Workgroup: QUIC
Internet-Draft:
draft-ietf-quic-qlog-main-schema-08
Published: 4 March 2024
Intended Status: Standards Track
Expires: 5 September 2024
Authors: R. Marx, Ed.   L. Niccolini, Ed.   M. Seemann, Ed.
Akamai   Meta
L. Pardue, Ed.
Cloudflare

Main logging schema for qlog

Abstract

This document defines qlog, an extensible high-level schema for a standardized logging format. It allows easy sharing of data, benefitting common debug and analysis methods and tooling. The high-level schema is independent of protocol; separate documents extend qlog for protocol-specific data. The schema is also independent of serialization format, allowing logs to be represented in many ways such as JSON, CSV, or protobuf.

Note to Readers

Note to RFC editor: Please remove this section before publication.

Feedback and discussion are welcome at https://github.com/quicwg/qlog. Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

Concrete examples of integrations of this schema in various programming languages can be found at https://github.com/quiclog/qlog/.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction
  1.1. Notational Conventions
    1.1.1. Schema definition
    1.1.2. Serialization examples
2. Design goals
3. QlogFile schema
  3.1. Traces
  3.2. Trace
  3.3. TraceError
4. QlogFileSeq schema
  4.1. TraceSeq
5. VantagePoint
6. Events
  6.1. Timestamps
  6.2. Names
  6.3. Data
  6.4. Path
  6.5. ProtocolType
  6.6. Triggers
  6.7. Grouping
  6.8. SystemInformation
  6.9. CommonFields
7. Raw packet and frame information
8. Common events and data classes
  8.1. Generic events
    8.1.1. error
    8.1.2. warning
    8.1.3. info
    8.1.4. debug
    8.1.5. verbose

8.2. Simulation events
8.2.1. scenario
8.2.2. marker
9. Event definition guidelines
9.1. Event design
9.2. Event importance levels
9.3. Custom fields
10. Serializing qlog
10.1. qlog to JSON mapping
10.2. qlog to JSON Text Sequences mapping
10.2.1. Supporting JSON Text Sequences in tooling
10.3. JSON Interoperability
10.4. Truncated values
10.5. Optimization of serialized data
11. Methods of access and generation
11.1. Set file output destination via an environment variable
12. Tooling requirements
13. Security and privacy considerations
13.1. Data at risk
13.2. Operational implications and recommendations
13.3. Data minimization or anonymization
14. IANA Considerations
15. References
15.1. Normative References
15.2. Informative References
Acknowledgements
Change Log
Since draft-ietf-quic-qlog-main-schema-07:
Since draft-ietf-quic-qlog-main-schema-06:
Since draft-ietf-quic-qlog-main-schema-05:
Since draft-ietf-quic-qlog-main-schema-04:
Since draft-ietf-quic-qlog-main-schema-03:
Since draft-ietf-quic-qlog-main-schema-02:
Since draft-ietf-quic-qlog-main-schema-01:
Since draft-ietf-quic-qlog-main-schema-00:
Since draft-marx-qlog-main-schema-draft-02:
Since draft-marx-qlog-main-schema-01:
Since draft-marx-qlog-main-schema-00:
Authors' Addresses

1. Introduction

Endpoint logging is a useful strategy for capturing and understanding how applications using network protocols are behaving, particularly where protocols have an encrypted wire image that restricts observers' ability to see what is happening.

Many applications implement logging using a custom, non-standard logging format. This has an effect on the tools and methods that are used to analyze the logs, for example to perform root cause analysis of an interoperability failure between distinct implementations. A lack of a common format impedes the development of common tooling that can be used by all parties that have access to logs.

This document defines qlog, an extensible high-level schema and harness that provides a shareable, aggregatable and structured logging format. This high-level schema is independent of protocol, with logging entries for specific protocols and use cases being defined in other documents (see for example [QLOG-QUIC] for QUIC and [QLOG-H3] for HTTP/3-related event definitions).

The goal of this high-level schema is to provide amenities and default characteristics that each logging file should contain (or should be able to contain), such that generic and reusable toolsets can be created that can deal with logs from a variety of different protocols and use cases.

As such, qlog provides versioning, metadata inclusion, log aggregation, event grouping and log file size reduction techniques.

The qlog schema can be serialized in many ways (e.g., JSON, CBOR, protobuf, etc). This document describes only how to employ [JSON], its subset [I-JSON], and its streamable derivative [JSON-Text-Sequences].

1.1. Notational Conventions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.1.1. Schema definition

To define events and data structures, all qlog documents use the Concise Data Definition Language [CDDL]. This document uses the basic syntax, the specific text, uint, float32, float64, bool, and any types, as well as the .default, .size, and .regexp control operators, the ~ unwrapping operator, and the $ extension point syntax from [CDDL].

Additionally, this document defines the following custom types for clarity:

```
; CDDL's uint is defined as being 64-bit in size
; but for many protocol fields it is better to be restrictive
; and explicit
uint8 = uint .size 1
uint16 = uint .size 2
uint32 = uint .size 4
uint64 = uint .size 8

; an even-length lowercase string of hexadecimally encoded bytes
; examples: 82dc, 027339, 4cdbfd9bf0
; this is needed because the default CDDL binary string (bytes/bstr)
; is only CBOR and not JSON compatible
hexstring = text .regexp "([0-9a-f]{2})*"
```

Figure 1: Additional CDDL type definitions

All timestamps and time-related values (e.g., offsets) in qlog are logged as float64 in the millisecond resolution.

Other qlog documents can define their own CDDL-compatible (struct) types (e.g., separately for each Packet type that a protocol supports).

> Note to RFC editor: Please remove the following text in this section before publication.

The main general CDDL syntax conventions in this document a reader should be aware of for easy reading comprehension are:

- ? obj : this object is optional

- TypeName1 / TypeName2 : a union of these two types (object can be either type 1 OR type 2)

- obj: TypeName : this object has this concrete type

- obj: [* TypeName] : this object is an array of this type with minimum size of 0 elements

- obj: [+ TypeName] : this object is an array of this type with minimum size of 1 element

- TypeName = ... : defines a new type

- EnumName = "entry1" / "entry2" / entry3 / ...: defines an enum

- StructName = { ... } : defines a new struct type

- ; : single-line comment

** text => any : special syntax to indicate 0 or more fields that
 have a string key that maps to any value. Used to indicate a
 generic JSON object.

All timestamps and time-related values (e.g., offsets) in qlog are logged as float64 in the millisecond resolution.

Other qlog documents can define their own CDDL-compatible (struct) types (e.g., separately for each Packet type that a protocol supports).

1.1.2. Serialization examples

Serialization examples in this document use JSON ([JSON]) unless otherwise indicated.

2. Design goals

The main tenets for the qlog schema design are:

* Streamable, event-based logging

* A flexible format that can reduce log producer overhead, at the cost of increased complexity for consumers (e.g. tools)

* Extensible and pragmatic

* Aggregation and transformation friendly (e.g., the top-level element for the non-streaming format is a container for individual traces, group_ids can be used to tag events to a particular context)

* Metadata is stored together with event data

3. QlogFile schema

A qlog using the QlogFile schema can contain several individual traces and logs from multiple vantage points that are in some way related. The top-level element in this schema defines only a small set of "header" fields and an array of component traces, defined in Figure 2 as:

```
QlogFile = {
    qlog_version: text
    ? qlog_format: text .default "JSON"
    ? title: text
    ? description: text
    ? traces: [+ Trace /
                TraceError]
}
```

Figure 2: QlogFile definition

The required "qlog_version" field **MUST** have the value "0.4".

The optional "qlog_format" field indicates the serialization format. Its value **MUST** either be one of the options defined in this document (i.e., Section 10) or the field **MUST** be omitted entirely. When the field is omitted the default value of "JSON" applies.

The optional "title" and "description" fields provide additional free-text information about the file.

The optional "traces" field contains an array of qlog traces (Section 3.2), each of which contain metadata and an array of qlog events (Section 6).

In order to make it easier to parse and identify qlog files and their serialization format, the "qlog_version" and "qlog_format" fields and their values **SHOULD** be in the first 256 characters/bytes of the resulting log file.

Where a qlog file is serialized to a JSON format, one of the downsides is that it is inherently a non-streamable format. Put differently, it is not possible to simply append new qlog events to a log file without "closing" this file at the end by appending "]}]}". Without these closing tags, most JSON parsers will be unable to parse the file entirely. The alternative QlogFileSeq (Section 4) is better suited to streaming.

JSON serialization example:

```
{
    "qlog_version": "0.4",
    "qlog_format": "JSON",
    "title": "Name of this particular qlog file (short)",
    "description": "Description for this group of traces (long)",
    "traces": [...]
}
```

Figure 3: QlogFile example

3.1. Traces

It can be advantageous to group several related qlog traces together in a single file. For example, it is possible to simultaneously perform logging on the client, on the server, and on a single point on their common network path. For analysis, it is useful to aggregate these three individual traces together into a single file, so it can be uniquely stored, transferred, and annotated.

The QlogFile "traces" field is an array that contains a list of individual qlog traces. When capturing a qlog at a vantage point, it is expected that the traces field contains a single entry. Files can be aggregated, for example as part of a post-processing operation, by copying the traces in component to files into the combined "traces" array of a new, aggregated qlog file.

3.2. Trace

The exact conceptual definition of a Trace can be fluid. For example, a trace could contain all events for a single connection, for a single endpoint, for a single measurement interval, for a single protocol, etc. In the normal use case however, a trace is a log of a single data flow collected at a single location or vantage point. For example, for QUIC, a single trace only contains events for a single logical QUIC connection for either the client or the server.

A Trace contains some metadata in addition to qlog events, defined in Figure 4 as:

```
Trace = {
    ? title: text
    ? description: text
    ? common_fields: CommonFields
    ? vantage_point: VantagePoint
    events: [* Event]
}
```

Figure 4: Trace definition

The optional "title" and "description" fields provide additional free-text information about the trace.

The optional "common_fields" field is described in Section 6.9.

The optional "vantage_point" field is described in Section 5.

The semantics and context of the trace can mainly be deduced from the entries in the "common_fields" list and "vantage_point" field.

JSON serialization example:

```
{
    "title": "Name of this particular trace (short)",
    "description": "Description for this trace (long)",
    "common_fields": {
        "ODCID": "abcde1234",
        "time_format": "absolute"
    },
    "vantage_point": {
        "name": "backend-67",
        "type": "server"
    },
    "events": [...]
}
```

Figure 5: Trace example

3.3. TraceError

A TraceError indicates that an attempt to find/convert a file for inclusion in the aggregated qlog was made, but there was an error during the process. Rather than silently dropping the erroneous file, it can be explicitly included in the qlog file as an entry in the "traces" array, defined in Figure 6 as:

```
TraceError = {
    error_description: text

    ; the original URI used for attempted find of the file
    ? uri: text
    ? vantage_point: VantagePoint
}
```

Figure 6: TraceError definition

JSON serialization example:

```
{
    "error_description": "File could not be found",
    "uri": "/srv/traces/today/latest.qlog",
    "vantage_point": { type: "server" }
}
```

Figure 7: TraceError example

Note that another way to combine events of different traces in a single qlog file is through the use of the "group_id" field, discussed in Section 6.7.

4. QlogFileSeq schema

A qlog file using the QlogFileSeq schema can be serialized to a streamable JSON format called JSON Text Sequences (JSON-SEQ) ([RFC7464]). The top-level element in this schema defines only a small set of "header" fields and an array of component traces, defined in Figure 2 as:

```
QlogFileSeq = {
    qlog_format: "JSON-SEQ"
    qlog_version: text
    ? title: text
    ? description: text
    trace: TraceSeq
}
```

Figure 8: QlogFileSeq definition

The required "qlog_format" field **MUST** have the value "JSON-SEQ".

The required "qlog_version" field **MUST** have the value "0.4".

The optional "title" and "description" fields provide additional free-text information about the file.

The optional "trace" field contains a singular trace metadata. All qlog events in the file are related to this trace.

JSON-SEQ serialization example:

```
// list of qlog events, serialized in accordance with RFC 7464,
// starting with a Record Separator character and ending with a
// newline.
// For display purposes, Record Separators are rendered as <RS>

<RS>{
    "qlog_version": "0.4",
    "qlog_format": "JSON-SEQ",
    "title": "Name of JSON Text Sequence qlog file (short)",
    "description": "Description for this trace file (long)",
    "trace": {
      "common_fields": {
        "protocol_type": ["QUIC","HTTP3"],
        "group_id":"127ecc830d98f9d54a42c4f0842aa87e181a",
        "time_format":"relative",
        "reference_time": 1553986553572
      },
      "vantage_point": {
        "name":"backend-67",
        "type":"server"
      }
    }
}
<RS>{"time": 2, "name": "quic:parameters_set", "data": { ... } }
<RS>{"time": 7, "name": "quic:packet_sent", "data": { ... } }
...
```

Figure 9: Top-level element

For further information about serialization, see Section 10.2.

4.1. TraceSeq

TraceSeq is used with QlogFileSeq. It is conceptually similar to a Trace, with the exception that qlog events are not contained within it, but rather appended after it in a QlogFileSeq.

```
TraceSeq = {
    ? title: text
    ? description: text
    ? common_fields: CommonFields
    ? vantage_point: VantagePoint
}
```

Figure 10: TraceSeq definition

5. VantagePoint

A VantagePoint describes the vantage point from which a trace originates, defined in Figure 11 as:

```
VantagePoint = {
    ? name: text
    type: VantagePointType
    ? flow: VantagePointType
}

; client = endpoint which initiates the connection
; server = endpoint which accepts the connection
; network = observer in between client and server
VantagePointType = "client" /
                   "server" /
                   "network" /
                   "unknown"
```

Figure 11: VantagePoint definition

JSON serialization examples:

```
{
    "name": "aioquic client",
    "type": "client"
}

{
    "name": "wireshark trace",
    "type": "network",
    "flow": "client"
}
```

Figure 12: VantagePoint example

The flow field is only required if the type is "network" (for example, the trace is generated from a packet capture). It is used to disambiguate events like "packet sent" and "packet received". This is indicated explicitly because for multiple reasons (e.g., privacy) data from which the flow direction can be otherwise inferred (e.g., IP addresses) might not be present in the logs.

Meaning of the different values for the flow field: * "client" indicates that this vantage point follows client data flow semantics (a "packet sent" event goes in the direction of the server). * "server" indicates that this vantage point follow server data flow semantics (a "packet sent" event goes in the direction of the client). * "unknown" indicates that the flow's direction is unknown.

Depending on the context, tools confronted with "unknown" values in the vantage_point can either try to heuristically infer the semantics from protocol-level domain knowledge (e.g., in QUIC, the client always sends the first packet) or give the user the option to switch between client and server perspectives manually.

6. Events

A qlog event is specified as a generic object with a number of member fields and their associated data. Depending on the protocol and use case, the exact member field names and their formats can differ across implementations. This section lists the main, pre-defined and reserved field names with specific semantics and expected corresponding value formats.

An Event is defined in Figure 13 as:

```
Event = {
    time: float64
    name: text
    data: $ProtocolEventData
    ? path: PathID
    ? time_format: TimeFormat
    ? protocol_type: ProtocolType
    ? group_id: GroupID
    ? system_info: SystemInformation

    ; events can contain any amount of custom fields
    * text => any
}
```

Figure 13: Event definition

Each qlog event **MUST** contain the mandatory fields: "time" (Section 6.1), "name" (Section 6.2), and "data" (Section 6.3).

Each qlog event **MAY** contain the optional fields: "time_format" (Section 6.1), "protocol_type" (Section 6.5), "trigger" (Section 6.6), and "group_id" (Section 6.7).

Multiple events can appear in a Trace or TraceSeq and they might contain fields with identical values. It is possible to optimize out this duplication using "common_fields" (Section 6.9).

The specific values for each of these fields and their semantics are defined in separate documents, depending on protocol or use case.

For example: event definitions for QUIC and HTTP/3 can be found in [QLOG-QUIC] and [QLOG-H3].

Events are intended to be extended with custom fields, therefore they **MAY** contain other fields not defined in this document. Custom fields may be known or unknown to tools. Tools **SHOULD** allow for the presence of unknown event fields, but their semantics depend on the context of the log usage.

JSON serialization:

```
{
    "time": 1553986553572,

    "name": "quic:packet_sent",
    "data": { ... },

    "protocol_type":  ["QUIC","HTTP3"],
    "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",

    "time_format": "absolute",

    "ODCID": "127ecc830d98f9d54a42c4f0842aa87e181a"
}
```

Figure 14: Event example

6.1. Timestamps

An event's "time" field indicates the timestamp at which the event occurred. Its value is typically the Unix timestamp since the 1970 epoch (number of milliseconds since midnight UTC, January 1, 1970, ignoring leap seconds). However, qlog supports two more succinct timestamps formats to allow reducing file size. The employed format is indicated in the "time_format" field, which allows one of three values: "absolute", "delta" or "relative".

```
TimeFormat = "absolute" /
             "delta" /
             "relative"
```

Figure 15: TimeFormat definition

*Absolute: Include the full absolute timestamp with each event. This approach uses the largest amount of characters. This is also the default value of the "time_format" field.

*Delta: Delta-encode each time value on the previously logged value. The first event in a trace typically logs the full absolute timestamp. This approach uses the least amount of characters.

*Relative: Specify a full "reference_time" timestamp (typically this is done up-front in "common_fields", see Section 6.9) and include only relatively-encoded values based on this reference_time with each event. The "reference_time" value is typically the first absolute timestamp. This approach uses a medium amount of characters.

The first option is good for stateless loggers, the second and third for stateful loggers. The third option is generally preferred, since it produces smaller files while being easier to reason about. An example for each option can be seen in Figure 16.

```
The absolute approach will use:
1500, 1505, 1522, 1588

The delta approach will use:
1500, 5, 17, 66

The relative approach will:
- set the reference_time to 1500 in "common_fields"
- use: 0, 5, 22, 88
```

Figure 16: Three different approaches for logging timestamps

One of these options is typically chosen for the entire trace (put differently: each event has the same value for the "time_format" field). Each event **MUST** include a timestamp in the "time" field.

Events in each individual trace **SHOULD** be logged in strictly ascending timestamp order (though not necessarily absolute value, for the "delta" format). Tools **MAY** sort all events on the timestamp before processing them, though are not required to (as this could impose a significant processing overhead). This can be a problem especially for multi-threaded and/or streaming loggers, who could consider using a separate post-processor to order qlog events in time if a tool do not provide this feature.

Timestamps do not have to use the UNIX epoch timestamp as their reference. For example for privacy considerations, any initial reference timestamps (for example "endpoint uptime in ms" or "time since connection start in ms") can be chosen. Tools **SHOULD NOT** assume the ability to derive the absolute Unix timestamp from qlog traces, nor allow on them to relatively order events across two or more separate traces (in this case, clock drift should also be taken into account).

6.2. Names

Events differ mainly in the type of metadata associated with them. The "name" field is an identifier that parsers can use to decide how to interpret the event metadata contained in the "data" field (see Section 6.3).

Event names indicate a category and type. The "name" field **MUST** contain a non-empty character sequence representing a category, followed by a colon (':'), followed by a non-empty character sequence representing a type.

Category allows a higher-level grouping of events per specific event type. For example for QUIC and HTTP/3, the different categories could be "quic", "h3", "qpack", and "recovery". Within these categories, the event type provides additional granularity. For example for QUIC and HTTP/3, within the "quic" category, there would be "packet_sent" and "packet_received" events.

JSON serialization example:

```
{
    "name": "quic:packet_sent"
}
```

Figure 17: An event with category "quic" and type "packet_sent".

6.3. Data

An event's "data" field is a generic object. It contains the per-event metadata and its form and semantics are defined per specific sort of event. For example, data field value definitions for QUIC and HTTP/3 can be found in [QLOG-QUIC] and [QLOG-H3].

This field is defined here as a CDDL extension point (a "socket" or "plug") named $ProtocolEventData. Other documents **MUST** properly extend this extension point when defining new data field content options to enable automated validation of aggregated qlog schemas.

The only common field defined for the data field is the trigger field, which is discussed in Section 6.6.

```
; The ProtocolEventData is any key-value map (e.g., JSON object)
; only the optional trigger field is defined in this document
$ProtocolEventData /= {
    ? trigger: text
    * text => any
}
; event documents are intended to extend this socket by using:
; NewProtocolEventData = EventType1 /
;                        EventType2 /
;                        ... /
;                        EventTypeN
; $ProtocolEventData /= NewProtocolEventData
```

Figure 18: ProtocolEventData definition

One purely illustrative example for a QUIC "packet_sent" event is shown in Figure 19:

```
TransportPacketSent = {
    ? packet_size: uint16
    header: PacketHeader
    ? frames:[* QuicFrame]
    ? trigger: "pto_probe" /
               "retransmit_timeout" /
               "bandwidth_probe"
}

could be serialized as

{
    "packet_size": 1280,
    "header": {
        "packet_type": "1RTT",
        "packet_number": 123
    },
    "frames": [
        {
            "frame_type": "stream",
            "length": 1000,
            "offset": 456
        },
        {
            "frame_type": "padding"
        }
    ]
}
```

Figure 19: Example of the 'data' field for a QUIC packet_sent event

6.4. Path

A qlog event can be associated with a single "network path" (usually, but not always, identified by a 4-tuple of IP addresses and ports). In many cases, the path will be the same for all events in a given trace, and does not need to be logged explicitly with each event. In this case, the "path" field can be omitted (in which case the default value of "" is assumed) or reflected in "common_fields" instead (see Section 6.9).

However, in some situations, such as during QUIC's Connection Migration or when using Multipath features, it is useful to be able to split events across multiple (concurrent) paths.

Definition:

```
PathID = text .default ""
```

Figure 20: PathID definition

The "path" field is an identifier that is associated with a single network path. This document intentionally does not define further how to choose this identifier's value per-path or how to potentially log other parameters that can be associated with such a path. This is left for other documents. Implementers are free to encode path information directly into the PathID or to log associated info in a separate event. For example, QUIC has the "path_assigned" event to couple the PathID value to a specific path configuration, see [QLOG-QUIC].

6.5. ProtocolType

An event's "protocol_type" array field indicates to which protocols (or protocol "stacks") this event belongs. This allows a single qlog file to aggregate traces of different protocols (e.g., a web server offering both TCP+HTTP/2 and QUIC+HTTP/3 connections).

```
ProtocolType = [+ text]
```

Figure 21: ProtocolType definition

For example, QUIC and HTTP/3 events have the "QUIC" and "HTTP3" protocol_type entry values, see [QLOG-QUIC] and [QLOG-H3].

Typically however, all events in a single trace are of the same few protocols, and this array field is logged once in "common_fields", see Section 6.9.

6.6. Triggers

Sometimes, additional information is needed in the case where a single event can be caused by a variety of other events. In the normal case, the context of the surrounding log messages gives a hint as to which of these other events was the cause. However, in highly-parallel and optimized implementations, corresponding log messages might separated in time. Another option is to explicitly indicate these "triggers" in a high-level way per-event to get more fine-grained information without much additional overhead.

In qlog, the optional "trigger" field contains a string value describing the reason (if any) for this event instance occurring, see Section 6.3. While this "trigger" field could be a property of the qlog Event itself, it is instead a property of the "data" field instead. This choice was made because many event types do not include a trigger value, and having the field at the Event-level would cause overhead in some serializations. Additional information on the trigger can be added in the form of additional member fields of the "data" field value, yet this is highly implementation-specific, as are the trigger field's string values.

One purely illustrative example of some potential triggers for QUIC's "packet_dropped" event is shown in Figure 22:

```
TransportPacketDropped = {
    ? packet_type: PacketType
    ? raw_length: uint16
    ? trigger: "key_unavailable" /
               "unknown_connection_id" /
               "decrypt_error" /
               "unsupported_version"
}
```

Figure 22: Trigger example

6.7. Grouping

As discussed in Section 3.2, a single qlog file can contain several traces taken from different vantage points. However, a single trace from one endpoint can also contain events from a variety of sources. For example, a server implementation might choose to log events for all incoming connections in a single large (streamed) qlog file. As such, a method for splitting up events belonging to separate logical entities is required.

The simplest way to perform this splitting is by associating a "group id" to each event that indicates to which conceptual "group" each event belongs. A post-processing step can then extract events per group. However, this group identifier can be highly protocol and context-specific. In the example above, the QUIC "Original Destination Connection ID" could be used to uniquely identify a connection. As such, they might add a "ODCID" field to each event. However, a middlebox logging IP or TCP traffic might rather use four-tuples to identify connections, and add a "four_tuple" field.

As such, to provide consistency and ease of tooling in cross-protocol and cross-context setups, qlog instead defines the common "group_id" field, which contains a string value. Implementations are free to use their preferred string serialization for this field, so long as it contains a unique value per logical group. Some examples can be seen in Figure 24.

```
GroupID = text
```

Figure 23: GroupID definition

JSON serialization example for events grouped by four tuples and QUIC connection IDs:

```
"events": [
    {
        "time": 1553986553579,
        "protocol_type": ["TCP", "TLS", "HTTP2"],
        "group_id": "ip1=2001:67c:1232:144:9498:6df6:f450:110b,
                    ip2=2001:67c:2b0:1c1::198,port1=59105,port2=80",
        "name": "quic:packet_received",
        "data": { ... }
    },
    {
        "time": 1553986553581,
        "protocol_type": ["QUIC","HTTP3"],
        "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
        "name": "quic:packet_sent",
        "data": { ... }
    }
]
```

Figure 24: GroupID example

Note that in some contexts (for example a Multipath transport protocol) it might make sense to add additional contextual per-event fields (for example "path_id"), rather than use the group_id field for that purpose.

Note also that, typically, a single trace only contains events belonging to a single logical group (for example, an individual QUIC connection). As such, instead of logging the "group_id" field with an identical value for each event instance, this field is typically logged once in "common_fields", see Section 6.9.

6.8. SystemInformation

The "system_info" field can be used to record system-specific details related to an event. This is useful, for instance, where an application splits work across CPUs, processes, or threads and events for a single trace occur on potentially different combinations thereof. Each field is optional to support deployment diversity.

```
SystemInformation = {
  ? processor_id: uint32
  ? process_id: uint32
  ? thread_id: uint32
}
```

6.9. CommonFields

As discussed in the previous sections, information for a typical qlog event varies in three main fields: "time", "name" and associated data. Additionally, there are also several more advanced fields that allow mixing events from different protocols and contexts inside of the same trace (for example "protocol_type" and "group_id"). In most "normal" use cases however, the values of these advanced fields are consistent for each event instance (for example, a single trace contains events for a single QUIC connection).

To reduce file size and making logging easier, qlog uses the "common_fields" list to indicate those fields and their values that are shared by all events in this component trace. This prevents these fields from being logged for each individual event. An example of this is shown in Figure 25.

```
JSON serialization with repeated field values
per-event instance:

{
    "events": [{
            "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
            "protocol_type": ["QUIC","HTTP3"],
            "time_format": "relative",
            "reference_time": 1553986553572,

            "time": 2,
            "name": "quic:packet_received",
            "data": { ... }
        },{
            "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
            "protocol_type": ["QUIC","HTTP3"],
            "time_format": "relative",
            "reference_time": 1553986553572,

            "time": 7,
            "name": "http:frame_parsed",
            "data": { ... }
        }
    ]
}

JSON serialization with repeated field values instead
extracted to common_fields:

{
    "common_fields": {
        "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
        "protocol_type": ["QUIC","HTTP3"],
        "time_format": "relative",
        "reference_time": 1553986553572
    },
    "events": [
        {
            "time": 2,
            "name": "quic:packet_received",
            "data": { ... }
        },{
            "time": 7,
            "name": "http:frame_parsed",
            "data": { ... }
        }
    ]
}
```

Figure 25: CommonFields example

An event's "common_fields" field is a generic dictionary of key-value pairs, where the key is always a string and the value can be of any type, but is typically also a string or number. As such, unknown entries in this dictionary **MUST** be disregarded by the user and tools (i.e., the presence of an unknown field is explicitly NOT an error).

The list of default qlog fields that are typically logged in common_fields (as opposed to as individual fields per event instance) are shown in the listing below:

```
CommonFields = {
    ? path: PathID
    ? time_format: TimeFormat
    ? reference_time: float64
    ? protocol_type: ProtocolType
    ? group_id: GroupID
    * text => any
}
```

Figure 26: CommonFields definition

Tools **MUST** be able to deal with these fields being defined either on each event individually or combined in common_fields. Note that if at least one event in a trace has a different value for a given field, this field **MUST NOT** be added to common_fields but instead defined on each event individually. Good example of such fields are "time" and "data", who are divergent by nature.

7. Raw packet and frame information

While qlog is a high-level logging format, it also allows the inclusion of most raw wire image information, such as byte lengths and byte values. This is useful when for example investigating or tuning packetization behavior or determining encoding/framing overheads. However, these fields are not always necessary, can take up considerable space, and can have a considerable privacy and security impact (see Section 13). Where applicable, these fields are grouped in a separate, optional, field named "raw" of type RawInfo. The exact definition of entities, headers, trailers and payloads depend on the protocol used.

```
RawInfo = {

    ; the full byte length of the entity (e.g., packet or frame),
    ; including possible headers and trailers
    ? length: uint64

    ; the byte length of the entity's payload,
    ; excluding possible headers or trailers
    ? payload_length: uint64

    ; the (potentially truncated) contents of the full entity,
    ; including headers and possibly trailers
    ? data: hexstring
}
```

Figure 27: RawInfo definition

The RawInfo:data field can be truncated for privacy or security purposes, see Section 10.4. In this case, the length and payload_length fields should still indicate the non-truncated lengths when used for debugging purposes.

This document does not specify explicit header_length or trailer_length fields. In protocols without trailers, header_length can be calculated by subtracting the payload_length from the length. In protocols with trailers (e.g., QUIC's AEAD tag), event definition documents **SHOULD** define how to support header_length calculation.

8. Common events and data classes

There are some event types and data classes that are common across protocols, applications, and use cases. This section specifies such common definitions.

8.1. Generic events

In typical logging setups, users utilize a discrete number of well-defined logging categories, levels or severities to log freeform (string) data. This generic events category replicates this approach to allow implementations to fully replace their existing text-based logging by qlog. This is done by providing events to log generic strings for the typical well-known logging levels (error, warning, info, debug, verbose).

For the events defined below, the "category" is "generic" and their "type" is the name of the heading in lowercase (e.g., the "name" of the error event is "generic:error").

8.1.1. error

Used to log details of an internal error that might not get reflected on the wire. It has Core importance level; see Section 9.2.

```
GenericError = {
    ? code: uint64
    ? message: text
}
```

Figure 28: GenericError definition

8.1.2. warning

Used to log details of an internal warning that might not get reflected on the wire. It has Base importance level; see Section 9.2.

```
GenericWarning = {
    ? code: uint64
    ? message: text
}
```

Figure 29: GenericWarning definition

8.1.3. info

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages. The event has Extra importance level; see Section 9.2.

```
GenericInfo = {
    message: text
}
```

Figure 30: GenericInfo definition

8.1.4. debug

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages. The event has Extra importance level; see Section 9.2.

```
GenericDebug = {
    message: text
}
```

Figure 31: GenericDebug definition

8.1.5. verbose

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages. The event has Extra importance level; see Section 9.2.

```
GenericVerbose = {
    message: text
}
```

Figure 32: GenericVerbose definition

8.2. Simulation events

When evaluating a protocol implementation, one typically sets up a series of interoperability or benchmarking tests, in which the test situations can change over time. For example, the network bandwidth or latency can vary during the test, or the network can be fully disable for a short time. In these setups, it is useful to know when exactly these conditions are triggered, to allow for proper correlation with other events.

For the events defined below, the "category" is "simulation" and their "type" is the name of the heading in lowercase (e.g., the "name" of the scenario event is "simulation:scenario").

8.2.1. scenario

Used to specify which specific scenario is being tested at this particular instance. This supports, for example, aggregation of several simulations into one trace (e.g., split by group_id). It has Extra importance level; see Section 9.2.

```
SimulationScenario = {
    ? name: text
    ? details: {* text => any }
}
```

Figure 33: SimulationScenario definition

8.2.2. marker

Used to indicate when specific emulation conditions are triggered at set times (e.g., at 3 seconds in 2% packet loss is introduced, at 10s a NAT rebind is triggered). It has Extra importance level; see Section 9.2.

```
SimulationMarker = {
    ? type: text
    ? message: text
}
```

Figure 34: SimulationMarker definition

9. Event definition guidelines

This document defines the main schema for the qlog format together with some common events, which on their own do not provide much logging utility. It is expected that logging is extended with specific, per-protocol event definitions that specify the name (category + type) and data needed for each individual event. Examples include the QUIC event definitions [QLOG-QUIC] and HTTP/3 event definitions [QLOG-H3].

This section defines some basic annotations and concepts that **SHOULD** be used by event definition documents. Doing so ensures a measure of consistency that makes it easier for qlog implementers to support a wide variety of protocols.

9.1. Event design

There are several ways of defining qlog events. In practice, two main types of approach have been observed: a) those that map directly to concepts seen in the protocols (e.g., packet_sent) and b) those that act as aggregating events that combine data from several possible protocol behaviors or code paths into one (e.g., parameters_set). The latter are typically used as a means to reduce the amount of unique event definitions, as reflecting each possible protocol event as a separate qlog entity would cause an explosion of event types.

Additionally, logging duplicate data is typically prevented as much as possible. For example, packet header values that remain consistent across many packets are split into separate events (for example spin_bit_updated or connection_id_updated for QUIC).

Finally, when logging additional state change events, those state changes can often be directly inferred from data on the wire (for example flow control limit changes). As such, if the implementation is bug-free and spec-compliant, logging additional events is typically avoided. Exceptions have been made for common events that benefit from being easily identifiable or individually logged (for example packets_acked).

9.2. Event importance levels

Depending on how events are designed, it may be that several events allow the logging of similar or overlapping data. For example the separate QUIC connection_started event overlaps with the more generic connection_state_updated. In these cases, it is not always clear which event should be logged or used, and which event should take precedence if e.g., both are present and provide conflicting information.

To aid in this decision making, qlog defines three event importance levels, in decreasing order of importance and expected usage:

* Core

* Base

* Extra

Events definitions **SHOULD** assign an importance level.

Core-level events **SHOULD** be present in all qlog files for a given protocol. These are typically tied to basic packet and frame parsing and creation, as well as listing basic internal metrics. Tool implementers **SHOULD** expect and add support for these events, though **SHOULD NOT** expect all Core events to be present in each qlog trace.

Base-level events add additional debugging options and **MAY** be present in qlog files. Most of these can be implicitly inferred from data in Core events (if those contain all their properties), but for many it is better to log the events explicitly as well, making it clearer how the implementation behaves. These events are for example tied to passing data around in buffers, to how internal state machines change, and used to help show when decisions are actually made based on received data. Tool implementers **SHOULD** at least add support for showing the contents of these events, if they do not handle them explicitly.

Extra-level events are considered mostly useful for low-level debugging of the implementation, rather than the protocol. They allow more fine-grained tracking of internal behavior. As such, they **MAY** be present in qlog files and tool implementers **MAY** add support for these, but they are not required to.

Note that in some cases, implementers might not want to log for example data content details in Core-level events due to performance or privacy considerations. In this case, they **SHOULD** use (a subset of) relevant Base-level events instead to ensure usability of the qlog output. As an example, implementations that do not log QUIC packet_received events and thus also not which (if any) ACK frames the packet contains, **SHOULD** log packets_acked events instead.

Finally, for event types whose data (partially) overlap with other event types' definitions, where necessary the event definition document should include explicit guidance on which to use in specific situations.

9.3. Custom fields

Event definition documents are free to define new category and event types, top-level fields (e.g., a per-event field indicating its privacy properties or path_id in multipath protocols), as well as values for the "trigger" property within the "data" field, or other member fields of the "data" field, as they see fit.

They however **SHOULD NOT** expect non-specialized tools to recognize or visualize this custom data. However, tools **SHOULD** make an effort to visualize even unknown data if possible in the specific tool's context. If they do not, they **MUST** ignore these unknown fields.

10. Serializing qlog

qlog schema definitions in this document are intentionally agnostic to serialization formats. The choice of format is an implementation decision.

Other documents related to qlog (for example event definitions for specific protocols), **SHOULD** be similarly agnostic to the employed serialization format and **SHOULD** clearly indicate this. If not, they **MUST** include an explanation on which serialization formats are supported and on how to employ them correctly.

Serialization formats make certain tradeoffs between usability, flexibility, interoperability, and efficiency. Implementations should take these into consideration when choosing a format. Some examples of possible formats are JSON, CBOR, CSV, protocol buffers, flatbuffers, etc. which each have their own characteristics. For instance, a textual format like JSON can be more flexible than a binary format but more verbose, typically making it less efficient than a binary format. A plaintext readable (yet relatively large) format like JSON is potentially more usable for users operating on the logs directly, while a more optimized yet restricted format can better suit the constraints of a large scale operation. A custom or restricted format could be more efficient for analysis with custom tooling but might not be interoperable with general-purpose qlog tools.

Considering these tradeoffs, JSON-based serialization formats provide features that make them a good starting point for qlog flexibility and interoperability. For these reasons, JSON is a recommended default and expanded considerations are given to how to map qlog to JSON (Section 10.1, and its streaming counterpart JSON Text Sequences (Section 10.2. Section 10.3 presents interoperability considerations for both formats, and Section 10.5 presents potential optimizations.

Serialization formats require appropriate deserializers/parsers. The "qlog_format" field (Section 3) is used to indicate the chosen serialization format.

10.1. qlog to JSON mapping

As described in Section 3, JSON is the default qlog serialization. When mapping qlog to normal JSON, QlogFile (Figure 2) is used and the "qlog_format" field **MUST** have the value "JSON". The file extension/suffix **SHOULD** be ".qlog". The Media Type, if any, **SHOULD** be "application/qlog+json" per [RFC6839].

In accordance with Section 8.1 of [RFC8259], JSON files are required to use UTF-8 both for the file itself and the string values it contains. In addition, all qlog field names **MUST** be lowercase when serialized to JSON.

In order to serialize CDDL-based qlog event and data structure definitions to JSON, the official CDDL-to-JSON mapping defined in Appendix E of [CDDL] **SHOULD** be employed.

10.2. qlog to JSON Text Sequences mapping

One of the downsides of using normal JSON is that it is inherently a non-streamable format. A qlog serializer could work around this by opening a file, writing the required opening data, streaming qlog events by appending them, and then finalizing the log by appending appropriate closing tags e.g., "]}]}". However, failure to append closing tags, could lead to problems because most JSON parsers will fail if a document is malformed. Some streaming JSON parsers are able to handle missing closing tags, however they are not widely deployed in popular environments (e.g., Web browsers)

To overcome the issues related to JSON streaming, a qlog mapping to a streamable JSON format called JSON Text Sequences (JSON-SEQ) ([RFC7464]) is provided.

JSON Text Sequences are very similar to JSON, except that objects are serialized as individual records, each prefixed by an ASCII Record Separator (<RS>, 0x1E), and each ending with an ASCII Line Feed character (\n, 0x0A). Note that each record can also contain any amount of newlines in its body, as long as it ends with a newline character before the next <RS> character.

In order to leverage the streaming capability, each qlog event is serialized and interpreted as an individual JSON Text Sequence record, that is appended as a new object to the back of an event stream or log file. Put differently, unlike default JSON, it does not require a document to be wrapped as a full object with "{ ... }" or "[...]".

This alternative record streaming approach cannot be accommodated by QlogFile (Figure 2). Instead, QlogFileSeq is defined in Figure 8, which notably includes only a single trace (TraceSeq) and omits an explicit "events" array. An example is provided in Figure 9. The "group_id" field can still be used on a per-event basis to include events from conceptually different sources in a single JSON-SEQ qlog file.

When mapping qlog to JSON-SEQ, the "qlog_format" field **MUST** have the value "JSON-SEQ". The file extension/suffix **SHOULD** be ".sqlog" (for "streaming" qlog). The Media Type, if any, **SHOULD** be "application/qlog+json-seq" per [RFC8091].

While not specifically required by the JSON-SEQ specification, all qlog field names **MUST** be lowercase when serialized to JSON-SEQ.

In order to serialize all other CDDL-based qlog event and data structure definitions to JSON-SEQ, the official CDDL-to-JSON mapping defined in Appendix E of [CDDL] **SHOULD** be employed.

10.2.1. Supporting JSON Text Sequences in tooling

Note that JSON Text Sequences are not supported in most default programming environments (unlike normal JSON). However, several custom JSON-SEQ parsing libraries exist in most programming languages that can be used and the format is easy enough to parse with existing implementations (i.e., by splitting the file into its component records and feeding them to a normal JSON parser individually, as each record by itself is a valid JSON object).

10.3. JSON Interoperability

Some JSON implementations have issues with the full JSON format, especially those integrated within a JavaScript environment (e.g., Web browsers, NodeJS). I-JSON (Internet-JSON) is a subset of JSON for such environments; see [I-JSON]. One of the key limitations of

JavaScript, and thus I-JSON, is that it cannot represent full 64-bit integers in standard operating mode (i.e., without using BigInt extensions), instead being limited to the range $-(2^{53})+1$ to $(2^{53})-1$.

To accommodate such constraints in CDDL, Appendix E of [CDDL] recommends defining new CDDL types for int64 and uint64 that limit their values to the restricted 64-bit integer range. However, some of the protocols that qlog is intended to support (e.g., QUIC, HTTP/3), can use the full range of uint64 values.

As such, to support situations where I-JSON is in use, seralizers **MAY** encode uint64 values using JSON strings. qlog parsers, therefore, **SHOULD** support parsing of uint64 values from JSON strings or JSON numbers unless there is out-of-band information indicating that neither the serializer nor parser are constrained by I-JSON.

10.4. Truncated values

For some use cases (e.g., limiting file size, privacy), it can be necessary not to log a full raw blob (using the hexstring type) but instead a truncated value. For example, one might only store the first 100 bytes of an HTTP response body to be able to discern which file it actually contained. In these cases, the original byte-size length cannot be obtained from the serialized value directly.

As such, all qlog schema definitions **SHOULD** include a separate, length-indicating field for all fields of type hexstring they specify, see for example Section 7. This not only ensures the original length can always be retrieved, but also allows the omission of any raw value bytes of the field completely (e.g., out of privacy or security considerations).

To reduce overhead however and in the case the full raw value is logged, the extra length-indicating field can be left out. As such, tools **MUST** be able to deal with this situation and derive the length of the field from the raw value if no separate length-indicating field is present. The main possible permutations are shown by example in Figure 35.

```
// both the content's value and its length are present
// (length is redundant)
{
    "content_length": 5,
    "content": "051428abff"
}

// only the content value is present, indicating it
// represents the content's full value. The byte
// length is obtained by calculating content.length / 2
{
    "content": "051428abff"
}

// only the length is present, meaning the value
// was omitted
{
    "content_length": 5,
}

// both value and length are present, but the lengths
// do not match: the value was truncated to
// the first three bytes.
{
    "content_length": 5,
    "content": "051428"
}
```

Figure 35: Example for serializing truncated hexstrings

10.5. Optimization of serialized data

Both the JSON and JSON-SEQ formatting options described above are serviceable in general small to medium scale (debugging) setups. However, these approaches tend to be relatively verbose, leading to larger file sizes. Additionally, generalized JSON(-SEQ) (de)serialization performance is typically (slightly) lower than that of more optimized and predictable formats. Both aspects present challenges to large scale setups, though they may still be practical to deploy; see [ANRW-2020]. JSON and JSON-SEQ compress very well using commonly-available algorithms such as GZIP or Brotli.

During the development of qlog, a multitude of alternative formatting and optimization options were assessed and the results are summarized on the qlog github repository.

Formal definition of additional qlog formats or encodings that use the optimization techniques described here, or any other optimization technique is left to future activity that can apply the following guidelines.

In order to help tools correctly parse and process serialized qlog, it is **RECOMMENDED** that new formats also define suitable file extensions and media types. This provides a clear signal and avoids the need to provide out-of-band information or to rely on heuristic fallbacks; see Section 12.

11. Methods of access and generation

Different implementations will have different ways of generating and storing qlogs. However, there is still value in defining a few default ways in which to steer this generation and access of the results.

11.1. Set file output destination via an environment variable

To provide users control over where and how qlog files are created, two environment variables are defined. The first, QLOGFILE, indicates a full path to where an individual qlog file should be stored. This path **MUST** include the full file extension. The second, QLOGDIR, sets a general directory path in which qlog files should be placed. This path **MUST** include the directory separator character at the end.

In general, QLOGDIR should be preferred over QLOGFILE if an endpoint is prone to generate multiple qlog files. This can for example be the case for a QUIC server implementation that logs each QUIC connection in a separate qlog file. An alternative that uses QLOGFILE would be a QUIC server that logs all connections in a single file and uses the "group_id" field (Section 6.7) to allow post-hoc separation of events.

Implementations **SHOULD** provide support for QLOGDIR and **MAY** provide support for QLOGFILE.

When using QLOGDIR, it is up to the implementation to choose an appropriate naming scheme for the qlog files themselves. The chosen scheme will typically depend on the context or protocols used. For example, for QUIC, it is recommended to use the Original Destination Connection ID (ODCID), followed by the vantage point type of the logging endpoint. Examples of all options for QUIC are shown in Figure 36.

```
Command: QLOGFILE=/srv/qlogs/client.qlog quicclientbinary

Should result in the the quicclientbinary executable logging a
single qlog file named client.qlog in the /srv/qlogs directory.
This is for example useful in tests when the client sets up
just a single connection and then exits.

Command: QLOGDIR=/srv/qlogs/ quicserverbinary

Should result in the quicserverbinary executable generating
several logs files, one for each QUIC connection.
Given two QUIC connections, with ODCID values "abcde" and
"12345" respectively, this would result in two files:
/srv/qlogs/abcde_server.qlog
/srv/qlogs/12345_server.qlog

Command: QLOGFILE=/srv/qlogs/server.qlog quicserverbinary

Should result in the the quicserverbinary executable logging
a single qlog file named server.qlog in the /srv/qlogs directory.
Given that the server handled two QUIC connections before it was
shut down, with ODCID values "abcde" and "12345" respectively,
this would result in event instances in the qlog file being
tagged with the "group_id" field with values "abcde" and "12345".
```

Figure 36: Environment variable examples for a QUIC implementation

12. Tooling requirements

Tools ingestion qlog **MUST** indicate which qlog version(s), qlog format(s), compression methods and potentially other input file formats (for example .pcap) they support. Tools **SHOULD** at least support .qlog files in the default JSON format (Section 10.1). Additionally, they **SHOULD** indicate exactly which values for and properties of the name (category and type) and data fields they look for to execute their logic. Tools **SHOULD** perform a (high-level) check if an input qlog file adheres to the expected qlog schema. If a tool determines a qlog file does not contain enough supported information to correctly execute the tool's logic, it **SHOULD** generate a clear error message to this effect.

Tools **MUST NOT** produce breaking errors for any field names and/or values in the qlog format that they do not recognize. Tools **SHOULD** indicate even unknown event occurrences within their context (e.g., marking unknown events on a timeline for manual interpretation by the user).

Tool authors should be aware that, depending on the logging implementation, some events will not always be present in all traces. For example, using a circular logging buffer of a fixed size, it could be that the earliest events (e.g., connection setup events) are later overwritten by "newer" events. Alternatively, some events can be intentionally omitted out of privacy or file size considerations. Tool authors are encouraged to make their tools robust enough to still provide adequate output for incomplete logs.

13. Security and privacy considerations

Protocols such as TLS [RFC8446] and QUIC [RFC9000] offer secure protection for the wire image [RFC8546]. Logging can reveal aspects of the wire image that would ordinarily be protected, creating tension between observability, security and privacy, especially if data can be correlated across data sources.

Depending on the observability use case any data could be logged or captured. As per [RFC6973], operators must be aware that such data could be compromised, risking the privacy of all participants. Entities that expect protocol features to ensure data privacy might unknowingly be subject to broader privacy risks, undermining their ability to assess or respond effectively.

13.1. Data at risk

qlog operators and implementers need to consider security and privacy risks when handling qlog data, including logging, storage, usage, and more. The considerations presented in this section may pose varying risks depending on the the data itself or its handling.

The following is a non-exhaustive list of example data types that could contain sensitive information that might allow identification or correlation of individual connections, endpoints, users or sessions across qlog or other data sources (e.g., captures of encrypted packets):

* IP addresses and transport protocol port numbers.

* Session, Connection, or User identifiers e.g., QUIC Connection IDs Section 9.5 of [RFC9000]).

* System-level information e.g., CPU, process, or thread identifiers.

* Stored State e.g., QUIC address validation and retry tokens, TLS session tickets, and HTTP cookies.

* TLS decryption keys, passwords, and HTTP-level API access or authorization tokens.

* High-resolution event timestamps or inter-event timings, event counts, packet sizes, and frame sizes.

* Full or partial raw packet and frame payloads that are encrypted.
* Full or partial raw packet and frame payloads that are plaintext e.g., HTTP Field values, HTTP response data, or TLS SNI field values.

13.2. Operational implications and recommendations

Operational considerations should focus on authorizing capture and access to logs. Logging of Internet protocols using qlog can be equivalent to the ability to store or read plaintext communications. Without a more detailed analysis, all of the security considerations of plaintext access apply.

It is recommended that qlog capture is subject to access control and auditing. These controls should support granular levels of information capture based on role and permissions (e.g., capture of more-sensitive data requires higher privileges).

It is recommended that access to stored qlogs is subject to access control and auditing.

End users might not understand the implications of qlog to security or privacy, and their environments might limit access control techniques. Implementations should make enabling qlog conspicuous (e.g., requiring clear and explicit actions to start a capture) and resistant to social engineering, automation, or drive-by attacks; for example, isolation or sandboxing of capture from other activities in the same process or component.

It is recommended that data retention policies are defined for the storage of qlog files.

It is recommended that qlog files are encrypted in transit and at rest.

13.3. Data minimization or anonymization

Applying data minimization or anonymization techniques to qlog might help address some security and privacy risks. However, removing or anonymizing data without sufficient care might not enhance privacy or security and could diminish the utility of qlog data.

Operators and implementers should balance the value of logged data with the potential risks of (involuntary) disclosure, which can depend on use cases (e.g., research datasets might have different requirements to live operational troubleshooting).

The most extreme form of minimization or anonymization is deleting a field, equivalent to not logging it. qlog implementations should offer fine-grained control for this on a per-use-case or per-connection basis.

Data can undergo anonymization, pseudonymization, permutation, truncation, re-encryption, or aggregation; see Appendix B of [DNS-PRIVACY] for techniques, especially regarding IP addresses. However, operators should be cautious because many anonymization methods have been shown to be insufficient to safeguard user privacy or identity, particularly with large or easily correlated data sets.

Operators should consider end user rights and preferences. Active user participation (as indicated by [RFC6973]) on a per-qlog basis is challenging but aligning qlog capture, storage, and removal with existing user preference and privacy controls is crucial. Operators should consider agressive approaches to deletion or aggregation.

The most sensitive data in qlog is typically contained in RawInfo type fields (see Section 7). Therefore, qlog users should exercise caution and limit the inclusion of such fields for all but the most stringent use cases.

14. IANA Considerations

There are no IANA considerations.

15. References

15.1. Normative References

[CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[DNS-PRIVACY] Dickinson, S., Overeinder, B., van Rijswijk-Deij, R., and A. Mankin, "Recommendations for DNS Privacy Service Operators", BCP 232, RFC 8932, DOI 10.17487/RFC8932, October 2020, <https://www.rfc-editor.org/rfc/rfc8932>.

[I-JSON] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/rfc/rfc7493>.

[JSON] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/

RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[JSON-Text-Sequences]
Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <https://www.rfc-editor.org/rfc/rfc7464>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC6839] Hansen, T. and A. Melnikov, "Additional Media Type Structured Syntax Suffixes", RFC 6839, DOI 10.17487/RFC6839, January 2013, <https://www.rfc-editor.org/rfc/rfc6839>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC7464] Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <https://www.rfc-editor.org/rfc/rfc7464>.

[RFC8091] Wilde, E., "A Media Type Structured Syntax Suffix for JSON Text Sequences", RFC 8091, DOI 10.17487/RFC8091, February 2017, <https://www.rfc-editor.org/rfc/rfc8091>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

15.2. Informative References

[ANRW-2020]

Marx, R., Piraux, M., Quax, P., and W. Lamotte, "Debugging QUIC and HTTP/3 with qlog and qvis", September 2020, <https://qlog.edm.uhasselt.be/anrw/>.

[QLOG-H3] Marx, R., Niccolini, L., Seemann, M., and L. Pardue, "HTTP/3 qlog event definitions", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-h3-events-06, 5 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-h3-events-06>.

[QLOG-QUIC] Marx, R., Niccolini, L., Seemann, M., and L. Pardue, "QUIC event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-quic-events-06, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-quic-events-06>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8546] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/rfc/rfc8546>.

Acknowledgements

Much of the initial work by Robin Marx was done at the Hasselt and KU Leuven Universities.

Thanks to Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, and Jeremy Laine for their feedback and suggestions.

Change Log

This section is to be removed before publishing as an RFC.

Since draft-ietf-quic-qlog-main-schema-07:

* Added path and PathID (#336)
* Removed custom definition of uint64 type (#360, #388)
* ProtocolEventBody is now called ProtocolEventData (#352)
* Editorial changes (#364, #289, #353, #361, #362)

Since draft-ietf-quic-qlog-main-schema-06:

* Editorial reworking of the document (#331, #332)

* Updated IANA considerations section (#333)

Since draft-ietf-quic-qlog-main-schema-05:

* Updated qlog_version to 0.4 (due to breaking changes) (#314)

* Renamed 'transport' category to 'quic' (#302)

* Added 'system_info' field (#305)

* Removed 'summary' and 'configuration' fields (#308)

* Editorial and formatting changes (#298, #303, #304, #316, #320, #321, #322, #326, #328)

Since draft-ietf-quic-qlog-main-schema-04:

* Updated RawInfo definition and guidance (#243)

Since draft-ietf-quic-qlog-main-schema-03:

* Added security and privacy considerations discussion (#252)

Since draft-ietf-quic-qlog-main-schema-02:

* No changes - new draft to prevent expiration

Since draft-ietf-quic-qlog-main-schema-01:

* Change the data definition language from TypeScript to CDDL (#143)

Since draft-ietf-quic-qlog-main-schema-00:

* Changed the streaming serialization format from NDJSON to JSON Text Sequences (#172)

* Added Media Type definitions for various qlog formats (#158)

* Changed to semantic versioning

Since draft-marx-qlog-main-schema-draft-02:

* These changes were done in preparation of the adoption of the drafts by the QUIC working group (#137)

* Moved RawInfo, Importance, Generic events and Simulation events to this document.

* Added basic event definition guidelines

* Made protocol_type an array instead of a string (#146)

Since draft-marx-qlog-main-schema-01:

* Decoupled qlog from the JSON format and described a mapping instead (#89)

  - Data types are now specified in this document and proper definitions for fields were added in this format

  - 64-bit numbers can now be either strings or numbers, with a preference for numbers (#10)

  - binary blobs are now logged as lowercase hex strings (#39, #36)

  - added guidance to add length-specifiers for binary blobs (#102)

* Removed "time_units" from Configuration. All times are now in ms instead (#95)

* Removed the "event_fields" setup for a more straightforward JSON format (#101,#89)

* Added a streaming option using the NDJSON format (#109,#2,#106)

* Described optional optimization options for implementers (#30)

* Added QLOGDIR and QLOGFILE environment variables, clarified the .well-known URL usage (#26,#33,#51)

* Overall tightened up the text and added more examples

Since draft-marx-qlog-main-schema-00:

* All field names are now lowercase (e.g., category instead of CATEGORY)

* Triggers are now properties on the "data" field value, instead of separate field types (#23)

* group_ids in common_fields is now just also group_id

Authors' Addresses

Robin Marx (editor)
Akamai

Email: rmarx@akamai.com

Luca Niccolini (editor)
Meta

Email: lniccolini@meta.com

Marten Seemann (editor)

Email: martenseemann@gmail.com

Lucas Pardue (editor)
Cloudflare

Email: lucas@lucaspardue.com