HOW CAN WE HELP?

Search our knowledge base articles

Article Detail (?tabset-530... Attachments (?tabset-5302... Subscribe

SDL API - addEvents

Last Updated: Aug 27, 2024

This endpoint inserts one or more structured or unstructured log events. It can scale horizontally to ingest 100s of TBs per day. The request body must be 6MB or less. While capable, and suitable for direct integrations, this endpoint is usually not fire-and-forget. We recommend you read SDL API Integration (https://community.sentinelone.com/s/article/000006766) for best practices and code examples.

If you want to import unstructured, raw log text, it is easier to use uploadLogs (/s/article/000006783#UUID-ee197e34-ab15-c3db-b109-db7bab3f19db).

All events must be part of a session. Only one request can be in-flight at a time for a session. We recommend you conservatively set each session to 2.5 MB/sec. Distribute the total volume of events fairly evenly across sessions. For example, if you have a server that sends 20 MB/sec, we recommend 8 sessions at 2.5 MB/sec per session. Each session cannot exceed 10 MB/sec.

If you generate many events per second, group them into batches and send a batch every few seconds for each of your servers.

There is a maximum of 50K sessions per five minute period per account.

URL

https://yourConsole.net/api/addEvents (https://yourConsole.net/api/addEvents)

Replace `https://yourConsole.net (https://yourConsole.net)` with the URL for the Singularity™ Data Lake Console. For example, *https://xdr.us1.sentinelone.net (https://xdr.us1.sentinelone.net)*.

See Services and Ports for Management (https://community.sentinelone.com/s/article/000004961) for the Singularity™ Data Lake URL for your Datacenter.

Example

Consult the URL above for your specific address, and use POST:

```
curl -X POST https://yourConsole.net/api/addEvents
(https://yourConsole.net/api/addEvents) \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer {token}"  \
    -d '{
    "session": "149d8290-7871-11e1-b0c4-0800200c9a66",
    "sessionInfo": {
      "serverType": "frontend",
      "serverId": "prod-front-2"
    },
    "events": [
      {
        "thread": "1",
        "ts": "1667443947000000000",
        "sev": 3,
        "attrs": {
          "message": "record retrieved",
          "recordId": 39217,
          "latency": 19.4,
          "length": 39207
        }
      }
    ],
    "threads": [
      {"id": "1", "name": "request handler thread"},
      {"id": "2", "name": "background thread"}
    ]
  }'
```

Format

```
{
  "token":       "xxx", // A "Log Write Access" key. We recommend setting
the key in the header.
  "session":     "yyy", // Required.
  "sessionInfo": {...}, // Optional. Lets you set fields related to the
upload process, ex. "serverHost".
  "threads":     [...], // Optional. Lets you create a readable name for
each thread in events.
  "events":      [...], // Zero or more events (log messages) to upload.
  "logs":        [...]  // Optional. Lets you decrease redundant data in
the request.
}
```

`token`

A **Log Write Access** SDL API key (https://community.sentinelone.com/s/article/000006763). We recommend you send the key in the header, in the format `Authorization: Bearer {token}` .

SentinelOne Console user API tokens also support the Singularity™ Data Lake SDL API. Because these tokens have an expiration date, we recommend a **Log Write Access** key for an integration. See SDL API Keys (https://community.sentinelone.com/s/article/000006763) for more.

`session`

An arbitrary string (up to 200 characters) that uniquely identifies the lifetime of the upload process. You can generate a UUID (http://en.wikipedia.org/wiki/Universally_unique_identifier) at process startup, then store the value in a global variable. For example, in Python:

```
import uuid
guid = str(uuid.uuid4())

data = {
  "session": guid,
  # more request properties
}
```

Do not create a new session identifier for each request. If there are too many, we rate-limit your account. Rate limiting can also occur if the throughput per session exceeds approximately 10 MB/s. If you receive "backoff" (429) errors during sustained periods of high throughput, separate your events into multiple sessions.

`sessionInfo`

Optional. Lets you set fields related to the upload process that are useful for querying. For example `serverHost`, or `region`.

```
"sessionInfo": {
  "serverHost": "front-1",
  "serverType": "frontend",
  "region":     "us-east-1"
}
```

Set at least the `serverHost` field with the hostname, or a stable server identifier.

`sessionInfo` must be the same for all requests with the same `session` value. If not, changes to the value may be ignored, and all events for the session will have the first `sessionInfo` value.

`threads`

Optional. Objects with `id` and `name` key-values, `{"id": "...", "name": "..."}`, one for each unique thread ID in the `events` property. This lets you create a readable name for each thread. Shows in the UI as the `threadName` field.

`events`

An array of zero or more events to upload:

```
"events": [
  {
    "ts": "event timestamp, as a string (nanoseconds since 1/1/1970)",
    "sev": nnn,   // Optional
    "thread": "thread id you set in the events field",  // Optional
    "attrs": {
      // Set arbitrary attributes
    }
  },
  {
    // One more event
  }
]
```

`ts` is the timestamp for the event. **This is a string, not a number.** (Some JSON packages convert numbers to floating-point, and a standard 64-bit floating-point value is not large enough for a nanosecond timestamp.) We discard events older than your account's retention period.

For example, in Python:

```
from datetime import datetime
import time

# Get the timestamp in nanoseconds
currentdate = str(int(time.time() * 1e9)) # must be string
```

For non-nanosecond timestamps, for example an ISO-8601 Unix epoch, you can ensure chronological order by incrementing the value by 1 for each event in the session, or by ordering the events in a batch.

`sev` is the severity of the event. This field is optional (defaults to `3` , "info"). Set from 0 to 6. The scale is the classic "finest, finer, fine, info, warning, error, and fatal":

```
0: "finest"
1: "finer", "trace", "TRC"
2: "fine", "debug", "d", "dbg", "DBG"
3: "info", "inf", "notice", "i", "INF", "NOT"
4: "warn", "warning", "w", "wrn", "WRN"
5: "error", "err", "e", "ERR"
6: "fatal", "emerg", "emergency", "crit", "critical", "panic", ""alert",
"f", "CRT"
```

`thread` is optional, and is the applicable thread id you set in the `threads` property.

`attrs` has the "content" of the event. You can put the text of the event in the `message` field. For example:

```
"attrs": { "message": "record 39217 retrieved in 19.4ms; 39207 bytes" }
```

Only the `message` field is available for parsing. The `parser` field creates a configuration file in the UI where you can set rules to parse the `message` . Example:

```
"attrs": {
  "message": "{\"field 1\":\"x\",\"field 2\":\"y\"}",
  "parser": "myParserName"
}
```

After parsing rules are set for `myParserName` , the event becomes:

```
{
  message: "{"field 1":"x","field 2":"y"}",
  field 1: "x",
  field 2: "y"
}
```

If your application is configured to extract fields, you can send structured events as key-value pairs. These can be queried without a parser. For example:

```
"attrs": {
  "message": "record retrieved",
  "recordId": 39217,
  "latency": 19.4,
  "length": 39207
}
```

Nested attributes are flattened. For example:

```
"attrs": {
    "nested": {
        "foo": "1",
        "bar": "2",
        "baz": "3"
    }
```

When ingested, `nested` has the value `{"foo":"1","bar":"2","baz":"3"}` .

Pass numeric values as JSON numbers, not quoted strings.

With respect to log volume, you will be charged for the bytes consumed by the `message` text, plus one byte for the field. For example, `"attrs": {"message": "record retrieved"}` incurs a charge of 17 bytes (16 characters plus 1 byte for the field); and `"attrs": {recordId": 39217}` incurs a charge of 9 bytes (8 for the number and 1 for the field). We do not charge for escape backslashes.

When you send structured events as key-value pairs, you will be charged 1 byte for each field, plus the bytes consumed by the field values. In the example below, only 4 bytes are charged, one for each field, and one for each character:

```
"attrs": {
  "field 1": "x",
  "field 2": "y"
}
```

When you omit the `message` field, the tradeoff is that you cannot search the full text of the message. In the above example, you cannot search for "field 1" or "field 1:x" as a text search. (You can search in "field 1" for value "x".) Charges incurred for structured events may be less, as the `message` usually has field names, values, and formatting characters such as quotes. Structured events only have field values.

`logs`

Optional. Lets you set constant metadata, whose value does not change in multiple events in the request. This decreases redundant data, and improves throughput.

When a request includes multiple events from the same log file, constant metadata in the event's `attrs` field is repeated. This can significantly decrease throughput. For example, Docker and Kubernetes events often include pod and container names, namespaces, node names, and more. Repeating this data for each event can consume a large portion of the request size.

The `logs` property lets you separate constant metadata from multiple events. For example:

```
{
  "events": [
    {
      "log": "1",
      "attrs": {
        "raw_timestamp": "2019-09-26T03:34:53.522532294Z",
        "message": "Thirteen old orange elephants cartwheel over the old
foxes."
      },
    },
    {
      "log": "1",
      "attrs": {
        "raw_timestamp": "2019-09-26T04:45:35.413474923Z",
        "message": "Star light, start bright."
      },
    },
  ],
  "logs": [
    {
      "id": "1",
      "attrs": {
        "_k8s_ck": "Deployment",
        "_k8s_cn": "test-cluster"
      }
    }
  ]
}
```

`logs` is an array of `{...}` objects, each with an `id` , and `attrs` properties. `id` is an identifier, while `attrs` sets the key-value attributes that are constant in multiple events in the request.

You must add the `log` property to `"events": [...]` , and map its value to the applicable `id` value in `"logs": [...]` .

In the example above, two messages are included in the request. The attributes `_k8s_ck` and `_k8s_cn` , with constant values "Deployment" and "test-cluster", are set in `logs: [...]` , with an `id` value of `"1"` . The `log` property is added to each event in `"events": [...]` , and has the same value as `id` , thus mapping the constant attributes to each event.

During ingestion, constant metadata is attached to each applicable event. This decreases redundant data, and improves throughput.

Request Response

When you first integrate with this endpoint, we recommend you confirm your data is available in the UI. You can search for `tag='ingestionFailure'` to find issues recorded by the metalog.

If the request is a success:

```
{"bytesCharged":0,"status": "success"}
```

A "success" may have a warning message.

Troubleshooting:

Status Code	Description
400	Check your request syntax. If the request exceeds 6 MB the data is truncated. The addEvents endpoint cannot handle malformed JSON.
401	If you receive 401 errors with a `status` of `error/client/noPermission` or `error/client/noPermission/accountDisabled`, the `token` is incorrect or invalid.
429	If you receive 429 "backoff" errors during sustained periods of high throughput, separate your events into multiple sessions. We recommend you conservatively set each session to 2.5 MB/sec. Each session cannot exceed 10 MB/sec, and there is a maximum of 50000 sessions per five minute period. Only one request can be in-flight at a time for a session.
5xx	Retry the request after a short delay. See SDL API Integration (https://community.sentinelone.com/article/000006766).

Was this article helpful? Yes No

Related Articles

HTTP Plugin
(/s/article/000008655)

Working with HTTP Monitors
(/s/article/000006759)

HTTP Event Collector (HEC)
(/s/article/000008671)

SDL API Keys
(/s/article/000006763)

SDL API Overview
(/s/article/000006771)

(https://twitter.com/SentinelOne)
(https://www.linkedin.com/company/sentinelone/)
(https://www.facebook.com/SentinelOne/)
(https://www.youtube.com/c/Sentinelone-inc)

444 Castro Street Suite 400 Mountain View, CA 94041
+1-855-868-3733
community@sentinelone.com (mailto:community@sentinelone.com)

Privacy Policy (https://www.sentinelone.com/legal/privacy-policy/)
Support Terms (https://www.sentinelone.com/legal/support-terms/)
Customer Community Terms of Use (https://www.sentinelone.com/legal/customer-community-terms-of-use/)