` ***String***`):` ***boolean***

The value is `true` if the threats have been found for the specified URI. If no intelligence information is available on the ExtraHop system, the value is `null`.

Properties

`isAvailable:` ***boolean***

The value is `true` if threat intelligence information is available on the ExtraHop system.

`Trigger`

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

Properties

`isDebugEnabled:` ***boolean***

The value is `true` if debugging is enabled for the trigger. The value is determined by the state of the **Enable debug log** checkbox in the Edit Trigger pane in the ExtraHop system.

`VLAN`

The `VLAN` class represents a VLAN on the network.

Instance properties

`id:` ***Number***

The numerical ID for a VLAN.

Protocol and network data classes

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

Class	Description
AAA	Enables you to store metrics and access properties on AAA_REQUEST or AAA_RESPONSE events.
ActiveMQ	Enables you to store metrics and access properties on ACTIVEMQ_MESSAGE events.
AJP	The AJP class enables you to store metrics and access properties on AJP_REQUEST and AJP_RESPONSE events.
CDP	The CDP class enables you to store metrics and access properties on CDP_FRAME events.
CIFS	Enables you to store metrics and access properties on CIFS_REQUEST and CIFS_RESPONSE events.
DB	Enables you to store metrics and access properties on DB_REQUEST and DB_RESPONSE events.
DHCP	Enables you to store metrics and access properties on DHCP_REQUEST and DHCP_RESPONSE events.
DICOM	Enables you to store metrics and access properties on DICOM_REQUEST and DICOM_RESPONSE events.
DNS	Enables you to store metrics and access properties on DNS_REQUEST and DNS_RESPONSE events.
FIX	Enables you to store metrics and access properties on FIX_REQUEST and FIX_RESPONSE events.
FTP	Enables you to store metrics and access properties on FTP_REQUEST and FTP_RESPONSE events.
HL7	Enables you to store metrics and access properties on HL7_REQUEST and HL7_RESPONSE events.
HTTP	Enables you to store metrics and access properties on HTTP_REQUEST and HTTP_RESPONSE events.
IBMMQ	Enables you to store metrics and access properties on IBMMQ_REQUEST and IBMMQ_RESPONSE events.
ICA	Enables you to store metrics and access properties on ICA_OPEN, ICA_AUTH, ICA_TICK, and ICA_CLOSE events.
ICMP	Enables you to store metrics and access properties on ICMP_MESSAGE events.

Class	Description
Kerberos	Enables you to store metrics and access properties on KERBEROS_REQUEST and KERBEROS_RESPONSE events.
LDAP	Enables you to store metrics and access properties on LDAP_REQUEST and LDAP_RESPONSE events.
LLDP	Enables you to access properties on LLDP_FRAME events.
Memcache	Enables you to store metrics and access properties on MEMCACHE_REQUEST and MEMCACHE_RESPONSE events.
Modbus	Enables you to store metrics and access properties on MODBUS_REQUEST and MODBUS_RESPONSE events.
MongoDB	The MongoDB class enables you to store metrics and access properties on MONGODB_REQUEST and MONGODB_RESPONSE events.
MSMQ	The MSMQ class enables you to store metrics and access properties on MSMQ_MESSAGE event.
NetFlow	Enables you to store metrics and access properties on NETFLOW_RECORD events.
NFS	Enables you to store metrics and access properties on NFS_REQUEST and NFS_RESPONSE events.
NTLM	Enables you to store metrics and access properties on NTLM_MESSAGE events.
POP3	Enables you to store metrics and access properties on POP3_REQUEST and POP3_RESPONSE events.
RDP	Enables you to store metrics and access properties on RDP_OPEN, RDP_CLOSE, and RDP_TICK events.
Redis	Enables you to store metrics and access properties on REDIS_REQUEST and REDIS_RESPONSE events.
RPC	Enables you to store metrics and access properties on RPC_REQUEST and RPC_RESPONSE events.
RTCP	Enables you to store metrics and access properties on RTCP_MESSAGE events.
RTP	Enables you to store metrics and access properties on RTP_OPEN, RTP_CLOSE, and RTP_TICK events.
SCCP	Enables you to store metrics and access properties on SCCP_MESSAGE events.
SDP	Enables you to access properties on SIP_REQUEST and SIP_RESPONSE events.
SFlow	Enables you to store metrics and access properties on SFLOW_RECORD events.

Class	Description
SIP	Enables you to store metrics and access properties on SIP_REQUEST and SIP_RESPONSE events.
SMPP	Enables you to store metrics and access properties on SMPP_REQUEST and SMPP_RESPONSE events.
SMTP	Enables you to store metrics and access properties on SMTP_REQUEST and SMTP_RESPONSE events.
SSH	Enables you to store metrics and access properties on SSH_CLOSE, SSH_OPEN and SSH_TICK events.
SSL	Enables you to store metrics and access properties on SSL_OPEN, SSL_CLOSE, SSL_ALERT, SSL_RECORD, SSL_HEARTBEAT, and SSL_RENEGOTIATE events.
TCP	Enables you to access properties and retrieve metrics from TCP events and on FLOW_TICK and FLOW_TURN events.
Telnet	Enables you to store metrics and access properties on TELNET_MESSAGE events.
Turn	Enables you to store metrics and access properties on FLOW_TURN events.
UDP	Enables you to access properties and retrieve metrics from UDP events and on FLOW_TICK and FLOW_TURN events.
WebSocket	Enables you to access properties on WEBSOCKET_OPEN, WEBSOCKET_CLOSE, and WEBSOCKET_MESSAGE events.

AAA

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

Events

AAA_REQUEST

Runs when the ExtraHop system finishes processing an AAA request .

AAA_RESPONSE

Runs on every AAA response processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on either an AAA_REQUEST or AAA_RESPONSE event.

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

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

Properties

`authenticator:` ***String***

The value of the authenticator field (RADIUS only).

`avps:` ***Array***

An array of AVP objects with the following properties:

`avpLength:` ***Number***

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

`id:` ***Number***

The numeric ID of the attribute represented as an integer.

`isGrouped:` ***Boolean***

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

`name:` ***String***

The name for the given AVP.

`vendor:` ***String***

The vendor name for vendor AVPs (Diameter only).

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

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

`isDiameter:` ***Boolean***

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

`isError:` ***Boolean***

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

Access only on `AAA_RESPONSE` events; otherwise, an error will occur.

`isRadius:` ***Boolean***

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

`isRspAborted:` ***Boolean***

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

Access only on `AAA_RESPONSE` events; otherwise, an error will occur.

`method:` ***Number***

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

The following table contains valid Diameter command codes:

Command name	Abbr.	Code
`AA-Request`	`AAR`	`265`
`AA-Answer`	`AAA`	`265`
`Diameter-EAP-Request`	`DER`	`268`
`Diameter-EAP-Answer`	`DEA`	`268`
`Abort-Session-Request`	`ASR`	`274`
`Abort-Session-Answer`	`ASA`	`274`
`Accounting-Request`	`ACR`	`271`

Command name	Abbr.	Code
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-Answer	RAA	258
Re-Auth-Request	RAR	258
Session-Termination-Request	STR	275
Session-Termination-Answer	STA	275
User-Authorization-Request	UAR	300
User-Authorization-Answer	UAA	300
Server-Assignment-Request	SAR	301
Server-Assignment-Answer	SAA	301
Location-Info-Request	LIR	302
Location-Info-Answer	LIA	302
Multimedia-Auth-Request	MAR	303
Multimedia-Auth-Answer	MAA	303
Registration-Termination-Request	RTR	304
Registration-Termination-Answer	RTA	304
Push-Profile-Request	PPR	305
Push-Profile-Answer	PPA	305
User-Data-Request	UDR	306
User-Data-Answer	UDA	306
Profile-Update-Request	PUR	307
Profile-Update-Answer	PUA	307
Subscribe-Notifications-Request	SNR	308
Subscribe-Notifications-Answer	SNA	308
Push-Notification-Request	PNR	309
Push-Notification-Answer	PNA	309
Bootstrapping-Info-Request	BIR	310
Bootstrapping-Info-Answer	BIA	310

Command name	Abbr.	Code
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

`processingTime:` ***Number***

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

Access only on AAA_RESPONSE events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `AAA.commitRecord()` on either an AAA_REQUEST or AAA_RESPONSE event.

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

AAA_REQUEST	AAA_RESPONSE
authenticator	authenticator
clientIsExternal	clientIsExternal
clientZeroWnd	clientZeroWnd
method	isError
receiverIsExternal	isRspAborted
reqBytes	method
reqL2Bytes	processingTime

AAA_REQUEST	AAA_RESPONSE
reqPkts	receiverIsExternal
reqRTO	roundTripTime
senderIsExternal	rspBytes
serverIsExternal	rspL2Bytes
serverZeroWnd	rspPkts
txId	rspRTO
	statusCode
	senderIsExternal
	serverIsExternal
	serverZeroWnd
	txId

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on AAA_REQUEST events; otherwise, an error will occur.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last AAA_REQUEST or AAA_RESPONSE event ran. The value is NaN if there are no RTT samples.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on AAA_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`statusCode:` ***String***

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

Access only on AAA_RESPONSE events; otherwise, an error will occur.

`txId:` ***Number***

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

ActiveMQ

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

Events

ACTIVEMQ_MESSAGE

Runs on every JMS message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an ACTIVEMQ_MESSAGE event.

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

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

Properties

`correlationId:` ***String***

The JMSCorrelationID field of the message.

`exceptionResponse:` ***Object | Null***

The JMSException field of the message. If the command of the message is not `ExceptionResponse`, the value is null. The object contains the following fields:

`message:` ***String***

The exception response message.

`class:` ***String***

The subclass of the JMSException.

`expiration:` ***Number***

The JMSExpiration field of the message.

`msg:` ***Buffer***

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

`msgFormat:` ***String***

The message format. Possible values are:

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

`msgId:` ***String***

The JMSMessageID field of the message.

`persistent:` ***Boolean***

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

`priority:` ***Number***

The JMSPriority field of the message.

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

`properties:` ***Object***

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

`queue:` ***String***

The JMSDestination field of the message.

`receiverBytes:` ***Number***

The number of application-level bytes from the receiver.

`receiverIsBroker:` ***Boolean***

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

`receiverL2Bytes:` ***Number***

The number of L2 bytes from the receiver.

`receiverPkts:` ***Number***

The number of packets from the receiver.

`receiverRTO:` ***Number***

The number of RTOs from the receiver.

`receiverZeroWnd:` ***Number***

The number of zero windows sent by the receiver.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `ActiveMQ.commitRecord()` on an `ACTIVEMQ_MESSAGE` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `correlationId`
- `expiration`
- `msgFormat`
- `msgId`
- `persistent`
- `priority`
- `queue`
- `receiverBytes`
- `receiverIsBroker`
- `receiverIsExternal`
- `receiverL2Bytes`
- `receiverPkts`
- `receiverRTO`
- `receiverZeroWnd`

- redeliveryCount
- replyTo
- roundTripTime
- senderBytes
- senderIsBroker
- senderIsExternal
- senderL2Bytes
- senderPkts
- senderRTO
- senderZeroWnd
- serverIsExternal
- 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. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `ACTIVEMQ_MESSAGE` event ran. The value is `NaN` if there are no RTT samples.

`senderBytes:` ***Number***

The number of application-level bytes from the sender.

`senderIsBroker:` ***Boolean***

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

`senderL2Bytes:` ***Number***

The number of L2 bytes from the sender.

`senderPkts:` ***Number***

The number of packets from the sender.

`senderRTO:` ***Number***

The number of RTOs from the sender.

`senderZeroWnd:` ***Number***

The number of zero windows sent by the sender.

`timestamp:` ***Number***

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

`totalMsgLength:` ***Number***

The length of the message, expressed in bytes.

AJP

Apache JServ Protocol (AJP) proxies inbound requests from a web server to an application server and is often applied to load-balanced environments where one or more front-end web servers feed requests into one or more application servers. The `AJP` class enables you to store metrics and access properties on `AJP_REQUEST` and `AJP_RESPONSE` events.

Events

`AJP_REQUEST`

Runs after the web server sends an AJP Forward Request message to a servlet container, and then transfers any subsequent request body.

`AJP_RESPONSE`

Runs after a servlet container sends an AJP End Response message to signal that the servlet container has finished processing an AJP Forward Request and has sent back the requested information.

Methods

`commitRecord()`: ***Void***

Sends a record to the configured recordstore on an `AJP_RESPONSE` event. Record commits on `AJP_REQUEST` events are not supported.

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

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

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

Accesses AJP header values and returns an array of header objects (with name and value properties) where the names match the prefix of the specified string. Accesses request headers on `AJP_REQUEST` events and response headers on `AJP_RESPONSE` requests.

Properties

`attributes:` ***Object***

An array of optional AJP attributes sent with the request, such as remote_user, auth_type, query_string, jvm_route, ssl_cert, ssl_cipher, and ssl_session.

Access only on `AJP_REQUEST` events; otherwise, an error will occur.

`fwdReqClientAddr:` ***IPAddress***

The IPAddress of the HTTP client that made the original request to the server. The value is `null` if the available information cannot be parsed to an IP address.

`fwdReqHost:` ***String***

The HTTP host specified by the HTTP client that made the original request to the server.

`fwdReqIsEncrypted:` ***Boolean***

The value is `true` if TLS encryption was applied by the HTTP client that made the original request to the server.

`fwdReqServerName:` ***String***

The name of the server to which the HTTP client made the original request.

fwdReqServerPort: ***Number***

The TCP port on the server to which the HTTP client made the original request.

`headers:` ***Object***

When accessed on `AJP_REQUEST` events, an array of header names and values sent with the request.

When accessed on `AJP_RESPONSE` events, an array of headers conveyed in the AJP Send Headers message by the server to the end user browser.

`method:` ***String***

The HTTP method of the request, such as POST or GET, from the server to the servlet container.

`processingTime:` ***Number***

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

Access only on `AJP_RESPONSE` events; otherwise, an error will occur.

`protocol:` ***String***

The protocol of the request from the server to the servlet container. Not set for other message types.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `AJP.commitRecord()` on an `AJP_RESPONSE` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `fwdReqClientAddr`
- `fwdReqHost`
- `fwdReqIsEncrypted`
- `fwdReqServerName`
- `fwdReqServerPort`
- `method`
- `processingTime`
- `protocol`
- `receiverIsExternal`
- `reqSize`
- `rspSize`
- `statusCode`
- `senderIsExternal`
- `serverIsExternal`
- `uri`

Access only on `AJP_RESPONSE` events; otherwise, an error will occur.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `AJP_RESPONSE` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes, excluding AJP headers.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `AJP_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on AJP_RESPONSE events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on AJP_RESPONSE events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on AJP_RESPONSE events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding AJP headers.

Access only on AJP_RESPONSE events; otherwise, an error will occur.

`statusCode:` ***Number***

The HTTP status code returned by the servlet container for responses to AJP Forward Request messages.

Access only on AJP_RESPONSE events; otherwise, an error will occur.

`uri:` ***String***

The URI for the request from the server to the servlet container. Not set for non-AJP message types.

BACnet

The Building Automation Control Network (BACnet) class enables you to store metrics and access properties on BACNET_MESSAGE events.

Events

BACKNET_MESSAGE

Runs on every BACnet message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a BACNET_MESSAGE event. To view the default properties committed to the record object, see the record property below.

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

Properties

`dstAddr:` ***Buffer | Null***

A Buffer object containing the address of the destination device. The value is null if the Network Protocol Data Unit (NPDU) does not specify a destination address.

`dstNetwork:` ***Buffer | Null***

A buffer object containing the ID of the destination network. The value is null if the NPDU does not specify a destination network.

`hopCount:` ***Number | Null***

A field specified in the network protocol data unit (NPDU) that tracks how many network hops the BACnet message has passed through. The value starts at 255 and decrements with each network hop.

`invokeId:` ***Number | Null***

The ID of the BACnet request message, which correlates the request with the response. The value is null if the request does not require a response.

`pduType:` ***String***

The application protocol data unit (APDU) type.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `BACnet.commitRecord()` on a `BACNET_MESSAGE` event.

The default record object can contain the following properties:

- `application`
- `dstAddr`
- `dstNetwork`
- `flowId`
- `hopCount`
- `invokeId`
- `pduType`
- `proto`
- `receiver`
- `receiverAddr`
- `receiverIsExternal`
- `receiverPort`
- `sender`
- `senderAddr`
- `senderIsExternal`
- `senderPort`
- `serviceChoice`
- `srcAddr`
- `srcNetwork`
- `vlan`

`serviceChoice:` ***Number***

The numeric identifier for the requested BACnet service.

`srcAddr:` ***Buffer | Null***

A Buffer object that contains the address of the source device. The value is null if the NPDU does not specify a source address.

`srcNetwork:` ***Buffer | Null***

A Buffer object that contains the ID of the source network. The value is null if the NPDU does not specify a source network.

`CDP`

Cisco Discovery Protocol (CDP) is a proprietary protocol that enables connected Cisco devices to send information to each other. The `CDP` class enables you to access properties on `CDP_FRAME` events.

Events

`CDP_FRAME`

Runs on every CDP frame processed by the device.

Properties

`destination:` ***String***

The destination MAC address. The most common destination is `01:00:0c:cc:cc:cc`, indicating a multicast address.

`checksum:` ***Number***

The CDP checksum.

`source:` ***Device***

The device sending the CDP 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.

`tlvs:` ***Array of Objects***

An array containing each type, length, value (TLV) field. A TLV field contains information such as the device ID, address, and platform. Each field is an object with the following properties:

`type:` ***Number***

The type of TLV.

`value:` ***Buffer***

The value of the TLV.

`version:` ***Number***

The CDP protocol version.

`CIFS`

The `CIFS` class enables you to store metrics and access properties on `CIFS_REQUEST` and `CIFS_RESPONSE` events.

Events

`CIFS_REQUEST`

Runs on every SMB request processed by the device.

`CIFS_RESPONSE`

Runs on every SMB response processed by the device.

Note: The `CIFS_RESPONSE` event runs after every `CIFS_REQUEST` event, even if the corresponding response is never observed by the ExtraHop system.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a `CIFS_RESPONSE` event. Record commits on `CIFS_REQUEST` events are not supported.

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

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

Properties

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

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

`accessMask:` ***Number***

A numeric representation of the hexadecimal number that specifies the access mask for the request.

Access only on `CIFS_REQUEST` events; otherwise, an error will occur.

`accessTime:` ***Number***

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

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`createOptions:` ***Number***

A numeric representation of the hexadecimal number that specifies the options for creating or opening a file.

Access only on `CIFS_REQUEST` events; otherwise, an error will occur.

`dialect:` ***String***

The dialect of SMB negotiated between the client and the server.

`encryptedBytes:` ***Number***

The number of encrypted bytes in the request or response.

`encryptionProtocol:` ***String***

The protocol that the transaction is encrypted with.

`error:` ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`filename:` ***String***

The name of the file being transferred.

`isCommandCreate:` ***Boolean***

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

`isCommandClose:` ***Boolean***

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

`isCommandDelete:` ***Boolean***

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

`isCommandFileInfo:` ***Boolean***

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

`isCommandLock:` ***Boolean***

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

`isCommandRead:` ***Boolean***

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

`isCommandRename:` ***Boolean***

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

`isCommandWrite:` ***Boolean***

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

`isDecrypted:` ***Boolean***

The value is true if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`isEncrypted:` ***Boolean***

The value is true if the transaction is encrypted.

`isRspAborted:` ***Boolean***

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

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`isRspSigned:` ***Boolean***

The value is true if the response is signed by the SMB server.

`method:` ***String***

The SMB method. Correlates to the methods listed under the SMB metric in the ExtraHop system.

`msgID:` ***Number***

The SMB transaction identifier.

`payload:` ***Buffer***

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

The buffer contains the *N* first bytes of the payload, where *N* is the number of payload bytes specified by the L7 Payload Bytes to Buffer option when the trigger was configured through the ExtraHop WebUI. The default number of bytes is 2048. For more information, see Advanced trigger options.

Note: The buffer cannot contain more than 4 KB, even if the L7 Payload Bytes to Buffer option is set to a higher value.

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

`payloadMediaType:` ***String | Null***

The type of media contained in the payload. The value is null if there is no payload or the media type is unknown.

`payloadOffset:` ***Number***

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

`payloadSHA256:` ***String | Null***

The hexadecimal representation of the SHA-256 hash of the payload. The string contains no delimiters, as shown in the following example:

```
468c6c84db844821c9ccb0983c78d1cc05327119b894b5ca1c6a1318784d3675
```

If there is no payload, the value is null.

`processingTime:` ***Number***

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

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `CIFS.commitRecord` on a `CIFS_RESPONSE` event.

The default record object can contain the following properties:

- accessTime
- clientIsExternal
- clientZeroWnd
- error
- isCommandCreate
- isCommandDelete
- isCommandFileInfo
- isCommandLock
- isCommandRead
- isCommandRename
- isCommandWrite
- isHighEntropy
- method
- processingTime
- receiverIsExternal
- reqPayloadMediaType
- reqPayloadSHA256
- reqSize
- reqXfer
- resource
- rspBytes
- rspPayloadMediaType
- rspPayloadSHA256
- rspXfer
- senderIsExternal
- serverIsExternal
- serverZeroWnd
- share
- statusCode
- user
- warning

Access only on CIFS_RESPONSE events; otherwise, an error will occur.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on CIFS_RESPONSE events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on CIFS_RESPONSE events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of request packets.

Access only on CIFS_RESPONSE events; otherwise, an error will occur.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on CIFS_RESPONSE events; otherwise, an error will occur.

`reqSize:` ***Number***

The number of L7 request bytes, excluding SMB headers.

`reqTransferTime:` ***Number***

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

Access only on `CIFS_REQUEST` events; otherwise, an error will occur.

`reqVersion:` ***String***

The version of SMB running on the request.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`resource:` ***String***

The share, path, and filename, concatenated together.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `CIFS_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding SMB headers.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspTransferTime:` ***Number***

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

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspVersion:` ***String***

The version of SMB running on the response.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`sessionId:` ***Number***

The ID of the SMB session.

`share:` ***String***

The name of the share the user is connected to.

`statusCode:` ***Number***

The numeric status code of the response (SMB1 and SMB2 only).

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

`user:` ***String***

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

`warning:` ***String***

The detailed warning message recorded by the ExtraHop system.

Access only on `CIFS_RESPONSE` events; otherwise, an error will occur.

Trigger Examples

- Example: Monitor SMB actions on devices

DB

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

Events

`DB_REQUEST`

Runs on every database request processed by the device.

`DB_RESPONSE`

Runs on every database response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a `DB_RESPONSE` event. Record commits on `DB_REQUEST` events are not supported.

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

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

Properties

`appName:` ***String***

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

`correlationId:` ***Number***

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

`database:` ***String***

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

`encryptionProtocol:` ***String***

The protocol that the transaction is encrypted with.

`error:` ***String***

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`errors:` ***Array of strings***

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`isDecrypted:` ***Boolean***

The value is true if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`isEncrypted:` ***Boolean***

The value is true if the transaction is encrypted.

`isReqAborted:` ***Boolean***

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

`isRspAborted:` ***Boolean***

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`method:` ***String***

The database method that correlates to the methods listed under the Database metric in the ExtraHop system.

`params:` ***Array***

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

The array contains each of the following parameters:

`name:` ***String***

The optional name of the supplied RPC parameter.

`value:` ***String | Number***

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

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

`procedure:` ***String***

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

`processingTime:` ***Number***

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `DB.commitRecord` on a `DB_RESPONSE` event.

The default record object can contain the following properties:

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of request packets.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`reqSize:` ***Number***

The number of L7 request bytes, excluding database protocol headers.

`reqTimeToLastByte:` ***Number***

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

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `DB_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding database protocol headers.

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspTimeToFirstByte:` ***Number***

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on `DB_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`serverVersion:` ***String***

The MS SQL server version.

`statement:` ***String***

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

`table:` ***String***

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

- `Sybase`
- `Sybase IQ`
- `MySQL`
- `PostgreSQL`
- `IBM Informix`
- `MS SQL TDS`

- `Oracle TNS`
- `DB2`

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

`user:` ***String***

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

Trigger Examples

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

DHCP

The DHCP class enables you to store metrics and access properties on `DHCP_REQUEST` and `DHCP_RESPONSE` events.

Events

`DHCP_REQUEST`

Runs on every DHCP request processed by the device.

`DHCP_RESPONSE`

Runs on every DHCP response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either a `DHCP_REQUEST` or `DHCP_RESPONSE` event.

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

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

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

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

`code:` ***Number***

The DHCP option code.

`name:` ***String***

The DHCP option name.

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

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

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

Properties

`chaddr:` ***String***

The client hardware address of the DHCP client.

`clientReqDelay:` ***Number***

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

Access only on `DHCP_REQUEST` events; otherwise, an error will occur.

`error:` ***String***

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

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`gwAddr:` ***IPAddress***

The IP address through which routers relay request and response messages.

`htype:` ***Number***

The hardware type code.

`msgType:` ***String***

The DHCP message type. Supported message types are:

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

`offeredAddr:` ***IPAddress***

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

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`options:` ***Array of Objects***

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

`code:` ***Number***

The DHCP option code.

`name:` ***String***

The DHCP option name.

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

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

`paramReqList:` ***String***

A comma-separated list of numbers that represents the DHCP options requested from the server by the client. For a complete list of DHCP options, see https://www.iana.org/assignments/bootp-dhcp-parameters/bootp-dhcp-parameters.xhtml.

`processingTime:` ***Number***

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

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `DHCP.commitRecord` on either a `DHCP_REQUEST` or `DHCP_RESPONSE` event.

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

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

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of request packets.

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `DHCP_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on DHCP_RESPONSE events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on DHCP_RESPONSE events; otherwise, an error will occur.

`txId:` ***Number***

The transaction ID.

`vendor:` ***String***

The Vendor Class Identifier (VCI) that specifies the vendor running on the client or server.

DICOM

The DICOM (Digital Imaging and Communications in Medicine) class enables you to store metrics and access properties on DICOM_REQUEST and DICOM_RESPONSE events.

Events

`DICOM_REQUEST`

Runs on every DICOM request processed by the device.

`DICOM_RESPONSE`

Runs on every DICOM response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a DICOM_REQUEST or DICOM_RESPONSE event.

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

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

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

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

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

`groupTag:` ***Number***

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

`elementTag:` ***Number***

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

Properties

`calledAETitle:` ***String***

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

`callingAETitle:` ***String***

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

`elements:` ***Array***

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

`error:` ***String***

The detailed error message recorded by the ExtraHop system.

`isReqAborted:` ***Boolean***

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

Access only on `DICOM_REQUEST` events; otherwise, an error will occur.

`isRspAborted:` ***Boolean***

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

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`isSubOperation:` ***Boolean***

The value is `true` if the timing metric on an L7 protocol message is not available because the primary request or response is not complete.

`methods:` ***Array of Strings***

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

`processingTime:` ***Number***

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

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `DICOM.commitRecord` on either a `DICOM_REQUEST` or `DICOM_RESPONSE` event.

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

DICOM_REQUEST	DICOM_RESPONSE
calledAETitle	calledAETitle
callingAETitle	callingAETitle
clientIsExternal	clientIsExternal
clientZeroWnd	clientZeroWnd
error	error
isReqAborted	isRspAborted
isSubOperation	isSubOperation
method	method
receiverIsExternal	processingTime
reqPDU	receiverIsExternal
reqSize	rspPDU
reqTransferTime	rspSize
senderIsExternal	rspTransferTime
serverIsExternal	senderIsExternal

DICOM_REQUEST	DICOM_RESPONSE
serverZeroWnd	serverIsExternal
version	serverZeroWnd
	version

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `DICOM_REQUEST` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPDU:` ***String***

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

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes.

Access only on `DICOM_REQUEST` events; otherwise, an error will occur.

`reqTransferTime:` ***Number***

The request transfer time, expressed in milliseconds.

Access only on `DICOM_REQUEST` events; otherwise, an error will occur.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `DICOM_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPDU:` ***String***

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

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspSize:` ***Number***

The number of L7 response bytes.

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`rspTransferTime:` ***Number***

The response transfer time, expressed in milliseconds.

Access only on `DICOM_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`version:` ***Number***

The DICOM version number.

DNP3

The Distributed Network Protocol (DNP3) class enables you to store metrics and access properties on `DNP3_REQUEST` and `DNP3_RESPONSE` events.

Events

`DNP3_REQUEST`

Runs on every DNP3 request processed by the device.

`DNP3_RESPONSE`

Runs on every DNP3 response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a `DNP3_RESPONSE` event. Record commits on `DNP3_REQUEST` events are not supported.To view the default properties committed to the record object, see the record property below.

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

Properties

`control:` ***Number***

The numeric code that specifies control flags for the request or response.

`dstAddr:` ***Number***

The address of the station that the request or response is being sent to.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `DNP3.commitRecord` on a `DNP3_RESPONSE` event.

The default record object can contain the following properties:

- `application`
- `client`
- `clientAddr`
- `clientIsExternal`
- `clientPort`
- `control`
- `dstAddr`

- flowId
- receiverIsExternal
- senderIsExternal
- server
- serverAddr
- serverIsExternal
- serverPort
- srcAddr
- vlan

Access only on DNP3_RESPONSE events; otherwise, an error will occur.

reqPayload: ***Buffer***

A Buffer object that contains the raw payload bytes of the request.

rspPayload: ***Buffer***

A Buffer object that contains the raw payload bytes of the response.

Access only on DNP3_RESPONSE events; otherwise, an error will occur.

srcAddr: ***Number***

The address of the station that the request or response is being sent from.

DNS

The DNS class enables you to store metrics and access properties on DNS_REQUEST and DNS_RESPONSE events.

Events

DNS_REQUEST

Runs on every DNS request processed by the device.

DNS_RESPONSE

Runs on every DNS response processed by the device.

Methods

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

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

Can be called only on DNS_RESPONSE events.

commitRecord(): ***void***

Sends a record to the configured recordstore on a DNS_REQUEST or DNS_RESPONSE event.

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

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

Properties

answers: ***Array***

An array of objects that correspond to answer resource records.

Access only on DNS_RESPONSE events; otherwise, an error will occur.

The objects contain the following properties:

`data:` ***String*** | ***IPAddress***

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

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

`name:` ***String***

The record name.

`ttl:` ***Number***

The time-to-live value.

`type:` ***String***

The DNS record type.

`typeNum:` ***Number***

The numeric representation of the DNS record type.

`error:` ***String***

The name of the DNS error code, in accordance with IANA DNS parameters.

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

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`errorNum:` ***Number***

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

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`isAuthenticData:` ***Boolean***

The value is `true` if the response was validated through DNSSEC.

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`isAuthoritative:` ***Boolean***

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

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`isCheckingDisabled:` ***Boolean***

The value is `true` if a response should be returned even though the request could not be authenticated.

Access only on `DNS_REQUEST` events; otherwise, an error will occur.

`isDGADomain:` ***Boolean***

The value is `true` if the domain of the server might have been generated by a domain generation algorithm (DGA). Some forms of malware produce large numbers of domain names with DGAs to hide command and control servers. The value is `null` if the domain was not suspicious.

`isRecursionAvailable:` ***Boolean***

The value is `true` if the name server supports recursive queries.

Access only on DNS_RESPONSE events; otherwise, an error will occur.

`isRecursionDesired:` ***Boolean***

The value is `true` if the name server should perform the query recursively.

Access only on DNS_REQUEST events; otherwise, an error will occur.

`isReqTimeout:` ***Boolean***

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

Access only on DNS_REQUEST events; otherwise, an error will occur.

`isRspTruncated:` ***Boolean***

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

Access only on DNS_RESPONSE events; otherwise, an error will occur.

`opcode:` ***String***

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

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

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

`opcodeNum:` ***Number***

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

`payload:` ***Buffer***

The Buffer object that contains the raw payload bytes of the event transaction.

`processingTime:` ***Number***

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

Access only on DNS_RESPONSE events; otherwise, an error will occur.

`qname:` ***String*** | ***null***

The hostname queried.

This value is `null` if the `opcode` property is UPDATE.

`qtype:` ***String*** | ***null***

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

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

This value is `null` if the `opcode` property is UPDATE.

`qtypeNum:` ***Number*** | ***null***

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

This value is `null` if the `opcode` property is `UPDATE`.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `DNS.commitRecord()` on either a `DNS_REQUEST` or `DNS_RESPONSE` event.

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

DNS_REQUEST	DNS_RESPONSE
clientIsExternal	answers
clientZeroWnd	clientIsExternal
isCheckingDisabled	clientZeroWnd
isDGADomain	error
isRecursionDesired	isAuthoritative
isReqTimeout	isCheckingDisabled
opcode	isDGADomain
qname	isRecursionAvailable
qtype	isRspTruncated
receiverIsExternal	opcode
reqBytes	processingTime
reqL2Bytes	receiverIsExternal
reqPkts	qname
senderIsExternal	qtype
serverIsExternal	rspBytes
serverZeroWnd	rspL2Bytes
	rspPkts
	senderIsExternal
	serverIsExternal
	serverZeroWnd

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `DNS_REQUEST` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on `DNS_REQUEST` events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of request packets.

Access only on `DNS_REQUEST` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of application-level response bytes.

Access only on `DNS_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`txId:` ***Number***

The transaction ID of the DNS request or response.

`zname:` ***String*** | ***null***

The DNS zone being updated.

This value is `null` if the `opcode` property is not `UPDATE`.

`ztype:` ***String*** | ***null***

The type of DNS zone being updated. Returns `OTHER` for types that are unrecognized by the system.

This value is `null` if the `opcode` property is not `UPDATE`.

`ztypeNum:` ***Number*** | ***null***

The numeric representation of the DNS zone type.

This value is `null` if the `opcode` property is not `UPDATE`.

`FIX`

The FIX class enables you to store metrics and access properties on `FIX_REQUEST` and `FIX_RESPONSE` events.

Events

`FIX_REQUEST`

Runs on every FIX request processed by the device.

`FIX_RESPONSE`

Runs on every FIX response processed by the device.

Note: The `FIX_RESPONSE` event is matched with a request based on order ID. There is no one-to-one correlation between request and response. There might be requests without a response, and sometimes data is pushed to the client, which limits request data availability on response event. However, you can invoke the session table to solve complex scenarios such as submission order id.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either a `FIX_REQUEST` or `FIX_RESPONSE` event.

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

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

Properties

`fields:` ***Array***

A list of FIX fields. Because 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.

In the following 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 enter 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 is display in the trigger's Debug Log:

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

`msgType:` ***String***

The value of the MessageCompID key.

`processingTime:` ***Number***

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

Access only on `FIX_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `FIX.commitRecord` on either a `FIX_REQUEST` or `FIX_RESPONSE` event.

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

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

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`sender:` ***String***

The value of the SenderCompID key.

`target:` ***String***

The value of the TargetCompID key.

`version:` ***String***

The protocol version.

FTP

The FTP class enables you to store metrics and access properties on `FTP_REQUEST` and `FTP_RESPONSE` events.

Events

`FTP_REQUEST`

Runs on every FTP request processed by the device.

`FTP_RESPONSE`

Runs on every FTP response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `FTP_RESPONSE` event. Record commits on `FTP_REQUEST` events are not supported.

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

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

Properties

`args:` ***String***

The arguments to the command.

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`cwd:` ***String***

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

- The value is `/` when method == "CWD".
- The value is `/subdir` for subsequent commands (rather than CWD becoming the changed to directory as part of the CWD response trigger).

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

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

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`error:` ***string***

The detailed error message recorded by the ExtraHop system.

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`filename:` ***String***

The name of the file being transferred.

`isReqAborted:` ***Boolean***

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

`isRspAborted:` ***Boolean***

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

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`method:` ***String***

The FTP method.

`path:` ***String***

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

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`payloadMediaType:` ***String | Null***

The type of media contained in the payload. The value is null if there is no payload or the media type is unknown.

`processingTime:` ***Number***

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

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `FTP.commitRecord()` on an `FTP_RESPONSE` event.

The default record object can contain the following properties:

- `args`
- `clientIsExternal`
- `clientZeroWnd`
- `cwd`
- `error`
- `isReqAborted`
- `isRspAborted`
- `method`
- `path`
- `processingTime`
- `receiverIsExternal`
- `reqBytes`
- `reqL2Bytes`
- `reqPayloadMediaType`
- `reqPayloadSHA256`
- `reqPkts`
- `reqRTO`
- `roundTripTime`
- `rspBytes`
- `rspL2Bytes`
- `rspPayloadMediaType`
- `rspPayloadSHA256`
- `rspPkts`

- rspRTO
- senderIsExternal
- serverIsExternal
- serverZeroWnd
- statusCode
- transferBytes
- user

Access the record object only on FTP_RESPONSE events; otherwise, an error will occur.

reqBytes: ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

reqL2Bytes: ***Number***

The number of L2 request bytes, including L2 headers.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

reqPkts: ***Number***

The number of request packets.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

Access only on FTP_RESPONSE events; otherwise, an error will occur.

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last FTP_RESPONSE event ran. The value is NaN if there are no RTT samples.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

rspBytes: ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

rspL2Bytes: ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

rspPkts: ***Number***

The number of response packets.

Access only on FTP_RESPONSE events; otherwise, an error will occur.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

Access only on FTP_RESPONSE events; otherwise, an error will occur.

rspZeroWnd: ***Number***

The number of zero windows in the response.

`statusCode:` ***Number***

The FTP status code of the response.

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

The following codes are valid:

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

Code	Description
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.

`transferBytes:` ***Number***

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

Access only on `FTP_RESPONSE` events; otherwise, an error will occur.

`user:` ***String***

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

`HL7`

The HL7 class enables you to store metrics and access properties on `HL7_REQUEST` and `HL7_RESPONSE` events.

Events

`HL7_REQUEST`

Runs on every HL7 request processed by the device.

`HL7_RESPONSE`

Runs on every HL7 response processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on an `HL7_RESPONSE` event. Record commits on `HL7_REQUEST` events are not supported.

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

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

Properties

`ackCode`: ***String***

The two character acknowledgment code.

Access only on `HL7_RESPONSE` events; otherwise, an error will occur.

`ackId`: ***String***

The identifier for the message being acknowledged.

Access only on `HL7_RESPONSE` events; otherwise, an error will occur.

`msgId`: ***String***

The unique identifier for this message.

`msgType`: ***String***

The entire message type field, including the msgId subfield.

`processingTime`: ***Number***

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

Access only on `HL7_RESPONSE` events; otherwise, an error will occur.

`record`: ***Object***

The record object that can be sent to the configured recordstore through a call to `HL7.commitRecord()` on an `HL7_RESPONSE` event.

The default record object can contain the following properties:

- `ackCode`
- `ackId`
- `clientIsExternal`
- `clientZeroWnd`
- `msgId`
- `msgType`
- `receiverIsExternal`
- `roundTripTime`
- `processingTime`
- `senderIsExternal`
- `serverIsExternal`
- `serverZeroWnd`
- `version`

Access the record object only on `HL7_RESPONSE` events; otherwise, an error will occur.

`reqZeroWnd`: ***Number***

The number of zero windows in the request.

`roundTripTime`: ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet.

The median value is calculated by sampling the RTTs observed since the last HL7_RESPONSE event ran. The value is NaN if there are no RTT samples.

Access only on HL7_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`segments:` ***Array***

An array of segment objects with the following fields:

`name:` ***String***

The name of the segment.

`fields:` ***Array of Strings***

The segment field values. Because the indices of the array start at 0, and HL7 field numbers start at 1, the index is the HL7 field number minus 1. For example, to select field 16 of a PRT segment (the participation device ID), specify 15, as shown in the following example code:

```
HL7.segments[5].fields[15]
```

Note: If a segment is blank, the array contains an empty string at the segment index.

`subfieldDelimiter:` ***String***

Supports non-standard field delimiters.

`version:` ***String***

The version advertised in the MSH segment.

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

HTTP

The HTTP class enables you to store metrics and access properties on HTTP_REQUEST and HTTP_RESPONSE events.

Events

`HTTP_REQUEST`

Runs on every HTTP request processed by the device.

`HTTP_RESPONSE`

Runs on every HTTP response processed by the device.

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

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an HTTP_REQUEST or HTTP_RESPONSE event. To view the default properties committed to the record object, see the `record` property below.

If the `commitRecord()` method is called on an HTTP_REQUEST event, the record is not created until the HTTP_RESPONSE event runs. If the `commitRecord()` method is called on both the HTTP_REQUEST and the corresponding HTTP_RESPONSE, only one record is created for request and response, even if the `commitRecord()` method is called multiple times on the same trigger events.

`findHeaders(name: String): Array`

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

`parseQuery(String): Object`

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

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

Note: If the query string contains repeated keys, the corresponding values are returned in an array. For example, the query string `event_type=status_update_event&event_type=api_post_event` returns the following object:

```
{
  "event_type": ["status_update_event", "api_post_event"]
}
```

Properties

`age: Number`

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

`contentType: String`

The value of the content-type HTTP header.

`cookies: Array`

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

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

`encryptionProtocol: String`

The protocol that the transaction is encrypted with.

`filename: String | Null`

The name of the file being transferred. If the HTTP request or response did not transfer a file, the value is null.

`headers: Object`

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

`length: Number`

The number of headers.

`string property:`

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

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

`numeric property:`

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

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

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

`headersRaw:` ***String***

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

`host:` ***String***

The value in the HTTP host header.

`isClientReset:` ***Boolean***

The value is `true` if the HTTP/2 stream is reset by the client. If the protocol is HTTP1.1, the value is `false`.

`isContinued:` ***Boolean***

The value is `true` if the client sent an initial HTTP/1.1 request with an `Expect: 100-continue` header and received a 100 status code from the server as part of the transaction. If the protocol is HTTP/2, the value is `false`

`isDesync:` ***Boolean***

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

`isEncrypted:` ***Boolean***

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

`isDecrypted:` ***Boolean***

The value is `true` if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`isPipelined:` ***Boolean***

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

`isReqAborted:` ***Boolean***

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

`isRspAborted:` ***Boolean***

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

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`isRspChunked:` ***Boolean***

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

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`isRspCompressed:` ***Boolean***

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

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`isServerPush:` ***Boolean***

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

`isServerReset:` ***Boolean***

The value is `true` if the HTTP/2 stream is reset by the server.

`isSQLi:` ***Boolean***

The value is true if the request included one or more suspicious SQL fragments. These fragments indicate a potential SQL injection (SQLi). SQLi is a technique where an attacker can access and tamper with data by inserting malicious SQL statements into a SQL query.

`isXSS:` ***Boolean***

The value is true if the HTTP request included potential cross-site scripting (XSS) attempts. A successful XSS attempt can inject a malicious client-side script or payload into a trusted website or application. When a victim visits the website, the malicious script is then injected into the victim's browser.

`method:` ***String***

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

`oauthBearerToken:` ***String***

The OAuth token sent by the client to the server for authorization.

`origin:` ***IPAddress | String***

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

`path:` ***String***

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

`payload:` ***Buffer | Null***

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

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

The following script is an example of HTTP payload analysis:

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

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

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

`payloadParts:` ***Array of Objects | Null***

An array of objects that contain the individual payloads of a multipart HTTP request or response. The value is null if the content type is not multipart. Each object contains the following fields:

`headers:` ***Object***

A header object that specifies HTTP headers. For more information, see the description of the `HTTP.headers` property.

`payloadSHA256:` ***String***

The hexadecimal representation of the SHA-256 hash of the payload. The string contains no delimiters.

`payloadMediaType:` ***String | Null***

The media type of the payload. The value is null if the media type is unknown.

`payload:` ***Buffer***

The Buffer object containing the raw payload bytes.

`size:` ***Number***

The size of the payload, expressed in bytes.

`filename:` ***String***

The filename specified in the Content-Disposition header.

`processingTime:` ***Number***

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

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`query:` ***String***

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

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `HTTP.commitRecord()`.

The default record object can contain the following properties:

- `clientIsExternal`
- `clientZeroWnd`
- `contentType`
- `host`
- `isPipelined`
- `isReqAborted`
- `isRspAborted`
- `isRspChunked`
- `isRspCompressed`
- `method`
- `oauthAlgorithm`
- `oauthAudience`
- `oauthClientId`
- `oauthIssuer`
- `oauthJWTId`
- `origin`
- `query`
- `receiverIsExternal`
- `referer`
- `reqBytes`
- `reqL2Bytes`

- reqPayloadMediaType
- reqPayloadSHA256
- reqPkts
- reqRTO
- reqSize
- reqTimeToLastByte
- roundTripTime
- rspBytes
- rspL2Bytes
- rspPayloadMediaType
- rspPayloadSHA256
- rspPkts
- rspRTO
- rspSize
- rspTimeToFirstHeader
- rspTimeToFirstPayload
- rspTimeToLastByte
- rspVersion
- samlRspAudience
- samlRspCertificateSubject
- samlRspDigestMethodAlgorithm
- samlRspIssuer
- samlRspNameID
- samlRspSignatureMethodAlgorithm
- samlRspStatusCode
- senderIsExternal
- serverIsExternal
- serverZeroWnd
- statusCode
- thinkTime
- title
- processingTime
- uri
- userAgent

Access the record object only on HTTP_RESPONSE events; otherwise, an error will occur.

`referer:` ***String***

The value in the HTTP referrer header.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of request packets.

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`reqSize:` ***Number***

The number of L7 request bytes, excluding HTTP headers.

`reqTimeToLastByte:` ***Number***

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

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `HTTP_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding HTTP headers.

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspTimeToFirstHeader:` ***Number***

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

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspTimeToFirstPayload:` ***Number***

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

Access only on `HTTP_RESPONSE` events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`rspVersion:` ***String***

The HTTP version of the response.

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`samlRequestXML:` ***Buffer | Null***

The Buffer object that contains the raw XML bytes of the SAML request. If the HTTP request or response did not contain a SAML request, the value is `null`.

`samlResponseXML:` ***Buffer | Null***

The Buffer object that contains the raw XML bytes of the SAML response. If the HTTP request or response did not contain a SAML response, the value is `null`.

`sqli:` ***Array of Strings***

An array of suspicious SQL fragments included in the request. These fragments might contain a potential SQL injection (SQLi). SQLi is a technique where an attacker can access and tamper with data by inserting malicious SQL statements into a SQL query.

`statusCode:` ***Number***

The HTTP status code of the response.

Access only on HTTP_RESPONSE events; otherwise, an error will occur.

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

`streamId:` ***Number***

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

`title:` ***String***

The value in the title element of the HTML content, if present. If the title was compressed, the decompressed content is returned.

`thinkTime:` ***Number***

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

`uri:` ***String***

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

`userAgent:` ***String***

The value in the HTTP user-agent header.

`xss:` ***Array of Strings***

An array of suspicious HTTP request fragments included in the request. These fragments might inject a malicious client-side script or payload into a trusted website or application. When a victim visits the website, the malicious script is then injected into the victim's browser.

Trigger Examples

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

IBMMQ

The IBMMQ class enables you to store metrics and access properties on IBMMQ_REQUEST and IBMMQ_RESPONSE events.

Note: The IBMMQ protocol supports EBCDIC encoding.

Events

IBMMQ_REQUEST

Runs on every IBMMQ request processed by the device.

IBMMQ_RESPONSE

Runs on every IBMMQ response processed by the device.

Methods

commitRecord(): ***void***

Sends a record to the configured recordstore on either an IBMMQ_REQUEST or IBMMQ_RESPONSE event.

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

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

Properties

channel: ***String***

The communication channel name.

conversationId: ***Number***

The identifier for the MQ conversation.

correlationId: ***String***

The IBMMQ correlation ID.

error: ***String***

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

method: ***String***

The wire protocol request or response method name.

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

ExtraHop	Wireshark
ASYNC_MSG_V7	ASYNC_MESSAGE
MQCLOSEv7	SOCKET_ACTION
MQGETv7	REQUEST_MSGS
MQGETv7_REPLY	NOTIFICATION

msg: ***Buffer***

A Buffer object containing MQPUT, MQPUT1, MQGET_REPLY, ASYNC_MSG_V7, and MESSAGE_DATA messages.

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

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

`msgFormat:` ***String***

The message format.

`msgId:` ***String***

The IBMMQ message ID.

`pcfError:` ***String***

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

`pcfMethod:` ***String***

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

`pcfWarning:` ***String***

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

`putAppName:` ***String***

The application name associated with the MQPUT message.

`queue:` ***String***

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

`queueMgr:` ***String***

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

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `IBMMQ.commitRecord()` on either an `IBMMQ_REQUEST` or `IBMMQ_RESPONSE` event.

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

IBMMQ_REQUEST	IBMMQ_RESPONSE
channel	channel
clientIsExternal	clientIsExternal
clientZeroWnd	clientZeroWnd
correlationId	correlationId
msgId	error
method	msgId
msgFormat	method
msgSize	msgFormat
queue	msgSize
queueMgr	queue
receiverIsExternal	queueMgr

IBMMQ_REQUEST	IBMMQ_RESPONSE
reqBytes	receiverIsExternal
reqL2Bytes	resolvedQueue
reqPkts	resolvedQueueMgr
reqRTO	roundTripTime
resolvedQueue	rspBytes
resolvedQueueMgr	rspL2Bytes
senderIsExternal	rspPkts
serverIsExternal	rspRTO
serverZeroWnd	senderIsExternal
	serverIsExternal
	serverZeroWnd
	warning

`reqBytes:` ***Number***

The number of application-level request bytes.

`reqL2Bytes:` ***Number***

The number of L2 request bytes.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`resolvedQueue:` ***String***

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

`resolvedQueueMgr:` ***String***

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

`rfh:` ***Array of Strings***

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

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `IBMMQ_REQUEST` or `IBMMQ_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

`rspBytes:` ***Number***

The number of application-level response bytes.

`rspL2Bytes:` ***Number***

The number of L2 response bytes.

`rspPkts:` ***Number***

The number of request packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`totalMsgLength:` ***Number***

The total length of the message, expressed in bytes.

`warning:` ***String***

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

Trigger Examples

- Example: Collect IBMMQ metrics

ICA

The ICA class enables you to store metrics and access properties on `ICA_OPEN`, `ICA_AUTH`, `ICA_TICK`, and `ICA_CLOSE` events.

Events

`ICA_AUTH`

Runs when the ICA authentication is complete.

`ICA_CLOSE`

Runs when the ICA session is closed.

`ICA_OPEN`

Runs immediately after the ICA application is initially loaded.

`ICA_TICK`

Runs periodically while the user interacts with the ICA application.

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

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either an `ICA_OPEN`, `ICA_TICK`, or `ICA_CLOSE` event. Record commits on `ICA_AUTH` events are not supported.

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

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

Properties

`application:` ***String***

The name of the application being launched.

`authDomain:` ***String***

The Windows authentication domain to which the user belongs.

`channels:` ***Array***

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

Access only on `ICA_TICK`events; otherwise, an error will occur.

Each object contains the following properties:

`name:` ***String***

The name of the virtual channel.

`description:` ***String***

The friendly description of the channel name.

`clientBytes:` ***Number***

The total number of bytes sent by the client for the channel since the last `ICA_TICK` event ran.

`serverBytes:` ***Number***

The total number of bytes sent by the server for the channel since the last `ICA_TICK` event ran.

`clientMachine:` ***String***

The name of the client machine. The name is displayed by the ICA client and is typically the hostname of the client machine.

`clientBytes:` ***Number***

The total number of bytes sent by the client since the last `ICA_TICK` event ran. Note that this property does not return the total number of bytes for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`clientCGPMsgCount:` ***Number***

The number of client CGP messages since the last `ICA_TICK` event.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`clientLatency:` ***Number***

The latency of the client, expressed in milliseconds, as reported by the End User Experience Management (EUEM) beacon.

Client latency is reported when a packet from the client on the EUEM channel reports the result of a single ICA round trip measurement.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`clientL2Bytes:` ***Number***

The total number of L2 client bytes observed since the last `ICA_TICK` event ran. Note that this property does not return the total number of bytes for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`clientMsgCount:` ***Number***

The number of client messages since the last `ICA_TICK` event.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`clientPkts:` ***Number***

The total number of packets sent by the client since the last `ICA_TICK` event ran. Note that this property does not return the total number of packets for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`clientRTO:` ***Number***

The total number of client retransmission timeouts (RTOs) observed since the last `ICA_TICK` event ran. Note that this property does not return the total number of client RTOs for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`clientZeroWnd:` ***Number***

The total number of zero windows sent by the client since the last `ICA_TICK` event ran. Note that this property does not return the total number of zero windows for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`clientType:` ***String***

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

`clipboardData:` ***Buffer***

A Buffer object containing raw data from the clipboard transfer.

The value is `null` if the `ICA_TICK` event did not result from a clipboard data transfer, or if the channel specified by the `tickChannel` property is not a clipboard channel.

The maximum number of bytes in the buffer is specified by the Clipboard Bytes to Buffer field when the trigger was configured through the ExtraHop system. The default maximum object size is 1024 bytes. For more information, see the Advanced trigger options.

To determine the direction of the clipboard data transfer, access this property through `Flow.sender`, `Flow.receiver`, `Flow.client`, or `Flow.server`.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`clipboardDataType:` ***String***

The type of data on the clipboard transfer. The following clipboard types are supported:

- `TEXT`
- `BITMAP`
- `METAFILEPICT`
- `SYMLINK`
- `DIF`
- `TIFF`
- `OEMTEXT`
- `DIB`
- `PALLETTE`
- `PENDATA`
- `RIFF`
- `WAVE`
- `UNICODETEXT`
- `EHNMETAFILE`
- `OWNERDISPLAY`
- `DSPTEXT`
- `DSPBITMAP`
- `DSPMETAFILEPICT`
- `DSPENHMETAFILE`

The value is `null` if the `ICA_TICK` event did not result from a clipboard data transfer, or if the channel specified by the `tickChannel` property is not a clipboard channel.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`frameCutDuration:` ***Number***

The frame cut duration, as reported by the EUEM beacon.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`frameSendDuration:` ***Number***

The frame send duration, as reported by the EUEM beacon.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`host:` ***String***

The host name of the Citrix server.

`isAborted:` ***Boolean***

The value is `true` if the application fails to launch successfully.

Access only on `ICA_CLOSE` events; otherwise, an error will occur.

`isCleanShutdown:` ***Boolean***

The value is `true` if the application shuts down cleanly.

Access only on `ICA_CLOSE` events; otherwise, an error will occur.

`isClientDiskRead:` ***Boolean***

The value is `true` if a file was read from the client disk to the Citrix server. The value is `null` if the command is not a file operation, or if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`isClientDiskWrite:` ***Boolean***

The value is `true` if a file was written from the Citrix server to the client disk. The value is `null` if the command is not a file operation, or if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`isEncrypted:` ***Boolean***

The value is `true` if the application is encrypted with RC5 encryption.

`isSharedSession:` ***Boolean***

The value is `true` if the application is launched over an existing connection.

`launchParams:` ***String***

The string that represents the parameters.

`loadTime:` ***Number***

The load time of the given application, expressed in milliseconds.

Note: The load time is recorded only for the initial application load. The ExtraHop system does not measure load time for applications launched over existing sessions and instead reports the initial load time on subsequent application loads. Choose `ICA.isSharedSession` to distinguish between initial and subsequent application loads.

`loginTime:` ***Number***

The user login time, expressed in milliseconds.

Access only on `ICA_OPEN`, `ICA_CLOSE`, or `ICA_TICK` events; otherwise, an error will occur.

Note: The login time is recorded only for the initial application load. The ExtraHop system does not measure login time for applications launched over existing sessions and instead reports the initial login time on subsequent application loads. Choose `ICA.isSharedSession` to distinguish between initial and subsequent application loads.

`networkLatency:` ***Number***

The current latency advertised by the client, expressed in milliseconds.

Network latency is reported when a specific ICA packet from the client contains latency information.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`payload:` ***Buffer***

The Buffer object that contains the raw payload bytes of the file that was read or written on the event.

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

The value is `null` if the channel specified by the `tickChannel` property is not a file channel.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`printerName:` ***String***

The name of the printer driver.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`program:` ***String***

The name of the program, or application, that is being launched.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `ICA.commitRecord()` on either an `ICA_OPEN`, `ICA_TICK`, or `ICA_CLOSE` event.

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

ICA_CLOSE	ICA_OPEN	ICA_TICK
authDomain	authDomain	authDomain
clientBytes	clientIsExternal	clientIsExternal
clientIsExternal	clientMachine	clientBytes
clientL2Bytes	clientType	clientCGPMsgCount
clientMachine	clientZeroWnd	clientL2Bytes
clientPkts	host	clientLatency
clientRTO	isEncrypted	clientMachine
clientType	isSharedSession	clientMsgCount
clientZeroWnd	launchParams	clientPkts
host	loadTime	clientRTO
isAborted	loginTime	clientType
isCleanShutdown	program	clientZeroWnd
isEncypted	receiverIsExternal	frameCutDuration
isSharedSession	senderIsExternal	frameSendDuration
launchParams	serverIsExternal	host
loadTime	serverZeroWnd	isClientDiskRead

ICA_CLOSE	ICA_OPEN	ICA_TICK
loginTime	user	isClientDiskWrite
program		isEncrypted
receiverIsExternal		isSharedSession
roundTripTime		launchParams
senderIsExternal		loadTime
serverBytes		loginTime
serverIsExternal		networkLatency
serverL2Bytes		program
serverPkts		receiverIsExternal
serverRTO		resource
serverZeroWnd		roundTripTime
user		senderIsExternal
		serverBytes
		serverCGPMsgCount
		serverIsExternal
		serverL2Bytes
		serverMsgCount
		serverPkts
		serverRTO
		serverZeroWnd
		tickChannel
		user

Access the record object only on ICA_OPEN, ICA_CLOSE, and ICA_TICK events; otherwise, an error will occur.

`resource:` ***String***

The path of the file that was read or written on the event, if known. The value is `null` if the channel specified by the `tickChannel` property is not a file channel.

Access only on ICA_TICK events; otherwise, an error will occur.

`resourceOffset:` ***Number***

The offset of the file that was read or written on the event, if known. The value is `null` if the channel specified by the `tickChannel` property is not a file channel.

Access only on ICA_TICK events; otherwise, an error will occur.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last ICA_CLOSE or ICA_TICK event ran. The value is `NaN` if there are no RTT samples.

Access only on ICA_CLOSE or ICA_TICK events; otherwise, an error will occur.

`serverBytes:` ***Number***

The total number of bytes sent by the client since the last `ICA_TICK` event ran. Note that this property does not return the total number of bytes for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`serverCGPMsgCount:` ***Number***

The number of CGP server messages since the last `ICA_TICK` event.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`serverL2Bytes:` ***Number***

The total number of L2 server bytes observed since the last `ICA_TICK` event ran. Note that this property does not return the total number of bytes for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`serverMsgCount:` ***Number***

The number of server messages since the last `ICA_TICK` event.

Access only on `ICA_TICK` events; otherwise, an error will occur.

`serverPkts:` ***Number***

The total number of packets sent by the server since the last `ICA_TICK` event ran. Note that this property does not return the total number of packets for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`serverRTO:` ***Number***

The total number of server retransmission timeouts (RTOs) observed since the last `ICA_TICK` event ran. Note that this property does not return the total number of server RTOs for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`serverZeroWnd:` ***Number***

The total number of zero windows sent by the server since the last `ICA_TICK` event ran. Note that this property does not return the total number of zero windows for the entire ICA session.

Access only on `ICA_CLOSE` or `ICA_TICK` events; otherwise, an error will occur.

`tickChannel:` ***String***

The name of the virtual channel that resulted in the current `ICA_TICK` event. The following channels are supported:

- **CTXCLI:** Clipboard
- **CTXCDM:** File
- **CTXEUE:** End user experience monitoring

Access only on `ICA_TICK` events; otherwise, an error will occur.

`user:` ***String***

The name of the user, if available.

ICMP

The ICMP class enables you to store metrics and access properties on `ICMP_MESSAGE` events.

Events

`ICMP_MESSAGE`

Runs on every ICMP message processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on an `ICMP_MESSAGE` event.

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

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

Properties

`gwAddr`: ***IPAddress***

For a redirect message, returns the address of the gateway to which traffic for the network specified in the internet destination network field of the original datagram's data should be sent. Returns null for all other messages.

Message	ICMPv4 Type	ICMPv6 Type
`Redirect Message`	`5`	`n/a`

`hopLimit`: ***Number***

The ICMP packet time to live or hop count.

`isError`: ***Boolean***

The value is `true` for message types in the following table.

Message	ICMPv4 Type	ICMPv6 Type
`Destination Unreachable`	`3`	`1`
`Redirect`	`5`	`n/a`
`Source Quench`	`4`	`n/a`
`Time Exceeded`	`11`	`3`
`Parameter Problem`	`12`	`4`
`Packet Too Big`	`n/a`	`2`

`isQuery`: ***Boolean***

The value is `true` for message types in the following table.

Message	ICMPv4 Type	ICMPv6 Type
`Echo Request`	`8`	`128`
`Information Request`	`15`	`n/a`
`Timestamp request`	`13`	`n/a`
`Address Mask Request`	`17`	`n/a`
`Router Discovery`	`10`	`151`
`Multicast Listener Query`	`n/a`	`130`
`Router Solicitation (NDP)`	`n/a`	`133`
`Neighbor Solicitation`	`n/a`	`135`
`ICMP Node Information Query`	`n/a`	`139`

Message	ICMPv4 Type	ICMPv6 Type
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***

The value is `true` for message types in the following table.

Message	ICMPv4 Type	ICMPv6 Type
Echo Reply	0	129
Information Reply	16	n/a
Timestamp Reply	14	n/a
Address Mask Reply	18	n/a
Multicast Listener Done	n/a	132
Multicast Listener Report	n/a	131
Router Advertisement (NDP)	n/a	134
Neighbor Advertisement	n/a	136
ICMP Node Information Response	n/a	140
Inverse Neighbor Discovery Advertisement	n/a	142
Home Agent Address Discovery Reply Message	n/a	145
Mobile Prefix Advertisement	n/a	147
Certification Path Advertisement	n/a	149

`msg:` ***Buffer***

A buffer object containing up to `message_length_max` bytes of the ICMP message. The `message_length_max` option is configured in the ICMP profile in the running config.

The following running config example changes the ICMP `message_length_max` from its default of 4096 bytes to 1234 bytes:

```
"capture": {
    "app_proto": {
        "ICMP": {
            "message_length_max": 1234
        }
    }
}
```

Tip: You can convert the buffer object to a string through the String.fromCharCode method. To view the string in the runtime log, run the JSON.stringify method, as shown in the following example code:

```
const icmp_msg = String.fromCharCode.apply(String,
 ICMP.msg);
debug('ICMP message text: ' + JSON.stringify(icmp_msg,
 null, 4));
```

You can also search the ICMP message strings with the includes and test methods, as shown in the following example code:

```
const substring_search = 'search term';
const regex_search = '^search term$';
const icmp_msg = String.fromCharCode.apply(String,
 ICMP.msg);

if (icmp_msg.includes(substring_search){
    debug('ICMP message includes substring');
}
if (regex_search.test(icmp_msg)){
    debug('ICMP message matches regex');
}
```

`msgCode:` ***Number***

The ICMP message code.

`msgId:` ***Number***

The ICMP message identifier for Echo Request, Echo Reply, Timestamp Request, Timestamp Reply, Information Request, and Information Reply messages. The value is `null` for all other message types.

The following table displays type IDs for the ICMP messages:

Message	ICMPv4 Type	ICMPv6 Type
Echo Request	8	128
Echo Reply	0	129
Timestamp Request	13	n/a
Timestamp Reply	14	n/a
Information Request	15	n/a
Information Reply	16	n/a

`msgLength:` ***Number***

The length of the ICMP message, expressed in bytes.

`msgText:` ***String***

The descriptive text for the message (for example, echo request or port unreachable).

`msgType:` ***Number***

The ICMP message type.

The following table displays the ICMPv4 message types available:

Type	Message
0	Echo Reply

Type	Message
1 and 2	Unassigned
3	Destination Unreachable
4	Source Quench
5	Redirect Message
6	Alternate Host Address (deprecated)
7	Unassigned
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)
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	Extended Echo Request
43	Extended Echo Reply

Type	Message
44-255	Unassigned

The following table displays the ICMPv6 message types available:

Type	Message
1	Destination Unreachable
2	Packet Too Big
3	Time Exceeded
4	Parameter Problem
100	Private Experimentation
101	Private Experimentation
127	Reserved for expansion of ICMPv6 error messages
128	Echo Request
129	Echo Reply
130	Multicast Listener Query
131	Multicast Listener Report
132	Multicast Listener Done
133	Router Solicitation
134	Router Advertisement
135	Neighbor Solicitation
136	Neighbor Advertisement
137	Redirect Message
138	Router Renumbering
139	ICMP Node Information Query
140	ICMP Node Information Response
141	Inverse Neighbor Discovery Solicitation Message
142	Inverse Neighbor Discovery Advertisement Message
143	Multicast Listener Discovery (MLDv2) reports
144	Home Agent Address Discovery Request Message
145	Home Agent Address Discovery Reply Message
146	Mobile Prefix Solicitation
147	Mobile Prefix Advertisement
148	Certification Path Solicitation
149	Certification Path Advertisement
150	ICMP messages utilized by experimental mobility protocols such as Seamoby

Type	Message
151	Multicast Router Advertisement
152	Multicast Router Solicitation
153	Multicast Router Termination
155	RPL Control Message
156	ILNPv6 Locator Update Message
157	Duplicate Address Request
158	Duplicate Address Confirmation
159	MPL Control Message
160	Extended Echo Request - No Error
161	Extended Echo Reply
200	Private Experimentation
201	Private Experimentation
255	Reserved for expansion of ICMPv6 informational messages

`nextHopMTU:` ***Number***

An ICMPv4 `Destination Unreachable` or an ICMPv6 `Packet Too Big` message, the maximum transmission unit of the next-hop link. The value is `null` for all other messages.

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

`original:` ***Object***

An object containing the following elements from the IP datagram that caused the ICMP message to be sent:

`ipproto:` ***String***

The IP protocol of the datagram, such as TCP, UDP, ICMP, or ICMPv6.

`ipver:` ***String***

The IP version of the datagram, such as IPv4 or IPv6.

`srcAddr:` ***IPAddress***

The IPAddress of the datagram sender.

`srcPort:` ***Number***

The port number of the datagram sender.

`dstAddr:` ***IPAddress***

The IPAddress of the datagram receiver.

`dstPort:` ***Number***

The port number of the datagram receiver.

The value is `null` if the internet header and 64 bits of the Original Data datagram is not present in the message or if the IP protocol is not TCP or UDP.

Access only on `ICMP_MESSAGE` events; otherwise, an error will occur.

`pointer:` ***Number***

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

Message	ICMPv4 Type	ICMPv6 Type
`Parameter Problem`	`12`	`4`

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `ICMP.commitRecord()` on an `ICMP_MESSAGE` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `gwAddr`
- `hopLimit`
- `msgCode`
- `msgId`
- `msgLength`
- `msgText`
- `msgType`
- `nextHopMTU`
- `pointer`
- `receiverIsExternal`
- `senderIsExternal`
- `serverIsExternal`
- `seqNum`
- `version`

`seqNum:` ***Number***

The ICMP sequence number for Echo Request, Echo Reply, Timestamp Request, Timestamp Reply, Information Request, and Information Reply messages. The value is `null` for all other messages.

`version:` ***Number***

The version of the ICMP message type, which can be ICMPv4 or ICMPv6.

`Kerberos`

The Kerberos class enables you to store metrics and access properties on `KERBEROS_REQUEST` and `KERBEROS_RESPONSE` events.

Events

`KERBEROS_REQUEST`

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

`KERBEROS_RESPONSE`

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

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either a `KERBEROS_REQUEST` or `KERBEROS_RESPONSE` event.

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

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

Properties

`addresses:` ***Array of Objects***

The addresses from which the requested ticket is valid.

Access only on KERBEROS_REQUEST events; otherwise, an error will occur.

`apOptions:` ***Object***

An object containing boolean values for each option flag in AP_REQ messages.

Access only on KERBEROS_REQUEST events; otherwise, an error will occur.

`clientPrincipalName:` ***String***

The client principal name.

`cNames:` ***Array of Strings***

The name portions of the principal identifier.

`cNameType:` ***String***

The type for the cNames field.

`cRealm:` ***String***

The client realm.

`eData:` ***Buffer***

Additional information about the error returned in the response.

Access only on KERBEROS_RESPONSE events; otherwise, an error will occur.

`error:` ***String***

The error returned.

Access only on KERBEROS_RESPONSE events; otherwise, an error will occur.

`from:` ***String***

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

Access only on KERBEROS_REQUEST events; otherwise, an error will occur.

`isAccountPrivileged:` ***Boolean***

The value is true if the account specified in the `clientPrincipalName` property is privileged.

`kdcOptions:` ***Object***

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

Access only on KERBEROS_REQUEST events; otherwise, an error will occur.

`msgType:` ***String***

The message type. Possible values are:

- AP_REP
- AP_REQ
- AS_REP
- AS_REQAUTHENTICATOR
- ENC_AS_REP_PART
- ENC_KRB_CRED_PART
- ENC_KRB_PRIV_PART
- ENC_P_REP_PART
- ENC_TGS_REP_PART

- ENC_TICKET_PART
- KRB_CRED
- KRB_ERROR
- KRB_PRIV
- KRB_SAFE
- TGS_REP
- TGS_REQ
- TICKET

`paData:` ***Array of Objects***

The pre-authentication data.

`processingTime:` ***Number***

The processing time, expressed in milliseconds.

Access only on KERBEROS_RESPONSE events; otherwise, an error will occur.

`realm:` ***String***

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

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Kerberos.commitRecord()` on either a KERBEROS_REQUEST or KERBEROS_RESPONSE event.

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

KERBEROS_REQUEST	KERBEROS_RESPONSE
clientIsExternal	clientIsExternal
cNames	cNames
cNameType	cNameType
cRealm	cRealm
clientZeroWnd	clientZeroWnd
encryptedTicketLength	encryptedTicketLength
eType	error
from	msgType
isAccountPrivileged	isAccountPrivileged
msgType	processingTime
realm	realm
receiverIsExternal	receiverIsExternal
reqBytes	roundTripTime
reqL2Bytes	rspBytes
reqPkts	rspL2Bytes
reqRTO	rspPkts
senderIsExternal	rspRTO
serverZeroWnd	senderIsExternal
sNames	serverIsExternal

KERBEROS_REQUEST	KERBEROS_RESPONSE
sNameType	sNames
ticketETypeName	sNameType
till	ticketETypeName
	serverZeroWnd

`reqETypes:` ***Array of Numbers***

An array of numbers that correspond to preferred encryption methods.

Encryption method	Number
ntlm-hash	-150
aes256-cts-hmac-sha1-96-plain	-149
aes128-cts-hmac-sha1-96-plain	-148
rc4-plain-exp	-141
rc4-plain	-140
rc4-plain-old-exp	-136
rc4-hmac-old-exp	-135
rc4-plain-old	-134
rcr-hmac-old	-133
des-plain	-132
rc4-sha	-131
rc4-lm	-130
rc4-plain2	-129
rc4-md4	-128
null	0
des-cbc-crc	1
des-cbc-md4	2
des-cbc-md5	3
des3-cbc-md5	5
des3-cbc-sha1	7
dsaWithSHA1-CmsOID	9
md5WithRSAEncryption-CmsOID	10
sha1WithRSAEncryption-CmsOID	11
rc2CBC-EnvOID	12
rsaEncryption-EnvOID	13
rsaES-OAEP-ENV-OID	14
des-ede3-cbc-Env-OID	15

Encryption method	Number
des3-cbc-sha1-kd	16
aes128-cts-hmac-sha1-96	17
aes256-cts-hmac-sha1-96	18
aes128-cts-hmac-sha256-128	19
aes256-cts-hmac-sha384-192	20
rc4-hmac	23
rc4-hmac-exp	24
camellia128-cts-cmac	25
camellia256-cts-cmac	26
subkey-keymaterial	65

`reqETypeNames:` ***Array of Strings***

An array of the preferred encryption methods.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`serverPrincipalName:` ***String***

The server principal name (SPN).

`sNames:` ***Array of Strings***

The name portions of the server principal identifier.

`sNameType:` ***String***

The type for the sNames field.

`ticket:` ***Object***

A newly generated ticket in an AP_REP message or a ticket to authenticate the client to the server in an AP_REQ message.

`till:` ***String***

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

Access only on KERBEROS_REQUEST events; otherwise, an error will occur.

LDAP

The LDAP class enables you to store metrics and access properties on LDAP_REQUEST and LDAP_RESPONSE events.

Events

LDAP_REQUEST

Runs on every LDAP request processed by the device.

LDAP_RESPONSE

Runs on every LDAP response processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on either an `LDAP_REQUEST` or `LDAP_RESPONSE` event.

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

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

Properties

`bindDN`: ***String***

The bind DN of the LDAP request.

Access only on `LDAP_REQUEST` events; otherwise, an error will occur.

`controls`: ***Array of Objects***

An array of objects containing the LDAP controls of the LDAP request. Each object contains the following properties:

`controlType`: ***String***

The OID of the LDAP control.

`criticality`: ***Boolean***

Indicates whether the control is required. If `criticality` is set to `true`, the server should process the control or fail the operation.

`controlValue`: ***Buffer***

The optional control value, which specifies additional information about how the control should be processed.

Access only on `LDAP_REQUEST` events; otherwise, an error will occur.

`dn`: ***String***

The LDAP distinguished name (DN). If no DN is set, `<ROOT>` will be returned instead.

`encryptionProtocol`: ***String***

The protocol that the transaction is encrypted with.

`error`: ***String***

The LDAP short error string as defined in the protocol (for example, noSuchObject).

Access only on `LDAP_RESPONSE` events; otherwise, an error will occur.

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

Result Code	Result String
16	noSuchAttribute
17	undefinedAttributeType
18	inappropriateMatching
19	constraintViolation
20	attributeOrValueExists
21	invalidAttributeSyntax
32	NoSuchObject
33	aliasProblem
34	invalidDNSSyntax
36	aliasDeferencingProblem
48	inappropriateAuthentication
49	invalidCredentials
50	insufficientAccessRights
51	busy
52	unavailable
53	unwillingToPerform
54	loopDetect
64	namingViolation
65	objectClassViolation
66	notAllowedOnNonLeaf
67	notAllowedOnRDN
68	entryAlreadyExists
69	objectClassModsProhibited
71	affectsMultipleDSAs
80	other

`errorDetail:` ***String***

The LDAP error detail, if available for the error type. For example, "protocolError : historical protocol version requested, use LDAPv3 instead."

Access only on LDAP_RESPONSE events; otherwise, an error will occur.

`isEncrypted:` ***Boolean***

The value is true if the transaction is encrypted with TLS.

`isDecrypted:` ***Boolean***

The value is true if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`isPasswordEmpty:` ***Boolean***

The value is true if the request does not specify a password for authentication.

Access only on LDAP_REQUEST events; otherwise, an error will occur.

`isSigned:` ***Boolean***

The value is true if the LDAP transaction has been signed by the source machine.

`method:` ***String***

The LDAP method.

`msgId:` ***Number***

The LDAP message ID, which correlates LDAP requests and responses.

`msgSize:` ***Number***

The size of the LDAP message, expressed in bytes.

`processingTime:` ***Number***

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

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

Applies only to LDAP_RESPONSE events.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to LDAP.commitRecord() on either an LDAP_REQUEST or LDAP_RESPONSE event.

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

LDAP_REQUEST	LDAP_RESPONSE
bindDN	clientIsExternal
clientIsExternal	clientZeroWnd
clientZeroWnd	dn
dn	error
isSigned	isSigned
method	errorDetail
msgSize	method
receiverIsExternal	msgSize
reqBytes	processingTime
reqL2Bytes	receiverIsExternal
reqPkts	roundTripTime
reqRTO	rspBytes
saslMechanism	rspL2Bytes
searchFilter	rspPkts

LDAP_REQUEST	LDAP_RESPONSE
searchScope	rspRTO
senderIsExternal	saslMechanism
serverIsExternal	senderIsExternal
serverZeroWnd	serverIsExternal
	serverZeroWnd

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `LDAP_REQUEST` or `LDAP_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`saslMechanism:` ***String***

The string that defines the SASL mechanism that identifies and authenticates a user to a server.

`searchAttributes:` ***Array***

The attributes to return from objects that match the filter criteria.

Access only on `LDAP_REQUEST` events; otherwise, an error will occur.

`searchFilter:` ***String***

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

Access only on `LDAP_REQUEST` events; otherwise, an error will occur.

`searchResults:` ***Array of Objects***

An array of objects containing the search results returned in an LDAP response. Each object contains the following properties:

`type:` ***String***

The type of search result.

`values:` ***Array of Buffers***

An array of Buffer objects containing the search result values.

Access only on `LDAP_REQUEST` events; otherwise, an error will occur.

`searchScope:` ***String***

The depth of a search within the search base.

Access only on `LDAP_REQUEST` events; otherwise, an error will occur.

`LLDP`

The LLDP class enables you to access properties on `LLDP_FRAME` events.

Events

`LLDP_FRAME`

Runs on every LLDP frame processed by the device.

Properties

`chassisId:` ***Buffer***

The chassis ID, obtained from the chassisId data field, or type-length-value (TLV).

`chassisIdSubtype:` ***Number***

The chassis ID subtype, obtained from the chassisID TLV.

`destination:` ***String***

The destination MAC address. The destination MAC address. The most common destinations are `01-80-C2-00-00-00`, `01-80-C2-00-00-03` and `01-80-C2-00-00-0E`, indicating multicast addresses.

`optTLVs:` ***Array***

An array containing the optional TLVs. Each TLV is an object with the following properties:

`customSubtype:` ***Number***

The subtype of an organizationally specific TLV.

`isCustom:` ***Boolean***

Returns true if the object is an organizationally specific TLV.

`oui:` ***Number***

The organizationally unique identifier for organizationally specific TLVs.

`type:` ***Number***

The type of TLV.

`value:` ***String***

The value of the TLV.

`portId:` ***Buffer***

The port ID, obtained from the portId TLV.

`portIdSubtype:` ***Number***

The port ID subtype, obtained from the portId TLV.

`source:` ***Device***

The device sending the LLDP frame.

`ttl:` ***Number***

The time to live, expressed in seconds. This is the length of time during which the information in this frame is valid, starting with when the information is received.

LLMNR

The `LLMNR` class enables you to store metrics and access properties on `LLMNR_REQUEST` and `LLMNR_RESPONSE` events.

Events

`LLMNR_REQUEST`

Runs on every LLMNR request processed by the device.

`LLMNR_RESPONSE`

Runs on every LLMNR response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `LLMNR_REQUEST` or `LLMNR_RESPONSE` event.

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

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

Properties

`answer:` ***Object***

An object that corresponds to an answer resource record.

Access only on `LLMNR_RESPONSE` events; otherwise, an error will occur.

The objects contain the following properties:

`data:` ***String | IPAddress***

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

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

`name:` ***String***

The record name.

`ttl:` ***Number***

The time-to-live value.

`type:` ***String***

The LLMNR record type.

`error:` ***String***

The name of the LLMNR error code, in accordance with IANA LLMNR parameters.

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

Access only on `LLMNR_RESPONSE` events; otherwise, an error will occur.

`errorNum:` ***Number***

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

Access only on `LLMNR_RESPONSE` events; otherwise, an error will occur.

`opcode:` ***String***

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

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

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

`opcodeNum:` ***Number***

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

`qname:` ***String***

The hostname queried.

`qtype:` ***String***

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

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

`qtypeNum:` ***Number***

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

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `LLMNR.commitRecord()` on either an `LLMNR_REQUEST` or `LLMNR_RESPONSE` event.

The default record object can contain the following properties:

LLMNR_REQUEST	LLMNR_RESPONSE
clientIsExternal	answer
opcode	clientIsExternal
qname	error

LLMNR_REQUEST	LLMNR_RESPONSE
qtype	opcode
receiverIsExternal	qname
reqBytes	qtype
reqL2Bytes	receiverIsExternal
reqPkts	rspBytes
senderIsExternal	rspL2Bytes
serverIsExternal	rspPkts
	senderIsExternal
	serverIsExternal

`reqBytes`: ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `LLMNR_REQUEST` events; otherwise, an error will occur.

`reqL2Bytes`: ***Number***

The number of L2 request bytes, including L2 headers.

Access only on `LLMNR_REQUEST` events; otherwise, an error will occur.

`reqPkts`: ***Number***

The number of request packets.

Access only on `LLMNR_REQUEST` events; otherwise, an error will occur.

`rspBytes`: ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `LLMNR_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes`: ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `LLMNR_RESPONSE` events; otherwise, an error will occur.

`rspPkts`: ***Number***

The number of application-level response bytes.

Access only on `LLMNR_RESPONSE` events; otherwise, an error will occur.

`Memcache`

The Memcache class enables you to store metrics and access properties on `MEMCACHE_REQUEST` and `MEMCACHE_RESPONSE` events.

Events

`MEMCACHE_REQUEST`

Runs on every memcache request processed by the device.

`MEMCACHE_RESPONSE`

Runs on every memcache response processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on either a `MEMCACHE_REQUEST` or `MEMCACHE_RESPONSE` event.

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

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

Properties

`accessTime`: ***Number***

The access time, expressed in milliseconds. Available only if the first key that was requested produced a hit.

Access only on `MEMCACHE_RESPONSE` events; otherwise, an error will occur.

`error`: ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `MEMCACHE_RESPONSE` events; otherwise, an error will occur.

`hits`: ***Array***

An array of objects containing the Memcache key and key size.

Access only on `MEMCACHE_RESPONSE` events; otherwise, an error will occur.

`key`: ***String | null***

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

`size`: ***Number***

The size of the value returned for the key, expressed in bytes.

`isBinaryProtocol`: ***Boolean***

The value is `true` if the request/response corresponds to the binary version of the memcache protocol.

`isNoReply`: ***Boolean***

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

Access only on `MEMCACHE_REQUEST` events; otherwise, an error will occur.

`isRspImplicit`: ***Boolean***

The value is `true` if the response was implied by a subsequent response from the server (binary protocol only).

Access only on `MEMCACHE_RESPONSE` events; otherwise, an error will occur.

`method`: ***String***

The Memcache method as recorded in Metrics section of the ExtraHop system.

`misses`: ***Array***

An array of objects containing the Memcache key.

Access only on `MEMCACHE_RESPONSE` events; otherwise, an error will occur.

`key`: ***String | null***

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

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Memcache.commitRecord()` on either a `MEMCACHE_REQUEST` or `MEMCACHE_RESPONSE` event.

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

MEMCACHE_REQUEST	MEMCACHE_RESPONSE
clientIsExternal	accessTime
clientZeroWnd	clientIsExternal
isBinaryProtocol	clientZeroWnd
isNoReply	error
method	hits
receiverIsExternal	isBinaryProtocol
reqBytes	isRspImplicit
reqL2Bytes	method
reqPkts	misses
reqRTO	receiverIsExternal
reqSize	roundTripTime
senderIsExternal	rspBytes
serverIsExternal	rspL2Bytes
serverZeroWnd	rspPkts
vbucket	rspRTO
	senderIsExternal
	serverIsExternal
	serverZeroWnd
	statusCode
	vbucket

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqKeys:` ***Array***

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

The value of the `reqKeys` property is the same when accessed on either the `MEMCACHE_REQUEST` or the `MEMCACHE_RESPONSE` event.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on MEMCACHE_REQUEST events; otherwise, an error will occur.

`reqSize:` ***Number***

The number of L7 request bytes, excluding Memcache headers. The value is NaN for requests with no payload, such as GET and DELETE.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last MEMCACHE_REQUEST or MEMCACHE_RESPONSE event ran. The value is NaN if there are no RTT samples.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on MEMCACHE_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`statusCode:` ***String***

The Memcache status code. For the binary protocol, the ExtraHop system metrics prepend the method to status codes other than NO_ERROR, but the statusCode property does not. Refer to the examples for code that matches the behavior of the ExtraHop system metrics.

Access only on MEMCACHE_RESPONSE events; otherwise, an error will occur.

`vbucket:` ***Number***

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

Trigger Examples

- Example: Record Memcache hits and misses
- Example: Parse memcache keys

Modbus

The Modbus class enables you to access properties from MODBUS_REQUEST and MODBUS_RESPONSE events. Modbus is a serial communications protocol that enables connections between multiple devices on the same network.

Events

MODBUS_REQUEST

Runs on every request sent by a Modbus client. A Modbus client in the ExtraHop system is the Modbus master device.

`MODBUS_RESPONSE`

Runs on every response sent by a Modbus server. A Modbus server in the ExtraHop system is the Modbus slave device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on a `MODBUS_RESPONSE` event. Record commits on `MODBUS_REQUEST` events are not supported.

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

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

Properties

`error`: ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `MODBUS_RESPONSE` events; otherwise, an error will occur.

`functionId`: ***Number***

The Modbus function code contained in the request or response.

Function ID	Function name
1	Read Coil
2	Read Discrete Inputs
3	Read Holding Registers
4	Read Input Registers
5	Write Single Coil
6	Write Single Holding Register
15	Write Multiple Coils
16	Write Multiple Holding Registers

`functionName`: ***String***

The name of the Modbus function code contained in the request or response.

`isReqAborted`: ***Boolean***

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

`isRspAborted`: ***Boolean***

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

Access only on `MODBUS_RESPONSE` events; otherwise, an error will occur.

`payload`: ***Buffer***

The Buffer object containing the body of the request or response.

`payloadOffset`: ***Number***

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

`processingTime`: ***Number***

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

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Modbus.commitRecord` on a MODBUS_RESPONSE event.

The default record object can contain the following properties:

- clientIsExternal
- error
- functionId
- functionName
- protocolId
- reqL2Bytes
- rspL2Bytes
- receiverIsExternal
- reqPkts
- rspPkts
- reqBytes
- rspBytes
- reqRTO
- rspRTO
- roundTripTime
- clientZeroWnd
- senderIsExternal
- serverIsExternal
- serverZeroWnd
- statusCode
- txId
- unitId

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of packets in the request.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`reqRTO:` ***Number***

The number of retransmission timeouts (RTOs) in the request.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`reqSize:` ***Number***

The number of L7 request bytes, excluding Modbus headers.

`reqTransferTime:` ***Number***

The transfer time of the request, expressed in milliseconds. If the request is contained in a single packet, the transfer time is zero. If the request spans multiple packets, the value is the amount of time between detection of the first request packet and detection of the last packet by the ExtraHop

system. A high value might indicate a large request or a network delay. The value is NaN if there is no valid measurement, or if the timing is invalid.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last MODBUS_RESPONSE event ran. The value is NaN if there are no RTT samples.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of packets in the response.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of retransmission timeouts (RTOs) in the response.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding Modbus protocol headers.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`rspTransferTime:` ***Number***

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

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`statusCode:` ***Number***

The numeric status code of the response.

Status code number	Status description
1	Illegal Function
2	Illegal Data Address
3	Illegal Data Value
4	Slave Device Failure

Status code number	Status description
5	Acknowledge
6	Slave Device Busy
7	Negative Acknowledge
8	Memory Parity Error
10	Gateway Path Unavailable
11	Gateway Target Device Failed to Respond

Access only on MODBUS_RESPONSE events; otherwise, an error will occur.

`txId:` ***Number***

The transaction identifier of the request or response.

`unitId:` ***Number***

The unit identifier of the Modbus server responding to the Modbus client.

MongoDB

The MongoDB class enables you to store metrics and access properties on MONGODB_REQUEST and MONGODB_RESPONSE events.

Events

MONGODB_REQUEST

Runs on every MongoDB request processed by the device.

MONGODB_RESPONSE

Runs on every MongoDB response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either a MONGODB_REQUEST or MONGODB_RESPONSE event.

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

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

Properties

`collection:` ***String***

The name of the database collection specified in the current request.

`database:` ***String***

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

`error:` ***String***

The detailed error message recorded by the ExtraHop system.

Access only on MONGODB_RESPONSE events; otherwise, an error will occur.

`isReqAborted:` ***Boolean***

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

`isReqTruncated:` ***Boolean***

The value is `true` if the request document(s) size is greater than the maximum payload document size.

`isRspAborted:` ***Boolean***

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

Access only on `MONGODB_RESPONSE` events; otherwise, an error will occur.

`method:` ***String***

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

`opcode:` ***String***

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

`processingTime:` ***Number***

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

Access only on `MONGODB_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `MongoDB.commitRecord()` on either a `MONGODB_REQUEST` or `MONGODB_RESPONSE` event.

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

MONGODB_REQUEST	MONGODB_RESPONSE
clientIsExternal	clientIsExternal
clientZeroWnd	clientZeroWnd
collection	collection
database	database
isReqAborted	error
isReqTruncated	isRspAborted
method	method
opcode	opcode
receiverIsExternal	processingTime
reqBytes	receiverIsExternal
reqL2Bytes	roundTripTime
reqPkts	rspBytes
reqRTO	rspL2Bytes
reqSize	rspPkts
reqTimeToLastByte	rspRTO
senderIsExternal	rspSize
serverIsExternal	rspTimeToFirstByte
serverZeroWnd	rspTimeToLastByte

MONGODB_REQUEST	MONGODB_RESPONSE
user	senderIsExternal
	serverIsExternal
	serverZeroWnd
	user

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes, excluding MongoDB headers.

`reqTimeToLastByte:` ***Number***

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

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`request:` ***Array***

An array of JS objects parsed from MongoDB request payload documents. Total document size is limited to 4K.

If BSON documents are truncated, `isReqTruncated` flag is set. Truncated values are represented as follows:

- Primitive string values like code, code with scope, and binary data are partially extracted.
- Objects and Arrays are partially extracted.
- All other primitive values like Numbers, Dates, RegExp, etc., are substituted with `null`.

If no documents are included in the request, an empty array is returned.

The value of the `request` property is the same when accessed on either the `MONGODB_REQUEST` or the `MONGODB_RESPONSE` event.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `MONGODB_REQUEST` or `MONGODB_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspSize:` ***Number***

The number of L7 response bytes, excluding MongoDB headers.

Access only on `MONGODB_RESPONSE` events; otherwise, an error will occur.

`rspTimeToFirstByte:` ***Number***

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

Access only on `MONGODB_RESPONSE` events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on `MONGODB_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`user:` ***String***

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

`MSMQ`

The MSMQ class enables you to store metrics and access properties on `MSMQ_MESSAGE` events.

Events

`MSMQ_MESSAGE`

Runs on every MSMQ user message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `MSMQ_MESSAGE` event.

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

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

Properties

`adminQueue:` ***String***

The name of the administration queue of the message.

`correlationId:` ***Buffer***

The application-generated correlation ID of the message.

`dstQueueMgr:` ***String***

The destination message broker of the message.

`isEncrypted:` ***Boolean***

The value is `true` if the payload is encrypted.

`label:` ***String***

The label or description of the message.

`msgClass:` ***String***

The message class of the message. The following values are valid:

- `MQMSG_CLASS_NORMAL`
- `MQMSG_CLASS_ACK_REACH_QUEUE`
- `MQMSG_CLASS_NACK_ACCESS_DENIED`
- `MQMSG_CLASS_NACK_BAD_DST_Q`
- `MQMSG_CLASS_NACK_BAD_ENCRYPTION`
- `MQMSG_CLASS_NACK_BAD_SIGNATURE`
- `MQMSG_CLASS_NACK_COULD_NOT_ENCRYPT`
- `MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED`
- `MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG`
- `MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q`
- `MQMSG_CLASS_NACK_PURGED`
- `MQMSG_CLASS_NACK_Q_EXCEEDED_QUOTA`
- `MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT`
- `MQMSG_CLASS_NACK_SOURCE_COMPUTER_GUID_CHANGED`
- `MQMSG_CLASS_NACK_UNSUPPORTED_CRYPTO_PROVIDER`
- `MQMSG_CLASS_ACK_RECEIVE`
- `MQMSG_CLASS_NACK_Q_DELETED`
- `MQMSG_CLASS_NACK_Q_PURGED`
- `MQMSG_CLASS_NACK_RECEIVE_TIMEOUT`
- `MQMSG_CLASS_NACK_RECEIVE_TIMEOUT_AT_SENDER`
- `MQMSG_CLASS_REPORT`

`msgId:` ***Number***

The MSMQ message id of the message.

`payload:` ***Buffer***

The body of the MSMQ message.

`priority:` ***Number***

The priority of the message. This can be a number between 0 and 7.

`queue:` ***String***

The name of the destination queue of the message.

`receiverBytes:` ***Number***

The number of L4 receiver bytes.

`receiverL2Bytes:` ***Number***

The number of L2 receiver bytes.

`receiverPkts:` ***Number***

The number of receiver packets.

`receiverRTO:` ***Number***

The number of retransmission timeouts (RTOs) from the receiver.

`receiverZeroWnd:` ***Number***

The number of zero windows sent by the receiver.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `MSMQ.commitRecord()` on an `MSMQ_MESSAGE` event.

The default record object can contain the following properties:

- `adminQueue`

- clientIsExternal
- dstQueueMgr
- isEncrypted
- label
- msgClass
- msgId
- priority
- queue
- receiverBytes
- receiverIsExternal
- receiverL2Bytes
- receiverPkts
- receiverRTO
- receiverZeroWnd
- responseQueue
- roundTripTime
- senderBytes
- senderIsExternal
- serverIsExternal
- senderL2Bytes
- senderPkts
- senderRTO
- serverZeroWnd
- srcQueueMgr

`responseQueue:` ***String***

The name of the response queue of the message.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `MSMQ_MESSAGE` event ran. The value is `NaN` if there are no RTT samples.

`senderBytes:` ***Number***

The number of sender L4 bytes.

`senderL2Bytes:` ***Number***

The number of sender L2 bytes.

`senderPkts:` ***Number***

The number of sender packets.

`senderRTO:` ***Number***

The number of retransmission timeouts (RTOs) from the sender.

`senderZeroWnd:` ***Number***

The number of zero windows sent by the sender.

`srcQueueMgr:` ***String***

The source message broker of the message.

`NetFlow`

The `NetFlow` class object enables you to store metrics and access properties on `NETFLOW_RECORD` events.

Events

`NETFLOW_RECORD`

Runs upon receipt of a flow record from a flow network.

Methods

`commitRecord(): void`

Sends a record to the configured recordstore on a `NETFLOW_RECORD` event.

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

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

`findField(field: Number, enterpriseId: Number): String | Number | IPAddress | Buffer | Boolean`

Searches the NetFlow record and returns the specified field. Returns a null value if the field is not in the record. If the optional `enterpriseId` argument is included, the specified field is returned only if the enterprise ID is a match, otherwise the method returns a null value.

`hasField(field: Number): Boolean`

Determines whether the specified field is in the NetFlow record.

Properties

`age: Number`

The amount of time elapsed, expressed in seconds, between the `first` and `last` property values reported in the NetFlow record.

`deltaBytes: Number`

The number of L3 bytes in the flow since the last `NETFLOW_RECORD` event.

`deltaPkts: Number`

The number of packets in the flow since the last `NETFLOW_RECORD` event.

`dscp: Number`

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

`dscpName: String`

The name associated with the DSCP value of the flow packet. The following table displays well-known DSCP names:

Number	Name
8	CS1
10	AF11
12	AF12
14	AF13
16	CS2
18	AF21
20	AF22
22	AF23
24	CS3
26	AF31
28	AF32

Number	Name
30	AF33
32	CS4
34	AF41
36	AF42
38	AF43
40	CS5
44	VA
46	EF
48	CS6
56	CS7

egressInterface: ***FlowInterface***

The FlowInterface object that identifies the output device.

fields: ***Array***

An array of objects that contain information fields found in the flow packets. Each object can contain the following properties:

fieldID: ***Number***

The ID number that represents the field type.

enterpriseID: ***Number***

The ID number that represents enterprise-specific information.

first: ***Number***

The amount of time elapsed, expressed in milliseconds, since the epoch of the first packet in the flow.

format: ***String***

The format of the NetFlow record. Valid values are `NetFlow v5`, `NetFlow v9`, and `IPFIX`.

ingressInterface: ***FlowInterface***

The FlowInterface object that identifies the input device.

ipPrecedence: ***Number***

The value of the IP precedence field associated with the DSCP of the flow packet.

ipproto: ***String***

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

last: ***Number***

The amount of time elapsed, expressed in milliseconds, since the epoch of the last packet in the flow.

network: ***FlowNetwork***

An object that identifies the FlowNetwork and contains the following properties:

id: ***String***

The identifier of the FlowNetwork.

ipaddr: ***IPAddress***

The IP address of the FlowNetwork.

nextHop: ***IPAddress***

The IP address of the next hop router.

observationDomain: ***Number***

The ID of the observation domain for the template.

receiver: ***Object***

An object that identifies the receiver and contains the following properties:

asn: ***Number***

The autonomous system number (ASN) of the destination device.

ipaddr: ***IPAddress***

The IP address of the destination device.

prefixLength: ***Number***

The number of bits in the prefix of the destination address.

port: ***Number***

The TCP or UDP port number of the destination device.

record: ***Object***

The record object that can be sent to the configured recordstore through a call to `NetFlow.commitRecord()` on a `NETFLOW_RECORD` event.

The default record object can contain the following properties:

- age
- clientIsExternal
- dscpName
- deltaBytes
- deltaPkts
- egressInterface
- first
- format
- ingressInterface
- last
- network
- networkAddr
- nextHop
- proto
- receiverAddr
- receiverAsn
- receiverIsExternal
- receiverPort
- receiverPrefixLength
- senderAddr
- senderAsn
- senderIsExternal
- serverIsExternal
- senderPort
- senderPrefixLength
- tcpFlagName
- tcpFlags

sender: ***Object***

An object that identifies the sender and contains the following properties:

asn: ***Number***

The autonomous system number (ASN) of the source device.

ipaddr: ***IPAddress***

The IP address of the source device.

prefixLength: ***Number***

The number of bits in the prefix of the source address.

port: ***Number***

The TCP or UDP port number of the source device.

tcpFlagNames: ***Array***

A string array of TCP flag names, such as SYN or ACK, found in the flow packets.

tcpFlags: ***Number***

The bitwise OR of all TCP flags set on the flow.

templateId: ***Number***

The ID of the template that is referred to by the record. Template IDs are applicable only to IPFIX and NetFlow v9 records.

tos: ***Number***

The type of service (ToS) number defined in the IP header.

NFS

The NFS class enables you to store metrics and access properties on NFS_REQUEST and NFS_RESPONSE events.

Events

NFS_REQUEST

Runs on every NFS request processed by the device.

NFS_RESPONSE

Runs on every NFS response processed by the device.

Note: The NFS_RESPONSE event runs after every NFS_REQUEST event, even if the corresponding response is never observed by the ExtraHop system.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an NFS_RESPONSE event. Record commits on NFS_REQUEST events are not supported.

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

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

Properties

`accessTime:` ***Number***

The amount of time taken by the server to access a file on disk, expressed in milliseconds. For NFS, it is the time from every non-pipelined READ and WRITE command in an NFS flow until the payload containing the response is recorded by the ExtraHop system. The value is NaN on malformed and aborted responses, or if the timing is invalid or is not applicable.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`authMethod:` ***String***

The method for authenticating users.

`error:` ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `NFS_RESPONSE` events; otherwise, an error will occur.

`fileHandle:` ***Buffer***

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

`isCommandFileInfo:` ***Boolean***

The value is `true` for file info commands.

`isCommandRead:` ***Boolean***

The value is `true` for READ commands.

`isCommandWrite:` ***Boolean***

The value is `true` for WRITE commands.

`isRspAborted:` ***Boolean***

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

Access only on `NFS_RESPONSE` events; otherwise, an error will occur.

`method:` ***String***

The NFS method. Valid methods are listed under the NFS metric in the ExtraHop system.

`offset:` ***Number***

The file offset associated with NFS READ and WRITE commands.

Access only on `NFS_REQUEST` events; otherwise, an error will occur.

`processingTime:` ***Number***

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

Access only on `NFS_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `NFS.commitRecord()` on a `NFS_RESPONSE` event.

The default record object can contain the following properties:

- `accessTime`
- `authMethod`
- `clientIsExternal`
- `clientZeroWnd`
- `error`
- `isCommandFileInfo`
- `isCommandRead`
- `isCommandWrite`
- `isRspAborted`
- `method`
- `offset`
- `processingTime`
- `receiverIsExternal`
- `renameDirChanged`
- `reqSize`
- `reqXfer`
- `resource`
- `rspSize`

- rspXfer
- senderIsExternal
- serverIsExternal
- serverZeroWnd
- statusCode
- txID
- user
- version

Access the record object only on NFS_RESPONSE events; otherwise, an error will occur.

renameDirChanged: ***Boolean***

The value is true if a resource rename request includes a directory move.

Access only on NFS_REQUEST events; otherwise, an error will occur.

reqBytes: ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

reqL2Bytes: ***Number***

The number of L2 request bytes, including L2 headers.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

reqPkts: ***Number***

The number of request packets.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

Access only on NFS_REQUEST events; otherwise, an error will occur.

reqSize: ***Number***

The number of L7 request bytes, excluding NFS headers.

reqTransferTime: ***Number***

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

Access only on NFS_REQUEST events; otherwise, an error will occur.

reqZeroWnd: ***Number***

The number of zero windows in the request.

resource: ***String***

The path and filename, concatenated together.

roundTripTime: ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last NFS_RESPONSE event ran. The value is NaN if there are no RTT samples.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

rspBytes: ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding NFS headers.

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`rspTransferTime:` ***Number***

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

Access only on NFS_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`statusCode:` ***String***

The NFS status code of the request or response.

`symlink:` ***Buffer | null***

The argument specified in an NFS SYMLINK request.

The value is null if this property is accessed on an event other than NFS_REQUEST or if the `NFS.method` is not `SYMLINK`.

`txId:` ***Number***

The transaction ID.

`user:` ***String***

The ID of the Linux user, formatted as `uid:xxxx`.

`verifierMethod:` ***String***

The method for verifying the sender of the request.

`version:` ***Number***

The NFS version.

NMF

The NET Message Framing Protocol (NMF) class enables you to store metrics and access properties on NMF_RECORD events.

Events

NMF_RECORD

Runs on every NMF record processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on an `NMF_RECORD` event. To view the default properties committed to the record object, see the record property below.

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

Properties

`envelope`: ***Buffer***

The Buffer object that contains the payload bytes of the message.

`wireSize`: ***Number***

The length of the raw record as it was observed, expressed in bytes. If the record is compressed, this property reflects the length of the compressed record.

`mode`: ***Number***

The numeric code for the communication mode. The following codes are valid:

Code	Description
1	Singleton-Unsized
2	Duplex
3	Simplex
4	Singleton-Sized

`via`: ***String***

The URI that subsequent messages will be sent to.

`version`: ***String***

The version of the NMF protocol.

NTLM

The `NTLM` class enables you to store metrics and access properties on `NTLM_MESSAGE` events.

Events

`NTLM_MESSAGE`

Runs on every NTLM message processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on an `NTLM_MESSAGE` event.

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

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

Properties

`containsMIC`: ***Boolean***

The value is true if the message includes a Message Integrity Code (MIC) that ensures that the message has not been tampered with.

`challenge:` ***String***

The hexadecimal-encoded challenge hash string.

`domain:` ***String***

The client domain name included in the challenge hash calculation.

`flags:` ***Number***

The bitwise OR of the NTLM negotiate flags. For more information, see the NTLM documentation on the Microsoft website.

`msgType:` ***String***

The type of NTLM message. The following message types are valid:

- `NTLM_AUTH`
- `NTLM_CHALLENGE`
- `NTLM_NEGOTIATE`

`ntlm2RspAVPairs:` ***Array***

An array of objects that contain NTLM attribute-value pairs. For more information, see the NTLM documentation on the Microsoft website.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `NTLM.commitRecord()` on a `NTLM_MESSAGE` event.

The default record object can contain the following properties:

- `challenge`
- `clientIsExternal`
- `domain`
- `flags`
- `l7proto`
- `msgType`
- `proto`
- `receiverAddr`
- `receiverIsExternal`
- `receiverPort`
- `senderAddr`
- `senderIsExternal`
- `senderPort`
- `serverIsExternal`
- `user`
- `windowsVersion`
- `workstation`

`rspVersion:` ***String***

The version of NTLM implemented in the NTLM_AUTH response. The value is `null` for non-authentication messages. The following versions are valid:

- `LM`
- `NTLMv1`
- `NTLMv2`

`user:` ***String***

The client username included in the challenge hash calculation.

`windowsVersion:` ***String***

The version of Windows running on the client included in the challenge hash calculation.

`workstation:` ***String***

The name of the client workstation included in the challenge hash calculation.

NTP

The Network Time Protocol (NTP) class enables you to store metrics and access properties on `NTP_MESSAGE` events.

Events

`NTP_MESSAGE`

Runs on every NTP message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `NTP_MESSAGE` event. To view the default properties committed to the record object, see the `record` property below.

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

Properties

`flags:` ***Number***

The decimal representation of the byte that contains information about the NTP flags. The leap indicator is contained in the first two bits of the byte, the NTP version is contained in the next three bits, and the NTP protocol operation mode is contained in the last three bits.

`leapIndicator:` ***Number***

Indicates whether an extra second will be added to or removed from the last minute of the day on the system clock. The following values are valid:

Value	Description
`0`	An extra second will not be added or removed.
`1`	An extra second will be added to the last minute of the day. The last minute will have 61 seconds.
`2`	An extra second will be removed from the last minute of the day. The last minute will have 59 seconds.
`3`	Unknown. Clocks are not currently synchronized.

`mode:` ***Number***

The numeric ID of the NTP protocol operation mode.

`modeName:` ***String***

The name of the NTP protocol operation mode. The following values are valid:

Value	Numeric ID
`reserved`	0
`symmetric active`	1
`symmetric passive`	2

Value	Numeric ID
client	3
server	4
broadcast	5
NTP control message	6
reserved for private use	7

`originTimestamp:` ***Number***

The local time of the client when the client sent the request to the server, expressed in fractional seconds since the NTP epoch.

`payload:` ***Buffer***

The Buffer object that contains the raw payload bytes of the NTP message.

`poll:` ***Number***

The maximum amount of time the system waits between NTP messages, expressed in fractional seconds.

`precision:` ***Number***

The precision of the system clock, expressed in fractional seconds.

`receiveTimestamp:` ***Number***

The local time of the server when the server received the request from the client, expressed in fractional seconds since the NTP epoch.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `NTP.commitRecord()` on an `NTP_MESSAGE` event.

The default record object can contain the following properties:

- `application`
- `extensionCount`
- `flowId`
- `modeName`
- `originTimestamp`
- `poll`
- `precision`
- `receiver`
- `receiverAddr`
- `receiverIsExternal`
- `receiverPort`
- `receiveTimestamp`
- `referenceId`
- `referenceIdCode`
- `referenceTimestamp`
- `rootDelay`
- `stratum`
- `sender`
- `senderAddr`
- `senderIsExternal`
- `senderPort`
- `transmitTimestamp`

- `version`
- `vlan`

`referenceId:` ***Number***

The numerical ID of the server or reference clock.

`referenceIdCode:` ***String | Null***

The string ID of the server or reference clock.

`referenceTimestamp:` ***Number***

The last time the system clock was set or corrected, expressed in fractional seconds since the NTP epoch.

`rootDelay:` ***Number***

The round-trip time delay to the reference clock, expressed in seconds.

`rootDispersion:` ***Number***

The maximum error relative to the reference clock, expressed in seconds.

`stratum:` ***Number***

The NTP stratum of the system clock.

`transmitTimestamp:` ***Number***

The local time of the server when the server sent the response to the client, expressed in fractional seconds since the NTP epoch.

`version:` ***Number***

The version of the NTP protocol.

POP3

The POP3 class enables you to store metrics and access properties on `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***

Sends a record to the configured recordstore on a `POP3_RESPONSE` event. Record commits on `POP3_REQUEST` events are not supported.

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

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

Properties

`dataSize:` ***Number***

The size of the message, expressed in bytes.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`error:` ***String***

The detailed error message recorded by the ExtraHop system.

Access only on POP3_RESPONSE events; otherwise, an error will occur.

`isEncrypted:` ***Boolean***

The value is `true` if the transaction is over a secure POP3 server.

`isReqAborted:` ***Boolean***

The value is `true` if the connection is closed before the POP3 request was complete.

`isRspAborted:` ***Boolean***

The value is `true` if the connection is closed before the POP3 response was complete.

Access only on POP3_RESPONSE events; otherwise, an error will occur.

`method:` ***String***

The POP3 method such as RETR or DELE.

`processingTime:` ***Number***

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

Access only on POP3_RESPONSE events; otherwise, an error will occur.

`recipientList:` ***Array***

An array that contains a list of recipient addresses.

Access only on POP3_RESPONSE events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `POP3.commitRecord()` on a POP3_RESPONSE event.

The default record object can contain the following properties:

- `clientIsExternal`
- `clientZeroWnd`
- `dataSize`
- `error`
- `isEncrypted`
- `isReqAborted`
- `isRspAborted`
- `method`
- `processingTime`
- `receiverIsExternal`
- `recipientList`
- `reqSize`
- `reqTimeToLastByte`
- `rspSize`
- `rspTimeToFirstByte`
- `rspTimeToLastByte`
- `sender`
- `senderIsExternal`
- `serverIsExternal`
- `serverZeroWnd`
- `statusCode`

Access the record object only on POP3_RESPONSE events; otherwise, an error will occur.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes, excluding POP3 headers.

`reqTimeToLastByte:` ***Number***

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

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median TCP round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `POP3_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding POP3 headers.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspTimeToFirstByte:` ***Number***

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

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`sender:` ***String***

The address of the sender of the message.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`status:` ***String***

The POP3 status message of the response which can be `OK`, `ERR` or `NULL`.

Access only on `POP3_RESPONSE` events; otherwise, an error will occur.

`QUIC`

The `QUIC` class enables you to store metrics and access properties on `QUIC_OPEN` and `QUIC_CLOSE` events.

Events

`QUIC_CLOSE`

Runs when a QUIC connection is closed.

`QUIC_OPEN`

Runs when a QUIC connection is opened.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either a `QUIC_OPEN` or `QUIC_CLOSE` event. To view the default properties committed to the record object, see the `record` property below.

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

Properties

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `QUIC.commitRecord()` on either a `QUIC_OPEN` or `QUIC_CLOSE` event.

The default record object can contain the following properties:

- `clientAddr`
- `clientIsExternal`
- `clientPort`
- `proto`
- `receiverIsExternal`
- `senderIsExternal`
- `serverAddr`
- `serverIsExternal`
- `serverPort`
- `sni`
- `version`
- `vlan`

`sni:` ***String***

The Server Name Indication (SNI), which identifies the name of the server the client is connecting to.

`version:` ***String***

The version of the QUIC protocol.

RDP

RDP (Remote Desktop Protocol) is a proprietary protocol created by Microsoft that enables a Windows computer to connect to another Windows computer on the same network or over the Internet. The RDP class enables you to store metrics and access properties on RDP_OPEN, RDP_CLOSE, or RDP_TICK events.

Events

RDP_CLOSE

Runs when an RDP connection is closed.

RDP_OPEN

Runs when a new RDP connection is opened.

RDP_TICK

Runs periodically while the user interacts with the RDP application.

Methods

commitRecord(): ***void***

Sends a record to the configured recordstore on an RDP_OPEN, RDP_CLOSE, or RDP_TICK event.

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

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

Properties

clientBuild: ***String***

The build number of the RDP client. This property is not available if the RDP connection is encrypted.

clientName: ***String***

The name of the client computer. This property is not available if the RDP connection is encrypted.

cookie: ***String***

The auto-connect cookie stored by the RDP client.

desktopHeight: ***Number***

The height of the desktop, expressed in pixels. This property is not available if the RDP connection is encrypted.

desktopWidth: ***Number***

The width of the desktop, expressed in pixels. This property is not available if the RDP connection is encrypted.

encryptionProtocol: ***String***

The protocol that the transaction is encrypted with.

error: ***String***

The detailed error message recorded by the ExtraHop system.

isDecrypted: ***Boolean***

The value is true if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

isEncrypted: ***Boolean***

The value is true if the RDP connection is encrypted.

`isError:` ***Boolean***

The value is `true` if an error occurred on the event.

`keyboardLayout:` ***String***

The keyboard layout, which indicates the arrangement of keys and the input language. This property is not available if the RDP connection is encrypted.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `RDP.commitRecord()` on either an `RDP_OPEN`, `RDP_CLOSE`, or `RDP_TICK` event.

The default record object can contain the following properties:

`RDP_OPEN` and `RDP_CLOSE`	`RDP_TICK`
`clientBuild`	`clientBuild`
`clientIsExternal`	`clientBytes`
`clientName`	`clientIsExternal`
`cookie`	`clientL2Bytes`
`desktopHeight`	`clientName`
`desktopWidth`	`clientPkts`
`error`	`clientRTO`
`isEncrypted`	`clientZeroWnd`
`keyboardLayout`	`cookie`
`receiverIsExternal`	`desktopHeight`
`requestedColorDepth`	`desktopWidth`
`requestedProtocols`	`error`
`selectedProtocol`	`isEncrypted`
`senderIsExternal`	`keyboardLayout`
`serverIsExternal`	`receiverIsExternal`
	`requestedColorDepth`
	`requestedProtocols`
	`roundTripTime`
	`selectedProtocol`
	`senderIsExternal`
	`serverBytes`
	`serverIsExternal`
	`serverL2Bytes`
	`serverPkts`
	`serverRTO`
	`serverZeroWnd`

`requestedColorDepth:` ***String***

The color depth requested by the RDP client. This property is not available if the RDP connection is encrypted.

`requestedProtocols:` ***Array of Strings***

The list of supported security protocols.

`reqBytes:` ***Number***

The number of L4 bytes in the request.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 bytes in the request.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of packets in the request.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`reqRTO:` ***Number***

The number of retransmission timeouts (RTOs) in the request.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `RDP_TICK` event ran. The value is `NaN` if there are no RTT samples.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of packets in the response.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of retransmission timeouts (RTOs) in the response.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

Access only on `RDP_TICK` events; otherwise, an error will occur.

`selectedProtocol:` ***String***

The selected security protocol.

user: String

The username, if available. In some cases, such as when login events are encrypted and the sensor has not been configured to decrypt the traffic, the username is unavailable.

Redis

Remote Dictionary Server (Redis) is an open-source, in-memory data structure server. The Redis class enables you to store metrics and access properties on REDIS_REQUEST and REDIS_RESPONSE events.

Events

REDIS_REQUEST

Runs on every Redis request processed by the device.

REDIS_RESPONSE

Runs on every Redis response processed by the device.

Methods

commitRecord(): ***void***

Sends a record to the configured recordstore on either a REDIS_REQUEST or REDIS_RESPONSE event.

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

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

Properties

errors: ***Array***

An array of detailed error messages recorded by the ExtraHop system.

Access only on REDIS_RESPONSE events; otherwise, an error will occur.

isReqAborted: ***Boolean***

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

isRspAborted: ***Boolean***

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

Access only on REDIS_RESPONSE events; otherwise, an error will occur.

method: ***String***

The Redis method such as GET or KEYS.

payload: ***Buffer***

The body of the response or request.

processingTime: ***Number***

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

Access only on REDIS_RESPONSE events; otherwise, an error will occur.

record: ***Object***

The record object that can be sent to the configured recordstore through a call to Redis.commitRecord() on either a REDIS_REQUEST or REDIS_RESPONSE event.

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

REDIS_REQUEST	REDIS_RESPONSE
clientIsExternal	clientIsExternal
clientZeroWnd	clientZeroWnd
method	error
receiverIsExternal	method
reqKey	processingTime
reqSize	receiverIsExternal
reqTransferTime	reqKey
isReqAborted	rspSize
senderIsExternal	rspTransferTime
serverZeroWnd	isRspAborted
	rspTimeToFirstByte
	rspTimeToLastByte
	senderIsExternal
	serverIsExternal
	serverZeroWnd

`reqKey:` ***Array***

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

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes, excluding Redis headers.

`reqTransferTime:` ***Number***

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

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median TCP round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `REDIS_REQUEST` or `REDIS_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspTransferTime:` ***Number***

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

Access only on `REDIS_RESPONSE` events; otherwise, an error will occur.

`rspSize:` ***Number***

The number of L7 response bytes, excluding Redis headers.

Access only on `REDIS_RESPONSE` events; otherwise, an error will occur.

`rspTimeToFirstByte:` ***Number***

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

Access only on `REDIS_RESPONSE` events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on `REDIS_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

RFB

The RFB class enables you to store metrics and access properties on `RFB_OPEN`, `RFB_CLOSE`, and `RFB_TICK` events.

Events

`RFB_CLOSE`

Runs when an RFB connection is closed.

`RFB_OPEN`

Runs when a new RFB connection is opened.

`RFB_TICK`

Runs periodically on RFB flows.

Methods

`commitRecord()`: ***void***

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

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

Properties

`authType`: ***Number***

The number that corresponds to the security type negotiated by the client and server.

Access only on `RFB_OPEN` events; otherwise, an error will occur.

Security type	Number
Invalid	0
None	1
VNC Authentication	2
RealVNC	3-15
Tight	16
Ultra	17
TLS	18
VeNCrypt	19
GTK-VNC SASL	20
MD5 hash authentication	21
Colin Dean xvp	22
RealVNC	128-255

`authResult`: ***Number***

Indicates whether authentication was successful.

Value	Description
0	Succeeded
1	Failed

`duration`: ***Number***

The duration of the RFB session, expressed in seconds.

Access only on `RFB_CLOSE` events; otherwise, an error will occur.

`error`: ***String***

The detailed error message recorded by the ExtraHop system.

Access only on `RFB_OPEN` events; otherwise, an error will occur.

`record`: ***Object***

The record object committed to the recordstore through a call to `RFB.commitRecord()`.

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

RFB_OPEN	RFB_TICK	RFB_CLOSE
authType	clientIsExternal	clientIsExternal
authResult	reqBytes	duration
clientIsExternal	receiverIsExternal	receiverIsExternal
error	reqL2Bytes	senderIsExternal
receiverIsExternal	reqPkts	serverIsExternal
senderIsExternal	reqRTO	
serverIsExternal	reqZeroWnd	
version	roundTripTime	
	rspBytes	
	rspL2Bytes	
	rspPkts	
	rspRTO	
	rspZeroWnd	
	senderIsExternal	
	serverIsExternal	

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`reqPkts:` ***Number***

The number of request packets.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

Access only on `RFB_TICK` events; otherwise, an error will occur.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`roundTripTime:` ***Number***

The median TCP round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `RFB_TICK` event ran. The value is `NaN` if there are no RTT samples.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`rspPkts:` ***Number***

The number of response packets.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

Access only on `RFB_TICK` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

Access only on `RFB_TICK` events; otherwise, an error will occur.

`version:` ***String***

The version of the RFB protocol negotiated by the client and server.

Access only on `RFB_OPEN` events; otherwise, an error will occur.

RPC

The `RPC` class enables you to store metrics and access properties from Microsoft Remote Procedure Call (MSRPC) activity on `RPC_REQUEST` and `RPC_RESPONSE` events.

Events

`RPC_REQUEST`

Runs on every RPC request processed by the device.

`RPC_RESPONSE`

Runs on every RPC response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `RPC_REQUEST` or `RPC_RESPONSE` event.

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

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

Properties

`authType:` ***String***

The security type negotiated by the client and server. The following types are valid:

- `DIGEST`
- `DPA`
- `GSS_KERBEROS`
- `GSS_SCHANNEL`

- KRB5
- MSN
- MQ
- NONE
- NTLMSSP
- SEC_CHAN
- SPNEGO

Access only on RPC_RESPONSE events; otherwise, an error will occur.

`commandLine:` ***String | Null***

The full command line specified in the RPC request for the following operations:

Interface	Operation
Service Control Manager	• RCreateServiceW • RCreateServiceA • RCreateServiceWOW64A • RCreateServiceWOW64W
IWbemServices	• ExecMethod • ExecMethodAsync
IDispatch	• Invoke
ITaskSchedulerService	• RegisterTask

If the operation is not included in the table above, or the request did not specify a command line, the value is null.

`encryptionProtocol:` ***String***

The protocol that the transaction is encrypted with.

`interface:` ***String***

The name of the RPC interface, such as drsuapi and epmapper.

`interfaceGUID:` ***String***

The GUID of the RPC interface. The format of the GUID includes hyphens, as shown in the following example:

```
367abb81-9844-35f2-ad32-98f038001004
```

`isEncrypted:` ***Boolean***

The value is true if the payload is encrypted.

`isDecrypted:` ***Boolean***

The value is true if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`isNDR64:` ***Boolean | null***

Indicates whether the request or response was transmitted with the NDR64 transfer syntax. If the pduType property is not request or response, the value is null.

`operation:` ***String***

The name of the RPC operation, such as DRSGetNCChanges and ept_map.

`opnum:` ***Number***

The opnum of the RPC operation. The opnum is the numerical ID of the RPC operation.

`payload:` ***Buffer | null***

The Buffer object containing the body of the request or response. If the `pduType` property is not request or response, the value is null.

`pduType:` ***String***

The PDU type, which indicates the purpose of the RPC message. The following values are valid:

- `ack`
- `alter_context`
- `alter_context_resp`
- `auth`
- `bind`
- `bind_ack`
- `bind_nak`
- `cancel_ack`
- `cl_cancel`
- `co_cancel`
- `fack`
- `fault`
- `nocall`
- `orphaned`
- `ping`
- `response`
- `request`
- `reject`
- `shutdown`
- `working`

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `RPC.commitRecord()` on an `RPC_REQUEST` or `RPC_RESPONSE` event.

The default record object can contain the following properties:

- `clientAddr`
- `clientBytes`
- `clientIsExternal`
- `clientL2Bytes`
- `clientPkts`
- `clientPort`
- `clientRTO`
- `clientZeroWnd`
- `interface`
- `operation`
- `proto`
- `receiverIsExternal`
- `roundTripTime`
- `senderIsExternal`
- `serverAddr`
- `serverBytes`
- `serverIsExternal`
- `serverL2Bytes`
- `serverPkts`

- serverPort
- serverRTO
- serverZeroWnd
- user

registryKey: ***String | Null***

The Windows registry key specified in the RPC request for the following operations:

Interface	Operation
WinReg	• BaseRegCloseKey • BaseRegCreateKey • BaseRegOpenKey • BaseRegQueryValue • BaseRegSetValue

If the operation is not included in the table above, or the request did not specify a registry key, the value is null.

reqBytes: ***Number***

The number of L4 request bytes, excluding L4 headers.

reqL2Bytes: ***Number***

The number of L2 request bytes, including L2 headers.

reqPkts: ***Number***

The number of request packets.

reqRTO: ***Number***

The number of request retransmission timeouts (RTOs).

reqZeroWnd: ***Number***

The number of zero windows in the request.

roundTripTime: ***Number***

The median TCP round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last RPC_REQUEST or RPC_RESPONSE event ran. The value is NaN if there are no RTT samples.

rspBytes: ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

rspL2Bytes: ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

rspPkts: ***Number***

The number of response packets.

rspRTO: ***Number***

The number of response retransmission timeouts (RTOs).

rspZeroWnd: ***Number***

The number of zero windows in the response.

sessionId: ***Number***

The ID of the associated SMB session.

serviceName: ***String | Null***

The name of the Windows service specified in the RPC request for the following operations:

Interface	Operation
Service Control Manager	• RCreateServiceW • RCreateServiceA • RCreateServiceWOW64A • RCreateServiceWOW64W

If the operation is not included in the table above, or the request did not specify a service, the value is null.

user: ***String***

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

RTCP

The RTCP class enables you to store metrics and access properties on RTCP_MESSAGE events.

Events

RTCP_MESSAGE

Runs on every RTCP UDP packet processed by the device.

Methods

commitRecord(): ***void***

Sends a record to the configured recordstore on an RTCP_MESSAGE event.

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

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

Properties

callId: ***String***

The Call ID for associating with a SIP flow.

packets: ***Array***

An array of RTCP packet objects where each object represents a packet and contains a packetType field. Each object has different fields based on the message type, as described below.

packetType: ***String***

The type of packet. If the packet type is not recognizable, then the packetType will be "Unknown N" where N is the RTP control packet type value.

Value	Type	Name
194	SMPTETC	SMPTE time-code mapping
195	IJ	Extended inter-arrival jitter report
200	SR	sender report
201	RR	receiver report
202	SDES	source description
203	BYE	goodbye

Value	Type	Name
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

The following list describes the fields for each type of packet object:

APP

`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

`packetType:` ***Number***

Contains the number 203 to identify this as an RTCP BYE packet.

SR

`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

reportBlocks: ***Array***

An array of report objects which contain:

fractionLost: ***Number***

The 8-bit number indicating the number of packets last divided by the number of packets expected.

jitter: ***Number***

An estimate of the statistical variance of the RTP data packet interarrival, expressed in milliseconds.

lastSR: ***Number***

The middle 32 bits of the ntp_Timestamp received as part of the most recent RTCP sender report (SR) packet from the source SSRC. If no SR has been received yet, this field is set to zero.

lastSRDelay: ***Number***

The delay between receiving the last SR packet from the source SSRC and sending this reception report block, expressed in units of 1/65536 seconds. If no SR packet has been received yet, this field is set to zero.

packetsLost: ***Number***

The total number of RTP data packets from the source SSRC that have been lost since the beginning of reception.

seqNum: ***Number***

The highest sequence number received from the source SSRC.

ssrc: ***Number***

The SSRC of the sender.

ssrc: ***Number***

The SSRC of the sender.

SDES

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`

`ssrc:` ***Number***

The SSRC of the sender.

`xrBlocks:` ***Array***

An array of report blocks which contain:

`statSummary:` ***Object***

Type 6 only. The `statSummary` object contains the following properties:

`beginSeq:` ***Number***

The beginning sequence number for the interval.

`devJitter:` ***Number***

The standard deviation of the relative transit time between each two packet series in the sequence interval.

`devTTLOrHL:` ***Number***

The standard deviation of TTL or Hop Limit values of data packets in the sequence number range.

`dupPackets:` ***Number***

The number of duplicate packets in the sequence number interval.

`endSeq:` ***Number***

The ending sequence number for the interval.

`lostPackets:` ***Number***

The number of lost packets in the sequence number interval.

`maxJitter:` ***Number***

The maximum relative transmit time between two packets in the sequence interval, expressed in milliseconds.

`maxTTLOrHL:` ***Number***

The maximum TTL or Hop Limit value of data packets in the sequence number range.

`meanJitter:` ***Number***

The mean relative transit time between two packet series in the sequence interval, rounded to the nearest value expressible as an RTP timestamp, expressed in milliseconds.

`meanTTLOrHL:` ***Number***

The mean TTL or Hop Limit value of data packets in the sequence number range.

`minJitter:` ***Number***

The minimum relative transmit time between two packets in the sequence interval, expressed in milliseconds.

`minTTLOrHL:` ***Number***

The minimum TTL or Hop Limit value of data packets in the sequence number range.

`ssrc:` ***Number***

The SSRC of the sender.

`type:` ***Number***

The XR block type.

Block Type	Name
1	Loss RTE Report Block
2	Duplicate RLE Report Block
3	Packet Receipt Times Report Block
4	Receiver Reference Time Report Block
5	DLRR Report Block
6	Statistics Summary Report Block
7	VoIP Metrics Report Block
8	RTCP XP
9	Texas Instruments Extended VoIP Quality Block
10	Post-repair Loss RLE Report Block
11	Multicast Acquisition Report Block
12	IBMS Report Block
13	ECN Summary Report

Block Type	Name
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***

The record object that can be sent to the configured recordstore through a call to `RTCP.commitRecord()` on an `RTCP_MESSAGE` event.

The default record object can contain the following properties:

- `callId`
- `clientIsExternal`
- `cName`
- `flowId`
- `receiverIsExternal`
- `senderIsExternal`
- `serverIsExternal`
- `signalingFlowId`

  The ID of the corresponding SIP or SCCP flow, which negotiates the VoIP call monitored by the RTCP flow.

RTP

The RTP class enables you to store metrics and access properties on `RTP_OPEN`, `RTP_CLOSE`, and `RTP_TICK` events.

Events

`RTP_CLOSE`

Runs when an RTP connection is closed.

`RTP_OPEN`

Runs when a new RTP connection is opened.

`RTP_TICK`

Runs periodically on RTP flows.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `RTP_TICK` event. Record commits on `RTP_OPEN` and `RTP_CLOSE` events are not supported.

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

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

Properties

`bytes:` ***Number***

The number of bytes sent.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`callId:` ***String***

The call ID associated with the SIP or SCCP flow.

`drops:` ***Number***

The number of dropped packets detected.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`dups:` ***Number***

The number of duplicate packets detected.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`jitter:` ***Number***

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

Access only on `RTP_TICK` events; otherwise, an error will occur.

`l2Bytes:` ***Number***

The number of L2 bytes.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`mos:` ***Number***

The estimated mean opinion score for quality.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`outOfOrder:` ***Number***

The number of out-of-order messaged detected.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`payloadType:` ***String***

The type of RTP payload.

Access only on `RTP_TICK` events; otherwise, an error will occur.

payloadTypeId	payloadType
0	ITU-T G.711 PCMU Audio
3	GSM 6.10 Audio
4	ITU-T G.723.1 Audio
5	IMA ADPCM 32kbit Audio
6	IMA ADPCM 64kbit Audio
7	LPC Audio
8	ITU-T G.711 PCMA Audio
9	ITU-T G.722 Audio
10	Linear PCM Stereo Audio
11	Linear PCM Audio
12	QCELP

payloadTypeId	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

`payloadTypeId:` ***Number***

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

Access only on `RTP_TICK` events; otherwise, an error will occur.

`pkts:` ***Number***

The number of packets sent.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `RTP.commitRecord()` on an `RTP_TICK` event.

The default record object can contain the following properties:

- `bytes`
- `callId`
- `clientIsExternal`
- `drops`
- `dups`
- `flowId`
- `jitter`
- `l2Bytes`
- `mos`
- `outOfOrder`
- `payloadType`
- `payloadTypeId`
- `pkts`
- `receiverIsExternal`
- `rFactor`
- `senderIsExternal`
- `serverIsExternal`

- `signalingFlowId`

  The ID of the corresponding SIP or SCCP flow, which negotiates the VoIP call streamed by the RTP flow.
- `ssrc`
- `version`

Access record objects only on `RTP_TICK` events; otherwise, an error will occur.

`rFactor:` ***Number***

The R factor quality metric.

Access only on `RTP_TICK` events; otherwise, an error will occur.

`ssrc:` ***Number***

The SSRC of sender.

`version:` ***Number***

The RTP version number.

SCCP

Skinny Client Control Protocol (SCCP) is a Cisco proprietary protocol for communicating with VoIP devices. The `SCCP` class enables you to store metrics and access properties on `SCCP_MESSAGE` events.

Events

`SCCP_MESSAGE`

Runs on every SCCP message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `SCCP_MESSAGE` event.

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

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

Properties

`callId:` ***String***

The call ID associated with the RTP flow.

`callInfo:` ***Object***

An object containing information about the current SCCP called. The object contains the following fields:

`callReference:` ***Number***

The unique identifier of the call.

`callType:` ***Number***

The ID of the call type.

ID	Call Type
1	Inbound
2	Outbound
3	Forward

`calledPartyName:` ***String***

The name of the recipient of the call.

`calledPartyNumber:` ***String***

The phone number of the recipient of the call.

`callingPartyName:` ***String***

The name of the caller.

`callingPartyNumber:` ***String***

The phone number of the caller.

`lineInstance:` ***Number***

The unique identifier of the line.

`callStats:` ***Object***

An object containing statistics for the SCCP call, as reported and calculated by the client. The object contains the following fields:

`reportedBytesIn:` ***Number***

The number of L7 bytes received.

`reportedBytesOut:` ***Number***

The number of L7 bytes sent.

`reportedJitter:` ***Number***

The level of packet jitter, or variation in latency, during the call.

`reportedLatency:` ***Number***

The level of packet latency, expressed in milliseconds, during the call.

`reportedPktsIn:` ***Number***

The number of packets received.

`reportedPktsLost:` ***Number***

The number of packets lost during the call.

`reportedPktsOut:` ***Number***

The number of packets sent.

`msgType:` ***String***

The decoded SCCP message type.

`receiverBytes:` ***Number***

The number of L4 bytes from the receiver.

`receiverL2Bytes:` ***Number***

The number of L2 bytes from the receiver.

`receiverPkts:` ***Number***

The number of packets from the receiver.

`receiverRTO:` ***Number***

The number of retransmission timeouts (RTOs) from the receiver.

`receiverZeroWnd:` ***Number***

The number of zero windows from the receiver.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SCCP.commitRecord()` on an `SCCP_MESSAGE` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `msgType`

- receiverBytes
- receiverIsExternal
- receiverL2Bytes
- receiverPkts
- receiverRTO
- receiverZeroWnd
- roundTripTime
- senderBytes
- senderIsExternal
- senderL2Bytes
- senderPkts
- senderRTO
- senderZeroWnd
- serverIsExternal

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `SCCP_MESSAGE` event ran. The value is `NaN` if there are no RTT samples.

`senderBytes:` ***Number***

The number of L4 bytes from the sender.

`senderL2Bytes:` ***Number***

The number of L2 bytes from the sender.

`senderPkts:` ***Number***

The number of packets from the sender.

`senderRTO:` ***Number***

The number of retransmission timeouts (RTOs) from the sender.

`senderZeroWnd:` ***Number***

The number of zero windows from the sender.

SDP

The SDP class enables you to access properties on `SIP_REQUEST` and `SIP_RESPONSE` events.

The `SIP_REQUEST` and `SIP_RESPONSE` events are defined in the SIP section.

Properties

`mediaDescriptions:` ***Array***

An array of objects that contain the following fields:

`attributes:` ***Array of Strings***

The optional session attributes.

`bandwidth:` ***Array of Strings***

The optional proposed bandwidth type and bandwidth to be consumed by the session or media.

`connectionInfo:` ***String***

The connection data, including network type, address type and connection adddress. May also contain optional sub-fields, depending on the address type.

`description:` ***String***

The session description which may contain one or more media descriptions. Each media description consists of media, port and transport protocol fields.

`encryptionKey:` ***String***

The optional encryption method and key for the session.

`mediaTitle:` ***String***

The title of the media stream.

`sessionDescription:` ***Object***

An object that contains the following fields:

`attributes:` ***Array of Strings***

The optional session attributes.

`bandwidth:` ***Array of Strings***

The optional proposed bandwidth type and bandwidth to be consumed 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 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.

SFlow

The SFlow class object enables you to store metrics and access properties on SFLOW_RECORD events. sFlow is a sampling technology for monitoring traffic in data networks. sFlow samples every nth packet and sends it to the collector whereas NetFlow sends data from every flow to the collector. The primary difference between sFlow and NetFlow is that sFlow is network layer independent and can sample anything.

Events

SFLOW_RECORD

Runs upon receipt of an SFlow sample exported from a flow network.

Methods

commitRecord(): ***void***

Sends a flow record object, which indicates the sFlow format, to the configured recordstore on an SFLOW_RECORD event.

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

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

Properties

deltaBytes: ***Number***

The number of L3 bytes in the flow packet.

dscp: ***Number***

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

dscpName: ***String***

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

Number	Name
8	CS1
10	AF11
12	AF12
14	AF13
16	CS2
18	AF21
20	AF22
22	AF23
24	CS3
26	AF31
28	AF32
30	AF33
32	CS4

Number	Name
34	AF41
36	AF42
38	AF43
40	CS5
44	VA
46	EF
48	CS6
56	CS7

`egressInterface:` ***FlowInterface***
The FlowInterface object that identifies the output interface.

`format:` ***String***
The format of the SFlow record. Valid value is "sFlow v5".

`headerData:` ***Buffer***
The Buffer object containing the raw bytes of the entire flow packet header.

`ingressInterface:` ***FlowInterface***
The FlowInterface object that identifies the input interface.

`ipPrecedence:` ***Number***
The value of the IP precedence field associated with the DSCP of the flow packet.

`ipproto:` ***String***
The IP protocol associated with the flow, such as TCP or UDP.

`network:` ***FlowNetwork***
Returns a FlowNetwork object that identifies the exporter and contains the following properties:

`id:` ***String***
The identifier of the FlowNetwork.

`ipaddr:` ***IPAddress***
The IP address of the FlowNetwork.

`record:` ***Object***
The flow record object that can be sent to the configured recordstore through a call to `SFlow.commitRecord()` on an `SFLOW_RECORD` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `deltaBytes`
- `dscpName`
- `egressInterface`
- `format`
- `ingressInterface`
- `ipPrecedence`
- `ipproto`
- `network`
- `networkAddr`
- `receiverIsExternal`
- `senderIsExternal`

- serverIsExternal
- tcpFlagName
- tcpFlags

tcpFlagNames: ***Array***

A string array of TCP flag names, such as SYN or ACK, found in the flow packets.

tcpFlags: ***Number***

The bitwise OR of all TCP flags set on the flow.

tos: ***Number***

The type of service (ToS) number defined in the IP header.

SIP

The SIP class enables you to store metrics and access properties on SIP_REQUEST and SIP_RESPONSE events.

Events

SIP_REQUEST

Runs on every SIP request processed by the device.

SIP_RESPONSE

Runs on every SIP response processed by the device.

Methods

commitRecord(): ***void***

Sends a record to the configured recordstore on either a SIP_REQUEST or SIP_RESPONSE event.

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

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

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

Enables access to SIP header values. The result is an array of header objects (with name and value properties) where the names match the prefix of the string passed to findHeaders.

Properties

callId: ***String***

The call ID for this message.

from: ***String***

The contents of the From header.

hasSDP: ***Boolean***

The value is true if this event includes SDP information.

headers: ***Object***

An array-like object that enables access to SIP header names and values. Access a specific header with one of the following 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 acknowledgment.
PUBLISH	Publish an event to the server.
REFER	Ask recipient to issue a SIP request (call transfer).
REGISTER	Registers the address listed in the To header field with a SIP server.
SUBSCRIBE	Subscribes for an event of Notification from the Notifier.
UPDATE	Modifies the state of a session without changing the state of the dialog.

`payload:` ***Buffer*** | ***null***

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

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

`processingTime:` ***Number***

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

Access only on `SIP_RESPONSE` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SIP.commitRecord()` on either a `SIP_REQUEST` or `SIP_RESPONSE` event.

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

SIP_REQUEST	SIP_RESPONSE
callId	callId
clientIsExternal	clientIsExternal
clientZeroWnd	clientZeroWnd
from	from
hasSDP	hasSDP
method	processingTime
receiverIsExternal	receiverIsExternal
reqBytes	roundTripTime
reqL2Bytes	rspBytes
reqPkts	rspL2Bytes
reqRTO	rspPkts
reqSize	rspRTO
senderIsExternal	rspSize
serverIsExternal	senderIsExternal
serverZeroWnd	serverIsExternal
to	serverZeroWnd
uri	statusCode
	to

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes, excluding SIP headers.

Access only on `SIP_REQUEST` events; otherwise, an error will occur.

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `SIP_REQUEST` or `SIP_RESPONSE` event ran. The value is `NaN` if there are no RTT samples.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspSize:` ***Number***

The number of L7 response bytes, excluding SIP headers.

Access only on `SIP_RESPONSE` events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`statusCode:` ***Number***

The SIP response status code.

Access only on `SIP_RESPONSE` events; otherwise, an error will occur.

The following table displays provisional responses:

Number	Response
100	Trying
180	Ringing
181	Call is Being Forwarded
182	Queued
183	Session In Progress
199	Early Dialog Terminated

The following table displays successful responses:

Number	Response
200	OK
202	Accepted
204	No Notification

The following table displays redirection responses:

Number	Response
300	Multiple Choice
301	Moved Permanently
302	Moved Temporarily
305	Use Proxy
380	Alternative Service

The following table displays client failure responses:

Number	Response
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
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

Number	Response
436	Bad Identity Info
437	Unsupported Certificate
438	Invalid Identity Header
439	First Hop Lacks Outbound Support
470	Consent Needed
480	Temporarily Unavailable
481	Call/Transaction Does Not Exist
482	Loop Detected
483	Too Many Hops
484	Address Incomplete
485	Ambiguous
486	Busy Here
487	Request Terminated
488	Not Acceptable Here
489	Bad Event
491	Request Pending
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

Name	Response
606	Not Acceptable

`to:` ***String***

The contents of the To header.

`uri:` ***String***

The URI for SIP request or response.

SLP

The SLP class enables you to store metrics and access properties on SLP_MESSAGE events.

Events

`SLP_MESSAGE`

Runs on every SLP message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an SLP_MESSAGE event.

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

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

Properties

`attrList:` ***String | null***

The attributes for the SLP message, in a comma-separated list.

`functionId:` ***Number***

The numeric function ID of the SLP message, which corresponds to the message type string.

`msgType:` ***String***

The SLP message type string, which corresponds to the numeric function ID as shown in the following table:

Message Type	Function ID
Service Request	1
Service Reply	2
Service Registration	3
Service Deregister	4
Service Acknowledge	5
Attribute Request	6
Attribute Reply	7
DA Advertisement	8
Service Type Request	9
Service Type Reply	10

Message Type	Function ID
SA Advertisement	11

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SLP.commitRecord()` on an SLP_MESSAGE event. The default record object can contain the following properties:

- `clientIsExternal`
- `functionId`
- `msgType`
- `receiverIsExternal`
- `scopeList`
- `senderIsExternal`
- `serverIsExternal`

`scopeList:` ***String*** | ***null***

The scope for the SLP message, in a comma-separated list.

SMPP

The SMPP class enables you to store metrics and access properties on SMPP_REQUEST and SMPP_RESPONSE events.

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

Events

SMPP_REQUEST

Runs on every SMPP request processed by the device.

SMPP_RESPONSE

Runs on every SMPP response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a SMPP_RESPONSE event. Record commits on SMPP_REQUEST events are not supported.

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

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

Properties

`command:` ***String***

The SMPP command ID.

`destination:` ***String***

The destination address as specified in the SMPP_REQUEST. The value is `null` if this is not available for the current command type.

`error:` ***String***

The error code corresponding to command_status. If the command status is ROK, the value is `null`.

Access only on SMPP_RESPONSE events; otherwise, an error will occur.

`message:` ***Buffer***

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

Access only on SMPP_REQUEST events; otherwise, an error will occur.

`processingTime:` ***Number***

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

Access only on SMPP_RESPONSE events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SMPP.commitRecord()` on a SMPP_RESPONSE event.

The default record object can contain the following properties:

- `clientIsExternal`
- `clientZeroWnd`
- `command`
- `destination`
- `error`
- `receiverIsExternal`
- `reqSize`
- `reqTimeToLastByte`
- `rspSize`
- `rspTimeToFirstByte`
- `rspTimeToLastByte`
- `senderIsExternal`
- `serverIsExternal`
- `serverZeroWnd`
- `source`
- `processingTime`

`reqSize:` ***Number***

The number of L7 request bytes, excluding SMPP headers.

`reqTimeToLastByte:` ***Number***

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

`rspSize:` ***Number***

The number of L7 response bytes, excluding SMPP headers.

Access only on SMPP_RESPONSE events; otherwise, an error will occur.

`rspTimeToFirstByte:` ***Number***

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

Access only on SMPP_RESPONSE events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on SMPP_RESPONSE events; otherwise, an error will occur.

`source:` ***String***

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

SMTP

The SMTP class enables you to store metrics and access properties on SMTP_REQUEST and SMTP_RESPONSE events.

Events

SMTP_OPEN

Runs on every SMTP greeting processed by the device.

SMTP_REQUEST

Runs on every SMTP request processed by the device.

SMTP_RESPONSE

Runs on every SMTP response processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a SMTP_RESPONSE event. Record commits on SMTP_REQUEST events are not supported.

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

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

Properties

`dataSize:` ***Number***

The size of the attachment, expressed in bytes.

`domain:` ***String***

The domain of the address the message is coming from.

`error:` ***String***

The error code corresponding to status code.

Access only on SMTP_RESPONSE events; otherwise, an error will occur.

`headers:` ***Object***

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

The value of the `headers` property is the same when accessed on either the SMTP_REQUEST or the SMTP_RESPONSE event.

`isEncrypted:` ***Boolean***

The value is `true` if the application is encrypted with STARTTLS.

`isReqAborted:` ***Boolean***

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

`isRspAborted:` ***Boolean***

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

Access only on SMTP_RESPONSE events; otherwise, an error will occur.

`method:` ***String***

The SMTP method.

`processingTime:` ***Number***

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

Access only on `SMTP_RESPONSE` events; otherwise, an error will occur.

`recipientList:` ***Array of Strings***

A list of recipient addresses.

The value of the `recipientList` property is the same when accessed on either the `SMTP_REQUEST` or the `SMTP_RESPONSE` event.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SMTP.commitRecord()` on a `SMTP_RESPONSE` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `clientZeroWnd`
- `dataSize`
- `domain`
- `error`
- `isEncrypted`
- `isReqAborted`
- `isRspAborted`
- `method`
- `processingTime`
- `receiverIsExternal`
- `recipient`
- `recipientList`
- `reqBytes`
- `reqL2Bytes`
- `reqPkts`
- `reqRTO`
- `reqSize`
- `reqTimeToLastByte`
- `roundTripTime`
- `rspBytes`
- `rspL2Bytes`
- `rspPkts`
- `rspRTO`
- `rspSize`
- `rspTimeToFirstByte`
- `rspTimeToLastByte`
- `sender`
- `senderIsExternal`
- `serverIsExternal`
- `serverZeroWnd`
- `statusCode`
- `statusText`

Access the record object only on SMTP_RESPONSE events; otherwise, an error will occur.

`reqBytes:` ***Number***

The number of L4 request bytes, excluding L4 headers.

`reqL2Bytes:` ***Number***

The number of L2 request bytes, including L2 headers.

`reqPkts:` ***Number***

The number of request packets.

`reqRTO:` ***Number***

The number of request retransmission timeouts (RTOs).

`reqSize:` ***Number***

The number of L7 request bytes, excluding SMTP headers.

`reqTimeToLastByte:` ***Number***

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

`reqZeroWnd:` ***Number***

The number of zero windows in the request.

`roundTripTime:` ***Number***

The median TCP round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last SMTP_RESPONSE event ran. The value is NaN if there are no RTT samples.

Access only on SMTP_RESPONSE events; otherwise, an error will occur.

`rspBytes:` ***Number***

The number of L4 response bytes, excluding L4 protocol overhead, such as ACKs, headers, and retransmissions.

`rspL2Bytes:` ***Number***

The number of L2 response bytes, including protocol overhead, such as headers.

`rspPkts:` ***Number***

The number of response packets.

`rspRTO:` ***Number***

The number of response retransmission timeouts (RTOs).

`rspSize:` ***Number***

The number of L7 response bytes, excluding SMTP headers.

Access only on SMTP_RESPONSE events; otherwise, an error will occur.

`rspTimeToFirstByte:` ***Number***

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

Access only on SMTP_RESPONSE events; otherwise, an error will occur.

`rspTimeToLastByte:` ***Number***

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

Access only on SMTP_RESPONSE events; otherwise, an error will occur.

`rspZeroWnd:` ***Number***

The number of zero windows in the response.

`sender:` ***String***

The sender of the message.

`statusCode:` ***Number***

The SMTP status code of the response or greeting.

Access only on `SMTP_RESPONSE` or `SMTP_OPEN` events; otherwise, an error will occur.

`statusText:` ***String***

The multi-line response or greeting string.

Access only on `SMTP_RESPONSE` or `SMTP_OPEN` events; otherwise, an error will occur.

SNMP

The SNMP class enables you to store metrics and access properties on `SNMP_REQUEST`, `SNMP_RESPONSE`, and `SNMP_MESSAGE` events.

Events

`SNMP_REQUEST`

Runs on every SNMP request processed by the device.

`SNMP_RESPONSE`

Runs on every SNMP response processed by the device.

`SNMP_MESSAGE`

Runs on SNMP messages that do not adhere to typical request and response behavior. Neither the `SNMP_REQUEST` event nor the `SNMP_RESPONSE` event runs on these messages. These messages include requests sent from a server to a client and responses sent from a client to a server. These messages also include SNMP traps, which are messages sent from the server that do not prompt a response.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on an `SNMP_REQUEST`, `SNMP_RESPONSE`, or `SNMP_MESSAGE` event. To view the default properties committed to the record object, see the record property below.

If the `commitRecord()` method is called on an `SNMP_REQUEST` event, the record is not created until the `SNMP_RESPONSE` event runs. If the `commitRecord()` method is called on both the `SNMP_REQUEST` and the corresponding `SNMP_RESPONSE`, only one record is created for request and response, even if the `commitRecord()` method is called multiple times on the same trigger events.

Properties

`error:` ***String***

The SNMP error message.

`community:` ***String***

The SNMP community string.

`payload:` ***Buffer***

The Buffer object that contains the raw payload bytes of the event transaction. The buffer contains the first 1024 bytes of the payload.

`pduType:` ***String***

The protocol data unit (PDU) type.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to SNMP.commitRecord() on either an `SNMP_REQUEST`, `SNMP_RESPONSE`, or `SNMP_MESSAGE` event.

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

SNMP_REQUEST	SNMP_RESPONSE	SNMP_MESSAGE
client	client	community
clientAddr	clientAddr	error
clientIsExternal	clientIsExternal	flowId
clientPort	clientPort	pduType
community	community	receiver
flowId	error	receiverAddr
pduType	flowId	receiverPort
server	pduType	receiverIsExternal
serverAddr	server	sender
serverIsExternal	serverAddr	senderAddr
serverPort	serverIsExternal	senderPort
version	serverPort	senderIsExternal
vlan	version	version
	vlan	vlan

`version:` ***String***

The version of SNMP protocol.

SOCKS

The SOCKet Secure (SOCKS) class enables you to store metrics and access properties on SOCKS_REQUEST and SOCKS_RESPONSE events.

Events

SOCKS_REQUEST

Runs on every SOCKS message processed by the device.

SOCKS_RESPONSE

Runs on every SOCKS message processed by the device.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on a SOCKS_RESPONSE event. Record commits on SOCKS_REQUEST events are not supported. To view the default properties committed to the record object, see the record property below.

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

Properties

`authResult:` ***Number***

Indicates whether authentication was successful. The following values are valid.

Value	Description
`0`	Succeeded
`1`	Failed

Note: If the protocol is SOCKS4, the value is always `0` because SOCKS4 does not support authentication.

`authType:` ***Number***

The authentication method that was negotiated between the server and the client.

`command:` ***Number***

The numeric code for the SOCKS command that the client requested. The following command codes are valid.

Code	Description
`1`	Connect TCP stream
`2`	Bind TCP port
`3`	Associate UDP port

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SOCKS.commitRecord()` on an `SOCKS_RESPONSE` event.

- `application`
- `authResult`
- `authType`
- `client`
- `clientAddr`
- `clientIsExternal`
- `clientPort`
- `command`
- `flowId`
- `requestAddress`
- `requestPort`
- `responseAddress`
- `responsePort`
- `result`
- `server`
- `serverAddr`
- `serverIsExternal`
- `serverPort`
- `username`
- `version`
- `vlan`

Access the record object only on `SOCKS_RESPONSE` events; otherwise, an error will occur.

`requestAddress:` ***IPAddress***

The IPAddress object for the address specified by the client in the request.

`requestPort:` ***Number***

The port number specified by the client in the request.

`responseAddress:` ***IPAddress***

The IPAddress object for the address specified by the server in the response.

`responsePort:` ***Number***

The port number specified by the server in the response.

`result:` ***Number***

The status code specified by the server in the response.

`username:` ***String***

The name of the user specified by the client for authentication.

`version:` ***Number***

The SOCKS protocol version.

SSH

Secure Socket Shell (SSH) is a network protocol that provides a secure method for remote login and other network services over an unsecured network. The SSH class object enables you to store metrics and access properties on SSH_CLOSE, SSH_OPEN and SSH_TICK events.

Events

SSH_CLOSE

Runs when the SSH connection is shut down by being closed, expired, or aborted.

SSH_OPEN

Runs when the SSH connection is first fully established after negotiating session information. If the negotiation fails because the key exchange is invalid, the SSH_OPEN event runs when there is an invalid exchange, and then the SSH_TICK and SSH_CLOSE events run in immediate succession.

If a connection closes before SSH_OPEN runs, SSH_OPEN, SSH_TICK, and SSH_CLOSE run in immediate succession.

SSH_TICK

Runs periodically on SSH flows.

Methods

`commitRecord():` ***void***

Sends a record to the configured recordstore on either an SSH_OPEN, SSH_CLOSE, or SSH_TICK event.

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

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

Properties

`clientBytes:` ***Number***

The total number of bytes sent by the client since the last SSH event ran. For SSH_OPEN events, this property is the number of bytes sent by the client since the start of the flow.

`clientCipherAlgorithm:` ***String***

The encryption cipher algorithm on the SSH client.

`clientCompressionAlgorithm:` ***String***

The compression algorithm applied to data transferred over the connection by the SSH client.

`clientCompressionAlgorithmsClientToServer:` ***String***

The compression algorithms that the SSH client supports for client to server communications.

`clientCompressionAlgorithmsServerToClient:` ***String***

The compression algorithms that the SSH client supports for server to client communications.

`clientEncryptionAlgorithmsClientToServer:` ***String***

The encryption algorithms that the SSH client supports for client to server communications.

`clientEncryptionAlgorithmsServerToClient:` ***String***

The encryption algorithms that the SSH client supports for server to client communications.

`clientImplementation:` ***String***

The SSH implementation installed on the client, such as OpenSSH or PUTTY.

`clientKexAlgorithms:` ***String***

The SSH key exchange algorithms that the client supports.

`clientL2Bytes:` ***Number***

The total number of L2 client bytes observed since the last SSH event ran. For `SSH_OPEN` events, this property is the number of L2 client bytes observed since the start of the flow. Note that this property does not return the total number of bytes for the entire SSH session.

`clientMacAlgorithm:` ***String***

The Method Authentication Code (MAC) algorithm on the SSH client.

`clientMacAlgorithmsClientToServer:` ***String***

The Method Authentication Code (MAC) algorithms that the SSH client supports for client to server communications.

`clientMacAlgorithmsServerToClient:` ***String***

The Method Authentication Code (MAC) algorithms that the SSH client supports for server to client communications.

`clientPkts:` ***Number***

The total number of packets sent by the client since the last SSH event ran. For `SSH_OPEN` events, this property is the number of packets sent by the client since the start of the flow. Note that this property does not return the total number of packets for the entire SSH session.

`clientRTO:` ***Number***

The total number of client retransmission timeouts (RTOs) observed since the last SSH event ran. For `SSH_OPEN` events, this property is the number of client RTOs observed since the start of the flow. Note that this property does not return the total number of client RTOs for the entire SSH session.

`clientVersion:` ***String***

The version of SSH on the client.

`clientZeroWnd:` ***Number***

The total number of zero windows sent by the client since the last SSH event ran. For `SSH_OPEN` events, this property is the number of zero windows sent by the client since the start of the flow. Note that this property does not return the total number of zero windows for the entire SSH session.

`duration:` ***Number***

The duration, expressed in milliseconds, of the SSH connection.

Access only on `SSH_CLOSE` events; otherwise, an error will occur.

`hasshAlgorithms:` ***String***

A string containing the SSH key exchange, encryption, message authentication, and compression algorithms that the client supports for SSH communications. These algorithms are communicated in the SSH_MSG_KEXINIT packet sent at the start of an SSH connection.

`hassh:` ***String***

An MD5 hash of the hasshAlgorithms string.

`hasshServerAlgorithms:` ***String***

A string containing the SSH key exchange, encryption, message authentication, and compression algorithms that the server supports for SSH communications. These algorithms are communicated in the SSH_MSG_KEXINIT packet sent at the start of an SSH connection.

`hasshServer:` ***String***

An MD5 hash of the hasshServerAlgorithms string.

`kexAlgorithm:` ***String***

The Key Exchange (Kex) algorithm on the SSH connection.

`messageNumbers:` ***Array of Numbers***

The numeric IDs of the SSH messages exchanged, listed in chronological order. The array cannot contain more than 50 entries. If more than 50 messages are exchanged, the array contains the 50 most recent IDs.

Access only on `SSH_OPEN` events; otherwise, an error will occur.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SSH.commitRecord()` on either an `SSH_OPEN`, `SSH_CLOSE`, or `SSH_TICK` event.

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

SSH_TICK	SSH_OPEN	SSH_CLOSE
clientCipherAlgorithm	clientCipherAlgorithm	clientCipherAlgorithm
clientCompressionAlgorithm	clientCompressionAlgorithm	clientCompressionAlgorithm
clientImplementation	clientImplementation	clientImplementation
clientIsExternal	clientIsExternal	clientIsExternal
clientMacAlgorithm	clientMacAlgorithm	clientMacAlgorithm
clientVersion	clientVersion	clientVersion
clientZeroWnd	clientZeroWnd	clientZeroWnd
kexAlgorithm	kexAlgorithm	kexAlgorithm
receiverIsExternal	receiverIsExternal	receiverIsExternal
senderIsExternal	senderIsExternal	senderIsExternal
serverCipherAlgorithm	serverCipherAlgorithm	serverCipherAlgorithm
serverCompressionAlgorithm	serverCompressionAlgorithm	serverCompressionAlgorithm
serverImplementation	serverImplementation	serverImplementation
serverIsExternal	serverIsExternal	serverIsExternal
serverMacAlgorithm	serverMacAlgorithm	serverMacAlgorithm
serverVersion	serverVersion	serverVersion

SSH_TICK	SSH_OPEN	SSH_CLOSE
serverZeroWnd	serverZeroWnd	serverZeroWnd
		duration

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last SSH event ran; for `SSH_OPEN` events, the sample begins at the start of the flow. The value is `NaN` if there are no RTT samples.

`serverBytes:` ***Number***

The total number of bytes sent by the server since the last SSH event ran. For `SSH_OPEN` events, this property is the number of bytes sent by the server since the start of the flow.

`serverCipherAlgorithm:` ***String***

The encryption cipher algorithm on the SSH server.

`serverCompressionAlgorithm:` ***String***

Returns the type of compression applied to data transferred over the connection by the SSH server.

`serverCompressionAlgorithmsClientToServer:` ***String***

The compression algorithms that the SSH server supports for client to server communications.

`serverCompressionAlgorithmsServerToClient:` ***String***

The compression algorithms that the SSH server supports for server to client communications.

`serverEncryptionAlgorithmsClientToServer:` ***String***

The encryption algorithms that the SSH server supports for client to server communications.

`serverEncryptionAlgorithmsServerToClient:` ***String***

The encryption algorithms that the SSH server supports for server to client communications.

`serverHostKey:` ***String***

The base64 encoding of the public SSH key sent from the server to the client.

`serverHostKeyType:` ***String***

The type of public SSH key sent from the server to the client, such as ssh-rsa or ssh-ed25519.

`serverImplementation:` ***String***

The SSH implementation installed on the server, such as OpenSSH or PUTTY.

`serverKexAlgorithms:` ***String***

The SSH key exchange algorithms that the server supports.

`serverL2Bytes:` ***Number***

The total number of L2 server bytes observed since the last SSH event ran. For `SSH_OPEN` events, this property is the number of L2 server bytes observed since the start of the flow. Note that this property does not return the total number of bytes for the entire SSH session.

`serverMacAlgorithm:` ***String***

The Method Authentication Code (MAC) algorithm on the SSH server.

`serverMacAlgorithmsClientToServer:` ***String***

The Method Authentication Code (MAC) algorithms that the SSH server supports for client to server communications.

`serverMacAlgorithmsServerToClient:` ***String***

The Method Authentication Code (MAC) algorithms that the SSH server supports for server to client communications.

`serverPkts:` ***Number***

The total number of packets sent by the server since the last SSH event ran. For `SSH_OPEN` events, this property is the number of packets sent by the server since the start of the flow. Note that this property does not return the total number of packets for the entire SSH session.

`serverRTO:` ***Number***

The total number of server retransmission timeouts (RTOs) observed since the last SSH event ran. For `SSH_OPEN` events, this property is the number of server RTOs observed since the start of the flow. Note that this property does not return the total number of server RTOs for the entire SSH session.

`serverVersion:` ***String***

The version of SSH on the server.

`serverZeroWnd:` ***Number***

The total number of packets sent by the server since the last SSH event ran. For `SSH_OPEN` events, this property is the number of packets sent by the server since the start of the flow. Note that this property does not return the total number of zero windows for the entire SSH session.

SSL

The `SSL` class enables you to store metrics and access properties on `SSL_OPEN`, `SSL_CLOSE`, `SSL_ALERT`, `SSL_RECORD`, `SSL_HEARTBEAT`, and `SSL_RENEGOTIATE` events.

Events

`SSL_ALERT`

Runs when an TLS alert record is exchanged.

`SSL_CLOSE`

Runs when the TLS connection is shut down.

`SSL_HEARTBEAT`

Runs when an TLS heartbeat record is exchanged.

`SSL_OPEN`

Runs when the TLS connection is first established.

`SSL_PAYLOAD`

Runs when the decrypted TLS payload matches the criteria configured in the associated trigger.

Depending on the flow, the payload can be found in the following properties:

- `Flow.payload1`
- `Flow.payload2`
- `Flow.client.payload`
- `Flow.server.payload`
- `Flow.sender.payload`
- `Flow.receiver.payload`

Additional payload options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

`SSL_RECORD`

Runs when an TLS record is exchanged.

`SSL_RENEGOTIATE`

Runs on TLS renegotiation.

Methods

`addApplication(name: String): void`

Associates an TLS session with the named application to collect TLS metric data about the session. For example, you might call `SSL.addApplication()` to associate TLS certificate data in an application.

After an TLS session is associated with an application, that pairing is permanent for the lifetime of the session.

Call only on `SSL_OPEN` events; otherwise, an error will occur.

`commitRecord(): void`

Sends a record to the configured recordstore only on `SSL_ALERT`, `SSL_CLOSE`, `SSL_HEARTBEAT`, `SSL_OPEN`, or `SSL_RENEGOTIATE` events. Record commits on `SSL_PAYLOAD` and `SSL_RECORD` events are not supported.

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

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

`getClientExtensionData(extension_name | extension_id): Buffer | Null`

Returns the data for the specified extension if the extension was passed as part of the Hello message from the client. Returns `null` if the message does not contain data.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`getServerExtensionData(extension_name | extension_id): Buffer | Null`

Returns data for the specified extension if the extension was passed as part of the `Hello` message from the server. Returns `null` if the message does not contain data.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`hasClientExtension(extension_name | extension_id): boolean`

Returns `true` for the specified extension if the extension was passed as part of the `Hello` message from the client.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`hasServerExtension(extension_name | extension_id): boolean`

Returns `true` for the specified extension if the extension was passed as part of the `Hello` message from the server.

Call only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

The following table provides a list of known TLS 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	supported_groups
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
27	compress_certificate
28	record_size_limit
29	pwd_protect
30	pwd_clear
31	password_salt
35	session_ticket
41	pre_shared_key
42	early_data
43	supported_versions
44	cookie
45	psk_key_exchange_modes
47	certificate_authorities
48	oid_filters
49	post_handshake_auth
50	signature_algorithms_cert
51	key_share
65281	renegotiation_info
65486	encrypted_server_name

The following extensions are sent out by applications to test whether servers can handle unknown extensions. For more information about these extensions, see Applying GREASE to TLS Extensibility.

- 2570
- 6682
- 10794

- 14906
- 19018
- 23130
- 27242
- 31354
- 35466
- 39578
- 43690
- 47802
- 51914
- 56026
- 60138
- 64250

Properties

`alertCode`: ***Number***

The numeric representation of the TLS alert. The following table displays the possible TLS alerts, which are defined in the `AlertDescription` data structure in RFC 2246:

Alert	Number
close_notify	0
unexpected_message	10
bad_record_mac	20
decryption_failed	21
record_overflow	22
decompression_failure	30
handshake_failure	40
bad_certificate	42
unsupported_certificate	43
certificate_revoked	44
certificate_expired	45
certificate_unknown	46
illegal_parameter	47
unknown_ca	48
access_denied	49
decode_error	50
decrypt_error	51
export_restriction	60
protocol_version	70
insufficient_security	71
internal_error	80

Alert	Number
user_canceled	90
no_renegotiation	100

If the session is opaque, the value is SSL.ALERT_CODE_UNKNOWN (null).

Access only on SSL_ALERT events; otherwise, an error will occur.

alertCodeName: ***String***

The name of the TLS alert associated with the alert code. See the alertCode property for alert names associated with alert codes. The value is null if no name is available for the associated alert code.

Access only on SSL_ALERT events; otherwise, an error will occur.

alertLevel: ***Number***

The numeric representation of the TLS alert level. The following possible alert levels are defined in the AlertLevel data structure in RFC 2246:

- warning (1)
- fatal (2)

If the session is opaque, the value is SSL.ALERT_LEVEL_UNKNOWN (null).

Access only on SSL_ALERT events; otherwise, an error will occur.

certificate: ***SSLCert***

The TLS server certificate object associated with the communication. Each object contains the following properties:

authorityInfoAccess: ***Object***

An object that contains information from the Authority Information Access extension, which specifies information about the certificate authority (CA). The object contains the following fields:

location: ***String***

The URL of the Online Certificate Status Protocol (OCSP) Responder that can verify whether the certificate is valid.

method: ***String***

The OID of the method that the certificate issuer can be accessed with.

authorityKeyIdentifier: ***String | Null***

The identifier for the public key of the certificate authority (CA), expressed as an octet string.

Note: This field does not contain the authority certification issuer or serial number.

basicConstraints: ***Object***

An object that contains information from the Basic Constraints extension, which specifies the type of certificate subject. The object contains the following fields:

ca: ***Boolean***

Indicates whether the subject of the certificate is a CA.

pathlen: ***Number***

The maximum number of certificates that can appear in the certificate chain after this certificate.

certificatePolicies: ***Array of Strings***

An array of OIDs for the policies specified in the Certificate Policies extension. Qualifiers are not included in this array.

`crlDistributionPoints:` ***Array of Strings***

An array of objects that contain information about servers that host certificate revocation lists (CRLs) for the server certificate. The servers are specified in the CRL distribution point (CDP) extension. Each object contains the following fields:

crlIssuer: Array of Strings

An array of locations where the certificate of the CRL issuer can be retrieved.

`distPoint:` ***Array of Strings***

An array of locations where the CRL can be retrieved.

`reasons:` ***Array of Strings***

An array of reason codes that indicate the reasons that the certificate could be revoked by the CRL distribution point.

`extensionOIDs:` ***Array of Strings***

An array of OIDs for the X509 extensions specified in the certificate.

`extendedKeyUsage:` ***Array of Strings***

An array of uses for the public key of the server certificate specified in the Extended Key Usage extension. The array can contain the following strings:

- `serverAuth`
- `clientAuth`
- `emailProtection`
- `codeSigning`
- `OCSPSigning`
- `timeStamping`
- `anyExtendedKeyUsage`
- `nsSGC`

`fingerprint:` ***String***

The hexadecimal representation of the SHA-1 hash of the certificate. The string contains no delimiters, as shown in the following example:

```
55F30E6D49E19145CF680E8B7E3DC8FC7041DC81
```

The SHA-1 certificate hash appears in the server certificate dialog box of most browsers.

`fingerprintSHA256:` ***String***

The hexadecimal representation of the SHA-256 hash of the certificate. The string contains no delimiters, as shown in the following example:

```
468C6C84DB844821C9CCB0983C78D1CC05327119B894B5CA1C6A1318784D3675
```

The SHA-256 certificate hash appears in the server certificate dialog box of most browsers.

`getExtensionDataByOID(extension_oid):` ***Buffer***

Method that returns a buffer object containing the value of the specified extension, expressed as an octet string. Returns null if the OID does not exist or the server certificate does not contain the extension.

`inhibitAnyPolicy:` ***Number***

The number specified in the Inhibit anyPolicy extension, which limits the number of certificates that the anyPolicy extension is applied to. The number specifies how many additional, non-self-issued certificates in the chain are affected by the anyPolicy extension.

`isSelfSigned:` ***Boolean***

The value is `true` if the server certificate is self-signed.

`issuer:` ***String***

The common name of the server certificate issuer. The value is `null` if the issuer is not available.

`issuerAlternativeNames:` ***Array of Strings***

An array of Issuer Alternative Names (IANs) specified in the server certificate.

`issuerDistinguishedName:` ***Object***

An object that contains information about the distinguished name of the certificate issuer. Each object contains the following properties:

`commonName:` ***String***

The common name (CN).

`country:` ***Array of Strings***

The country name (C).

`emailAddress:` ***String***

The email address.

`organization:` ***Array of Strings***

The organization name (O).

`organizationalUnit:` ***Array of Strings***

The organizational unit name (OU).

`locality:` ***Array of Strings***

The locality name (L).

`stateOrProvince:` ***Array of Strings***

The state or province name (ST).

`keySize:` ***Number***

The key size of the server certificate.

`keyUsage:` ***Array of Strings***

An array of uses for the public key of the server certificate specified in the Key Usage extension. The array can contain the following strings:

- `digitalSignature`
- `nonRepudiation`
- `keyEncipherment`
- `dataEncipherment`
- `keyAgreement`
- `keyCertSign`
- `cRLSign`
- `encipherOnly`
- `decipherOnly`

`notAfter:` ***Number***

The expiration time of the server certificate, expressed in UTC.

`notBefore:` ***Number***

The start time of the server certificate, expressed in UTC. The server certificate is not valid before this time.

`nsComment:` ***String***

The comment specified in the Netscape Comment extension. This comment is sometimes displayed in browsers when users view the server certificate.

`ocspNoCheck:` ***Boolean***

Indicates whether the signing certificate can be trusted without verification from the OCSP responder.

`payload:` ***Buffer***

The Buffer object that contains the raw payload bytes of the server certificate.

`policyConstraints:` ***Object***

An object that contains information from the Policy Constraints extension, which specifies validation constraints for CA certificates.

`requireExplicitPolicy:` ***Number***

Specifies the maximum number of adjacent certificates in the chain that do not need to specify an explicit policy.

`inhibitPolicyMapping:` ***Number***

Specifies the maximum number of adjacent certificates in the certificate chain before policy mappings are ignored.

`policyMappings:` ***Array of Objects***

An array of objects that contains information from the Policy Mappings extension, which indicates policies that are equivalent to each other. Each object contains the following fields.

`issuerDomainPolicy:` ***String***

The OID of the issuer policy.

`subjectDomainPolicy:` ***String***

The OID of the subject policy.

`publicKeyCurveName:` ***String***

The name of the standard elliptic curve that the cryptography of the public key is based on. This value is determined by the OID or explicit curve parameters specified in the certificate.

`publicKeyExponent:` ***String | Null***

A string hex representation of the public key exponent. The string is shown in the client certificate dialog box of most browsers, but without spaces.

`publicKeyHasExplicitCurve:` ***Boolean | Null***

Indicates whether the certificate specifies explicit parameters for the elliptic curve of the public key.

`publicKeyModulus:` ***String | Null***

A string hex representation of the public key modulus. The string is shown in the client certificate dialog box of most browsers, but without space, such as `010001`

`serial:` ***String | Null***

The serial number assigned to the certificate by the Certificate Authority (CA).

`signatureAlgorithm:` ***String | Null***

The algorithm applied to sign the server certificate. The following table displays some of the possible values:

RFC	Algorithm
RFC 3279	• `md2WithRSAEncryption` • `md5WithRSAEncryption` • `sha1WithRSAEncryption`
RFC 4055	• `sha224WithRSAEncryption` • `sha256WithRSAEncryption \` • `sha384WithRSAEncryption` • `sha512WithRSAEncryption`
RFC 4491	• `id-GostR3411-94-with-Gost3410-94`

RFC	Algorithm
	• `id-GostR3411-94-with-Gost3410-2001`

`subject:` ***String***

The subject common name (CN) of the server certificate.

`subjectAlternativeNames:` ***Array***

An array of strings that correspond to Subject Alternative Names (SANs) included in the server certificate. Supported SANs are DNS names, email addresses, URIs, and IP addresses.

`subjectDistinguishedName:` ***Object***

An object that contains information about the distinguished name of the certificate subject. Each object contains the following properties:

`commonName:` ***String***

The common name (CN).

`country:` ***Array of Strings***

The country name (C).

`emailAddress:` ***String***

The email address.

`organization:` ***Array of Strings***

The organization name (O).

`organizationalUnit:` ***Array of Strings***

The organizational unit name (OU).

`locality:` ***Array of Strings***

The locality name (L).

`stateOrProvince:` ***Array of Strings***

The state or province name (ST).

`subjectKeyIdentifier:` ***String***

The identifier for the public key of the certificate subject, expressed as an octet string.

`certificates:` ***Array of Objects***

An array of certificate objects for each intermediate TLS certificate. The end-entity certificate, also known as the leaf certificate, is the first object in the array; this object is also returned by the `certificate` property.

`cipherSuite:` ***String***

A string representing the cryptographic cipher suite negotiated between the server and the client.

`cipherSuitesHex:` ***String***

A hexadecimal representation of the cryptographic cipher suite negotiated between the server and the client.

`cipherSuitesSupported:` ***Array of Objects | Null***

An array of objects with the following properties that specify the cipher suites supported by the TLS client:

`name:` ***String***

The name of cipher suite.

`type:` ***Number***

The cipher suite number.

Access only on `SSL_OPEN` or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`cipherSuiteType:` ***Number***

The numeric value that represents the cryptographic cipher suite negotiated between the server and the client. Possible values are defined by the IANA TLS Cipher Suite Registry.

`clientBytes:` ***Number***

The total number of bytes sent by the client since the last `SSL_RECORD` event ran. Note that this property does not return the total number of bytes for the entire SSL session.

Access only on `SSL_RECORD` or `SSL_CLOSE` events; otherwise, an error will occur.

`clientCertificate:` ***SSLCert***

The TLS client certificate object associated with the communication. Each object contains the following properties:

`authorityInfoAccess:` ***Object***

An object that contains information from the Authority Information Access extension, which specifies information about the certificate authority (CA). The object contains the following fields:

`location:` ***String***

The URL of the Online Certificate Status Protocol (OCSP) Responder that can verify whether the certificate is valid.

`method:` ***String***

The OID of the method that the certificate issuer can be accessed with.

`authorityKeyIdentifier:` ***String | Null***

The identifier for the public key of the certificate authority (CA), expressed as an octet string.

Note: This field does not contain the authority certification issuer or serial number.

`basicConstraints:` ***Object***

An object that contains information from the Basic Constraints extension, which specifies the type of certificate subject. The object contains the following fields:

`ca:` ***Boolean***

Indicates whether the subject of the certificate is a CA.

`pathlen:` ***Number***

The maximum number of certificates that can appear in the certificate chain after this certificate.

`certificatePolicies:` ***Array of Strings***

An array of OIDs for the policies specified in the Certificate Policies extension. Qualifiers are not included in this array.

`crlDistributionPoints:` ***Array of Strings***

An array of objects that contain information about servers that host certificate revocation lists (CRLs) for the client certificate. The servers are specified in the CRL distribution point (CDP) extension. Each object contains the following fields:

crlIssuer: Array of Strings

An array of locations where the certificate of the CRL issuer can be retrieved.

`distPoint:` ***Array of Strings***

An array of locations where the CRL can be retrieved.

`reasons:` ***Array of Strings***

An array of reason codes that indicate the reasons that the certificate could be revoked by the CRL distribution point.

`extensionOIDs:` ***Array of Strings***

An array of OIDs for the X509 extensions specified in the client certificate.

`extendedKeyUsage:` ***Array of Strings***

An array of uses for the public key of the client certificate specified in the Extended Key Usage extension. The array can contain the following strings:

- `serverAuth`
- `clientAuth`
- `emailProtection`
- `codeSigning`
- `OCSPSigning`
- `timeStamping`
- `anyExtendedKeyUsage`
- `nsSGC`

`fingerprint:` ***String***

The hexadecimal representation of the SHA-1 hash of the client certificate. The string contains no delimiters, as shown in the following example:

```
55F30E6D49E19145CF680E8B7E3DC8FC7041DC81
```

`fingerprintSHA256:` ***String***

The hexadecimal representation of the SHA-256 hash of the client certificate. The string contains no delimiters, as shown in the following example:

```
468C6C84DB844821C9CCB0983C78D1CC05327119B894B5CA1C6A1318784D3675
```

`getExtensionDataByOID(extension_oid):` ***Buffer***

Method that returns a buffer object containing the value of the specified extension, expressed as an octet string. Returns null if the OID does not exist or the client certificate does not contain the extension.

`keySize:` ***Number***

The key size of the client certificate.

`keyUsage:` ***Array of Strings***

An array of uses for the public key of the client certificate specified in the Key Usage extension. The array can contain the following strings:

- `digitalSignature`
- `nonRepudiation`
- `keyEncipherment`
- `dataEncipherment`
- `keyAgreement`
- `keyCertSign`
- `cRLSign`
- `encipherOnly`
- `decipherOnly`

`inhibitAnyPolicy:` ***Number***

The number specified in the Inhibit anyPolicy extension, which limits the number of certificates that the anyPolicy extension is applied to. The number specifies how many additional, non-self-issued certificates in the chain are affected by the anyPolicy extension.

`isSelfSigned:` ***Boolean***

The value is `true` if the client certificate is self-signed.

`issuer:` ***String | Null***

The common name of the client certificate issuer. The value is `null` if the issuer is not available.

`issuerDistinguishedName:` ***Object***

An object that contains information about the distinguished name of the certificate issuer. Each object contains the following properties:

`commonName:` ***String***

The common name (CN).

`country:` ***Array of Strings***

The country name (C).

`emailAddress:` ***String***

The email address.

`organization:` ***Array of Strings***

The organization name (O).

`organizationalUnit:` ***Array of Strings***

The organizational unit name (OU).

`locality:` ***Array of Strings***

The locality name (L).

`stateOrProvince:` ***Array of Strings***

The state or province name (ST).

`issuerAlternativeNames:` ***Array of Strings***

An array of Issuer Alternative Names (IANs) specified in the client certificate.

`notAfter:` ***Number***

The expiration time of the client certificate, expressed in UTC.

`notBefore:` ***Number***

The start time of the client certificate, expressed in UTC. The client certificate is not valid before this time.

`nsComment:` ***String***

The comment specified in the Netscape Comment extension. This comment is sometimes displayed in browsers when users view the client certificate.

`ocspNoCheck:` ***Boolean***

Indicates whether the signing certificate can be trusted without verification from the OCSP responder.

`payload:` ***Buffer***

The Buffer object that contains the raw payload bytes of the client certificate.

`policyConstraints:` ***Object***

An object that contains information from the Policy Constraints extension, which specifies validation constraints for CA certificates.

`requireExplicitPolicy:` ***Number***

Specifies the maximum number of adjacent certificates in the chain that do not need to specify an explicit policy.

`inhibitPolicyMapping:` ***Number***

Specifies the maximum number of adjacent certificates in the certificate chain before policy mappings are ignored.

`publicKeyCurveName:` ***String***

The name of the standard elliptic curve that the cryptography of the public key is based on. This value is determined by the OID or explicit curve parameters specified in the certificate.

`publicKeyExponent:` ***String | Null***

A string hex representation of the public key exponent.

`publicKeyHasExplicitCurve:` ***Boolean | Null***

Indicates whether the certificate specifies explicit parameters for the elliptic curve of the public key.

`publicKeyModulus:` ***String | Null***

A string hex representation of the public key modulus, such as `010001`.

`policyMappings:` ***Array of Objects***

An array of objects that contains information from the Policy Mappings extension, which indicates policies that are equivalent to each other. Each object contains the following fields.

`issuerDomainPolicy:` ***String***

The OID of the issuer policy.

`subjectDomainPolicy:` ***String***

The OID of the subject policy.

`signatureAlgorithm:` ***String | Null***

The algorithm applied to sign the client certificate. The following table displays some of the possible values:

RFC	Algorithm
`RFC 3279`	• `md2WithRSAEncryption` • `md5WithRSAEncryption` • `sha1WithRSAEncryption`
`RFC 4055`	• `sha224WithRSAEncryption` • `sha256WithRSAEncryption` • `sha384WithRSAEncryption` • `sha512WithRSAEncryption`
`RFC 4491`	• `id-GostR3411-94-with-Gost3410-94` • `id-GostR3411-94-with-Gost3410-2001`

`subject:` ***String***

The subject common name (CN) of the client certificate.

`subjectAlternativeNames:` ***Array***

An array of strings that correspond to Subject Alternative Names (SANs) included in the client certificate. Supported SANs are DNS names, email addresses, URIs, and IP addresses.

`subjectDistinguishedName:` ***Object***

An object that contains information about the distinguished name of the certificate subject. Each object contains the following properties:

`commonName:` ***String***

The common name (CN).

`country:` ***Array of Strings***

The country name (C).

`emailAddress:` ***String***

The email address.

`organization:` ***Array of Strings***

The organization name (O).

`organizationalUnit:` ***Array of Strings***

The organizational unit name (OU).

`locality:` ***Array of Strings***

The locality name (L).

`stateOrProvince:` ***Array of Strings***

The state or province name (ST).

`subjectKeyIdentifier:` ***String***

The identifier for the public key of the client certificate subject, expressed as an octet string.

`clientCertificates:` ***Array of Objects***

An array of certificate objects for each intermediate TLS client certificate. The end-entity certificate, also known as the leaf certificate, is the first object in the array; this object is also returned by the `clientCertificate` property.

`clientCertificateRequested:` ***Boolean***

The value is `true` if the TLS server requested a client certificate.

Access only on `SSL_OPEN`, `SSL_ALERT`, or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`clientExtensions:` ***Array | Null***

An array of client extension objects that contain the following properties:

`id:` ***Number***

The ID number of the TLS client extension.

`length:` ***Number***

The full length of the TLS client extension, expressed in bytes.

Note: An extension might be truncated if the length exceeds the maximum size. The default is 512 bytes. Truncation has occurred if the value of this property is smaller than the buffer returned by the `getClientExtensionData()` method.

`name:` ***String***

The name of the TLS client extension, if known. Otherwise, the value indicates that the extension is unknown. See the table of known TLS extensions in the Methods section.

Access only on `SSL_OPEN` or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`clientExtensionsHex:` ***String***

A hexadecimal representation of the sorted list of client extensions.

Note: The Generate Random Extensions And Sustain Extensibility (GREASE) values are removed from the list before encoding.

Access only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`clientHelloVersion:` ***Number***

The version of TLS specified by the client in the client hello packet.

`clientL2Bytes:` ***Number***

The total number of L2 client bytes observed since the last `SSL_RECORD` event ran. Note that this property does not return the total number of bytes for the entire SSL session.

Access only on `SSL_RECORD` or `SSL_CLOSE` events; otherwise, an error will occur.

`clientPkts:` ***Number***

The total number of packets sent by the client since the last `SSL_RECORD` event ran. Note that this property does not return the total number of packets for the entire SSL session.

Access only on `SSL_RECORD` or `SSL_CLOSE` events; otherwise, an error will occur.

`clientSessionId:` ***String***

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

`clientZeroWnd:` ***Number***

The total number of zero windows sent by the client since the last `SSL_RECORD` event ran. Note that this property does not return the total number of zero windows for the entire SSL session.

Access only on `SSL_RECORD` or `SSL_CLOSE` events; otherwise, an error will occur.

`contentType:` ***String***

The content type for the current record.

Access only on `SSL_RECORD` events; otherwise, an error will occur.

`ecPointFormatsHex:` ***String***

A hexadecimal representation of the elliptic-curve point formats that the client can parse.

Access only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`encryptionProtocol:` ***String***

The TLS protocol version that the transaction is encrypted with.

`handshakeTime:` ***Number***

The amount of time required to negotiate the TLS connection, expressed in milliseconds. Specifically, the amount of time between when the client sends a `ClientHello` message and the server sends `ChangeCipherSpec` values as specified in RFC 2246.

Access only on `SSL_OPEN` or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`heartbeatPayloadLength:` ***Number***

The value of the payload length field of the HeartbeatMessage data structure as specified in RFC 6520.

Access only on `SSL_HEARTBEAT` events; otherwise, an error will occur.

`heartbeatType:` ***Number***

The numeric representation of the HeartbeatMessageType field of the HeartbeartMessage data structure as specified in RFC 6520. Valid values are `SSL.HEARTBEAT_TYPE_REQUEST (1)`, `SSL.HEARTBEAT_TYPE_RESPONSE (2)`, or `SSL.HEARTBEAT_TYPE_UNKNOWN (255)`.

Access only on `SSL_HEARTBEAT` events; otherwise, an error will occur.

`host:` ***String | Null***

The TLS Server Name Indication (SNI), if available.

Access only on `SSL_OPEN` or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`isAborted:` ***Boolean***

The value is `true` if the TLS session is aborted.

Access only on `SSL_CLOSE`, `SSL_OPEN`, and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`isCompressed:` ***Boolean***

The value is `true` if the TLS record is compressed.

`isDecrypted:` ***Boolean***

The value is `true` if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`isEncrypted:` ***Boolean***

The value is `true` if the TLS connection is encrypted.

`isPostQuantumKeyAgreement:` ***Boolean***

Indicates whether the TLS session was encrypted with a post-quantum cryptography (PQC) algorithm. PQC is designed to resist attacks from quantum computers.

`isResumed:` ***Boolean***

The value is `true` if the connection is resumed from an existing TLS session and is not a new TLS session.

Access only on `SSL_OPEN`, `SSL_CLOSE`, `SSL_ALERT`, `SSL_HEARTBEAT`, or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`isStartTLS:` ***Boolean***

The value is `true` if negotiation of the TLS session was initiated by the STARTTLS mechanism of the protocol.

Access only on `SSL_OPEN`, `SSL_CLOSE`, `SSL_ALERT`, `SSL_HEARTBEAT`, or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`isV2ClientHello:` ***Boolean***

The value is `true` if the Hello record corresponds to SSLv2.

`isWeakCipherSuite:` ***Boolean***

The value is `true` if the cipher suite encrypting the TLS session is considered weak. NULL, anonymous, and EXPORT cipher suites are considered weak, as are suites that encrypt with CBC, DES, 3DES, MD5, or RC4.

Access only on `SSL_OPEN`, `SSL_CLOSE`, `SSL_ALERT`, `SSL_HEARTBEAT`, or `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`ja3Text:` ***String | Null***

The complete JA3 string for the client, including the client hello TLS version, accepted ciphers, SSL extensions, elliptic curves, and elliptic curve formats.

`ja3Hash:` ***String | Null***

The MD5 hash of the JA3 string for the client.

`ja3sText:` ***String | Null***

The complete JA3S string for the server, including the server hello SSL version, accepted ciphers, and TLS extensions.

`ja3sHash:` ***String | Null***

The MD5 hash of the JA3S string for the server.

`ja4Fingerprint:` ***String | Null***

The complete JA4 fingerprint for the client, which includes the following information:

- The transport layer (L4) protocol
- The TLS version
- Whether the Server Name Indicator (SNI) extension was specified
- The number of cipher suites
- The number of extensions
- The first Application Layer Protocol Negotiation (ALPN) value listed
- The truncated SHA256 hash of cipher suites
- The truncated SHA256 hash of extensions

`keyAgreement:` ***String***

The details of the key agreement or exchange algorithm established for the TLS session. For the RSA algorithm, the property specifies the key size (`RSA-2048`). For the Elliptic-Curve Diffe-Hellman Ephemeral (ECDHE) algorithm, the property specifies the key-exchange group (`ECDHE-secp256r1`). For post-quantum cryptography (PQC) algorithms, the property specifies that the algorithm includes PQC (`PQC-ECDHE-Kyber-768-X25519`).

`privateKeyId:` ***String | Null***

The string ID associated with the private key if the ExtraHop system is decrypting TLS traffic. The value is `null` if the ExtraHop system is not decrypting SSL traffic.

To find the private key ID in the Administration settings, click **Capture** from the System Configuration section, click **SSL Decryption**, and then click a certificate. The pop-up window displays all identifiers for the certificate.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `SSL.commitRecord()` on either an `SSL_OPEN`, `SSL_CLOSE`, `SSL_ALERT`, `SSL_HEARTBEAT`, or `SSL_RENEGOTIATE` event.

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

Event	Available properties
`SSL_ALERT`	• `alertCode` • `alertLevel` • `certificateFingerprint` • `certificateIsSelfSigned` • `certificateIssuer` • `certificateKeySize` • `certificateNotAfter` • `certificateNotBefore` • `certificateSignatureAlgorithm` • `certificateSubject` • `cipherSuite` • `clientAddr` • `clientBytes` • `clientCertificateRequested` • `clientIsExternal` • `clientL2Bytes` • `clientPkts` • `clientPort` • `clientRTO` • `clientZeroWnd` • `isCompressed` • `isWeakCipherSuite` • `proto` • `receiverIsExternal` • `reqBytes` • `reqL2Bytes` • `reqPkts` • `reqRTO` • `rspBytes` • `rspL2Bytes` • `rspPkts` • `rspRTO` • `senderIsExternal` • `serverAddr` • `serverBytes` • `serverIsExternal` • `serverL2Bytes` • `serverPkts` • `serverPort`

Event	Available properties
	• serverRTO • serverZeroWnd • version
SSL_CLOSE	• certificateIsSelfSigned • certificateIssuer • certificateFingerprint • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • clientAddr • clientBytes • clientIsExternal • clientL2Bytes • clientPkts • clientPort • clientRTO • clientZeroWnd • isAborted • isCompressed • isWeakCipherSuite • proto • receiverIsExternal • reqBytes • reqPkts • reqL2Bytes • reqRTO • rspBytes • rspL2Bytes • rspPkts • rspRTO • senderIsExternal • serverAddr • serverBytes • serverIsExternal • serverL2Bytes • serverPkts • serverPort • serverRTO • serverZeroWnd • version
SSL_HEARTBEAT	• certificateFingerprint • certificateIssuer • certificateKeySize • certificateNotAfter

Event	Available properties
	• certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • clientIsExternal • clientZeroWnd • heartbeatPayloadLength • heartbeatType • isCompressed • receiverIsExternal • senderIsExternal • serverIsExternal • serverZeroWnd • version
SSL_OPEN	• certificateFingerprint • certificateIsSelfSigned • certificateIssuer • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • certificateSubjectAlternativeNames • cipherSuite • clientAddr • clientAlpn • clientBytes • clientCertificateRequested • clientIsExternal • clientL2Bytes • clientPkts • clientPort • clientRTO • clientZeroWnd • handshakeTime • host • isAborted • isCompressed • isPostQuantumKeyAgreement • isRenegotiate • isWeakCipherSuite • ja3Hash • ja3sHash • ja4Fingerprint • keyAgreement • proto • receiverIsExternal • reqBytes

Event	Available properties
	• reqL2Bytes • reqPkts • reqRTO • rspBytes • rspL2Bytes • rspPkts • rspRTO • senderIsExternal • serverAddr • serverAlpn • serverBytes • serverIsExternal • serverL2Bytes • serverPkts • serverPort • serverRTO • serverZeroWnd • version
SSL_RENEGOTIATE **Note:** The SSL_OPEN record format is applied to records committed on this event.	• certificateFingerprint • certificateKeySize • certificateNotAfter • certificateNotBefore • certificateSignatureAlgorithm • certificateSubject • cipherSuite • clientAlpn • clientIsExternal • handshakeTime • host • isAborted • isCompressed • receiverIsExternal • senderIsExternal • serverAlpn • serverIsExternal • version

`recordLength`: ***Number***

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

Access only on SSL_RECORD, SSL_ALERT, or SSL_HEARTBEAT events; otherwise, an error will occur.

`recordType`: ***Number***

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

Access only on SSL_RECORD, SSL_ALERT, and SSL_HEARTBEAT events; otherwise, an error will occur.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last SSL_ALERT, SSL_CLOSE, SSL_HEARTBEAT, SSL_OPEN, SSL_PAYLOAD, SSL_RECORD, or SSL_RENEGOTIATE event ran. The value is NaN if there are no RTT samples.

Access only on SSL_RECORD or SSL_CLOSE events; otherwise, an error will occur.

`serverExtensions:` ***Array | Null***

An array of server extension objects that contain the following properties:

`id:` ***Number***

The ID number of the SSL server extension.

`length:` ***Number***

The full length of the SSL server extension, expressed in bytes.

Note: An extension might be truncated if the length exceeds the maximum size. The default is 512 bytes. Truncation has occurred if the value of this property is smaller than the buffer returned by the `getClientExtensionData()` method.

`name:` ***String***

The name of the TLS server extension, if known. Otherwise, the value indicates that the extension is unknown. See the table of known TLS extensions in the Methods section.

Access only on SSL_OPEN or SSL_RENEGOTIATE events; otherwise, an error will occur.

`serverExtensionsHex:` ***String***

A hexadecimal representation of the sorted list of server extensions.

Note: The Generate Random Extensions And Sustain Extensibility (GREASE) values are removed from the list before encoding.

Access only on SSL_OPEN and SSL_RENEGOTIATE events; otherwise, an error will occur.

`serverBytes:` ***Number***

The total number of bytes sent by the server since the last SSL_RECORD event ran. Note that this property does not return the total number of bytes for the entire SSL session.

Access only on SSL_RECORD or SSL_CLOSE events; otherwise, an error will occur.

`serverHelloVersion:` ***Number***

The version of TLS specified by the server in the server hello packet.

`serverL2Bytes:` ***Number***

The total number of L2 server bytes observed since the last SSL_RECORD event ran.Note that this property does not return the total number of bytes for the entire SSL session.

Access only on SSL_RECORD or SSL_CLOSE events; otherwise, an error will occur.

`serverPkts:` ***Number***

The total number of packets sent by the server since the last SSL_RECORD event ran. Note that this property does not return the total number of packets for the entire SSL session.

Access only on SSL_RECORD or SSL_CLOSE events; otherwise, an error will occur.

`serverSessionId:` ***String***

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

`serverZeroWnd:` ***Number***

The total number of zero windows sent by the server since the last SSL_RECORD event ran. Note that this property does not return the total number of zero windows for the entire SSL session.

Access only on SSL_RECORD or SSL_CLOSE events; otherwise, an error will occur.

`startTLSProtocol:` ***String | Null***

The protocol from which the client sent a STARTTLS command.

`supportedGroupsHex:` ***String***

A hexadecimal representation of the elliptic-curve Diffie-Hellman (ECDH) groups that the client supports.

Access only on `SSL_OPEN` and `SSL_RENEGOTIATE` events; otherwise, an error will occur.

`version:` ***Number***

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

Version	Hex	Decimal
SSLv2	0x200	2
SSLv3	0x300	768
TLS 1.0	0x301	769
TLS 1.1	0x302	770
TLS 1.2	0x303	771
TLS 1.3	0x304	772

TCP

The `TCP` class enables you to access properties and retrieve metrics from TCP events and from `FLOW_TICK` and `FLOW_TURN` events.

The `FLOW_TICK` and `FLOW_TURN` events are defined in the Flow section.

Events

`TCP_CLOSE`

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

`TCP_OPEN`

Runs when the TCP connection is first fully established.

The `FLOW_CLASSIFY` event runs after the `TCP_OPEN` event to determine the L7 protocol of the TCP flow.

Note: If a TCP connection stalls for a long period of time, the TCP_OPEN event runs again when the connection resumes. The following TCP properties and methods are null when the event runs for a resumed connection:

- `getOption`
- `handshakeTime`
- `hasECNEcho`
- `hasECNEcho1`
- `hasECNEcho2`
- `initRcvWndSize`
- `initRcvWndSize1`
- `initRcvWndSize2`
- `initSeqNum`
- `initSeqNum1`
- `initSeqNum2`

- options
- options1
- options2

TCP_PAYLOAD

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

Depending on the Flow, the TCP payload can be found in the following properties:

- Flow.client.payload
- Flow.payload1
- Flow.payload2
- Flow.receiver.payload
- Flow.sender.payload
- Flow.server.payload

Additional payload options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

Methods

getOption(kind: ***Number***): ***Object*** | ***Null***

Returns a TCP option object that matches the specified option kind. For a list of valid option kinds, see TCP options. Specify the TCP client or the TCP server in the syntax—for example, TCP.client.getOption(1) or TCP.server.getOption(1).

Applies only to TCP_OPEN events.

Properties

handshakeTime: ***Number***

The amount of time required to negotiate the TCP connection, expressed in milliseconds.

Access only on TCP_OPEN events; otherwise, an error will occur.

hasECNEcho: ***Boolean***

The value is true if the ECN flag is set on a device during the three-way handshake. Specify the TCP client or the TCP server in the syntax—for example, TCP.client.hasECNEcho or TCP.server.hasECNEcho.

Access only on TCP_OPEN events; otherwise, an error will occur.

hasECNEcho1: ***Boolean***

The value is true if the ECN flag is set during the three-way handshake associated with one of two devices in the connection; the other device is represented by hasECNEcho2. The device represented by hasECNEcho1 remains consistent for the connection.

Access only on TCP_OPEN events; otherwise, an error will occur.

hasECNEcho2: ***Boolean***

The value is true if the ECN flag is set during the three-way handshake associated with one of two devices in the connection; the other device is represented by hasECNEcho1. The device represented by hasECNEcho2 remains consistent for the connection.

Access only on TCP_OPEN events; otherwise, an error will occur.

initRcvWndSize: ***Number***

The initial size of the TCP sliding window on a device negotiated during the three-way handshake. Specify the TCP client or the TCP server in the syntax—for example, TCP.client.initRcvWndSize or TCP.server.initRcvWndSize.

Access only on TCP_OPEN events; otherwise, an error will occur.

`initRcvWndSize1:` ***Number***

The initial size of the TCP sliding window negotiated during the three-way handshake associated with one of two devices in the connection; the other device is represented by `initRcvWndSize2`. The device represented by `initRcvWndSize1` remains consistent for the connection.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`initRcvWndSize2:` ***Number***

The initial size of the TCP sliding window negotiated during the three-way handshake associated with one of two devices in the connection; the other device is represented by `initRcvWndSize1`. The device represented by `initRcvWndSize2` remains consistent for the connection.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`initSeqNum:` ***Number***

The initial sequence number sent from a device during the three-way handshake. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.initSeqNum` or `TCP.server.initSeqNum`.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`initSeqNum1:` ***Number***

The initial sequence number during the three-way handshake associated with one of two devices in the connection; the other device is represented by `initSeqNum2`. The device represented by `initSeqNum1` remains consistent for the connection.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`initSeqNum2:` ***Number***

The initial sequence number during the three-way handshake associated with one of two devices in the connection; the other device is represented by `initSeqNum1`. The device represented by `initSeqNum2` remains consistent for the connection.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`isAborted:` ***Boolean***

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

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

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

`isExpired:` ***Boolean***

The value is `true` if the TCP connection expired at the time of the event. If applicable, specify TCP client or the TCP server in the syntax—for example, `TCP.client.isExpired` or `TCP.server.isExpired`.

Access only on `TCP_CLOSE` events; otherwise, an error will occur.

`isReset:` ***Boolean***

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

`nagleDelay:` ***Number***

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

Access only on `FLOW_TICK` and `FLOW_TURN` events; otherwise, an error will occur.

`nagleDelay1:` ***Number***

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

Access only on `FLOW_TICK` and `FLOW_TURN` events; otherwise, an error will occur.

`nagleDelay1:` ***Number***

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

Access only on `FLOW_TICK` and `FLOW_TURN` events; otherwise, an error will occur.

`options:` ***Array***

An array of objects representing the TCP options of a device in the initial handshake packets. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.options` or `TCP.server.options`. For more information, see the TCP options section below.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`options1:` ***Array***

An array of options representing the TCP options in the initial handshake packets associated with one of two devices in the connection; the other device is represented by `options2`. The device represented by `options1` remains consistent for the connection. For more information, For more information, see the TCP options section below.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`options2:` ***Array***

An array of options representing the TCP options in the initial handshake packets associated with one of two devices in the connection; the other device is represented by `options1`. The device represented by `options2` remains consistent for the connection. For more information, For more information, see the TCP options section below.

Access only on `TCP_OPEN` events; otherwise, an error will occur.

`overlapSegments:` ***Number***

The number of non-identical TCP segments, transmitted by a device in the flow, where two or more TCP segments contain data for the same part of the flow. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.overlapSegments` or `TCP.server.overlapSegments`.

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

`overlapSegments1:` ***Number***

The number of non-identical TCP segments where two or more segments contain data for the same part of the flow. The TCP segments are transmitted by one of two devices in the flow; the other device is represented by `overlapSegments2`. The device represented by `overlapSegments1` remains consistent for the flow.

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

`overlapSegments2:` ***Number***

The number of non-identical TCP segments where two or more segments contain data for the same part of the flow. The TCP segments are transmitted by one of two devices in the flow; the other device is represented by `overlapSegments1`. The device represented by `overlapSegments2` remains consistent for the flow.

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

`rcvWndThrottle:` ***Number***

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

Access only on FLOW_TICK and FLOW_TURN events; otherwise, an error will occur.

`rcvWndThrottle1:` ***Number***

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

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

`rcvWndThrottle2:` ***Number***

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

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

`retransBytes:` ***Number***

The number of bytes retransmitted over TCP by a client or server device in the flow. Specify the TCP client or the TCP server in the syntax—for example, `TCP.client.retransBytes` or `TCP.server.retransBytes`.

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

`retransBytes1:` ***Number***

The number of bytes retransmitted over TCP by one of two devices in the flow; the other device is represented by `retransBytes2`. The device represented by `retransBytes1` remains consistent for the connection.

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

`retransBytes2:` ***Number***

The number of bytes retransmitted over TCP by one of two devices in the flow; the other device is represented by `retransBytes1`. The device represented by `retransBytes2` remains consistent for the connection.

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

`zeroWnd:` ***Number***

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

Access only on FLOW_TICK and FLOW_TURN events; otherwise, an error will occur.

`zeroWnd1:` ***Number***

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

Access only on FLOW_TICK and FLOW_TURN events; otherwise, an error will occur.

`zeroWnd2:` ***Number***

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

Access only on FLOW_TICK and FLOW_TURN events; otherwise, an error will occur.

TCP options

All TCP Options objects have the following properties:

`kind`: ***Number***

The TCP option kind number.

Kind Number	Meaning
0	End of Option List
1	No-Operation
2	Maximum Segment Size
3	Window Scale
4	SACK Permitted
5	SACK
6	Echo (obsoleted by option 8)
7	Echo Reply (obsoleted by option 8)
8	Timestamps
9	Partial Order Connection Permitted (obsolete)
10	Partial Order Service Profile (obsolete)
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 acknowledgments
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)

Kind Number	Meaning
30	Multipath TCP (MPTCP)
31	Reserved (known authorized used without proper IANA assignment)
32	Reserved (known authorized used without proper IANA assignment)
33	Reserved (known authorized used without proper IANA assignment)
34	TCP Fast Open Cookie
35-75	Reserved
76	Reserved (known authorized used without proper IANA assignment)
77	Reserved (known authorized used without proper IANA assignment)
78	Reserved (known authorized used without proper IANA assignment)
79-252	Reserved
253	RFC3692-style Experiment 1 (also improperly used for shipping products)
254	RFC3692-style Experiment 2 (also improperly used for shipping products)

`name:` ***String***

The name of the TCP option.

The following list contains the names of common TCP options and their specific properties:

Maximum Segment Size (name 'mss', option kind 2)

`value:` ***Number***

The maximum segment size.

Window Scale (name 'wscale', kind 3)

`value:` ***Number***

The window scale factor.

Selective acknowledgment Permitted (name 'sack-permitted', kind 4)

No additional properties. Its presence indicates that the selective acknowledgment option was included in the SYN.

Timestamp (name 'timestamp', kind 8)

`tsval:` ***Number***

The TSVal field for the option.

`tsecr:` ***Number***

The TSecr field for the option.

Quickstart Response (name 'quickstart-rsp', kind 27)

`rate-request:` ***Number***

The requested rate for transport, expressed in bytes per second.

`ttl-diff:` ***Number***
The TTLDif.

qs-`nonce:` ***Number***
The QS Nonce.

Akamai Address (name 'akamai-addr', kind 28)

`value:` ***IPAddr***
The IP Address of the Akamai server.

User Timeout (name 'user-timeout', kind 28)

`value:` ***Number***
The user timeout.

Authentication (name 'tcp-ao', kind 29)

`keyId property:` ***Number***
The key id for the key in use.

`rNextKeyId:` ***Number***
The key id for the "receive next" key id.

`mac:` ***Buffer***
The message authentication code.

Multipath (name 'mptcp', kind 30)

`value:` ***Buffer***
The multipath value.

Note: The Akamai address and user timeout options are differentiated by the length of the option.

The following is an example of TCP options:

```
if (TCP.client.options != null) {

   var optMSS = TCP.client.getOption(2)

   if (optMSS && (optMSS.value > 1460)) {
       Network.metricAddCount('large_mss', 1);
       Network.metricAddDetailCount('large_mss_by_client_ip',
                                    Flow.client.ipaddr + " " + optMSS.value,
 1);
   }

}
```

`Telnet`

The `Telnet` class enables you to store metrics and access properties on `TELNET_MESSAGE` events.

Events

`TELNET_MESSAGE`
Runs on a telnet command or line of data from the telnet client or server.

Methods

`commitRecord():` ***void***
Sends a record to the configured recordstore on an `TELNET_MESSAGE` event.

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

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

Properties

`command:` ***String***

The command type. The value is `null` if the event was run due to a line of data being sent.

The following values are valid:

- `Abort`
- `Abort Output`
- `Are You There`
- `Break`
- `Data Mark`
- `DO`
- `DON'T`
- `End of File`
- `End of Record`
- `Erase Character`
- `Erase Line`
- `Go Ahead`
- `Interrupt Process`
- `NOP`
- `SB`
- `SE`
- `Suspend`
- `WILL`
- `WON'T`

`line:` ***String***

A line of the data sent by the client or server. Terminal escape sequences and special characters are filtered out. Cursor movement and line editing are not simulated except for backspace characters.

`option:` ***String***

The option being negotiated. The value is `null` if the option is invalid. The following values are valid:

- `3270-REGIME`
- `AARD`
- `ATCP`
- `AUTHENTICATION`
- `BM`
- `CHARSET`
- `COM-PORT-OPTION`
- `DET`
- `ECHO`
- `ENCRYPT`
- `END-OF-RECORD`
- `ENVIRON`
- `EXPOPL`
- `EXTEND-ASCII`
- `FORWARD-X`
- `GMCP`

- KERMIT
- LINEMODE
- LOGOUT
- NAOCRD
- NAOFFD
- NAOHTD
- NAOHTS
- NAOL
- NAOLFD
- NAOP
- NAOVTD
- NAOVTS
- NAWS
- NEW-ENVIRON
- OUTMRK
- PRAGMA-HEARTBEAT
- PRAGMA-LOGON
- RCTE
- RECONNECT
- REMOTE-SERIAL-PORT
- SEND-LOCATION
- SEND-URL
- SSPI-LOGON
- STATUS
- SUPDUP
- SUPDUP-OUTPUT
- SUPPRESS-GO-AHEAD
- TERMINAL-SPEED
- TERMINAL-TYPE
- TIMING-MARK
- TN3270E
- TOGGLE-FLOW-CONTROL
- TRANSMIT-BINARY
- TTYLOC
- TUID
- X-DISPLAY-LOCATION
- X.3-PAD
- XAUTH

`optionData:` ***Buffer***

For option subnegotiations (the SB command), the raw, option-specific data sent. The value is `null` if the command is not SB.

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `Telnet.commitRecord()` on an `TELNET_MESSAGE` event.

The default record object can contain the following properties:

- `clientIsExternal`
- `command`
- `option`
- `receiverBytes`

- receiverIsExternal
- receiverL2Bytes
- recieverPkts
- receiverRTO
- receiverZeroWnd
- roundTripTime
- senderBytes
- senderIsExternal
- senderL2Bytes
- senderPkts
- senderRTO
- senderZeroWnd
- serverIsExternal

`receiverBytes:` ***Number***

The number of application-level bytes from the receiver.

`receiverL2Bytes:` ***Number***

The number of L2 bytes from the receiver.

`receiverPkts:` ***Number***

The number of packets from the receiver.

`receiverRTO:` ***Number***

The number of retransmission timeouts (RTOs) from the receiver.

`receiverZeroWnd:` ***Number***

The number of zero windows sent by the receiver.

`roundTripTime:` ***Number***

The median round trip time (RTT), expressed in milliseconds. An RTT is the time it took for a device to send a single TCP packet and receive an immediate corresponding acknowledgment (ACK) packet. The median value is calculated by sampling the RTTs observed since the last `TELNET_MESSAGE` event ran. The value is `NaN` if there are no RTT samples.

`senderBytes:` ***Number***

The number of application-level bytes from the sender.

`senderL2Bytes:` ***Number***

The number of L2 bytes from the sender.

`senderPkts:` ***Number***

The number of packets from the sender.

`senderRTO:` ***Number***

The number of retransmission timeouts (RTOs) from the sender.

`senderZeroWnd:` ***Number***

The number of zero windows sent by the sender.

TFTP

The TFTP (Trivial File Transfer Protocol) class enables you to store metrics and access properties on `TFTP_REQUEST` and `TFTP_RESPONSE` events.

Events

`TFTP_REQUESTS`

Runs on every TFTP request processed by the device.

`TFTP_RESPONSE`

Runs on every TFTP response processed by the device.

Methods

`commitRecord()`: ***void***

Sends a record to the configured recordstore on a `TFTP_RESPONSE` event. Record commits on `TFTP_REQUEST` events are not supported.

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

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

Properties

`blocks`: ***Number***

The number of data blocks written or read.

Access only on `TFTP_RESPONSE` events; otherwise, an error will occur.

`error`: ***String | null***

The detailed error message recorded by the ExtraHop system.

Access only on `TFTP_RESPONSE` events; otherwise, an error will occur.

`fileComplete`: ***Boolean***

If the value is `false`, only part of the file was transferred, either because the client timed out during a write operation or the server timed out during a read operation.

Access only on `TFTP_RESPONSE` events; otherwise, an error will occur.

`filename`: ***String***

The name of the file transferred.

`mode`: ***String***

The mode that the file was transferred with. The following values are valid:

- `netascii`
- `octet`
- `mail`

`operation`: ***String***

The TFTP operation. The following values are valid:

- `READ`
- `WRITE`

`payload`: ***Buffer***

The Buffer object that contains the raw payload bytes of the first data block transferred. The maximum size of a block is 512 bytes.

`payloadMediaType`: ***String***

The type of file transferred.

Access only on `TFTP_RESPONSE` events; otherwise, an error will occur.

`payloadSHA256`: ***String***

The hexadecimal representation of the SHA-256 hash of the payload. The string contains no delimiters, as shown in the following example:

```
468c6c84db844821c9ccb0983c78d1cc05327119b894b5ca1c6a1318784d3675
```

Access only on `TFTP_RESPONSE` events; otherwise, an error will occur.

`size:` ***Number***

The size of the file transferred, expressed in bytes.

Access only on `TFTP_RESPONSE` events; otherwise, an error will occur.

`Turn`

`Turn` is a class that enables you to store metrics and access properties available on `FLOW_TURN` events.

The `FLOW_TURN` event is defined in the Flow section.

Properties

`clientBytes:` ***Number***

The total number of bytes sent by the client since the last `FLOW_TURN` event ran.

`clientTransferTime:` ***Number***

The client transfer time, expressed in milliseconds.

`processingTime:` ***Number***

The time elapsed between when the client transfers the request to the server and when the server begins to transfer the response back to the client, expressed in milliseconds.

`reqSize:` ***Number***

The size of the request payload, expressed in bytes.

`reqTransferTime:` ***Number***

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

`rspSize:` ***Number***

The size of the response payload, expressed in bytes.

`rspTransferTime:` ***Number***

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

`serverBytes:` ***Number***

The total number of bytes sent by the server since the last `SSL_RECORD` event ran.

`serverTransferTime:` ***Number***

The server transfer time, expressed in milliseconds.

`sourceDevice:` ***Device***

The source device object. See the Device class for more information.

`thinkTime:` ***Number***

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

UDP

The UDP class enables you to access properties and retrieve metrics from UDP events and from FLOW_TICK and FLOW_TURN events.

The FLOW_TICK and FLOW_TURN events are defined in the Flow section.

Events

UDP_PAYLOAD

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

Depending on the Flow, the UDP payload can be found in the following properties:

- Flow.client.payload
- Flow.payload1
- Flow.payload2
- Flow.receiver.payload
- Flow.sender.payload
- Flow.server.payload

Additional payload options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

WebSocket

The WebSocket class enables you to access properties on WEBSOCKET_OPEN, WEBSOCKET_CLOSE, and WEBSOCKET_MESSAGE events.

Events

WEBSOCKET_OPEN

Runs when a successful handshake has been observed.

WEBSOCKET_CLOSE

Runs when both close frames are observed, or when the underlying TCP connection is closed.

WEBSOCKET_MESSAGE

Runs when all frames of a text or binary message have been observed.

Properties

clientBytes: ***Number***

The total number of bytes sent by the client during the entire WebSockets session.

Access only on WEBSOCKET_MESSAGE events; otherwise, an error will occur.

clientL2Bytes: ***Number***

The total number of L2 client bytes observed during the entire WebSockets session.

Access only on WEBSOCKET_MESSAGE events; otherwise, an error will occur.

clientPkts: ***Number***

The total number of packets sent by the client during the entire WebSockets session.

Access only on WEBSOCKET_MESSAGE events; otherwise, an error will occur.

clientRTO: ***Number***

The total number of client retransmission timeouts (RTOs) observed during the WebSockets session.

Access only on WEBSOCKET_MESSAGE events; otherwise, an error will occur.

`clientZeroWnd:` ***Number***

The total number of zero windows sent by the client during the entire WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`closeReason:` ***String***

The text message included in the first observed close frame that describes the reason the connection was closed. The value is `null` if the frame does not contain this information.

Access only on `WEBSOCKET_CLOSE` events; otherwise, an error will occur.

`host:` ***String***

The host provided in the handshake request from the client. The value is `null` if no host is provided.

Access only on `WEBSOCKET_OPEN` events; otherwise, an error will occur.

`isClientClose:` ***Boolean***

The value is `true` if the initial close frame was sent by the client.

Access only on `WEBSOCKET_CLOSE` events; otherwise, an error will occur.

`isEncrypted:` ***Boolean***

The value is `true` if the WebSocket connection is TLS-encrypted.

`isMasked: Boolean`

The value is true if the frames of the WebSocket message are masked.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`isServerClose:` ***Boolean***

The value is `true` if the initial close frame was sent by the server. The value is `false` if the connection was terminated abnormally.

Access only on `WEBSOCKET_CLOSE` events; otherwise, an error will occur.

`msg:` ***Buffer***

The Buffer object containing the WebSocket message. If the message is compressed, the buffer contains the decompressed message. The buffer is `null` if the contents exceed the maximum length.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`msgLength:` ***Number***

The length of the message, expressed in bytes. If the message is compressed, the length reflects the total length of the decompressed message, even if the message exceeds the maximum length.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`msgType:` ***String***

The type of WebSocket message frame. Valid values are `TEXT` or `BINARY`.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`origin:` ***String***

The origin URL provided in the handshake request initiated by the client.

Access only on `WEBSOCKET_OPEN` events; otherwise, an error will occur.

`rawMsgLength:` ***Number***

The length of the raw message as it was observed, expressed in bytes. If the message is compressed, this property reflects the length of the compressed message.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`serverBytes:` ***Number***

The total number of bytes sent by the server during the entire WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`serverL2Bytes:` ***Number***

The total number of L2 server bytes observed during the entire WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`serverPkts:` ***Number***

The total number of packets sent by the server during the entire WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`serverRTO:` ***Number***

The total number of server retransmission timeouts (RTOs) observed during the WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`serverZeroWnd:` ***Number***

The total number of zero windows sent by the server during the entire WebSockets session.

Access only on `WEBSOCKET_MESSAGE` events; otherwise, an error will occur.

`statusCode:` ***Number***

The status code that represents the reason the connection was closed, as defined in RFC 6455.

The value is `NO_STATUS_RECVD` (`1005`) if the initial close frame does not include a status code. The value is `NaN` if connection was terminated abnormally.

Access only on `WEBSOCKET_CLOSE` events; otherwise, an error will occur.

`uri:` ***String***

The URI provided in the handshake request initiated by the client.

Access only on `WEBSOCKET_OPEN` events; otherwise, an error will occur.

`WSMAN`

The `WSMAN` class enables you to store metrics and access properties on `WSMAN_REQUEST` and `WSMAN_RESPONSE` events. Web Services-Management (WSMAN) and the Microsoft implementation Windows Remote Management (WinRM) are protocols that enable devices to exchange management information on a network.

Events

`WSMAN_REQUEST`

Runs on every `WSMAN_REQUEST` processed by the device.

`WSMAN_RESPONSE`

Runs on every `WSMAN_RESPONSE` processed by the device.

Methods

`commitRecord():`***void***

Sends a record to the configured recordstore on either a `WSMAN_REQUEST` or `WSMAN_RESPONSE` event. To view the default properties committed on each event, see the record property below.

If the `commitRecord()` method is called on an `WSMAN_REQUEST` event, the record is not created until the `WSMAN_RESPONSE` event runs. If the `commitRecord()` method is called on both the `WSMAN_REQUEST` and the corresponding `WSMAN_RESPONSE`, only one record is created for request and response, even if the `commitRecord()` method is called multiple times on the same trigger events.

Properties

`commandLine:` ***String | Null***

The full command line specified in the WSMAN request. If the WSMAN request did not specify a command line, the value is null.

`encryptionProtocol:` ***String***

The protocol that the transaction is encrypted with.

`isEncrypted:` ***Boolean***

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

`isDecrypted:` ***Boolean***

The value is `true` if the ExtraHop system securely decrypted and analyzed the transaction. Decrypted traffic analysis can expose advanced threats that hide within encrypted traffic.

`operationId:` ***String***

The unique identifier of the operation.

`payload:` ***Buffer***

A buffer object containing the XML message envelope. Messages longer than the maximum size are truncated. The maximum size is configured in the WSMAN profile in the running config. The following running config example changes the maximum message size from its default of 1024 bytes to 4096:

```
"capture": {
    "app_proto": {
        "wsman": {
            "payload_max_size": 4096
        }
    }
}
```

`record:` ***Object***

The record object that can be sent to the configured recordstore through a call to `WSMAN.commitRecord()`.

The default record object can contain the following properties:

- `clientAddr`
- `clientIsExternal`
- `clientPort`
- `serverAddr`
- `serverPort`
- `proto`
- `timestamp`
- `user`
- `vlan`
- `operationId`
- `receiverIsExternal`
- `reqAction`
- `reqResourceURI`
- `rspAction`
- `rspResourceURI`
- `senderIsExternal`
- `sequenceId`
- `serverIsExternal`

Access the record object only on `WSMAN_RESPONSE` events; otherwise, an error will occur.

`reqAction:` ***String***

The action requested by the client to be performed by the resource specified in the resourceURI.

Access only on `WSMAN_REQUEST` events; otherwise, an error will occur.

`reqCommand:` ***String | null***

The command specified in the request. If no command is specified, the value is null.

`reqResourceURI:` ***String***

The Uniform Resource Identifier (URI) of the resource that performs an action.

`rspAction:` ***String***

The server response to the action requested by the client.

Access only on `WSMAN_RESPONSE` events; otherwise, an error will occur.

`rspResourceURI:` ***String***

The Uniform Resource Identifier (URI) of the resource that performs an action.

`sequenceId:` ***String***

The string representation of a 64-bit integer that identifies a message in an operation.

`user:` ***String***

The username of the account that sent the request.

Open data stream classes

The Trigger API classes in this section enable you to send data to a third-party syslog, database, or server through an open data stream (ODS) you have configured in the Administration settings.

Class	Description
Remote.HTTP	Enables you to submit HTTP request data to a remote server through REST API endpoints.
Remote.Kafka	Enables you to submit message data to remote a Kafka server.
Remote.MongoDB	Enables you to insert, remove, and update document collections to a remote MongoDB database.
Remote.Raw	Enables you to submit raw data to a remote server through a TCP or UDP port.
Remote.Syslog	Enables you to send syslog data to a remote server.

`Remote.HTTP`

The `Remote.HTTP` class enables you to submit HTTP request data to an HTTP open data stream (ODS) target and provides access to HTTP REST API endpoints.

You must first configure an HTTP ODS target from the Administration settings, which requires system and access administration privileges. For configuration information, see the Open Data Streams section in the Sensor Administration Guide.

Methods

`delete`

Submits an HTTP REST delete request to a configured HTTP open data stream.

Syntax:

```
Remote.HTTP("name").delete({path: "path", headers: {header:
 "header"},
payload: "payload"})
```

```
Remote.HTTP.delete({path: "path", headers: {header: "header"},
 payload: "payload"})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`options:` ***Object***

The options object has the following properties:

`path:` ***String***

The string specifying the request path.

`headers:` ***Object***

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- `Connection`
- `Authorization`
- `Proxy-Connection`
- `Content-Length`
- `X-Forwarded-For`
- `Transfer-Encoding`

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Administration settings.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Administration settings. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests with 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.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

`get`

Submits an HTTP REST get request to a configured HTTP open data stream.

Syntax:

```
Remote.HTTP("name").get({path: "path", headers: {header:
 "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.get({path: "path", headers: {header: "header"},
 payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`options:` ***Object***

The options object has the following properties:

`path:` ***String***

The string specifying the request path.

`headers:` ***Object***

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- `Connection`
- `Authorization`
- `Proxy-Connection`
- `Content-Length`
- `X-Forwarded-For`
- `Transfer-Encoding`

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Administration settings.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Administration settings. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests with 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.

`enableResponseEvent:` ***Boolean***

Enables a trigger to run on the HTTP response that is sent by the ODS target by creating a REMOTE_RESPONSE event.

Important: Processing a large number of HTTP responses can affect trigger performance and efficiency. We recommend that you enable this option only if necessary.

`context:` ***Object*** | ***String*** | ***Number*** | ***Boolean*** | ***null***

An optional object that is sent to the trigger that is running on the HTTP response from the ODS target. You can access information stored in the object by specifying the `Remote.response.context` property.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

`patch`

Submits an HTTP REST patch request to a configured HTTP open data stream.

Syntax:

```
Remote.HTTP("name").patch({path: "path", headers: {header:
 "header"},
```

```
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.patch({path: "path", headers: {header: "header"},
 payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`options:` ***Object***

The options object has the following properties:

`path:` ***String***

The string specifying the request path.

`headers:` ***Object***

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- `Connection`
- `Authorization`
- `Proxy-Connection`
- `Content-Length`
- `X-Forwarded-For`
- `Transfer-Encoding`

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Administration settings.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Administration settings. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests with 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.

`enableResponseEvent:` ***Boolean***

Enables a trigger to run on the HTTP response that is sent by the ODS target by creating a REMOTE_RESPONSE event.

Important: Processing a large number of HTTP responses can affect trigger performance and efficiency. We recommend that you enable this option only if necessary.

`context:` ***Object*** | ***String*** | ***Number*** | ***Boolean*** | ***null***

An optional object that is sent to the trigger that is running on the HTTP response from the ODS target. You can access information stored in the object by specifying the `Remote.response.context` property.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

`post`

Submits an HTTP REST post request to a configured HTTP open data stream.

Syntax:

```
Remote.HTTP("name").post({path: "path", headers: {header:
 "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.post({path: "path", headers: {header: "header"},
 payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`options:` ***Object***

The options object has the following properties:

`path:` ***String***

The string specifying the request path.

`headers:` ***Object***

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- `Connection`
- `Authorization`
- `Proxy-Connection`
- `Content-Length`
- `X-Forwarded-For`
- `Transfer-Encoding`

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Administration settings.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Administration settings. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests with 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.

`enableResponseEvent:` ***Boolean***

Enables a trigger to run on the HTTP response that is sent by the ODS target by creating a REMOTE_RESPONSE event.

Important: Processing a large number of HTTP responses can affect trigger performance and efficiency. We recommend that you enable this option only if necessary.

`context:` ***Object*** | ***String*** | ***Number*** | ***Boolean*** | ***null***

An optional object that is sent to the trigger that is running on the HTTP response from the ODS target. You can access information stored in the object by specifying the `Remote.response.context` property.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

`put`

Submits an HTTP REST put request to a configured HTTP open data stream.

Syntax:

```
Remote.HTTP("name").put({path: "path", headers: {header:
 "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.put({path: "path", headers: {header: "header"},
 payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`options:` ***Object***

The options object has the following properties:

`path:` ***String***

The string specifying the request path.

`headers:` ***Object***

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- `Connection`
- `Authorization`
- `Proxy-Connection`
- `Content-Length`
- `X-Forwarded-For`
- `Transfer-Encoding`

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or

through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Administration settings.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Administration settings. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests with 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.

`enableResponseEvent:` ***Boolean***

Enables a trigger to run on the HTTP response that is sent by the ODS target by creating a REMOTE_RESPONSE event.

Important: Processing a large number of HTTP responses can affect trigger performance and efficiency. We recommend that you enable this option only if necessary.

`context:` ***Object*** | ***String*** | ***Number*** | ***Boolean*** | ***null***

An optional object that is sent to the trigger that is running on the HTTP response from the ODS target. You can access information stored in the object by specifying the `Remote.response.context` property.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

`request`

Submits an HTTP REST request to a configured HTTP open data stream.

Syntax:

```
Remote.HTTP("name").request("method", {path: "path", headers:
 {header: "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.request("method", {path: "path", headers: {header:
 "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`method:` ***String***

String that specifies the HTTP method.

- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `TRACE`
- `OPTIONS`
- `CONNECT`
- `PATCH`

`options:` ***Object***

The options object has the following properties:

`path:` ***String***

The string specifying the request path.

`headers:` ***Object***

The optional object specifying the request headers. The following headers are restricted and will result in an error if specified:

- `Connection`
- `Authorization`
- `Proxy-Connection`
- `Content-Length`
- `X-Forwarded-For`
- `Transfer-Encoding`

Note: Authorization headers must be specified by either a built-in authentication method, such as Amazon Web Services, or through the **Additional HTTP Header** field in the Open Data Streams configuration window in the Administration settings.

Headers configured in a trigger take precedence over an entry in the **Additional HTTP Header** field, which is located in the Open Data Streams configuration window in the Administration settings. For example, if the **Additional HTTP Header** field specifies `Content-Type: text/plain`, but a trigger script on the same ODS target specifies `Content-Type: application/json`, then `Content-Type: application/json` is included in the HTTP request.

You can compress the outgoing HTTP requests with 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.

`enableResponseEvent:` ***Boolean***

Enables a trigger to run on the HTTP response that is sent by the ODS target by creating a REMOTE_RESPONSE event.

Important: Processing a large number of HTTP responses can affect trigger performance and efficiency. We recommend that you enable this option only if necessary.

`context`: ***Object*** | ***String*** | ***Number*** | ***Boolean*** | ***null***

An optional object that is sent to the trigger that is running on the HTTP response from the ODS target. You can access information stored in the object by specifying the `Remote.response.context` property.

Return Values:

Returns `true` if the request is queued, otherwise returns `false`.

Helper methods

The following helper methods are available for 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: {header: "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.delete({path: "path", headers: {header: "header"}, payload:
 "payload", enableResponseEvent: "enableResponseEvent", context:
 "context"})
```

```
Remote.HTTP("name").get({path: "path", headers: {header: "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.get({path: "path", headers: {header: "header"}, payload:
 "payload", enableResponseEvent: "enableResponseEvent", context:
 "context"})
```

```
Remote.HTTP("name").patch({path: "path", headers: {header: "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.patch({path: "path", headers: {header: "header"}, payload:
 "payload", enableResponseEvent: "enableResponseEvent", context:
 "context"})
```

```
Remote.HTTP("name").post({path: "path", headers: {header: "header"},
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.post({path: "path", headers: {header: "header"}, payload:
 "payload", enableResponseEvent: "enableResponseEvent", context:
 "context"})
```

```
Remote.HTTP("name").put({path: "path", headers: {header: "header"},
```

```
payload: "payload", enableResponseEvent: "enableResponseEvent",
 context: "context"})
```

```
Remote.HTTP.put({path: "path", headers: {header: "header"}, payload:
 "payload", enableResponseEvent: "enableResponseEvent", context:
 "context"})
```

Return values:

Returns `true` if the request is queued, otherwise returns `false`.

Examples

`HTTP GET`

The following example will issue an HTTP GET request to the HTTP configuration called "my_destination" and a path that is the URI, including query string variables, that you want the request to be sent to.

```
Remote.HTTP("my_destination").get( { path: "/?
example=example1&example2=my_data" } );
```

`HTTP POST`

The following example will issue an HTTP POST request to the HTTP configuration called "my_destination", the path that is the URI you want the request to be sent to and a payload. The payload can be data similar to what an HTTP client would send, a JSON blob, XML, or whatever else you want to send.

```
Remote.HTTP("my_destination").post( { path: "/", payload: "data I want
 to
send" } );
```

Custom HTTP Headers

The following example defines a Javascript object with keys to represent the header names and their corresponding values and provide that in a call as the value for the headers key.

```
var my_json = { example: "my_data", example1: 42, example2: false };
var headers = { "Content-Type": "application/json" };
Remote.HTTP("my_destination").post( { path: "/", headers: headers,
 payload:
JSON.stringify(my_json) });
```

Trigger Examples

- Example: Send data to Elasticsearch with Remote.HTTP
- Example: Send data to Azure with Remote.HTTP

`Remote.Kafka`

The `Remote.Kafka` class enables you to submit message data to a Kafka server through a Kafka open data stream (ODS).

You must first configure a Kafka ODS target from the Administration settings, which requires system and access administration privileges. For configuration information, see the Open Data Streams section in the Sensor Administration Guide.

Methods

`send`

Sends an array of messages to a single topic with an option to indicate which Kafka partition the messages will be sent to.

Syntax:

```
Remote.Kafka.send({"topic": "topic", "messages":[messages],
"partition": partition})
```

```
Remote.Kafka("name").send({"topic": "topic", "messages":
[messages],
"partition": partition})
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`topic:` ***String***

A string corresponding to the topic associated with the Kafka `send` method. The topic string has the following restrictions:

- The string length must be between 1 and 249 characters.
- The string supports only alphanumeric characters and the following symbols: "-", "_", or ".".
- The string cannot be "." or "..".

`messages:` ***Array***

An optional array of messages to be sent. An element in this array cannot be an array itself.

`partition:` ***Number***

An optional non-negative integer corresponding to the Kafka partition the messages will be sent to. The `send` action will fail silently if the number provided exceeds the number of partitions on the Kafka cluster associated with the given target. This value is ignored unless **Manual Partitioning** is selected as the partitioning strategy when you configured the open data stream in the Administration settings.

Return values:

None

Examples:

```
Remote.Kafka.send({"topic": "my_topic", "messages": ["hello
 world", 42,
DHCP.msgType], "partition": 2});
```

```
Remote.Kafka("my-target").send({"topic": "my_topic", "messages":
 [HTTP.query,
HTTP.uri]});
```

`send`

Sends messages to a single topic.

Syntax:

```
Remote.Kafka.send("topic", message1, message2, etc...)
```

```
Remote.Kafka("my-target").send("topic", message1, message2,
 etc...)
```

Parameters:

If `Remote.Kafka.send` is called with multiple arguments, the following fields are required:

`topic:` ***String***

A string corresponding to the topic associated with the Kafka `send` method. The topic string has the following restrictions:

- The string length must be between 1 and 249 characters.
- The string supports only alphanumeric characters and the following symbols: "-", "_", or ".".
- The string cannot be "." or "..".

`messages:` ***String | Number***

The messages to be sent. This cannot be an array.

Return values:

None.

Examples:

```
Remote.Kafka.send("my_topic", HTTP.query, HTTP.uri);
```

```
Remote.Kafka("my-target").send("my_topic", HTTP.query, HTTP.uri);
```

`Remote.MongoDB`

The `Remote.MongoDB` class enables you to insert, remove, and update MongoDB document collections through a MongoDB open data stream (ODS).

You must first configure a MongoDB ODS target from the Administration settings, which requires system and access administration privileges. For configuration information, see the Open Data Streams section in the Sensor Administration Guide.

Methods

`insert`

Inserts a document or array of documents into a collection, and handles both add and modify operations.

Syntax:

```
Remote.MongoDB.insert("db.collection", document);
```

```
Remote.MongoDB("name").insert("db.collection", document);
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`collection:` ***String***

The name of a group of MongoDB documents.

`document:` ***Object***

The JSON-formatted document to insert into the collection.

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("collection", document, justOnce);
```

```
Remote.MongoDB("name").remove("collection", document, justOnce]);
```

Parameters:

`name:` ***String***

The optional name of the host specified when you configured the open data stream in the Administration settings. If no host is specified, the value is the default host.

`collection:` ***String***

The name of a group of MongoDB documents.

`document:` ***Object***

The JSON-formatted document to remove from the collection.

`justOnce:` ***Boolean***

An optional boolean parameter that limits the removal to just one document. Set to `true` to limit the deletion. The default value is `false`.

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("collection", document, update,
 {"upsert":true,
"multi":true});
```

```
Remote.MongoDB("name").update("collection", document, update,
{"upsert":true, "multi":true});
```

Parameters:

`collection:` ***String***

The name of a group of MongoDB documents.

`document:` ***Object***

The JSON-formatted document that specifies which documents to update or insert, if upsert option is set to true.

`update:` ***Object***

The JSON-formatted document that specifies how to update the specified documents.

`name:` ***String***

The name of the host specified when you configured the open data stream in the Administration settings. If no host was specified, the value is the default host.

`options:`

Optional flags that indicate the following additional update options:

`upsert:` ***Boolean***

An optional boolean parameter that creates a new document when no document matches the query data. Set to `true` to create a new document. The default value is `false`.

`multi:` ***Boolean***

An optional boolean parameter that updates all documents that match the query data. Set to `true` to update multiple documents. The default value is `false`, which updates only the first document returned.

Return Values:

The value is `true` if the request is queued, otherwise returns FALSE.

Examples:

```
var x = Remote.MongoDB.update('test.tbc', {_id: 1}, {$set:
 {example:2}},
{'upsert':true, 'multi':false} );
if (x) {
    Network.metricAddCount('perf_trigger_success', 1);
}
else {
    Network.metricAddCount('perf_trigger_error', 1);
}
```

Refer to http://docs.mongodb.org/manual/reference/method/db.collection.update/#db.collection.update for more information.

Trigger Examples

- Example: Parse syslog over TCP with universal payload analysis

Remote.Raw

The Remote.Raw class enables you to submit raw data to a Raw open data stream (ODS) target through a TCP or UDP port.

You must first configure a raw ODS target from the Administration settings, which requires system and access administration privileges. For configuration information, see the Open Data Streams section in the Sensor Administration Guide.

Note: If the Gzip feature is enabled for the raw data stream in the Administration settings, the Remote.Raw class will automatically compress the data with Gzip.

Methods

send

Sends raw data to a Raw open data stream (ODS) target through a TCP or UDP port.

Syntax:

```
Remote.Raw.send("data")
```

```
Remote.Raw("name").send("data")
```

Parameters:

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`data:` ***String***

The JavaScript string representing the bytes to send.

Return Values:

None

Examples

```
Remote.Raw.send("data over the wire");
```

```
Remote.Raw("my-target").send("extra data for my-target");
```

Remote.Syslog

The Remote.Syslog class enables you to create remote syslog messages and send message data to a Syslog open data stream (ODS).

You must first configure a syslog ODS target from the Administration settings, which requires system and access administration privileges. For configuration information, see the Open Data Streams section in the Sensor Administration Guide.

Note: 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, in other words, calling them with the wrong number or type of arguments, will still result in trigger execution stopping.

Methods

`emerg(message:String):void`

Sends a message to the remote syslog server with an emergency severity level.

Syntax:

```
Remote.Syslog.emerg("eh_event=web uri=" + HTTP.uri + " req_size="
 + HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").emerg("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`alert(message:String):void`

Sends a message to the remote syslog server with an alert severity level.

Syntax:

```
Remote.Syslog.alert("eh_event=web uri=" + HTTP.uri + " req_size="
 + HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").alert("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`crit(message:String):void`

Sends a message to the remote syslog server with a critical severity level.

Syntax:

```
Remote.Syslog.crit("eh_event=web uri=" + HTTP.uri + " req_size=" +
 HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").crit("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`error(message:`***String***`):`***void***

Sends a message to the remote syslog server with an error severity level.

Syntax:

```
Remote.Syslog.error("eh_event=web uri=" + HTTP.uri + " req_size="
 + HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").error("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`warn(message:`***String***`):`***void***

Sends a message to the remote syslog server with a warning severity level.

Syntax:

```
Remote.Syslog.warn("eh_event=web uri=" + HTTP.uri + " req_size=" +
 HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").warn("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`notice(message:`***String***`):`***void***

Sends a message to the remote syslog server with a notice severity level.

Syntax:

```
Remote.Syslog.notice("eh_event=web uri=" + HTTP.uri + " req_size="
 + HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").notice("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

`info(message:`***String***`):`***void***

Sends a message to the remote syslog server with an info severity level.

Syntax:

```
Remote.Syslog.info("eh_event=web uri=" + HTTP.uri + " req_size=" +
 HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").info("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

`name:` ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

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

Sends a message to the remote syslog server with a debug severity level.

Syntax:

```
Remote.Syslog.debug("eh_event=web uri=" + HTTP.uri + " req_size="
 + HTTP.reqSize + "
rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

```
Remote.Syslog("name").debug("eh_event=web uri=" + HTTP.uri + "
 req_size=" +
HTTP.reqSize + " rsp_size=" + HTTP.rspSize + " processingTime=" +
 HTTP.processingTime);
```

Parameters

name: ***String***

The name of the ODS target that requests are sent to. If this field is not specified, the name is set to `default`.

Message size

By default, the message sent to the remote server is limited to 1024 bytes, including the message header and trailer (if necessary). The message header always includes the priority and timestamp, which together are up to 30 bytes.

If you have system and access administration privileges, you can increase the default message size in the Administration settings. Click **Running Config** from the Appliance Settings section, and then click **Edit config**. Go to the "remote" section, and under the ODS target name, such as "rsyslog", add "message_length_max" as shown in the example below. The "message_length_max" setting applies only to the message passed to the Remote.Syslog APIs; the message header does not count against the maximum.

```
"remote": {
   "rsyslog": {
```

```
    "host": "hostname",
    "port": 54322,
    "ipproto": "tcp",
    "message_length_max": 4000
    }
}
```

Timestamp

The default timestamp format for rsyslog messages is UTC. You can change the timestamp to local time when you configure the open data stream in the Administration settings.

Trigger Examples

- Example: Send discovered device data to a remote syslog server
- Example: Parse syslog over TCP with universal payload analysis
- Example: Matching topnset keys

Remote

The Remote class enables you to send data to a third-party syslog, database, or server through an open data stream (ODS) and access responses returned by HTTP ODS targets.

Events

REMOTE_RESPONSE

Runs when the ExtraHop system receives a response from an HTTP ODS target.

Note: A trigger runs on the REMOTE_RESPONSE event only if the trigger created the ODS request that caused the response.

Properties

response: ***Object***

An object that contains information from the HTTP response returned by the ODS target. The response object has the following properties:

statusCode: ***Number***

The status code returned by the ODS target.

body: ***Buffer***

The body of the HTTP response sent by the ODS target.

headers: ***Object***

An object that contains the headers of the HTTP response sent by the ODS target. If the response contains multiple headers with the same name, the value for the header is an array. For example, if Set-Cookie is specified multiple times in the response, you can access the first cookie by specifying Remote.response.headers["Set-Cookie"][0].

context: ***Object*** | ***String*** | ***Number*** | ***Boolean*** | ***null***

The context information specified in the Remote.HTTP context parameter when the ODS request was sent. For more information see Remote.HTTP.

Datastore classes

The Trigger API classes in this section enable you to access datastore, or bridge, metrics.

Class	Description
AlertRecord	Enables you to access alert information on `ALERT_RECORD_COMMIT` events.
Dataset	Enables you to access raw dataset values and provides an interface for computing percentiles.
MetricCycle	Enables you to retrieve metrics published during a metric cycle interval represented by the `METRIC_CYCLE_BEGIN`, `METRIC_CYCLE_END`, and `METRIC_RECORD_COMMIT` events.
MetricRecord	Enables access to the current set of metrics on `METRIC_RECORD_COMMIT` events.
Sampleset	Enables you to retrieve summary data about metrics.
Topnset	Enables you to access data from a collection of metrics grouped by a key such as a URI or a client IP address.

`AlertRecord`

The AlertRecord class enables you to access alert information on `ALERT_RECORD_COMMIT` events.

Events

`ALERT_RECORD_COMMIT`

Runs when an alert occurs. Provides access to information about the alert.

Additional datastore options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

Important: This event runs only if the NPM module is enabled on the ExtraHop system. If your user account has not been granted NPM module access, you cannot configure a trigger to run on this event.

Properties

`description:` ***String***

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

`id:` ***String***

The ID of the alert record. Alert record IDs are named according to the following format:

```
extrahop.<object>.<alert_type>
```

<object> is the type of object that the alert applies to. For network objects, the <object> value is capture. If the alert is for a detail topnset metric, the <alert_type> is alert_detail; otherwise, the <alert_type> is alert. The following alert record IDs are valid:

- extrahop.capture.alert
- extrahop.capture.alert_detail
- extrahop.device.alert
- extrahop.device.alert_detail
- extrahop.application.alert
- extrahop.application.alert_detail
- extrahop.flow_network.alert
- extrahop.flow_network.alert_detail
- extrahop.flow_interface.alert
- extrahop.flow_interface.alert_detail

Note: You can restrict the trigger to only run for specified alert record types. Type a comma-separated list of alert record IDs in the **Metric types** field of the Advanced trigger options.

`name:` ***String***

The name of the alert.

`object:` ***Object***

The object the alert applies to. For device, application, capture, flow interface, or flow network alerts, this property will contain a Device, Application, Network, FlowInterface, or FlowNetwork object, respectively.

`time:` ***Number***

The time that the alert record will be published with.

`severityName:` ***String***

The name of the alert severity level. The following severity levels are supported:

Value	Description
emerg	Emergency
alert	Alert
crit	Critical
err	Error
warn	Warning
notice	Notice
info	Info
debug	Debug

`severityLevel:` ***Number***

The numeric alert severity level. The following severity levels are supported:

Value	Description
0	Emergency
1	Alert
2	Critical
3	Error

Value	Description
4	Warning
5	Notice
6	Info
7	Debug

Dataset

The dataset class enables you to access raw dataset values and provides an interface for computing percentiles.

Instance Methods

`percentile(...):` ***Array*** | ***Number***

Accepts a list of percentiles (either as an array or as multiple arguments) to compute and returns the computed percentile values for the dataset. If passed a single numeric argument, a number is returned. Otherwise an array is returned. The arguments must be in ascending order with no duplicates. Floating point values, such as 99.99, are allowed.

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.

MetricCycle

The `MetricCycle` class represents an interval during which metrics are published. The MetricCycle class is valid on `METRIC_CYCLE_BEGIN`, `METRIC_CYCLE_END`, and `METRIC_RECORD_COMMIT` events.

The `METRIC_RECORD_COMMIT` event is defined in the MetricRecord section.

Events

`METRIC_CYCLE_BEGIN`

Runs when a metric interval begins.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

`METRIC_CYCLE_END`

Runs when a metric interval ends.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

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

Properties

`id:` ***String***

A string representing the metric cycle. The only possible value is `30sec`.

`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 the `Flow.store` object. The `store` object is shared among triggers for `METRIC_* events`. It is cleared at the end of a metric cycle.

Trigger Examples

- Example: Add metrics to the metric cycle store

`MetricRecord`

The `MetricRecord` class enables you to access to the current set of metrics on `METRIC_RECORD_COMMIT` events.

Events

`METRIC_RECORD_COMMIT`

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

Additional datastore options are available when you create a trigger that runs on this event. See Advanced trigger options for more information.

Note: You cannot assign triggers that run only on this event to specific devices or device groups. Triggers that run on this event will run whenever this event occurs.

Properties

`fields:` ***Object***

An object containing metric values. The properties are the field names and the values can be numbers, Topnset, Dataset or Sampleset.

`id:` ***String***

The metric type, such as `extrahop.device.http_server`.

`object:` ***Object***

The object the metric applies to. For device, application, or VLAN alerts, this property contains a Device object, an Application object, or a VLAN instance, respectively. For capture metrics, such as `extrahop.capture.net`, the property contains a Network object. The following example code stores the ID of an application in a variable:

```
var app_id = MetricRecord.object.id;
```

Note: The example code above always generates the following warning in the trigger editor:

```
Property 'id' does not exist on type 'Device | Application
 | VLAN | Network'. ts(2339) [2, 33]
Property 'id' does not exist on type 'Network'.
```

The warning indicates that assigning the trigger to a network is not supported. You can ignore this warning when the trigger is assigned to an application.

`time:` ***Number***

The publish time of the metric record.

Trigger Examples

- Example: Matching topnset keys
- Example: Add metrics to the metric cycle store

Sampleset

The Sampleset class enables you to retrieve summary data about metrics.

Properties

`count:` ***Number***

The number of samples in the sampleset.

`mean:` ***Number***

The average value of the samples.

`sigma:` ***Number***

The standard deviation.

`sum:` ***Number***

The sum of the samples.

`sum2:` ***Number***

The sum of the squares of the samples.

Topnset

The `Topnset` class represents a collection of metrics grouped by a key such as a URI or a client IP address.

For custom metrics, keys in the topnset corresponds to the keys passed into `metricAddDetail*()` methods. Key values can be a number, string, Dataset, Sampleset, or another topnset.

Methods

`findEntries(key:` ***IPAddress*** | ***String*** | ***Object****`):` ***Array***

Returns all entries with matching keys.

`findKeys(key:` ***IPAddress*** | ***String*** | ***Object****`):` ***Array***

Returns all matching keys.

`lookup(key:` ***IPAddress*** | ***String*** | ***Object****`):` ***

Look up an item in the topnset and retrieves the first matching entry.

Properties

`entries:` ***Array***

An array of the topnset entries. The array contains at most `N` objects with key and value properties where `N` is currently set to 1000.

Keys in the `entries` array adhere to the following structure, or key pattern:

`type:` ***String***

The type of the topnset key. The following key types are supported:

- `int`
- `string`
- `device_id`
- `ipaddr`
- `addr_pair`

- `ether`

`value: *`

The key value, which varies depending on the key type.

- For `int`, `string`, and `device_id` keys, the value is a number, string, and device ID, respectively.
- For `ipaddr` keys, the value is an object containing the following properties:
  - `addr`
  - `proto`
  - `port`
  - `device_id`
  - `origin`
- For `addr_pair` keys, the value is an object containing the following properties:
  - `addr1`
  - `addr2`
  - `port1`
  - `port2`
  - `proto`
- For `ether` keys, the value is an object containing the following properties:
  - `ethertype`
  - `hwaddr`

Deprecated API elements

The API elements listed in this section have been deprecated. Each element includes an alternative and the version in which the element was deprecated.

If your trigger script contains a deprecated element, the syntax validator in the trigger editor lets you know which element is deprecated and suggests a replacement element, if available. You cannot save the trigger until you fix your code or you disable syntax validation. For better trigger performance, replace deprecated elements.

Deprecated advanced trigger options

Option	Replacement	Version
`5min`, `1hr`, and `24hr` metric cycles	There is no replacement for 5 minute, 1 hour, and 24 hour metric cycles. However, 30 second metric cycles are still supported.	9.6

Deprecated global functions

Function	Replacement	Version
`exit(): Void`	The return statement	4.0
`getTimestampMSec(): Number`	`getTimestamp(): Number`	4.0

Deprecated global function parameters

Function	Property	Replacement	Version
`commitDetection()`	`categories`	You can specify detection categories in the Detection Catalog.	9.3

Deprecated events

Event	Replacement	Version
`NEW_VLAN`	No replacement	6.1

Deprecated classes

Class	Replacement	Version
`RemoteSyslog`	Remote.Syslog	4.0
`XML`	Regular expressions	6.0
`TroubleGroup`	No replacement	6.0

Deprecated methods by class

Class	Method	Replacement	Version
Flow	`getApplication(): String`	`getApplications(): String`	5.3
	`setApplication(name: String, turnTiming: Boolean): void`	`addApplication(name: String, turnTiming: Boolean): void`	5.3

Class	Method	Replacement	Version
Session	`update(key: String, value: *, options: Object)*`	`replace(key: String, value: *, options: Object): *`	3.9
SSL	`setApplication(name: String): void`	`addApplication(name: String): void`	5.3

Deprecated properties by class

Class	Property	Replacement	Version
AAA	`error: String`	`isError: Boolean`	5.0
	`tprocess: Number`	`processingTime: Number`	5.2
DB	`tprocess: Number`	`processingTime: Number`	5.2
Detection	`participants.object_type: String`	instanceof operator	7.8
Discover	`vlan: VLAN`	No replacement	6.1
DNS	`tprocess: Number`	`processingTime: Number`	5.2
Flow	`isClientAborted: Boolean`	`isAborted: Boolean`	3.10
	`isServerAborted: Boolean`	`isAborted: Boolean`	3.10
	`turnInfo: String`	Top-level Turn object with attributes for the turn	3.9
FTP	`tprocess: Number`	`processingTime: Number`	5.2
HL7	tprocess: *Number*	processingTime: *Number*	5.2
HTTP	`payloadText: String`	`payload: Buffer`	4.0
	`tprocess: Number`	`processingTime: Number`	5.2
IBMMQ	`messageID: String`	`msgID: Buffer`	5.2
	`msgSize: Number`	`totalMsgLength: Number`	5.2
	`objectHandle: String`	No replacement	5.0
	`payload: Buffer`	`msg: Buffer`	5.2
ICA	`authTicket: String`	`user: String`	3.7
	`application: String`	`program: String`	5.2
	`client: String`	`clientMachine: String`	6.0
LDAP	`tprocess: Number`	`processingTime: Number`	5.2
MongoDB	`roundTripTime: Number` on the `MONGODB_REQUEST` event.	No replacement	25.2
	`tprocess: Number`	`processingTime: Number`	5.2
Netflow	`tos: Number`	`dscp: Number` `dscp: String`	6.1
NTLM	`ntlmRspVersion: String`	`rspVersion: String`	8.2

Class	Property	Replacement	Version
QUIC	`cyuFingerprint: String`	No replacement	9.6
	`tags: Array of Objects`	No replacement	9.6
	`record.cyuFingerprint: String`	No replacement	9.6
SMPP	`tprocess: Number`	`processingTime: Number`	5.2
SMTP	`recipient: String`	`recipientList: Array of Strings`	7.5
	`roundTripTime: Number` on the `SMTP_REQUEST` event.	No replacement	25.2
	`tprocess: Number`	`processingTime: Number`	5.2
SSL	`SSL.record.ja3Hash: String`	`SSL.ja3Hash: String`	9.7
	`SSL.record.ja3sHash: String`	`SSL.ja3sHash String`	9.7
	`reqBytes: Number`	`clientBytes: Number`	6.1
	`reqL2Bytes: Number`	`clientL2Bytes: Number`	6.1
	`reqPkts: Number`	`clientPkts: Number`	6.1
	`rspBytes: Number`	`serverBytes: Number`	6.1
	`rspL2Bytes: Number`	`serverL2Bytes: Number`	6.1
	`rspPkts: Number`	`serverPkts: Number`	6.1
TCP	`wndSize: Number`	`initRcvWndSize: Number`	6.2
	`wndSize1: Number`	`initRcvWndSize1: Number`	6.2
	`wndSize2: Number`	`initRcvWndSize2: Number`	6.2
Turn	`reqSize: Number`	`clientBytes: Number`	4.0
	`reqXfer: Number`	`clientTransferTime: Number`	4.0
	`respSize: Number`	`serverBytes: Number`	4.0
	`rspXfer: Number`	`serverTransferTime: Number`	4.0
	`tprocess: Number`	`processingTime: Number`	4.0

Advanced trigger options

You can configure advanced options for some events when you create a trigger.

The following table describes available advanced options and applicable events.

Option	Description	Supported events
Bytes Per Packet to Capture	Specifies the number of bytes to capture per packet. The capture starts with the first byte in the packet. Specify this option only if the trigger script performs packet capture. A value of 0 specifies that the capture should collect all bytes in each packet.	All events are supported except the following list: • `ALERT_RECORD_COMMIT` • `METRIC_CYCLE_BEGIN` • `METRIC_CYCLE_END` • `FLOW_REPORT` • `NEW_APPLICATION` • `NEW_DEVICE` • `SESSION_EXPIRE`
L7 Payload Bytes to Buffer	Specifies the maximum number of payload bytes to buffer. **Note:** If multiple triggers run on the same event, the trigge… with the highest L7 Paylo… Bytes to Buffer value determines the maximum payload for that event for each trigger.	• `CIFS_REQUEST` • `CIFS_RESPONSE` • `HTTP_REQUEST` • `HTTP_RESPONSE` • `ICA_TICK` • `LDAP_RESPONSE`
Clipboard Bytes	Specifies the number of bytes to buffer on a Citrix clipboard transfer.	• `ICA_TICK`
Metric cycle	Specifies the length of the metric cycle, expressed in seconds. The only valid value is `30sec`.	• `METRIC_CYCLE_BEGIN` • `METRIC_CYCLE_END` • `METRIC_RECORD_COMMIT`
Metric types	Specifies the metric type by the raw metric name, such as `extrahop.device.http_server`.	• `ALERT_RECORD_COMMIT` • `METRIC_RECORD_COMMIT`

Option	Description	Supported events
	Specify multiple metric types in a comma-delimited list.	
Run trigger on each flow turn	Enables packet capture on each flow turn. Per-turn analysis continuously analyzes communication between two endpoints to extract a single payload data point from the flow. If this option is enabled, any values specified for the **Client matching string** and **Server matching string** options are ignored.	• SSL_PAYLOAD • TCP_PAYLOAD
Client Port Range	Specifies the client port range. Valid values are 0 to 65535.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Client Bytes to Buffer	Specifies the number of client bytes to buffer. The value of this option cannot be set to 0 if the value of the **Server bytes to buffer** option is also set to 0.	• SSL_PAYLOAD • TCP_PAYLOAD
Client Buffer Search String	Specifies the format string that indicates when to begin buffering client data. Returns the entire packet upon a string match. You can specify the string as text or hexidecimal numbers. For example, both `ExtraHop` and `\x45\x78\x74\x72\x61\x48\x6F\x70` are equivalent. Hexidecimal numbers are not case sensitive. Any value specified for this option is ignored if the **Per Turn** or **Run**	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD

Option	Description	Supported events
	trigger on all UDP packets option is enabled.	
Server Port Range	Specifies the server port range. Valid values are 0 to 65535.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Server Bytes to Buffer	Specifies the number of server bytes to buffer. The value of this option cannot be set to 0 if the value of the **Client bytes to buffer** option is also set to 0.	• SSL_PAYLOAD • TCP_PAYLOAD
Server Buffer Search String	Specifies the format string that indicates when to begin buffering server data. You can specify the string as text or hexidecimal numbers. For example, both `ExtraHop` and `\x45\x78\x74\x72\x61\x48\x6F\x70` are equivalent. Hexidecimal numbers are not case sensitive. Any value specified for this option is ignored if the **Per Turn** or **Run trigger on all UDP** option is enabled.	• SSL_PAYLOAD • TCP_PAYLOAD • UDP_PAYLOAD
Run trigger on all UDP packets	Enables capture of all UDP datagrams.	• UDP_PAYLOAD
Run FLOW_CLASSIFY on expiring, unclassified flows	Enables running the event upon expiration to accumulate metrics for flows that were not classified before expiring.	• FLOW_CLASSIFY
External types	Specifies the types of external data the trigger processes. The trigger only runs if the payload contains a type field with one of the specified values. Specify multiple types in a comma-separated list.	1. EXTERNAL_DATA

Examples

The following examples are available:

Example: Collect ActiveMQ metrics

The trigger in this example records destination information from the Java Messaging Service (JMS). The trigger creates an application and collects custom metrics that include the whether the broker of an event is the sender or receiver and the JMS destination field specified on that event.

Run the trigger on the following events: `ACTIVEMQ_MESSAGE`

```
var app = Application("ActiveMQ Sample");
    if (ActiveMQ.senderIsBroker) {
       if (ActiveMQ.receiverIsBroker) {
          app.metricAddCount("amq_broker", 1);
          app.metricAddDetailCount("amq_broker", ActiveMQ.queue, 1);
       }
       else {
          app.metricAddCount("amq_msg_out", 1);
          app.metricAddDetailCount("amq_msg_out", ActiveMQ.queue, 1);
       }
}
else {
    app.metricAddCount("amq_msg_in", 1);
    app.metricAddDetailCount("amq_msg_in", ActiveMQ.queue, 1);
}
```

Related classes

- ActiveMQ
- Application

Example: Send data to Azure with Remote.HTTP

The trigger in this example sends data to the Microsoft Azure Table storage service through an HTTP open data stream (ODS).

You must first configure an HTTP open data stream from the Administration settings before you create the trigger. The ODS configuration contains the authentication information required to sign in to your Microsoft Azure service. For configuration information, see Configure an HTTP target for an open data stream in the ExtraHop Admin UI Guide.

Run the trigger on the following events: HTTP_RESPONSE

```
// The name of the HTTP destination defined in the ODS config
var REST_DEST = "my_table_storage";

// The name of the table within Azure Table storage
var TABLE_NAME = "TestTable";

/* If the header is not set to this value, Azure expects to receive XML;
 * however, it is easier for a trigger to send JSON.
 * The ODS config enables you to specify the datatype of fields; in this
 case
 * the timestamp (TS) field is a datetime even though it is serialized from
 a
 * Date to a String.
 */

var headers = { "Content-Type": "application/json;odata=minimalmetadata" };

var now = new Date(getTimestamp());
var msg = {
    "RowKey":        now.getTime().toString(), // must be a string
    "PartitionKey":  "my_key", // must be a string
    "HTTPMethod":    HTTP.method,
    "DestAddr":      Flow.server.ipaddr,
    "SrcAddr":       Flow.client.ipaddr,
    "SrcPort":       Flow.client.port,
    "DestPort":      Flow.server.port,
    "TS@odata.type": "Edm.DateTime", // metadata to describe format of TS
 field
    "TS":            now.toISOString(),
    "ServerTime":    HTTP.processingTime,
    "RspTTLB":       HTTP.rspTimeToLastByte,
    "RspCode":       HTTP.statusCode.toString(),
    "URI":           "http://" + HTTP.host + HTTP.path,
};

// debug(JSON.stringify(msg));
Remote.HTTP(REST_DEST).post( { path: "/" + TABLE_NAME, headers: headers,
 payload:
JSON.stringify(msg) } );
```

Related classes

- Remote.HTTP
- Flow
- HTTP

Example: Monitor SMB actions on devices

The trigger in this example monitors the SMB actions performed on devices, and then creates custom device metrics that collect the total number of bytes read and written, and the number of bytes written by SMB users that are not authorized to access a sensitive resource.

Run the trigger on the following events: `CIFS_RESPONSE`

```
var client = Flow.client.device,
   server = Flow.server.device,
   clientAddress = Flow.client.ipaddr,
   serverAddress = Flow.server.ipaddr,
   file = CIFS.resource,
   user = CIFS.user,
   resource,
   permissions,
   writeBytes,
   readBytes;

// Resource to monitor
resource = "\\Clients\\Confidential\\";
// Users of interest and their permissions
permissions = {
   "\\\\EXTRAHOP\\tom" : {read: false, write: false},
   "\\\\Anonymous" : {read: true, write: false},
   "\\\\WORKGROUP\\maria" : {read: true, write: true}
};

// Check if this is an action on your monitored resource
if ((file !== null) && (file.indexOf(resource) !== -1)) {
   if (CIFS.isCommandWrite) {
      writeBytes = CIFS.reqSize;
      // Record bytes written
      Device.metricAddCount("cifs_write_bytes", writeBytes);
      Device.metricAddDetailCount("cifs_write_bytes", user, writeBytes);
      // Record number of writes
      Device.metricAddCount("cifs_writes", 1);
      Device.metricAddDetailCount("cifs_writes", user, 1);
      // Record number of unauthorized writes
      if (!permissions[user] || !permissions[user].write) {
         Device.metricAddCount("cifs_unauth_writes", 1);
         Device.metricAddDetailCount("cifs_unauth_writes", user, 1);
      }
   }

   if (CIFS.isCommandRead) {
      readBytes = CIFS.reqSize;
      // Record bytes read
      Device.metricAddCount("cifs_read_bytes", readBytes);
      Device.metricAddDetailCount("cifs_read_bytes", user, readBytes);
      // Record number of reads
      Device.metricAddCount("cifs_reads", 1);
      Device.metricAddDetailCount("cifs_reads", user, 1);
      // Record number of unauthorized reads
      if (!permissions[user] || !permissions[user].read) {
         Device.metricAddCount("cifs_unauth_reads", 1);
         Device.metricAddDetailCount("cifs_unauth_reads", user, 1);
      }
   }
}
```

Related classes

- CIFS
- Device
- Flow

Example: Track 500-level HTTP responses by customer ID and URI

The trigger in this example tracks HTTP server responses that result in an error code of 500. The trigger also creates custom device metrics that collect the customer ID and URI in the header of each 500 response.

Run the trigger on the following events: HTTP_REQUEST and HTTP_RESPONSE

```
var custId,
   query,
   uri,
   key;

if (event === "HTTP_REQUEST") {
   custId = HTTP.headers["Cust-ID"];
   // Only keep the URI if there is a customer id
   if (custId !== null) {
      Flow.store.custId = custId;

      query = HTTP.query;

      /* Pull the complete URI (URI plus query string) and save it to
       * the Flow store for a subsequent response event.
       *
       * The query string data is only available on the request.
       */
      uri = HTTP.uri;
      if ((uri !== null) && (query !== null)) {
         uri = uri + "?" + query;
      }

      // Keep URIs for handling by HTTP_RESPONSE triggers
      Flow.store.uri = uri;
   }
}
else if (event === "HTTP_RESPONSE") {
   custId = Flow.store.custId;

   // Count total requests by customer ID
   Device.metricAddCount("custid_rsp_count", 1);
   Device.metricAddDetailCount("custid_rsp_count_detail", custId, 1);

   // If the status code is 500 or 503, record the URI and customer ID
   if ((HTTP.statusCode === 500) || (HTTP.statusCode === 503)){
      // Combine URI and customer ID to create the detail key
      key = custId;
      if (Flow.store.uri != null) {
        key += ", " + Flow.store.uri;
      }
      Device.metricAddCount("custid_error_count", 1);
      Device.metricAddDetailCount("custid_error_count_detail", key, 1);
   }
}
```

Related classes

- HTTP
- Flow
- Device

Example: Collect response metrics on database queries

The trigger in this example creates custom device metrics that collect the number of responses and the processing times on database queries.

Run the trigger on the following events: `DB_RESPONSE`

```
let stmt = DB.statement;
if (stmt === null) {
    return;
}

// Remove leading whitespace and truncate
stmt = stmt.trimLeft().substr(0, 1023);

// Record counts by statement
Device.metricAddCount("db_rsp_count", 1);
Device.metricAddDetailCount("db_rsp_count_detail", stmt, 1);

// Record processing times by statement
Device.metricAddSampleset("db_proc_time", DB.processingTime);
Device.metricAddDetailSampleset("db_proc_time_detail",
                                stmt, DB.processingTime);
```

Related classes

- DB
- Device

Example: Send discovered device data to a remote syslog server

The trigger in this example discovers when a new device is detected on the ExtraHop system and creates remote syslog messages that contain device attributes.

You must first configure a remote open data stream from the Administration settings before you create the trigger. The ODS configuration specifies the location of the remote syslog server. For configuration information, see Configure a syslog target for an open data stream in the ExtraHop Admin UI Guide.

Run the trigger on the following events: `NEW_DEVICE`

```
var dev = Discover.device;
Remote.Syslog.info('Discovered device ' + dev.id + ' (hwaddr: ' + dev.hwaddr
 + ')
');
```

Related classes

- Remote.Syslog
- Discover
- Device

Example: Send data to Elasticsearch with Remote.HTTP

The trigger in this example sends data to an Elasticsearch server through an HTTP open data stream (ODS).

You must first configure an HTTP open data stream from the Administration settings before you create the trigger. The ODS configuration specifies the Elasticsearch target and any required authentication credentials. For configuration information, see Configure an HTTP target for an open data stream in the ExtraHop Admin UI Guide.

Run the trigger on the following events: HTTP_REQUEST and HTTP_RESPONSE

```
var date = new Date();
var payload = {
    'ts' : date.toISOString(), // Timestamp recognized by Elasticsearch
    'eh_event' : 'http',
    'my_path' : HTTP.path};
var obj = {
    'path' : '/extrahop/http', // Add to ExtraHop index
    'headers' : {},
    'payload' : JSON.stringify(payload)} ;
Remote.HTTP('elasticsearch').request('POST', obj);
```

Related classes

- Remote.HTTP

Example: Access HTTP header attributes

The trigger in this example accesses HTTP event attributes from the header object, and creates custom device metrics that count header requests and attributes.

Run the trigger on the following events: HTTP_RESPONSE

```
var hdr,
   session,
   accept,
   results,
   headers = HTTP.headers,
   i;

// Header lookups are case-insensitive properties
session = headers["X-Session-Id"];

/* Session is a string representing the value of the header (or null
 * if the header is not present). Header values are always strings.
 */

// This syntax also works if the header is a legal property name
accept = headers.accept;

/*
 * In the event that there are multiple instances of a header,
 * accessing the header in the above manner (as a property)
 * will always return the value for the first appearance of the
 * header.
 */

if (session !== null)
{
   // Count requests per session ID
```

```
    Device.metricAddCount("req_count", 1);
    Device.metricAddDetailCount("req_count", session, 1);
}

/* Looping over all headers
 *
 * The "length" property is case-sensitive and is not
 * treated as a header lookup. Instead, it returns the number of
 * headers (as if HTTP.headers were an array). In the unlikely
 * event that there is a header called "Length," it would still be
 * accessible with HTTP.headers["Length"] (or HTTP.headers.Length).
 */

for (i = 0; i < headers.length; i++) {
    hdr = headers[i];
    debug("headers[" + i + "].name: " + hdr.name);
    debug("headers[" + i + "].value: " + hdr.value);
    Device.metricAddCount("hdr_count", 1);
    /* Count instances of each header */
    Device.metricAddDetailCount("hdr_count", hdr.name, 1);
}

// Searching for headers by prefix
results = HTTP.findHeaders("Content-");

/* The "results" property is an array (a real javascript array, as opposed
 * to an array-like object) of header objects (with name and value
 * properties) where the names match the prefix of the string passed
 * to findHeaders.
 */
for (i = 0; i < results.length; i++) {
    hdr = results[i];
    debug("results[" + i + "].name: " + hdr.name);
    debug("results[" + i + "].value: " + hdr.value);
}
```

Related classes

- HTTP
- Device

Example: Collect IBMMQ metrics

The triggers in this example work together to give a view of the flow of queue level messages through the IBMMQ protocol. The triggers create custom application metrics that count the number of messages in, out, and exchanged between brokers by different message queues.

Run the following trigger on the `IBMMQ_REQUEST` event.

```
if (IBMMQ.method == "MESSAGE_DATA") {
    var app = Application("IBMMQ Sample");
    app.metricAddCount("broker", 1);
    if (IBMMQ.queue !== null) {
        var ret = IBMMQ.queue.split(":");
        var queue = ret.length > 1 ? ret[1] : ret[0];
        app.metricAddDetailCount("broker", queue, 1);
    }
    else {
        app.metricAddCount("queueless_broker", 1);
    }
    if (IBMMQ.queue !== null && IBMMQ.queue.indexOf("QUEUE2") > -1) {
```

```
        app.metricAddCount("queue2_broker", 1);
    }
    app.commit();
}
elseif (IBMMQ.method == "MQPUT" || IBMMQ.method == "MQPUT1") {
    var app = Application("IBMMQ Sample");
    app.metricAddCount("msg_in", 1);
    if (IBMMQ.queue !== null) {
        var ret = IBMMQ.queue.split(":");
        var queue = ret.length > 1 ? ret[1] : ret[0];
        app.metricAddDetailCount("msg_in", queue, 1);
    }
    else {
        app.metricAddCount("queueless_msg_in", 1);
    }
    if (IBMMQ.queue !== null && IBMMQ.queue.indexOf("QUEUE2") > -1) {
        app.metricAddCount("queue2_msg_in", 1);
    }
    app.commit();
}
```

Run the following trigger on the IBMMQ_RESPONSE event.

```
if (IBMMQ.method == "ASYNC_MSG_V7" || IBMMQ.method == "MQGET_REPLY") {
    var app = Application("IBMMQ Sample");
    if (IBMMQ.payload === null) {
        app.metricAddCount("payloadless_msg_out", 1);
    }
    else {
        app.metricAddCount("msg_out", 1);
        if (IBMMQ.queue !== null) {
            var ret = IBMMQ.queue.split(":");
            var queue = ret.length > 1 ? ret[1] : ret[0];
            app.metricAddDetailCount("msg_out", queue, 1);
        }
        else {
            app.metricAddCount("queueless_msg_out", 1);
        }
        if (IBMMQ.queue !== null && IBMMQ.queue.indexOf("QUEUE2") > -1) {
            app.metricAddCount("queue2_msg_out", 1);
        }
    }
    app.commit();
}
```

Related classes

- IBMMQ
- Application

Example: Record Memcache hits and misses

The trigger in this example creates custom device metrics that record each memcache hit or miss and the access time of each hit.

Run the trigger on the following events: MEMCACHE_RESPONSE

```
var hits = Memcache.hits;
var misses = Memcache.misses;
var accessTime = Memcache.accessTime;
var i;
```

```
Device.metricAddCount('memcache_key_hit', hits.length);

for (i = 0; i < hits.length; i++) {
   var hit = hits[i];
   if (hit.key != null) {
      Device.metricAddDetailCount('memcache_key_hit_detail', hit.key, 1);
   }
}

if (!isNaN(accessTime)) {
   Device.metricAddSampleset('memcache_key_hit', accessTime);
   if ((hits.length > 0) && (hits[0].key != null)) {
      Device.metricAddDetailSampleset('memcache_key_hit_detail',
 hits[0].key,
                                      accessTime);
   }
}

Device.metricAddCount('memcache_key_miss', misses.length);

for (i = 0; i < misses.length; i++) {
   var miss = misses[i];
   if (miss.key != null) {
      Device.metricAddDetailCount('memcache_key_miss_detail', miss.key, 1);
   }
}
```

Related classes

- Memcache
- Device

Example: Parse memcache keys

Parses the memcache keys to extract detailed breakdowns, such as by ID module and class name, and creates custom device metrics to collect key details.

Keys are formatted as `"com.extrahop.<module>.<class>_<id>"`—for example: `"com.extrahop.widgets.sprocket_12345"`.

Run the trigger on the following events: `MEMCACHE_RESPONSE`

```
var method = Memcache.method;
var statusCode = Memcache.statusCode;
var reqKeys = Memcache.reqKeys;
var hits = Memcache.hits;
var misses = Memcache.misses;
var error = Memcache.error;
var hit;
var miss;
var key;
var size;
var reqKey;
var i;

// Record breakdown of hit count and value size by module and class
for (i = 0; i < hits.length; i++) {
   hit = hits[i];
   key = hit.key;
   size = hit.size;
```

```
   Device.metricAddCount("hit", 1);
   if (key != null) {
      var parts = key.split(".");

      if ((parts.length == 4) && (parts[0] == "com") &&
         (parts[1] == "extrahop")) {
         var module = parts[2];
         var subparts = parts[3].split("_");

         Device.metricAddDetailCount("hit_module", module, 1);
         Device.metricAddDetailSampleset("hit_module_size", module, size);

         if (subparts.length == 2) {
            var hitClass = module + "." + subparts[0];

            Device.metricAddDetailCount("hit_class", hitClass, 1);
            Device.metricAddDetailSampleset("hit_class_size", hitClass,
                                            size);
         }
      }
   }
}

// Record misses by ID to help identify caching issues
for (i = 0; i < misses.length; i++) {
   miss = misses[i];
   key = miss.key;
   if (key != null) {
      var parts = key.split(".");

      if ((parts.length == 4) && (parts[0] == "com") &&
         (parts[1] == "extrahop") && (parts[2] == "widgets")) {
         var subparts = parts[3].split("_");

         if ((subparts.length == 2) && (subparts[0] == "sprocket")) {
            Device.metricAddDetailCount("sprocket_miss_id", subparts[1], 1);
         }
      }
   }
}

// Record the keys that produced any errors
if (error != null && method != null) {
   for (i = 0; i < reqKeys.length; i++) {
      reqKey = reqKeys[i];
      if (reqKey != null) {
         var errDetail = method + " " + reqKey + " / " + statusCode + ": " +
                         error;
         Device.metricAddDetailCount("error_key", errDetail, 1);
      }
   }
}

// Record the status code, matching built-in metrics
if (Memcache.isBinaryProtocol && statusCode != "NO_ERROR") {
   Device.metricAddDetailCount("status_code",
                               method + "/" + statusCode, 1);
}
else {
   Device.metricAddDetailCount("status_code", statusCode, 1);
}
```

Related classes

- Memcache
- Device

Example: Add metrics to the metric cycle store

The trigger in this example illustrates how to temporarily store data from all metric record commits that occur during a metric cycle.

Run the trigger on the following events: `METRIC_CYCLE_BEGIN`, `METRIC_CYCLE_END`, `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.http_server, extrahop.device.tcp

```
var store = MetricCycle.store;

function processMetric() {
    var id = MetricRecord.id,
        deviceId = MetricRecord.object.id,
        fields = MetricRecord.fields;

    if (!store.metrics[deviceId]) {
        store.metrics[deviceId] = {};
    }
    if (id === 'extrahop.device.http_server') {
        store.metrics[deviceId].httpRspAborted= fields['rsp_abort'];
    }
    else if (id === 'extrahop.device.tcp') {
        store.metrics[deviceId].tcpAborted = fields['aborted_out'];
    }
}

function commitSyntheticMetrics() {
    var dev,
        metrics,
        abortPct,
        deviceId;
    for (deviceId in store.metrics) {
        metrics = store.metrics[deviceId];
        abortPct = (metrics.httpRspAborted / metrics.tcpAborted) * 100;
        dev = new Device(deviceId);
        dev.metricAddSnap('http-tcp-abort-pct', abortPct);
    }
}

switch (event) {
case 'METRIC_CYCLE_BEGIN':
    store.metrics = {};
    break;

case 'METRIC_RECORD_COMMIT':
    processMetric();
    break;
```

```
case 'METRIC_CYCLE_END':
    commitSyntheticMetrics();
    break;
}
```

Related classes

- MetricCycle
- MetricRecord
- Device

Example: Parse custom PoS messages with universal payload analysis

The trigger in this example parses TCP messages from a point-of-sale (PoS) system and creates custom device metrics that collect specific values in the 4th to 7th bytes of both response and request messages.

Run the trigger on the following events: TCP_PAYLOAD

```
// Define variables; store client or server payload into a Buffer object

var buf_client = Flow.client.payload,
    buf_server = Flow.server.payload,
    protocol = Flow.l7proto,

// PoS Message Type Structure Definition
    pos_message_type = {
        "0100" : "0100_Authorization_Request",
        "0101" : "0101_Authorization_Request_Repeat",
        "0110" : "0110_Authorization_Response",
        "0200" : "0200_Financial_Request",
        "0201" : "0201_Financial_Request_Repeat",
        "0210" : "0210_Financial_Response",
        "0220" : "0220_Financial_Transaction_Advice_Request",
        "0221" : "0221_Financial_Transaction_Advice_Request_Repeat",
        "0230" : "0230_Financial_Transaction_Advice_Response",
        "0420" : "0420_Reversal_Advice_Request",
        "0421" : "0421_Reversal_Advice_Request_Repeat",
        "0430" : "0430_Reversal_Advice_Response",
        "0600" : "0600_Administration_Request",
        "0601" : "0601_Administration_Request_Repeat",
        "0610" : "0610_Administration_Response",
        "0620" : "0620_Administration_Advice_Request",
        "0621" : "0621_Administration_Advice_Request_Repeat",
        "0630" : "0630_Administration_Advice_Response",
        "0800" : "0800_Administration_Request",
        "0801" : "0801_Administration_Request_Repeat",
        "0810" : "0810_Administration_Response"
    };

// Skip parsing if it is a protocol of no interest or there is no payload
if (protocol !== 'tcp:4015' || (buf_client === null && buf_server === null))
 {
    // debug('Protocol of no interest: ' + protocol);
    return;
} else {
    /* Store the data into variables for future access since there is some
 payload
     * to parse
     */
    var client_ip = Flow.client.ipaddr,
```

```
        server_ip = Flow.server.ipaddr,
        client_port = Flow.client.port,
        server_port = Flow.server.port;
        // client = new Device(Flow.client.device.id),
        // server = new Device(Flow.server.device.id);
}

if (buf_client !== null && buf_client.length >= 7) {

    // This is a client payload
    var cli_msg_type = buf_client.slice(3,7).decode('utf-8');
    debug('Client: ' + client_ip + ":" + client_port + " Type: " +
 pos_message_type[cli_msg_type]);
    Device.metricAddCount('UPA_Request', 1);
    Device.metricAddDetailCount('UPA_Request_by_Message',
 pos_message_type[cli_msg_type], 1);
    Device.metricAddDetailCount('UPA_Request_by_Client',
 client_ip.toString(), 1);

} else if (buf_server !== null && buf_server.length >= 7) {

    // This is a server payload
    var srv_msg_type = buf_server.slice(3,7).decode('utf-8');
    debug('Server: ' + server_ip + " Client: " + client_ip + ":" +
 client_port +"
Type: " + pos_message_type[srv_msg_type]);
    Device.metricAddCount('UPA_Response', 1);
    Device.metricAddDetailCount('UPA_Response_by_Message',
 pos_message_type[srv_msg_type], 1);
    Device.metricAddDetailCount('UPA_Response_by_Client',
 client_ip.toString(), 1);

} else {

    // No buffer captured situation
    //debug('Null or not enough buffer data');
    return;
}
```

Related classes

- Buffer
- Device
- Flow

Example: Parse syslog over TCP with universal payload analysis

The trigger in this example parses the syslog over TCP and counts the syslog activity over time, both network-wide and per device.

Note: You might need to edit the trigger example to make sure the network ports for your syslog server match the ports in your environment.

Run the trigger on the following events: TCP_PAYLOAD, UDP_PAYLOAD

```
// Global variables
var buffer         = Flow.client.payload,
    buffer_size    = Flow.client.payload.length + 1,
    client         = new Device(Flow.client.device.id),
    data_as_json   = { client_ip       : Flow.client.ipaddr.toString(),
                       client_port     : Flow.client.port.toString(),
```

```
                        server_ip       : Flow.server.ipaddr.toString(),
                        server_port     : Flow.server.port.toString(),
                        protocol        : 'syslog',
                        protocol_fields : {} },
    protocol        = Flow.l7proto,
    server          = new Device(Flow.server.device.id),
    syslog          = {},
    syslog_facility = {
        "0": "kern",
        "1": "user",
        "2": "mail",
        "3": "daemon",
        "4": "auth",
        "5": "syslog",
        "6": "lpr",
        "7": "news",
        "8": "uucp",
        "9": "clock_daemon",
        "10": "authpriv",
        "11": "ftp",
        "12": "ntp",
        "13": "log_audit",
        "14": "log_alert",
        "15": "cron",
        "16": "local0",
        "17": "local1",
        "18": "local2",
        "19": "local3",
        "20": "local4",
        "21": "local5",
        "22": "local6",
        "23": "local7",
    },
    syslog_priority = {
         "0": "emerg",
         "1": "alert",
         "2": "crit",
         "3": "err",
         "4": "warn",
         "5": "notice",
         "6": "info",
         "7": "debug",
    };

// Exit out early if not classified properly or no payload

if ( ( protocol != 'tcp:5141' ) || ( buffer === null ) ) {
    debug('Invalid protocol ' + protocol +
          ' or null buffer (' + buffer.unpack('z').join(' ') + ')');
    return;
}

// Get started parsing Syslog

var data = buffer.unpack('z');

// Separate the PRIO field from the rest of the message
var msg_part  = data[0].split('>')[1].split(' ');
var prio_part = data[0].split('>')[0].split('<')[1];

// Decode the PRIO field into Syslog facility and priority
var raw_facility = parseInt(prio_part) >> 3;
var raw_priority = parseInt(prio_part) & 7;
```

```
syslog.facility = syslog_facility[raw_facility];
syslog.priority = syslog_priority[raw_priority];

/* Timestamp and hostname are technically part of the HEADER field, but
 * treat the rest of the message as a <space> delimited
 * string, which it is (the syslog protocol is very basic)
 */
syslog.timestamp = msg_part.slice(0,3).join(' ');
syslog.hostname  = msg_part[3];
syslog.message   = msg_part.slice(4).join(' ');

/* At the network level, keep counts of who is sending messages by
 * both facility and priority
 */
Network.metricAddCount('syslog:priority_' + syslog.priority, 1);
Network.metricAddDetailCount('syslog:priority_' +
                             syslog.priority + '_detail',
                             Flow.client.ipaddr, 1);
Network.metricAddCount('syslog:facility_' + syslog.facility, 1);
Network.metricAddDetailCount('syslog:facility_' +
                             syslog.facility + '_detail',
                             Flow.client.ipaddr, 1);

/* Devices receiving messages keep a count of who sent those messages
 * by facility and priority
 */
server.metricAddCount('syslog:priority_' + syslog.priority, 1);
server.metricAddDetailCount('syslog:priority_' +
                            syslog.priority + '_detail',
                            Flow.client.ipaddr, 1);
server.metricAddCount('syslog:facility_' + syslog.facility, 1);
server.metricAddDetailCount('syslog:facility_' +
                            syslog.facility + '_detail',
                            Flow.client.ipaddr, 1);

/* Devices sending messages keep a count of who they sent those messages
 * to by facility and priority
 */
client.metricAddCount('syslog:priority_' + syslog.priority, 1);
client.metricAddDetailCount('syslog:priority_' +
                            syslog.priority + '_detail',
                            Flow.server.ipaddr, 1);
client.metricAddCount('syslog:facility_' + syslog.facility, 1);
client.metricAddDetailCount('syslog:facility_' +
                            syslog.facility + '_detail',
                            Flow.server.ipaddr, 1);

data_as_json.protocol_fields = syslog;
data_as_json.ts              = new Date();

//try {
//    Remote.MongoDB.insert('payload.syslog', data_as_json);
//}
//catch ( err ) {
//    Remote.Syslog.debug(JSON.stringify(data_as_json));
//}
debug('Syslog data: ' + JSON.stringify(data_as_json, null, 4));
```

Related classes

- Flow
- Network

- Buffer
- Remote.MongoDB
- Remote.Syslog

Example: Parse NTP with universal payload analysis

The trigger in the following example parses the network time protocol through universal payload analysis (UPA).

Run the trigger on the following events: UDP_PAYLOAD

```
var buf = Flow.server.payload,
    flags,
    values,
    fmt,
    offset = 0,
    ntpData = {},
    proto = Flow.l7proto;
if ((proto !== 'NTP') || (buf === null)) {
    return;
}
// Parse individual flag values from flags byte
function parseFlags(flags) {
    return {
        'LI': flags >> 6,
        'VN': (flags & 0x3f) >> 3,
        'mode': flags & 0x7
    };
}

// Convert from NTP short format
function ntpShort(n) {
    return n / 65536.0;
}

// Convert integral part of NTP timestamp format to Date
function ntpTimestamp(n) {
    /* NTP dates start at 1900, subtract the difference
     * and convert to milliseconds */
    var ms = (n - 0x83aa7e80) * 1000;
    return new Date(ms);
}

// First part of NTP header
fmt = ('B' + // Flags (LI, VN, mode)
       'B' + // Stratum
       'b' + // Polling interval (signed)
       'b' + // Precision (signed)
       'I' + // Root delay
       'I'); // Root dispersion

values = buf.unpack(fmt);

offset = values.bytes;

flags = parseFlags(values[0]);
if (flags.VN !== 4) {
    // Expecting NTPv4
    return;
}
```

```
ntpData.flags = flags;
ntpData.stratum = values[1];
ntpData.poll = values[2];
ntpData.precision = values[3];
ntpData.rootDelay = ntpShort(values[4]);
ntpData.rootDispersion = ntpShort(values[5]);

// The next field, the reference ID, depends upon the stratum field
switch (ntpData.stratum)
{
case 0:
case 1:
    // Identifier string (4 bytes), and 4 NTP timestamps in two parts
    fmt = '4s8I';
    break;
default:
    // Unsigned int (based on IP), and 4 NTP timestamps in two parts
    fmt = 'I8I';
    break;
}
// Passing in offset enables you to continue parsing where you left off
values = buf.unpack(fmt, offset);
ntpData.referenceId = values[0];

// Only the integral parts of the timestamp are referenced here
ntpData.referenceTimestamp = ntpTimestamp(values[1]);
ntpData.originTimestamp = ntpTimestamp(values[3]);
ntpData.receiveTimestamp = ntpTimestamp(values[5]);
ntpData.transmitTimestamp = ntpTimestamp(values[7]);

debug('NTP data:' + JSON.stringify(ntpData, null, 4));
```

Related classes

- Buffer
- Flow
- UDP

Example: Record data to a session table

The trigger in this example records specific HTTP transactions to the session table and creates custom network metrics that collect session expiration data.

Run the trigger on the following events: `HTTP_REQUEST`, `SESSION_EXPIRE`

```
// HTTP_REQUEST
if (event == "HTTP_REQUEST") {
   if (HTTP.userAgent === null) {
      return;
   }

   // Look for the OS name
   var re = /(Windows|Mac|Linux)/;
   var os = HTTP.userAgent.match(re);
   if (os === null) {
      return;
   }
   // Specify the matched string as the key for session table entry
   var os_name = os[0];

   var opts =
```

```
    {
      // Expire added entries after 30 seconds
      expire: 30,
      // Retain entries with normal priority if session table grows too
 large
      priority: Session.PRIORITY_NORMAL,
      // Make expired entries available on SESSION_EXPIRE events
      notify: true
    };
    // Ensure an entry for this key is present; an existing entry will not be
 replaced
    Session.add(os_name, 0, opts);
    // Increase the count for this entry
    var count = Session.increment(os_name);
    debug(os_name + ": " + count);
}

/* After 30 seconds, the accumulated per-OS counts appear in the
 Session.expiredKeys
 * list, accessible in the SESSION_EXPIRE event:
 */
    //SESSION_EXPIRE
if (event == "SESSION_EXPIRE"){
    var keys = Session.expiredKeys;
    for (var i = 0; i < keys.length; i++) {
       debug("count of " + keys[i].name + ": " + keys[i].value);
       if (keys[i].value > 500) {
          Network.metricAddCount("os-high-request-count", 1);
          Network.metricAddDetailCount("os-high-request-count",
                                       keys[i].name, 1);
       }
    }
}
```

Related classes

- HTTP
- Network
- Session

Example: Track SOAP requests

The trigger in this example tracks SOAP requests through the SOAPAction header, saves them into the flow store, and creates custom network metrics that collect data about the transactions.

Note: Before you begin, confirm your SOAP implementation passes the necessary information through the header.

Run the trigger on the following events: HTTP_REQUEST, HTTP_RESPONSE

```
var soapAction,
    headers = HTTP.headers,
    method,
    detailMethod,
    parts;

if (event === "HTTP_REQUEST") {
    soapAction = headers["SOAPAction"]
    if (soapAction != null) {
        Flow.store.soapAction = soapAction;
```

```
    }
}
else if (event === "HTTP_RESPONSE") {
    soapAction = Flow.store.soapAction;
    if (soapAction != null) {
        parts = soapAction.split("/");
        if (parts.length > 0) {
            method = soapAction.split("/")[1];
        }
        else {
            method = soapAction;
        }
        detailMethod = method + "_detail";
        Network.metricAddCount(method, 1);
        Network.metricAddDetailCount(detailMethod, Flow.client.ipaddr, 1);
        Network.metricAddSampleset("soap_proc", HTTP.processingTime);
        Network.metricAddDetailSampleset("soap_proc_detail", method,
                                         HTTP.processingTime);
    }
}
```

Related classes

- Flow
- HTTP
- Network

Example: Matching topnset keys

The triggers in this example illustrate topnset key matching by string and IPAddress, and includes advanced key mapping.

Topnset key matching by string

Run the trigger on the following events: `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.app

```
var stat = MetricRecord.fields['bytes_out'],
    id = MetricRecord.object.id,
    proto = 'HTTP2-SSL',
    entry;

entry = stat.lookup(proto);
if (entry !==null) {
    debug('Device ' + id + ' sent ' + entry.value + ' bytes over ' + proto);
}
```

Topnset key matching by IPAddress

Run the trigger on the following events: `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.net_detail

```
var stat = MetricRecord.fields['bytes_out'],
    total = 0,
    entry,
    entries,
    i,
    ip = new IPAddress('192.168.112.1');

entries = stat.findEntries(ip);
for (i = 0; i < entries.length; i++) {
    entry = entries[i];
    total += entry.value;
}
Remote.Syslog.alert('IP ' + ip + ' sent ' + total + ' bytes.');
```

Advanced topnset key matching

Run the trigger on the following events: `METRIC_RECORD_COMMIT`

Configure advanced trigger options as shown in the following table:

Option	Value
Metric Cycle	30sec
Metric Type	extrahop.device.net_detail

```
var stat = MetricRecord.fields['bytes_out'],
    entry,
    entries,
    key,
    i;

entries = stat.findEntries({addr: /192.168.112.1*/, proto: 17});

debug('matched ' + entries.length + '/' + stat.entries.length + '
 entries')};

for (i = 0; i < entries.length; i++) {
    entry = entries[i];
    key = entry.key;
    Remote.Syslog.alert('unexpected outbound UDP traffic from: ' +
                        JSON.stringify(key));
}
```

Related classes

- MetricRecord
- IPAddress
- Remote.Syslog

Example: Create an application container

The trigger in this example creates an application container based on traffic associated with a two-tier application, and creates custom application metrics collected on HTTP and database events.

Run the trigger on the following events: HTTP_RESPONSE and DB_RESPONSE

```
/* Initialize the application object against which you will
 * commit specific HTTP and DB transactions. After traffic is
 * committed, an application container called "My App" will appear
 * in the Applications tab in the ExtraHop system.
 */

var myApp = Application("My App");

/* These configurable properties describe features that define
 * your application traffic.
 */

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

   /* HTTP transactions can be committed to the application on
    * HTTP_RESPONSE events.
    */

   /* Commit this HTTP transaction only if the HTTP host header for
    * this response is defined and matches your application's HTTP host.
    */

   if (HTTP.host && (HTTP.host == myAppHTTPHost)) {
      myApp.commit();

      /* Capture custom metrics about user agents that experience
       * HTTP 40x or 50x responses.
       */

      if (HTTP.statusCode && (HTTP.statusCode >= 400))
{

         // Increment the overall count of 40x or 50x responses

         myApp.metricAddCount('myapp_40x_50x', 1);

         // Collect additional detail on referer, if any

         if (HTTP.referer) {
            myApp.metricAddDetailCount('myapp_40x_50x_refer_detail',
                                       HTTP.referer, 1);
         }
      }
   }

} else if (event == "DB_RESPONSE") {
   /* Database transactions can be committed to the application on
    * DB_RESPONSE events.
    *
    * Commit this database transaction only if the database name for
    * this response matches the name of our application database.
    */
   if (DB.database && (DB.database == myAppDatabaseName)) {
```

```
        myApp.commit();
    }
}
```

Related classes

- Application
- DB
- HTTP