

HTTPBIS
Internet-Draft
Intended status: Standards Track
Expires: 11 February 2022

M. Thomson
Mozilla
C.A. Wood
Cloudflare
10 August 2021

Binary Representation of HTTP Messages
draft-thomson-http-binary-message-01

Abstract

This document defines a binary format for representing HTTP messages.

Discussion Venues

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

Discussion of this document takes place on the HTTP Working Group mailing list (http@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/http/>.

Source for this draft and an issue tracker can be found at <https://github.com/unicorn-wg/oblivious-http>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 11 February 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
3. Format	3
3.1. Known Length Messages	4
3.2. Indeterminate Length Messages	5
3.3. Framing Indicator	6
3.4. Request Control Data	6
3.5. Response Control Data	7
3.5.1. Informational Status Codes	7
3.6. Header and Trailer Field Lines	8
3.7. Content	9
4. Invalid Messages	9
5. Examples	9
5.1. Request Example	9
5.2. Response Example	11
6. "message/bhttp" Media Type	13
7. Security Considerations	14
8. IANA Considerations	14
9. References	14
9.1. Normative References	14
9.2. Informative References	15
Acknowledgments	16
Authors' Addresses	16

1. Introduction

This document defines a simple format for representing an HTTP message ([HTTP]), either request or response. This allows for the encoding of HTTP messages that can be conveyed outside of an HTTP protocol. This enables the transformation of entire messages, including the application of authenticated encryption.

This format is informed by the framing structure of HTTP/2 ([H2]) and HTTP/3 ([H3]). In comparison, this format simpler by virtue of not including either header compression ([HPACK], [QPACK]) or a generic framing layer.

This format provides an alternative to the "message/http" content type defined in [MESSAGING]. A binary format permits more efficient encoding and processing of messages. A binary format also reduces exposure to security problems related to processing of HTTP messages.

Two modes for encoding are described:

- * a known-length encoding includes length prefixes for all major message components; and
- * an indefinite-length encoding enables efficient generation of messages where lengths are not known when encoding starts.

2. Conventions and Definitions

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.

This document uses terminology from HTTP ([HTTP]) and notation from QUIC (Section 1.3 of [QUIC]).

3. Format

Section 6 of [HTTP] defines five distinct parts to HTTP messages. A framing indicator is added to signal how these parts are composed:

1. Framing indicator. This format uses a single integer to describe framing, which describes whether the message is a request or response and how subsequent sections are formatted; see Section 3.3.
2. For a response, any number of interim responses, each consisting of an informational status code and header section.
3. Control data. For a request, this contains the request method and target. For a response, this contains the status code.
4. Header section. This contains zero or more header fields.
5. Content. This is a sequence of zero or more bytes.
6. Trailer section. This contains zero or more trailer fields.

All lengths and numeric values are encoded using the variable-length integer encoding from Section 16 of [QUIC].

3.1. Known Length Messages

A message that has a known length at the time of construction uses the format shown in Figure 1.

```
Message with Known-Length {
  Framing Indicator (i) = 0..1,
  Known-Length Informational Response (...) ...,
  Control Data (...),
  Known-Length Field Section (...),
  Known-Length Content (...),
  Known-Length Field Section (...),
}

Known-Length Field Section {
  Length (i) = 2..,
  Field Line (...) ...,
}

Known-Length Content {
  Content Length (i),
  Content (...)
}

Known-Length Informational Response {
  Informational Response Control Data (...),
  Known-Length Field Section (...),
}
```

Figure 1: Known-Length Message

That is, a known-length message consists of a framing indicator, a block of control data that is formatted according to the value of the framing indicator, a header section with a length prefix, binary content with a length prefix, and a trailer section with a length prefix.

Response messages that contain informational status codes result in a different structure; see Section 3.5.1.

Fields in the header and trailer sections consist of a length-prefixed name and length-prefixed value. Both name and value are sequences of bytes that cannot be zero length.

The format allows for the message to be truncated before any of the length prefixes that precede the field sections or content. This reduces the overall message size. A message that is truncated at any other point is invalid; see Section 4.

The variable-length integer encoding means that there is a limit of $2^{62}-1$ bytes for each field section and the message content.

3.2. Indeterminate Length Messages

A message that is constructed without encoding a known length for each section uses the format shown in Figure 2:

```
Indeterminate-Length Message {
  Framing Indicator (i) = 2..3,
  Indeterminate-Length Informational Response (...) ...,
  Control Data (...),
  Indeterminate-Length Field Section (...),
  Indeterminate-Length Content (...) ...,
  Indeterminate-Length Field Section (...),
}

Indeterminate-Length Content {
  Indeterminate-Length Content Chunk (...) ...,
  Content Terminator (i) = 0,
}

Indeterminate-Length Content Chunk {
  Chunk Length (i) = 1..,
  Chunk (...)
}

Indeterminate-Length Field Section {
  Field Line (...) ...,
  Content Terminator (i) = 0,
}

Indeterminate-Length Informational Response {
  Informational Response Control Data (...),
  Indeterminate-Length Field Section (...),
}
```

Figure 2: Indeterminate-Length Message

That is, an indeterminate length consists of a framing indicator, a block of control data that is formatted according to the value of the framing indicator, a header section that is terminated by a zero value, any number of non-zero-length chunks of binary content, a zero value, and a trailer section that is terminated by a zero value.

Response messages that contain informational status codes result in a different structure; see Section 3.5.1.

Indeterminate-length messages can be truncated in a similar way as known-length messages. Truncation occurs after the control data, or after the Content Terminator field that ends a field section or sequence of content chunks. A message that is truncated at any other point is invalid; see Section 4.

Indeterminate-length messages use the same encoding for field lines as known-length messages; see Section 3.6.

3.3. Framing Indicator

The start of each is a framing indicator that is a single integer that describes the structure of the subsequent sections. The framing indicator can take just four values:

- * A value of 0 describes a request of known length.
- * A value of 1 describes a response of known length.
- * A value of 2 describes a request of indeterminate length.
- * A value of 3 describes a response of indeterminate length.

Other values cause the message to be invalid; see Section 4.

3.4. Request Control Data

The control data for a request message includes four values that correspond to the values of the ":method", ":scheme", ":authority", and ":path" pseudo-header fields described in HTTP/2 (Section 8.3.1 of [H2]). These fields are encoded, each with a length prefix, in the order listed.

The rules in Section 8.3 of [H2] for constructing pseudo-header fields apply to the construction of these values. However, where the ":authority" pseudo-header field might be omitted in HTTP/2, a zero-length value is encoded instead.

The format of request control data is shown in Figure 3.

```
Request Control Data {  
    Method Length (i),  
    Method (...),  
    Scheme Length (i),  
    Scheme (...),  
    Authority Length (i),  
    Authority (...),  
    Path Length (i),  
    Path (...),  
}
```

Figure 3: Format of Request Control Data

3.5. Response Control Data

The control data for a request message includes a single field that corresponds to the ":status" pseudo-header field in HTTP/2; see Section 8.3.2 of [H2]. This field is encoded as a single variable length integer, not a decimal string.

The format of final response control data is shown in Figure 4.

```
Final Response Control Data {  
    Status Code (i) = 200..599,  
}
```

Figure 4: Format of Final Response Control Data

3.5.1. Informational Status Codes

Responses that include information status codes (see Section 15.2 of [HTTP]) are encoded by repeating the response control data and associated header section until the final status code is encoded.

The format of the informational response control data is shown in Figure 5.

```
Informational Response Control Data {  
    Status Code (i) = 100..199,  
}
```

Figure 5: Format of Informational Response Control Data

A response message can include any number of informational responses. If the response control data includes an informational status code (that is, a value between 100 and 199 inclusive), the control data is followed by a header section (encoded with known- or indeterminate-length according to the framing indicator). After the header section, another response control data block follows.

3.6. Header and Trailer Field Lines

Header and trailer sections consist of zero or more field lines; see Section 5 of [HTTP]. The format of a field section depends on whether the message is known- or intermediate-length.

Each field line includes a name and a value. Both the name and value are non-zero length sequences of bytes. The format of a field line is shown in Figure 6.

```
Field Line {  
    Name Length (i) = 1..  
    Name (...),  
    Value Length (i) = 1..  
    Value (...),  
}
```

Figure 6: Format of a Field Line

For field names, byte values that are not permitted in an HTTP field name cause the message to be invalid; see Section 5.1 of [HTTP] for a definition of what is valid and Section 4 for handling of invalid messages.

Field names and values MUST be constructed and validated according to the rules of Section 8.2.1 of [H2]. A recipient MUST treat a message that HTTP/2 regards as malformed by these rules as invalid; see Section 4.

The same field name can be repeated in multiple field lines; see Section 5.2 of [HTTP] for the semantics of repeated field names and rules for combining values.

Like HTTP/2, this format has an exception for the combination of multiple instances of the "Cookie" field. Instances of fields with the ASCII-encoded value of "cookie" are combined using a semicolon octet (0x3b) rather than a comma; see Section 8.2.3 of [H2].

This format provides fixed locations for content that would be carried in HTTP/2 pseudo-fields. Therefore, there is no need to include field lines containing a name of ":method", ":scheme",

":authority", ":path", or ":status". Fields that contain one of these names cause the message to be invalid; see Section 4. Pseudo-fields that are defined by protocol extensions MAY be included. Field lines containing pseudo-fields MUST precede other field lines; a message that contains a pseudo-field after any other field is invalid; see Section 4.

3.7. Content

The content of messages is a sequence of bytes of any length. Though a known-length message has a limit, this limit is large enough that it is unlikely to be a practical limitation. There is no limit to the size of content in an indeterminate length message.

Omitting content by truncating a message is only possible if the content is zero-length.

4. Invalid Messages

This document describes a number of ways that a message can be invalid. Invalid messages MUST NOT be processed except to log an error and produce an error response.

The format is designed to allow incremental processing. Implementations need to be aware of the possibility that an error might be detected after performing incremental processing.

5. Examples

This section includes example requests and responses encoded in both known-length and indefinite-length forms.

5.1. Request Example

The example HTTP/1.1 message in Figure 7 shows the content of a "message/http".

Valid HTTP/1.1 messages require lines terminated with CRLF (the two bytes 0x0a and 0x0d). For simplicity and consistency, the content of these examples is limited to text, which also uses CRLF for line endings.

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Figure 7: Sample HTTP Request

This can be expressed as a binary message (type "message/bhttp") using a known-length encoding as shown in hexadecimal in Figure 8. Figure 8 view includes some of the text alongside to show that most of the content is not modified.

```

00034745 54056874 74707300 0a2f6865 ..GET.https../he
6c6c6f2e 74787440 6c0a7573 65722d61 llo.txt@l.user-a
67656e74 34637572 6c2f372e 31362e33 gent4curl/7.16.3
206c6962 6375726c 2f372e31 362e3320 libcurl/7.16.3
4f70656e 53534c2f 302e392e 376c207a OpenSSL/0.9.7l z
6c69622f 312e322e 3304686f 73740f77 lib/1.2.3.host.w
77772e65 78616d70 6c652e63 6f6d0f61 ww.example.com.a
63636570 742d6c61 6e677561 67650665 ccept-language.e
6e2c206d 6900000 n, mi..

```

Figure 8: Known-Length Binary Encoding of Request

This example shows that the Host header field is not replicated in the :authority field, as is required for ensuring that the request is reproduced accurately; see Section 8.3.1 of [H2].

The same message can be truncated with no effect on interpretation. In this case, the last two bytes – corresponding to content and a trailer section – can each be removed without altering the semantics of the message.

The same message, encoded using an indefinite-length encoding is shown in Figure 9. As the content of this message is empty, the difference in formats is negligible.

```

02034745 54056874 74707300 0a2f6865 ..GET.https../he
6c6c6f2e 7478740a 75736572 2d616765 llo.txt.user-age
6e743463 75726c2f 372e3136 2e33206c nt4curl/7.16.3 l
69626375 726c2f37 2e31362e 33204f70 ibcurl/7.16.3 Op
656e5353 4c2f302e 392e376c 207a6c69 enSSL/0.9.7l zli
622f312e 322e3304 686f7374 0f777777 b/1.2.3.host.www
2e657861 6d706c65 2e636f6d 0f616363 .example.com.acc
6570742d 6c616e67 75616765 06656e2c ept-language.en,
206d6900 0000 mi...

```

Figure 9: Indefinite-Length Binary Encoding of Request

This indefinite-length encoding can be truncated by two bytes in the same way.

5.2. Response Example

Response messages can contain interim (1xx) status codes as the message in Figure 10 shows. Figure 10 includes examples of informational status codes defined in [RFC2518] and [RFC8297].

```
HTTP/1.1 102 Processing
Running: "sleep 15"
```

```
HTTP/1.1 103 Early Hints
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script
```

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My content includes a trailing CRLF.

Figure 10: Sample HTTP Response

As this is a longer example, only the indefinite-length encoding is shown in Figure 11. Note here that the specific text used in the reason phrase is not retained by this encoding.

```

03406607 72756e6e 696e670a 22736c65 .@f.running."sle
65702031 35220040 67046c69 6e6b233c ep 15".@g.link#<
2f737479 6c652e63 73733e3b 2072656c /style.css>; rel
3d707265 6c6f6164 3b206173 3d737479 =preload; as=sty
6c65046c 696e6b24 3c2f7363 72697074 le.link$</script
2e6a733e 3b207265 6c3d7072 656c6f61 .js>; rel=preloa
643b2061 733d7363 72697074 0040c804 d; as=script.@..
64617465 1d4d6f6e 2c203237 204a756c date.Mon, 27 Jul
20323030 39203132 3a32383a 35332047 2009 12:28:53 G
4d540673 65727665 72064170 61636865 MT.server.Apache
0d6c6173 742d6d6f 64696669 65641d57 .last-modified.W
65642c20 3232204a 756c2032 30303920 ed, 22 Jul 2009
31393a31 353a3536 20474d54 04657461 19:15:56 GMT.eta
67142233 34616133 38372d64 2d313536 g."34aa387-d-156
38656230 30220d61 63636570 742d7261 8eb00".accept-ra
6e676573 05627974 65730e63 6f6e7465 nges.bytes.conte
6e742d6c 656e6774 68023531 04766172 nt-length.51.var
790f4163 63657074 2d456e63 6f64696e y.Accept-Encodin
670c636f 6e74656e 742d7479 70650a74 g.content-type.t
6578742f 706c6169 6e003348 656c6c6f ext/plain.3Hello
20576f72 6c642120 4d792063 6f6e7465 World! My conte
6e742069 6e636c75 64657320 61207472 nt includes a tr
61696c69 6e672043 524c462e 0d0a0000 ailing CRLF.....

```

Figure 11: Binary Response including Interim Responses

A response that uses the chunked encoding (see Section 7.1 of [MESSAGING]) as shown for Figure 12 can be encoded using indefinite-length encoding, which minimizes buffering needed to translate into the binary format. However, chunk boundaries do not need to be retained and any chunk extensions cannot be conveyed using the binary format.

```

HTTP/1.1 200 OK
Transfer-Encoding: chunked

4
This
6
  conte
13;chunk-extension=foo
nt contains CRLF.

0
Trailer: text

```

Figure 12: Chunked Encoding Example

Figure 13 shows this message using the known-length coding. Note that the transfer-encoding header field is removed.

```
0140c800 1d546869 7320636f 6e74656e  .@...This conten
7420636f 6e746169 6e732043 524c462e  t contains CRLF.
0d0a0d07 74726169 6c657204 74657874  ....trailer.text
```

Figure 13: Known-Length Encoding of Response

6. "message/bhttp" Media Type

The message/http media type can be used to enclose a single HTTP request or response message, provided that it obeys the MIME restrictions for all "message" types regarding line length and encodings.

Type name: message

Subtype name: bhttp

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 7

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

7. Security Considerations

Many of the considerations that apply to HTTP message handling apply to this format; see Section 17 of [HTTP] and Section 11 of [MESSAGING] for common issues in handling HTTP messages.

Strict parsing of the format with no tolerance for errors can help avoid a number of attacks. However, implementations still need to be aware of the possibility of resource exhaustion attacks that might arise from receiving large messages, particularly those with large numbers of fields.

The format is designed to allow for minimal state when translating for use with HTTP proper. However, producing a combined value for fields, which might be necessary for the "Cookie" field when translating this format (like HTTP/1.1 [MESSAGING]), can require the commitment of resources. Implementations need to ensure that they aren't subject to resource exhaustion attack from a maliciously crafted message.

8. IANA Considerations

Please add the "Media Types" registry at <https://www.iana.org/assignments/media-types> (<https://www.iana.org/assignments/media-types>) with the registration information in Section 6 for the media type "message/bhttp".

9. References

9.1. Normative References

- [H2] Thomson, M. and C. Benfield, "Hypertext Transfer Protocol Version 2 (HTTP/2)", Work in Progress, Internet-Draft, draft-ietf-httpbis-http2bis-03, 12 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-http2bis-03>>.
- [HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-17, 25 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-17>>.

[MESSAGING]

Fielding, R. T., Nottingham, M., and J. Reschke,
"HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-17, 25 July 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-messaging-17>>.

[QUIC]

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>>.

[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>>.

[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>>.

9.2. Informative References

[H3]

Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.

[HPACK]

Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015,
<<https://www.rfc-editor.org/rfc/rfc7541>>.

[QPACK]

Krasic, C. '., Bishop, M., and A. Frindell, "QPACK: Header Compression for HTTP/3", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-21, 2 February 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qpack-21>>.

[RFC2518]

Goland, Y., Whitehead, E., Faizi, A., Carter, S., and D. Jensen, "HTTP Extensions for Distributed Authoring -- WEBDAV", RFC 2518, DOI 10.17487/RFC2518, February 1999,
<<https://www.rfc-editor.org/rfc/rfc2518>>.

[RFC8297]

Oku, K., "An HTTP Status Code for Indicating Hints", RFC 8297, DOI 10.17487/RFC8297, December 2017,
<<https://www.rfc-editor.org/rfc/rfc8297>>.

Acknowledgments

TODO: credit where credit is due.

Authors' Addresses

Martin Thomson
Mozilla

Email: mt@lowentropy.net

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

HTTPBIS
Internet-Draft
Intended status: Standards Track
Expires: 25 February 2022

M. Thomson
Mozilla
C.A. Wood
Cloudflare
24 August 2021

Oblivious HTTP
draft-thomson-http-oblivious-02

Abstract

This document describes a system for the forwarding of encrypted HTTP messages. This allows a client to make multiple requests of a server without the server being able to link those requests to the client or to identify the requests as having come from the same client.

Discussion Venues

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

Discussion of this document takes place on the HTTP Working Group mailing list (http@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/http/>.

Source for this draft and an issue tracker can be found at <https://github.com/unicorn-wg/oblivious-http>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 25 February 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
3. Overview	4
3.1. Applicability	6
4. Key Configuration	7
4.1. Key Configuration Encoding	8
4.2. Key Configuration Media Type	8
5. HPKE Encapsulation	9
5.1. Encapsulation of Requests	10
5.2. Encapsulation of Responses	12
6. HTTP Usage	13
6.1. Informational Responses	14
6.2. Errors	14
7. Media Types	15
7.1. message/ohhttp-req Media Type	15
7.2. message/ohhttp-res Media Type	16
8. Security Considerations	17
8.1. Client Responsibilities	18
8.2. Proxy Responsibilities	19
8.2.1. Denial of Service	20
8.2.2. Linkability Through Traffic Analysis	20
8.3. Server Responsibilities	21
8.4. Replay Attacks	21
8.5. Post-Compromise Security	23
9. Privacy Considerations	23
10. Operational and Deployment Considerations	23
11. IANA Considerations	24
12. References	24
12.1. Normative References	24
12.2. Informative References	25
Appendix A. Complete Example of a Request and Response	27
Acknowledgments	29

Authors' Addresses	29
--------------------	----

1. Introduction

The act of making a request using HTTP reveals information about the client identity to a server. Though the content of requests might reveal information, that is information under the control of the client. In comparison, the source address on the connection reveals information that a client has only limited control over.

Even where an IP address is not directly attributed to an individual, the use of an address over time can be used to correlate requests. Servers are able to use this information to assemble profiles of client behavior, from which they can make inferences about the people involved. The use of persistent connections to make multiple requests improves performance, but provides servers with additional certainty about the identity of clients in a similar fashion.

Use of an HTTP proxy can provide a degree of protection against servers correlating requests. Systems like virtual private networks or the Tor network [Dingledine2004], provide other options for clients.

Though the overhead imposed by these methods varies, the cost for each request is significant. Preventing request linkability requires that each request use a completely new TLS connection to the server. At a minimum, this requires an additional round trip to the server in addition to that required by the request. In addition to having high latency, there are significant secondary costs, both in terms of the number of additional bytes exchanged and the CPU cost of cryptographic computations.

This document describes a method of encapsulation for binary HTTP messages [BINARY] using Hybrid Public Key Encryption (HPKE; [HPKE]). This protects the content of both requests and responses and enables a deployment architecture that can separate the identity of a requester from the request.

Though this scheme requires that servers and proxies explicitly support it, this design represents a performance improvement over options that perform just one request in each connection. With limited trust placed in the proxy (see Section 8), clients are assured that requests are not uniquely attributed to them or linked to other requests.

2. Conventions and Definitions

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.

Encapsulated Request: An HTTP request that is encapsulated in an HPKE-encrypted message; see Section 5.1.

Encapsulated Response: An HTTP response that is encapsulated in an HPKE-encrypted message; see Section 5.2.

Oblivious Proxy Resource: An intermediary that forwards requests and responses between clients and a single oblivious request resource.

Oblivious Request Resource: A resource that can receive an encapsulated request, extract the contents of that request, forward it to an oblivious target resource, receive a response, encapsulate that response, then return that response.

Oblivious Target Resource: The resource that is the target of an encapsulated request. This resource logically handles only regular HTTP requests and responses and so might be ignorant of the use of oblivious HTTP to reach it.

This draft includes pseudocode that uses the functions and conventions defined in [HPKE].

Encoding an integer to a sequence of bytes in network byte order is described using the function "encode(n, v)", where "n" is the number of bytes and "v" is the integer value. The function "len()" returns the length of a sequence of bytes.

Formats are described using notation from Section 1.3 of [QUIC].

3. Overview

A client learns the following:

- * The identity of an oblivious request resource. This might include some information about oblivious target resources that the oblivious request resource supports.
- * The details of an HPKE public key that the oblivious request resource accepts, including an identifier for that key and the HPKE algorithms that are used with that key.

- * The identity of an oblivious proxy resource that will forward encapsulated requests and responses to the oblivious request resource.

This information allows the client to make a request of an oblivious target resource without that resource having only a limited ability to correlate that request with the client IP or other requests that the client might make to that server.

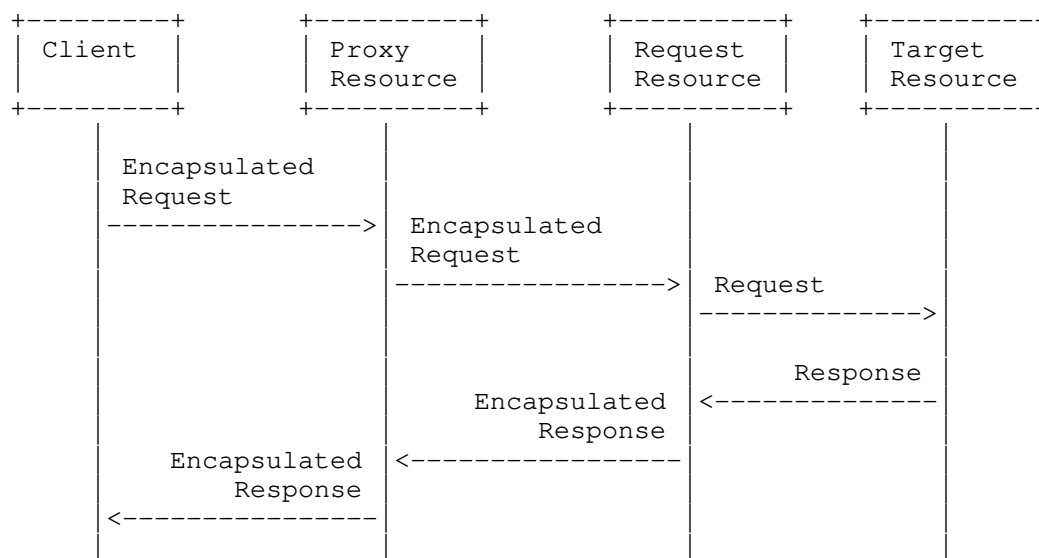


Figure 1: Overview of Oblivious HTTP

In order to make a request to an oblivious target resource, the following steps occur, as shown in Figure 1:

1. The client constructs an HTTP request for an oblivious target resource.
2. The client encodes the HTTP request in a binary HTTP message and then encapsulates that message using HPKE and the process from Section 5.1.
3. The client sends a POST request to the oblivious proxy resource with the encapsulated request as the content of that message.
4. The oblivious proxy resource forwards this request to the oblivious request resource.

5. The oblivious request resource receives this request and removes the HPKE protection to obtain an HTTP request.
6. The oblivious request resource makes an HTTP request that includes the target URI, method, fields, and content of the request it acquires.
7. The oblivious target resource answers this HTTP request with an HTTP response.
8. The oblivious request resource encapsulates the HTTP response following the process in Section 5.2 and sends this in response to the request from the oblivious proxy resource.
9. The oblivious proxy resource forwards this response to the client.
10. The client removes the encapsulation to obtain the response to the original request.

3.1. Applicability

Oblivious HTTP has limited applicability. Many uses of HTTP benefit from being able to carry state between requests, such as with cookies ([RFC6265]), authentication (Section 11 of [HTTP]), or even alternative services ([RFC7838]). Oblivious HTTP seeks to prevent this sort of linkage, which requires that applications not carry state between requests.

Oblivious HTTP is primarily useful where privacy risks associated with possible stateful treatment of requests are sufficiently negative that the cost of deploying this protocol can be justified. Oblivious HTTP is simpler and less costly than more robust systems, like Prio ([PRIO]) or Tor ([Dingledine2004]), which can provide stronger guarantees at higher operational costs.

Oblivious HTTP is more costly than a direct connection to a server. Some costs, like those involved with connection setup, can be amortized, but there are several ways in which oblivious HTTP is more expensive than a direct request:

- * Each oblivious request requires at least two regular HTTP requests, which adds latency.
- * Each request is expanded in size with additional HTTP fields, encryption-related metadata, and AEAD expansion.

- * Deriving cryptographic keys and applying them for request and response protection takes non-negligible computational resources.

Examples of where preventing the linking of requests might justify these costs include:

- * DNS queries. DNS queries made to a recursive resolver reveal information about the requester, particularly if linked to other queries.
- * Telemetry submission. Applications that submit reports about their usage to their developers might use oblivious HTTP for some types of moderately sensitive data.

4. Key Configuration

A client needs to acquire information about the key configuration of the oblivious request resource in order to send encapsulated requests.

In order to ensure that clients do not encapsulate messages that other entities can intercept, the key configuration **MUST** be authenticated and have integrity protection.

This document describes the "application/ohttp-keys" media type; see Section 4.2. This media type might be used, for example with HTTPS, as part of a system for configuring or discovering key configurations. Note however that such a system needs to consider the potential for key configuration to be used to compromise client privacy; see Section 9.

Specifying a format for expressing the information a client needs to construct an encapsulated request ensures that different client implementations can be configured in the same way. This also enables advertising key configurations in a consistent format.

A client might have multiple key configurations to select from when encapsulating a request. Clients are responsible for selecting a preferred key configuration from those it supports. Clients need to consider both the key encapsulation method (KEM) and the combinations of key derivation function (KDF) and authenticated encryption with associated data (AEAD) in this decision.

Evolution of the key configuration format is supported through the definition of new formats that are identified by new media types.

4.1. Key Configuration Encoding

A single key configuration consists of a key identifier, a public key, an identifier for the KEM that the public key uses, and a set of HPKE symmetric algorithms. Each symmetric algorithm consists of an identifier for a KDF and an identifier for an AEAD.

Figure 2 shows a single key configuration, `KeyConfig`, that is expressed using the TLS syntax; see Section 3 of [TLS].

```
opaque HpkePublicKey[Npk];
uint16 HpkeKemId;
uint16 HpkeKdfId;
uint16 HpkeAeadId;

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricAlgorithms;

struct {
    uint8 key_id;
    HpkeKemId kem_id;
    HpkePublicKey public_key;
    HpkeSymmetricAlgorithms cipher_suites<4..2^16-4>;
} KeyConfig;
```

Figure 2: A Single Key Configuration

The types `HpkeKemId`, `HpkeKdfId`, and `HpkeAeadId` identify a KEM, KDF, and AEAD respectively. The definitions for these identifiers and the semantics of the algorithms they identify can be found in [HPKE]. The `Npk` parameter corresponding to the `HpkeKdfId` can be found in [HPKE].

4.2. Key Configuration Media Type

The "application/ohttp-keys" format is a media type that identifies a serialized collection of key configurations. The content of this media type comprises one or more key configuration encodings (see Section 4.1) that are concatenated.

Type name: application

Subtype name: ohttp-keys

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 8

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

5. HPKE Encapsulation

HTTP message encapsulation uses HPKE for request and response encryption. An encapsulated HTTP message includes the following values:

1. A binary-encoded HTTP message; see [BINARY].
2. Padding of arbitrary length which MUST contain all zeroes.

The encoding of an HTTP message is as follows:

```
Plaintext Message {  
    Message Length (i),  
    Message (...),  
    Padding Length (i),  
    Padding (...),  
}
```

An Encapsulated Request is comprised of a length-prefixed key identifier and a HPKE-protected request message. HPKE protection includes an encapsulated KEM shared secret (or "enc"), plus the AEAD-protected request message. An Encapsulated Request is shown in Figure 3. Section 5.1 describes the process for constructing and processing an Encapsulated Request.

```
Encapsulated Request {  
    Key Identifier (8),  
    KEM Identifier (16),  
    KDF Identifier (16),  
    AEAD Identifier (16),  
    Encapsulated KEM Shared Secret (8*Nenc),  
    AEAD-Protected Request (...),  
}
```

Figure 3: Encapsulated Request

The Nenc parameter corresponding to the HpkeKdfId can be found in [HPKE].

Responses are bound to responses and so consist only of AEAD-protected content. Section 5.2 describes the process for constructing and processing an Encapsulated Response.

```
Encapsulated Response {  
    Nonce (Nk),  
    AEAD-Protected Response (...),  
}
```

Figure 4: Encapsulated Response

The size of the Nonce field in an Encapsulated Response corresponds to the size of an AEAD key for the corresponding HPKE ciphersuite.

5.1. Encapsulation of Requests

Clients encapsulate a request "request" using values from a key configuration:

- * the key identifier from the configuration, "keyID", with the corresponding KEM identified by "kemID",
- * the public key from the configuration, "pkR", and
- * a selected combination of KDF, identified by "kdfID", and AEAD, identified by "aeadID".

The client then constructs an encapsulated request, "enc_request", as follows:

1. Compute an HPKE context using "pkR", yielding "context" and encapsulation key "enc".
2. Construct associated data, "aad", by concatenating the values of "keyID", "kemID", "kdfID", and "aeadID", as one 8-bit integer and three 16-bit integers, respectively, each in network byte order.
3. Encrypt (seal) "request" with "aad" as associated data using "context", yielding ciphertext "ct".
4. Concatenate the values of "aad", "enc", and "ct", yielding an Encapsulated Request "enc_request".

Note that "enc" is of fixed-length, so there is no ambiguity in parsing this structure.

In pseudocode, this procedure is as follows:

```
enc, context = SetupBaseS(pkR, "request")
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID))
ct = context.Seal(aad, request)
enc_request = concat(aad, enc, ct)
```

Servers decrypt an Encapsulated Request by reversing this process. Given an Encapsulated Request "enc_request", a server:

1. Parses "enc_request" into "keyID", "kemID", "kdfID", "aeadID", "enc", and "ct" (indicated using the function "parse()" in pseudocode). The server is then able to find the HPKE private key, "skR", corresponding to "keyID".
 - a. If "keyID" does not identify a key matching the type of "kemID", the server returns an error.

- b. If "kdfID" and "aeadID" identify a combination of KDF and AEAD that the server is unwilling to use with "skR", the server returns an error.
2. Compute an HPKE context using "skR" and the encapsulated key "enc", yielding "context".
3. Construct additional associated data, "aad", from "keyID", "kdfID", and "aeadID" or as the first five bytes of "enc_request".
4. Decrypt "ct" using "aad" as associated data, yielding "request" or an error on failure. If decryption fails, the server returns an error.

In pseudocode, this procedure is as follows:

```
keyID, kemID, kdfID, aeadID, enc, ct = parse(enc_request)
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID))
context = SetupBaseR(enc, skR, "request")
request, error = context.Open(aad, ct)
```

5.2. Encapsulation of Responses

Given an HPKE context "context", a request message "request", and a response "response", servers generate an Encapsulated Response "enc_response" as follows:

1. Export a secret "secret" from "context", using the string "response" as context. The length of this secret is "max(Nn, Nk)", where "Nn" and "Nk" are the length of AEAD key and nonce associated with "context".
2. Generate a random value of length "max(Nn, Nk)" bytes, called "response_nonce".
3. Extract a pseudorandom key "prk" using the "Extract" function provided by the KDF algorithm associated with "context". The "ikm" input to this function is "secret"; the "salt" input is the concatenation of "enc" (from "enc_request") and "response_nonce".
4. Use the "Expand" function provided by the same KDF to extract an AEAD key "key", of length "Nk" - the length of the keys used by the AEAD associated with "context". Generating "key" uses a label of "key".

5. Use the same "Expand" function to extract a nonce "nonce" of length "Nn" – the length of the nonce used by the AEAD. Generating "nonce" uses a label of "nonce".
6. Encrypt "response", passing the AEAD function Seal the values of "key", "nonce", empty "aad", and a "pt" input of "request", which yields "ct".
7. Concatenate "response_nonce" and "ct", yielding an Encapsulated Response "enc_response". Note that "response_nonce" is of fixed-length, so there is no ambiguity in parsing either "response_nonce" or "ct".

In pseudocode, this procedure is as follows:

```
secret = context.Export("response", Nk)
response_nonce = random(max(Nn, Nk))
salt = concat(enc, response_nonce)
prk = Extract(salt, secret)
aead_key = Expand(prk, "key", Nk)
aead_nonce = Expand(prk, "nonce", Nn)
ct = Seal(aead_key, aead_nonce, "", response)
enc_response = concat(response_nonce, ct)
```

Clients decrypt an Encapsulated Request by reversing this process. That is, they first parse "enc_response" into "response_nonce" and "ct". They then follow the same process to derive values for "aead_key" and "aead_nonce".

The client uses these values to decrypt "ct" using the Open function provided by the AEAD. Decrypting might produce an error, as follows:

```
reponse, error = Open(aead_key, aead_nonce, "", ct)
```

6. HTTP Usage

A client interacts with the oblivious proxy resource by constructing an encapsulated request. This encapsulated request is included as the content of a POST request to the oblivious proxy resource. This request MUST only contain those fields necessary to carry the encapsulated request: a method of POST, a target URI of the oblivious proxy resource, a header field containing the content type (see (Section 7)), and the encapsulated request as the request content. Clients MAY include fields that do not reveal information about the content of the request, such as Alt-Used [ALT-SVC], or information that it trusts the oblivious proxy resource to remove, such as fields that are listed in the Connection header field.

The oblivious proxy resource interacts with the oblivious request resource by constructing a request using the same restrictions as the client request, except that the target URI is the oblivious request resource. The content of this request is copied from the client. The oblivious proxy resource **MUST NOT** add information about the client to this request.

When a response is received from the oblivious request resource, the oblivious proxy resource forwards the response according to the rules of an HTTP proxy; see Section 7.6 of [HTTP].

An oblivious request resource, if it receives any response from the oblivious target resource, sends a single 200 response containing the encapsulated response. Like the request from the client, this response **MUST** only contain those fields necessary to carry the encapsulated response: a 200 status code, a header field indicating the content type, and the encapsulated response as the response content. As with requests, additional fields **MAY** be used to convey information that does not reveal information about the encapsulated response.

An oblivious request resource acts as a gateway for requests to the oblivious target resource (see Section 7.6 of [HTTP]). The one exception is that any information it might forward in a response **MUST** be encapsulated, unless it is responding to errors it detects before removing encapsulation of the request; see Section 6.2.

6.1. Informational Responses

This encapsulation does not permit progressive processing of responses. Though the binary HTTP response format does support the inclusion of informational (1xx) status codes, the AEAD encapsulation cannot be removed until the entire message is received.

In particular, the Expect header field with 100-continue (see Section 10.1.1 of [HTTP]) cannot be used. Clients **MUST NOT** construct a request that includes a 100-continue expectation; the oblivious request resource **MUST** generate an error if a 100-continue expectation is received.

6.2. Errors

A server that receives an invalid message for any reason **MUST** generate an HTTP response with a 4xx status code.

Errors detected by the oblivious proxy resource and errors detected by the oblivious request resource before removing protection (including being unable to remove encapsulation for any reason) result in the status code being sent without protection in response to the POST request made to that resource.

Errors detected by the oblivious request resource after successfully removing encapsulation and errors detected by the oblivious target resource MUST be sent in an encapsulated response.

7. Media Types

Media types are used to identify encapsulated requests and responses.

Evolution of the format of encapsulated requests and responses is supported through the definition of new formats that are identified by new media types.

7.1. message/ohttp-req Media Type

The "message/ohttp-req" identifies an encapsulated binary HTTP request. This is a binary format that is defined in Section 5.1.

Type name: message

Subtype name: ohttp-req

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 8

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

7.2. message/ohhttp-res Media Type

The "message/ohhttp-res" identifies an encapsulated binary HTTP response. This is a binary format that is defined in Section 5.2.

Type name: message

Subtype name: ohhttp-res

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 8

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8. Security Considerations

In this design, a client wishes to make a request of a server that is authoritative for the oblivious target resource. The client wishes to make this request without linking that request with either:

1. The identity at the network and transport layer of the client (that is, the client IP address and TCP or UDP port number the client uses to create a connection).
2. Any other request the client might have made in the past or might make in the future.

In order to ensure this, the client selects a proxy (that serves the oblivious proxy resource) that it trusts will protect this information by forwarding the encapsulated request and response without passing the server (that serves the oblivious request resource).

In this section, a deployment where there are three entities is considered:

- * A client makes requests and receives responses
- * A proxy operates the oblivious proxy resource
- * A server operates both the oblivious request resource and the oblivious target resource

To achieve the stated privacy goals, the oblivious proxy resource cannot be operated by the same entity as the oblivious request resource. However, colocation of the oblivious request resource and oblivious target resource simplifies the interactions between those resources without affecting client privacy.

As a consequence of this configuration, Oblivious HTTP prevents linkability described above. Informally, this means:

1. Requests and responses are known only to clients and targets in possession of the corresponding response encapsulation key and HPKE keying material. In particular, the oblivious proxy knows the origin and destination of an encapsulated request and response, yet does not know the decapsulated contents. Likewise, targets know only the oblivious request origin, i.e., the proxy, and the decapsulated request. Only the client knows both the plaintext request and response.
2. Targets cannot link requests from the same client in the absence of unique per-client keys.

Traffic analysis that might affect these properties are outside the scope of this document; see Section 8.2.2.

A formal analysis of Oblivious HTTP is in [OHTTP-ANALYSIS].

8.1. Client Responsibilities

Clients MUST ensure that the key configuration they select for generating encapsulated requests is integrity protected and authenticated so that it can be attributed to the oblivious request resource; see Section 4.

Since clients connect directly to the proxy instead of the target, application configurations wherein clients make policy decisions about target connections, e.g., to apply certificate pinning, are incompatible with Oblivious HTTP. In such cases, alternative technologies such as HTTP CONNECT (Section 9.3.6 of [HTTP]) can be used. Applications could implement related policies on key configurations and proxy connections, though these might not provide the same properties as policies enforced directly on target connections. When this difference is relevant, applications can instead connect directly to the target at the cost of either privacy or performance.

Clients MUST NOT include identifying information in the request that is encapsulated. Identifying information includes cookies [COOKIES], authentication credentials or tokens, and any information that might reveal client-specific information such as account credentials.

Clients cannot carry connection-level state between requests as they only establish direct connections to the proxy responsible for the oblivious proxy resource. However, clients need to ensure that they construct requests without any information gained from previous requests. Otherwise, the server might be able to use that information to link requests. Cookies [COOKIES] are the most obvious feature that **MUST NOT** be used by clients. However, clients need to include all information learned from requests, which could include the identity of resources.

Clients **MUST** generate a new HPKE context for every request, using a good source of entropy ([RANDOM]) for generating keys. Key reuse not only risks requests being linked, reuse could expose request and response contents to the proxy.

The request the client sends to the oblivious proxy resource only requires minimal information; see Section 6. The request that carries the encapsulated request and is sent to the oblivious proxy resource **MUST NOT** include identifying information unless the client ensures that this information is removed by the proxy. A client **MAY** include information only for the oblivious proxy resource in header fields identified by the Connection header field if it trusts the proxy to remove these as required by Section 7.6.1 of [HTTP]. The client needs to trust that the proxy does not replicate the source addressing information in the request it forwards.

Clients rely on the oblivious proxy resource to forward encapsulated requests and responses. However, the proxy can only refuse to forward messages, it cannot inspect or modify the contents of encapsulated requests or responses.

8.2. Proxy Responsibilities

The proxy that serves the oblivious proxy resource has a very simple function to perform. For each request it receives, it makes a request of the oblivious request resource that includes the same content. When it receives a response, it sends a response to the client that includes the content of the response from the oblivious request resource. When generating a request, the proxy **MUST** follow the forwarding rules in Section 7.6 of [HTTP].

A proxy can also generate responses, though it assumed to not be able to examine the content of a request (other than to observe the choice of key identifier, KDF, and AEAD), so it is also assumed that it cannot generate an encapsulated response.

A proxy MUST NOT add information about the client identity when forwarding requests. This includes the Via field, the Forwarded field [FORWARDED], and any similar information. A client does not depend on the proxy using an authenticated and encrypted connection to the oblivious request resource, only that information about the client not be attached to forwarded requests.

8.2.1. Denial of Service

As there are privacy benefits from having a large rate of requests forwarded by the same proxy (see Section 8.2.2), servers that operate the oblivious request resource might need an arrangement with proxies. This arrangement might be necessary to prevent having the large volume of requests being classified as an attack by the server.

If a server accepts a larger volume of requests from a proxy, it needs to trust that the proxy does not allow abusive levels of request volumes from clients. That is, if a server allows requests from the proxy to be exempt from rate limits, the server might want to ensure that the proxy applies a rate limiting policy that is acceptable to the server.

Servers that enter into an agreement with a proxy that enables a higher request rate might choose to authenticate the proxy to enable the higher rate.

8.2.2. Linkability Through Traffic Analysis

As the time at which encapsulated request or response messages are sent can reveal information to a network observer. Though messages exchanged between the oblivious proxy resource and the oblivious request resource might be sent in a single connection, traffic analysis could be used to match messages that are forwarded by the proxy.

A proxy could, as part of its function, add delays in order to increase the anonymity set into which each message is attributed. This could latency to the overall time clients take to receive a response, which might not be what some clients want.

A proxy can use padding to reduce the effectiveness of traffic analysis.

A proxy that forwards large volumes of exchanges can provide better privacy by providing larger sets of messages that need to be matched.

8.3. Server Responsibilities

A server that operates both oblivious request and oblivious target resources is responsible for removing request encapsulation, generating a response the encapsulated request, and encapsulating the response.

Servers should account for traffic analysis based on response size or generation time. Techniques such as padding or timing delays can help protect against such attacks; see Section 8.2.2.

If separate entities provide the oblivious request resource and oblivious target resource, these entities might need an arrangement similar to that between server and proxy for managing denial of service; see Section 8.2.1. It is also necessary to provide confidentiality protection for the unprotected requests and responses, plus protections for traffic analysis; see Section 8.2.2.

An oblivious request resource needs to have a plan for replacing keys. This might include regular replacement of keys, which can be assigned new key identifiers. If an oblivious request resource receives a request that contains a key identifier that it does not understand or that corresponds to a key that has been replaced, the server can respond with an HTTP 422 (Unprocessable Content) status code.

A server can also use a 422 status code if the server has a key that corresponds to the key identifier, but the encapsulated request cannot be successfully decrypted using the key.

8.4. Replay Attacks

Encapsulated requests can be copied and replayed by the oblivious proxy resource. The design of oblivious HTTP does not assume that the oblivious proxy resource will not replay requests. In addition, if a client sends an encapsulated request in TLS early data (see Section 8 of [TLS] and [RFC8470]), a network-based adversary might be able to cause the request to be replayed. In both cases, the effect of a replay attack and the mitigations that might be employed are similar to TLS early data.

A client or oblivious proxy resource MUST NOT automatically attempt to retry a failed request unless it receives a positive signal indicating that the request was not processed or forwarded. The HTTP/2 REFUSED_STREAM error code (Section 8.1.4 of [RFC7540]), the HTTP/3 H3_REQUEST_REJECTED error code (Section 8.1 of [QUIC-HTTP]), or a GOAWAY frame with a low enough identifier (in either protocol version) are all sufficient signals that no processing occurred. Connection failures or interruptions are not sufficient signals that no processing occurred.

The anti-replay mechanisms described in Section 8 of [TLS] are generally applicable to oblivious HTTP requests. Servers can use the encapsulated keying material as a unique key for identifying potential replays. This depends on clients generating a new HPKE context for every request.

The mechanism used in TLS for managing differences in client and server clocks cannot be used as it depends on being able to observe previous interactions. Oblivious HTTP explicitly prevents such linkability. Applications can still include an explicit indication of time to limit the span of time over which a server might need to track accepted requests. Clock information could be used for client identification, so reduction in precision or obfuscation might be necessary.

The considerations in [RFC8470] as they relate to managing the risk of replay also apply, though there is no option to delay the processing of a request.

Limiting requests to those with safe methods might not be satisfactory for some applications, particularly those that involve the submission of data to a server. The use of idempotent methods might be of some use in managing replay risk, though it is important to recognize that different idempotent requests can be combined to be not idempotent.

Idempotent actions with a narrow scope based on the value of a protected nonce could enable data submission with limited replay exposure. A nonce might be added as an explicit part of a request, or, if the oblivious request and target resources are co-located, the encapsulated keying material can be used to produce a nonce.

The server-chosen "response_nonce" field ensures that responses have unique AEAD keys and nonces even when requests are replayed.

8.5. Post-Compromise Security

This design does not provide post-compromise security for responses. A client only needs to retain keying material that might be used compromise the confidentiality and integrity of a response until that response is consumed, so there is negligible risk associated with a client compromise.

A server retains a secret key that might be used to remove protection from messages over much longer periods. A server compromise that provided access to the oblivious request resource secret key could allow an attacker to recover the plaintext of all requests sent toward affected keys and all of the responses that were generated. Accessing requests and responses also requires access to requests and responses, which implies either compromise of TLS connections or collusion with the oblivious proxy resource.

The total number of affected messages affected by server key compromise can be limited by regular rotation of server keys.

9. Privacy Considerations

One goal of this design is that independent client requests are only linkable by the chosen key configuration. The oblivious proxy and request resources can link requests using the same key configuration by matching `KeyConfig.key_id`, or, if the oblivious target resource is willing to use trial decryption, a limited set of key configurations that share an identifier. An oblivious proxy can link requests using the public key corresponding to `KeyConfig.key_id`.

Request resources are capable of linking requests depending on how `KeyConfigs` are produced by servers and discovered by clients. Specifically, servers can maliciously construct key configurations to track individual clients. A specific method for a client to acquire key configurations is not included in this specification. Clients need to consider these tracking vectors when choosing a discovery method. Applications using this design should provide accommodations to mitigate tracking use key configurations.

10. Operational and Deployment Considerations

Using Oblivious HTTP adds both cryptographic and latency to requests relative to a simple HTTP request-response exchange. Deploying proxy services that are on path between clients and servers avoids adding significant additional delay due to network topology. A study of a similar system [ODoH] found that deploying proxies close to servers was most effective in minimizing additional latency.

Oblivious HTTP might be incompatible with network interception regimes, such as those that rely on configuring clients with trust anchors and intercepting TLS connections. While TLS might be intercepted successfully, interception middleboxes devices might not receive updates that would allow Oblivious HTTP to be correctly identified using the media types defined in Section 7.

Oblivious HTTP has a simple key management design that is not trivially altered to enable interception by intermediaries. Clients that are configured to enable interception might choose to disable Oblivious HTTP in order to ensure that content is accessible to middleboxes.

11. IANA Considerations

Please update the "Media Types" registry at <https://www.iana.org/assignments/media-types> (<https://www.iana.org/assignments/media-types>) with the registration information in Section 7 for the media types "message/ohttp-req", "message/ohttp-res", and "application/ohttp-keys".

12. References

12.1. Normative References

- [BINARY] Thomson, M., "Binary Representation of HTTP Messages", Work in Progress, Internet-Draft, draft-thomson-http-binary-message-00, 24 August 2021, <<https://datatracker.ietf.org/doc/html/draft-thomson-http-binary-message-00>>.
- [HPKE] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-11, 2 August 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-11>>.
- [HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-18, 18 August 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-18>>.
- [QUIC] 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>>.

[QUIC-HTTP]

Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.

[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>>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/rfc/rfc7540>>.

[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>>.

[RFC8470] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/rfc/rfc8470>>.

[TLS] 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>>.

12.2. Informative References

[ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.

[COOKIES] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.

[Dingledine2004]

Dingledine, R., Mathewson, N., and P. Syverson, "Tor: The Second-Generation Onion Router", August 2004, <<https://svn.torproject.org/svn/projects/design-paper/tor-design.html>>.

[FORWARDED]

Petersson, A. and M. Nilsson, "Forwarded HTTP Extension", RFC 7239, DOI 10.17487/RFC7239, June 2014, <<https://www.rfc-editor.org/rfc/rfc7239>>.

[ODOH]

Singanamalla, S., Chunhapanya, S., Vavrusa, M., Verma, T., Wu, P., Fayed, M., Heimerl, K., Sullivan, N., and C. A. Wood, "Oblivious DNS over HTTPS (ODOH): A Practical Privacy Enhancement to DNS", 7 January 2021, <<https://www.petsymposium.org/2021/files/papers/issue4/popets-2021-0085.pdf>>.

[ODOH]

Kinnear, E., McManus, P., Pauly, T., Verma, T., and C. A. Wood, "Oblivious DNS Over HTTPS", Work in Progress, Internet-Draft, draft-pauly-dprive-oblivious-doh-06, 8 March 2021, <<https://datatracker.ietf.org/doc/html/draft-pauly-dprive-oblivious-doh-06>>.

[OHTTP-ANALYSIS]

Hoyland, J., "Tamarin Model of Oblivious HTTP", 23 August 2021, <<https://github.com/cloudflare/ohttp-analysis>>.

[PRIO]

Corrigan-Gibbs, H. and D. Boneh, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics", 14 March 2017, <<https://crypto.stanford.edu/prio/paper.pdf>>.

[RANDOM]

Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/rfc/rfc4086>>.

[RFC6265]

Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.

[RFC7838]

Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.

[X25519]

Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/rfc/rfc7748>>.

Appendix A. Complete Example of a Request and Response

A single request and response exchange is shown here. Binary values (key configuration, secret keys, the content of messages, and intermediate values) are shown in hexadecimal. The request and response here are absolutely minimal; the purpose of this example is to show the cryptographic operations.

The oblivious request resource generates a key pair. In this example the server chooses DHKEM(X25519, HKDF-SHA256) and generates an X25519 key pair [X25519]. The X25519 secret key is:

```
cb14d538a70d8a74d47fb7e3ac5052a086da127c678d3585dcad72f98e3bfff83
```

The oblivious request resource constructs a key configuration that includes the corresponding public key as follows:

```
01002012a45279412ea6ef11e9f839bb5a422fc1262b5c023d787e4e636e70ae  
d3d56e00080001000100010003
```

This key configuration is somehow obtained by the client. Then when a client wishes to send an HTTP request of a GET request to "https://example.com", it constructs the following binary HTTP message:

```
00034745540568747470730b6578616d706c652e636f6d012f
```

The client then reads the oblivious request resource key configuration and selects a mutually supported KDF and AEAD. In this example, the client selects HKDF-SHA256 and AES-128-GCM. The client then generates an HPKE context that uses the server public key. This results in the following encapsulated key:

```
cd7786fd75143f12e03398dbe2bcfa8e01a8132e7b66050674db72730623ca3b
```

The corresponding private key is:

```
c20afd33a2f2663faf023acf5d56fc08fddd38aada29b21b3b96e16f4326ccf7
```

Applying the Seal operation from the HPKE context produces an encrypted message, allowing the client to construct the following encapsulated request:

```
01002000010001cd7786fd75143f12e03398dbe2bcfa8e01a8132e7b66050674  
db72730623ca3b68b9e75a0576745da12c4fa5053b7ec06d7f625197564a6087  
ec299f8d6fffa2a8addfc1c0f64b4b05
```

The client then sends this to the oblivious proxy resource in a POST request, which might look like the following HTTP/1.1 request:

```
POST /request.example.net/proxy HTTP/1.1
Host: proxy.example.org
Content-Type: message/ohttp-req
Content-Length: 78
```

<content is the encapsulated request above>

The oblivious proxy resource receives this request and forwards it to the oblivious request resource, which might look like:

```
POST /oblivious/request HTTP/1.1
Host: example.com
Content-Type: message/ohttp-req
Content-Length: 78
```

<content is the encapsulated request above>

The oblivious request resource receives this request, selects the key it generated previously using the key identifier from the message, and decrypts the message. As this request is directed to the same server, the oblivious request resource does not need to initiate an HTTP request to the oblivious target resource. The request can be served directly by the oblivious target resource, which generates a minimal response (consisting of just a 200 status code) as follows:

0140c8

The response is constructed by extracting a secret from the HPKE context:

9c0b96b577b9fc7a5beef536e0ff3a64

The key derivation for the encapsulated response uses both the encapsulated KEM key from the request and a randomly selected nonce. This produces a salt of:

cd7786fd75143f12e03398dbe2bcfa8e01a8132e7b66050674db72730623ca3b
061d62d5df5832c6c9fa4617ceb848a7

The salt and secret are both passed to the Extract function of the selected KDF (HKDF-SHA256) to produce a pseudorandom key of:

a0ab55d3b1811694943bb72c386f59bd030e1278692a3db2f30d8aac2f89a5fc

The pseudorandom key is used with the Expand function of the KDF and an info field of "key" to produce a 16-byte key for the selected AEAD (AES-128-GCM):

```
1dae9d7fe263d23e51a768bcaf310aa5
```

With the same KDF and pseudorandom key, an info field of "nonce" is used to generate a 12-byte nonce:

```
e520beec147740e4f8a3b553
```

The AEAD Seal function is then used to encrypt the response, which is added to the randomized nonce value to produce the encapsulated response:

```
061d62d5df5832c6c9fa4617ceb848a7a6f694da45accc3c32ad576cb204f7cd  
3bf23e
```

The oblivious request resource then constructs a response:

```
HTTP/1.1 200 OK  
Date: Wed, 27 Jan 2021 04:45:07 GMT  
Cache-Control: private, no-store  
Content-Type: message/ohhttp-res  
Content-Length: 38
```

<content is the encapsulated response>

The same response might then be generated by the oblivious proxy resource which might change as little as the Date header. The client is then able to use the HPKE context it created and the nonce from the encapsulated response to construct the AEAD key and nonce and decrypt the response.

Acknowledgments

This design is based on a design for oblivious DoH, described in [ODOH]. David Benjamin and Eric Rescorla made technical contributions.

Authors' Addresses

Martin Thomson
Mozilla

Email: mt@lowentropy.net

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

HTTPBIS
Internet-Draft
Intended status: Standards Track
Expires: 28 April 2022

M. Thomson
Mozilla
C.A. Wood
Cloudflare
25 October 2021

Oblivious HTTP
draft-thomson-ohai-ohttp-00

Abstract

This document describes a system for the forwarding of encrypted HTTP messages. This allows a client to make multiple requests of a server without the server being able to link those requests to the client or to identify the requests as having come from the same client.

Discussion Venues

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

Source for this draft and an issue tracker can be found at
<https://github.com/unicorn-wg/oblivious-http>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 28 April 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
3. Overview	4
3.1. Applicability	6
4. Key Configuration	7
4.1. Key Configuration Encoding	8
4.2. Key Configuration Media Type	8
5. HPKE Encapsulation	9
5.1. Encapsulation of Requests	10
5.2. Encapsulation of Responses	12
6. HTTP Usage	13
6.1. Informational Responses	14
6.2. Errors	14
7. Media Types	15
7.1. message/ohttp-req Media Type	15
7.2. message/ohttp-res Media Type	16
8. Security Considerations	17
8.1. Client Responsibilities	18
8.2. Proxy Responsibilities	19
8.2.1. Denial of Service	20
8.2.2. Linkability Through Traffic Analysis	20
8.3. Server Responsibilities	21
8.4. Replay Attacks	21
8.5. Post-Compromise Security	23
9. Privacy Considerations	23
10. Operational and Deployment Considerations	24
11. IANA Considerations	24
12. References	24
12.1. Normative References	24
12.2. Informative References	25
Appendix A. Complete Example of a Request and Response	27
Acknowledgments	29
Authors' Addresses	29

1. Introduction

The act of making a request using HTTP reveals information about the client identity to a server. Though the content of requests might reveal information, that is information under the control of the client. In comparison, the source address on the connection reveals information that a client has only limited control over.

Even where an IP address is not directly attributed to an individual, the use of an address over time can be used to correlate requests. Servers are able to use this information to assemble profiles of client behavior, from which they can make inferences about the people involved. The use of persistent connections to make multiple requests improves performance, but provides servers with additional certainty about the identity of clients in a similar fashion.

Use of an HTTP proxy can provide a degree of protection against servers correlating requests. Systems like virtual private networks or the Tor network [Dingledine2004], provide other options for clients.

Though the overhead imposed by these methods varies, the cost for each request is significant. Preventing request linkability requires that each request use a completely new TLS connection to the server. At a minimum, this requires an additional round trip to the server in addition to that required by the request. In addition to having high latency, there are significant secondary costs, both in terms of the number of additional bytes exchanged and the CPU cost of cryptographic computations.

This document describes a method of encapsulation for binary HTTP messages [BINARY] using Hybrid Public Key Encryption (HPKE; [HPKE]). This protects the content of both requests and responses and enables a deployment architecture that can separate the identity of a requester from the request.

Though this scheme requires that servers and proxies explicitly support it, this design represents a performance improvement over options that perform just one request in each connection. With limited trust placed in the proxy (see Section 8), clients are assured that requests are not uniquely attributed to them or linked to other requests.

2. Conventions and Definitions

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.

Encapsulated Request: An HTTP request that is encapsulated in an HPKE-encrypted message; see Section 5.1.

Encapsulated Response: An HTTP response that is encapsulated in an HPKE-encrypted message; see Section 5.2.

Oblivious Proxy Resource: An intermediary that forwards requests and responses between clients and a single oblivious request resource.

Oblivious Request Resource: A resource that can receive an encapsulated request, extract the contents of that request, forward it to an oblivious target resource, receive a response, encapsulate that response, then return that response.

Oblivious Target Resource: The resource that is the target of an encapsulated request. This resource logically handles only regular HTTP requests and responses and so might be ignorant of the use of oblivious HTTP to reach it.

This draft includes pseudocode that uses the functions and conventions defined in [HPKE].

Encoding an integer to a sequence of bytes in network byte order is described using the function `encode(n, v)`, where `n` is the number of bytes and `v` is the integer value. The function `len()` returns the length of a sequence of bytes.

Formats are described using notation from Section 1.3 of [QUIC].

3. Overview

A client learns the following:

- * The identity of an oblivious request resource. This might include some information about oblivious target resources that the oblivious request resource supports.
- * The details of an HPKE public key that the oblivious request resource accepts, including an identifier for that key and the HPKE algorithms that are used with that key.

- * The identity of an oblivious proxy resource that will forward encapsulated requests and responses to the oblivious request resource.

This information allows the client to make a request of an oblivious target resource without that resource having only a limited ability to correlate that request with the client IP or other requests that the client might make to that server.

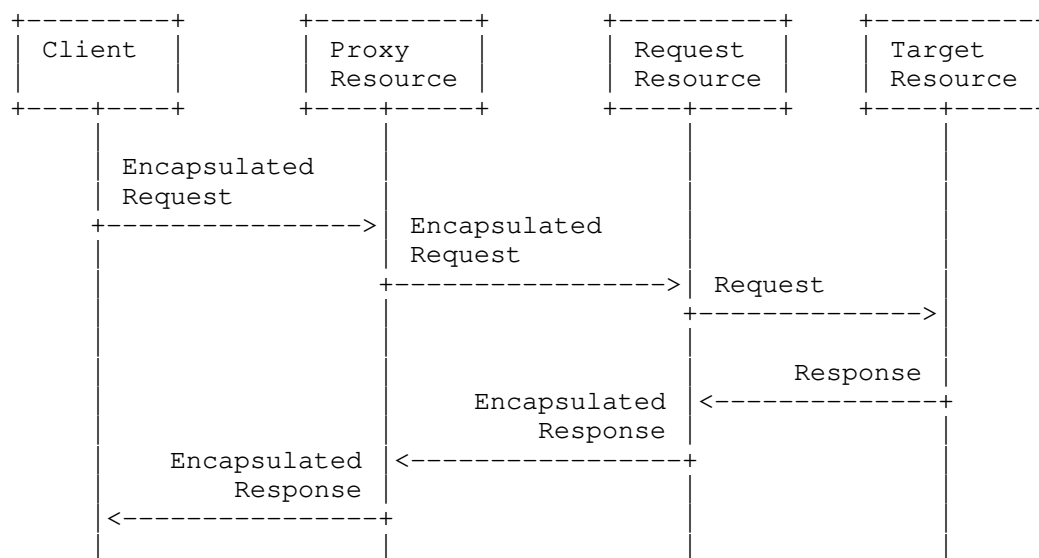


Figure 1: Overview of Oblivious HTTP

In order to make a request to an oblivious target resource, the following steps occur, as shown in Figure 1:

1. The client constructs an HTTP request for an oblivious target resource.
2. The client encodes the HTTP request in a binary HTTP message and then encapsulates that message using HPKE and the process from Section 5.1.
3. The client sends a POST request to the oblivious proxy resource with the encapsulated request as the content of that message.
4. The oblivious proxy resource forwards this request to the oblivious request resource.

5. The oblivious request resource receives this request and removes the HPKE protection to obtain an HTTP request.
6. The oblivious request resource makes an HTTP request that includes the target URI, method, fields, and content of the request it acquires.
7. The oblivious target resource answers this HTTP request with an HTTP response.
8. The oblivious request resource encapsulates the HTTP response following the process in Section 5.2 and sends this in response to the request from the oblivious proxy resource.
9. The oblivious proxy resource forwards this response to the client.
10. The client removes the encapsulation to obtain the response to the original request.

3.1. Applicability

Oblivious HTTP has limited applicability. Many uses of HTTP benefit from being able to carry state between requests, such as with cookies ([RFC6265]), authentication (Section 11 of [HTTP]), or even alternative services ([RFC7838]). Oblivious HTTP seeks to prevent this sort of linkage, which requires that applications not carry state between requests.

Oblivious HTTP is primarily useful where privacy risks associated with possible stateful treatment of requests are sufficiently negative that the cost of deploying this protocol can be justified. Oblivious HTTP is simpler and less costly than more robust systems, like Prio ([PRIO]) or Tor ([Dingledine2004]), which can provide stronger guarantees at higher operational costs.

Oblivious HTTP is more costly than a direct connection to a server. Some costs, like those involved with connection setup, can be amortized, but there are several ways in which oblivious HTTP is more expensive than a direct request:

- * Each oblivious request requires at least two regular HTTP requests, which adds latency.
- * Each request is expanded in size with additional HTTP fields, encryption-related metadata, and AEAD expansion.

- * Deriving cryptographic keys and applying them for request and response protection takes non-negligible computational resources.

Examples of where preventing the linking of requests might justify these costs include:

- * DNS queries. DNS queries made to a recursive resolver reveal information about the requester, particularly if linked to other queries.
- * Telemetry submission. Applications that submit reports about their usage to their developers might use oblivious HTTP for some types of moderately sensitive data.

4. Key Configuration

A client needs to acquire information about the key configuration of the oblivious request resource in order to send encapsulated requests.

In order to ensure that clients do not encapsulate messages that other entities can intercept, the key configuration **MUST** be authenticated and have integrity protection.

This document describes the "application/ohttp-keys" media type; see Section 4.2. This media type might be used, for example with HTTPS, as part of a system for configuring or discovering key configurations. Note however that such a system needs to consider the potential for key configuration to be used to compromise client privacy; see Section 9.

Specifying a format for expressing the information a client needs to construct an encapsulated request ensures that different client implementations can be configured in the same way. This also enables advertising key configurations in a consistent format.

A client might have multiple key configurations to select from when encapsulating a request. Clients are responsible for selecting a preferred key configuration from those it supports. Clients need to consider both the key encapsulation method (KEM) and the combinations of key derivation function (KDF) and authenticated encryption with associated data (AEAD) in this decision.

Evolution of the key configuration format is supported through the definition of new formats that are identified by new media types.

4.1. Key Configuration Encoding

A single key configuration consists of a key identifier, a public key, an identifier for the KEM that the public key uses, and a set of HPKE symmetric algorithms. Each symmetric algorithm consists of an identifier for a KDF and an identifier for an AEAD.

Figure 2 shows a single key configuration, `KeyConfig`, that is expressed using the TLS syntax; see Section 3 of [TLS].

```
opaque HpkePublicKey[Npk];
uint16 HpkeKemId;
uint16 HpkeKdfId;
uint16 HpkeAeadId;

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricAlgorithms;

struct {
    uint8 key_id;
    HpkeKemId kem_id;
    HpkePublicKey public_key;
    HpkeSymmetricAlgorithms cipher_suites<4..2^16-4>;
} KeyConfig;
```

Figure 2: A Single Key Configuration

The types `HpkeKemId`, `HpkeKdfId`, and `HpkeAeadId` identify a KEM, KDF, and AEAD respectively. The definitions for these identifiers and the semantics of the algorithms they identify can be found in [HPKE]. The `Npk` parameter corresponding to the `HpkeKdfId` can be found in [HPKE].

4.2. Key Configuration Media Type

The "application/ohttp-keys" format is a media type that identifies a serialized collection of key configurations. The content of this media type comprises one or more key configuration encodings (see Section 4.1) that are concatenated.

Type name: application

Subtype name: ohttp-keys

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 8

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

5. HPKE Encapsulation

HTTP message encapsulation uses HPKE for request and response encryption. An encapsulated HTTP message includes the following values:

1. A binary-encoded HTTP message; see [BINARY].
2. Padding of arbitrary length which MUST contain all zeroes.

The encoding of an HTTP message is as follows:

```
Plaintext Message {  
    Message Length (i),  
    Message (...),  
    Padding Length (i),  
    Padding (...),  
}
```

An Encapsulated Request is comprised of a length-prefixed key identifier and a HPKE-protected request message. HPKE protection includes an encapsulated KEM shared secret (or enc), plus the AEAD-protected request message. An Encapsulated Request is shown in Figure 3. Section 5.1 describes the process for constructing and processing an Encapsulated Request.

```
Encapsulated Request {  
    Key Identifier (8),  
    KEM Identifier (16),  
    KDF Identifier (16),  
    AEAD Identifier (16),  
    Encapsulated KEM Shared Secret (8*Nenc),  
    AEAD-Protected Request (...),  
}
```

Figure 3: Encapsulated Request

The Nenc parameter corresponding to the HpkeKdfId can be found in [HPKE].

Responses are bound to responses and so consist only of AEAD-protected content. Section 5.2 describes the process for constructing and processing an Encapsulated Response.

```
Encapsulated Response {  
    Nonce (Nk),  
    AEAD-Protected Response (...),  
}
```

Figure 4: Encapsulated Response

The size of the Nonce field in an Encapsulated Response corresponds to the size of an AEAD key for the corresponding HPKE ciphersuite.

5.1. Encapsulation of Requests

Clients encapsulate a request request using values from a key configuration:

- * the key identifier from the configuration, `keyID`, with the corresponding KEM identified by `kemID`,
- * the public key from the configuration, `pkR`, and
- * a selected combination of KDF, identified by `kdfID`, and AEAD, identified by `aeadID`.

The client then constructs an encapsulated request, `enc_request`, as follows:

1. Compute an HPKE context using `pkR`, yielding context and encapsulation key `enc`.
2. Construct associated data, `aad`, by concatenating the values of `keyID`, `kemID`, `kdfID`, and `aeadID`, as one 8-bit integer and three 16-bit integers, respectively, each in network byte order.
3. Encrypt (seal) request with `aad` as associated data using context, yielding ciphertext `ct`.
4. Concatenate the values of `aad`, `enc`, and `ct`, yielding an Encapsulated Request `enc_request`.

Note that `enc` is of fixed-length, so there is no ambiguity in parsing this structure.

In pseudocode, this procedure is as follows:

```
enc, context = SetupBaseS(pkR, "request")
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID))
ct = context.Seal(aad, request)
enc_request = concat(aad, enc, ct)
```

Servers decrypt an Encapsulated Request by reversing this process. Given an Encapsulated Request `enc_request`, a server:

1. Parses `enc_request` into `keyID`, `kemID`, `kdfID`, `aeadID`, `enc`, and `ct` (indicated using the function `parse()` in pseudocode). The server is then able to find the HPKE private key, `skR`, corresponding to `keyID`.
 - a. If `keyID` does not identify a key matching the type of `kemID`, the server returns an error.

- b. If `kdfID` and `aeadID` identify a combination of KDF and AEAD that the server is unwilling to use with `skR`, the server returns an error.
2. Compute an HPKE context using `skR` and the encapsulated key `enc`, yielding `context`.
3. Construct additional associated data, `aad`, from `keyID`, `kdfID`, and `aeadID` or as the first five bytes of `enc_request`.
4. Decrypt `ct` using `aad` as associated data, yielding `request` or an error on failure. If decryption fails, the server returns an error.

In pseudocode, this procedure is as follows:

```
keyID, kemID, kdfID, aeadID, enc, ct = parse(enc_request)
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID))
context = SetupBaseR(enc, skR, "request")
request, error = context.Open(aad, ct)
```

5.2. Encapsulation of Responses

Given an HPKE context `context`, a request message `request`, and a response `response`, servers generate an Encapsulated Response `enc_response` as follows:

1. Export a secret `secret` from `context`, using the string "response" as context. The length of this secret is $\max(N_n, N_k)$, where N_n and N_k are the length of AEAD key and nonce associated with `context`.
2. Generate a random value of length $\max(N_n, N_k)$ bytes, called `response_nonce`.
3. Extract a pseudorandom key `prk` using the Extract function provided by the KDF algorithm associated with `context`. The `ikm` input to this function is `secret`; the salt input is the concatenation of `enc` (from `enc_request`) and `response_nonce`.
4. Use the Expand function provided by the same KDF to extract an AEAD key `key`, of length N_k - the length of the keys used by the AEAD associated with `context`. Generating key uses a label of "key".

5. Use the same Expand function to extract a nonce of length N_n – the length of the nonce used by the AEAD. Generating nonce uses a label of "nonce".
6. Encrypt response, passing the AEAD function Seal the values of key, nonce, empty aad, and a pt input of request, which yields ct.
7. Concatenate response_nonce and ct, yielding an Encapsulated Response enc_response. Note that response_nonce is of fixed-length, so there is no ambiguity in parsing either response_nonce or ct.

In pseudocode, this procedure is as follows:

```
secret = context.Export("response", Nk)
response_nonce = random(max(Nn, Nk))
salt = concat(enc, response_nonce)
prk = Extract(salt, secret)
aead_key = Expand(prk, "key", Nk)
aead_nonce = Expand(prk, "nonce", Nn)
ct = Seal(aead_key, aead_nonce, "", response)
enc_response = concat(response_nonce, ct)
```

Clients decrypt an Encapsulated Request by reversing this process. That is, they first parse enc_response into response_nonce and ct. They then follow the same process to derive values for aead_key and aead_nonce.

The client uses these values to decrypt ct using the Open function provided by the AEAD. Decrypting might produce an error, as follows:

```
response, error = Open(aead_key, aead_nonce, "", ct)
```

6. HTTP Usage

A client interacts with the oblivious proxy resource by constructing an encapsulated request. This encapsulated request is included as the content of a POST request to the oblivious proxy resource. This request MUST only contain those fields necessary to carry the encapsulated request: a method of POST, a target URI of the oblivious proxy resource, a header field containing the content type (see (Section 7), and the encapsulated request as the request content. Clients MAY include fields that do not reveal information about the content of the request, such as Alt-Used [ALT-SVC], or information that it trusts the oblivious proxy resource to remove, such as fields that are listed in the Connection header field.

The oblivious proxy resource interacts with the oblivious request resource by constructing a request using the same restrictions as the client request, except that the target URI is the oblivious request resource. The content of this request is copied from the client. The oblivious proxy resource **MUST NOT** add information about the client to this request.

When a response is received from the oblivious request resource, the oblivious proxy resource forwards the response according to the rules of an HTTP proxy; see Section 7.6 of [HTTP].

An oblivious request resource, if it receives any response from the oblivious target resource, sends a single 200 response containing the encapsulated response. Like the request from the client, this response **MUST** only contain those fields necessary to carry the encapsulated response: a 200 status code, a header field indicating the content type, and the encapsulated response as the response content. As with requests, additional fields **MAY** be used to convey information that does not reveal information about the encapsulated response.

An oblivious request resource acts as a gateway for requests to the oblivious target resource (see Section 7.6 of [HTTP]). The one exception is that any information it might forward in a response **MUST** be encapsulated, unless it is responding to errors it detects before removing encapsulation of the request; see Section 6.2.

6.1. Informational Responses

This encapsulation does not permit progressive processing of responses. Though the binary HTTP response format does support the inclusion of informational (1xx) status codes, the AEAD encapsulation cannot be removed until the entire message is received.

In particular, the Expect header field with 100-continue (see Section 10.1.1 of [HTTP]) cannot be used. Clients **MUST NOT** construct a request that includes a 100-continue expectation; the oblivious request resource **MUST** generate an error if a 100-continue expectation is received.

6.2. Errors

A server that receives an invalid message for any reason **MUST** generate an HTTP response with a 4xx status code.

Errors detected by the oblivious proxy resource and errors detected by the oblivious request resource before removing protection (including being unable to remove encapsulation for any reason) result in the status code being sent without protection in response to the POST request made to that resource.

Errors detected by the oblivious request resource after successfully removing encapsulation and errors detected by the oblivious target resource MUST be sent in an encapsulated response.

7. Media Types

Media types are used to identify encapsulated requests and responses.

Evolution of the format of encapsulated requests and responses is supported through the definition of new formats that are identified by new media types.

7.1. message/ohttp-req Media Type

The "message/ohttp-req" identifies an encapsulated binary HTTP request. This is a binary format that is defined in Section 5.1.

Type name: message

Subtype name: ohttp-req

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 8

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

7.2. message/ohhttp-res Media Type

The "message/ohhttp-res" identifies an encapsulated binary HTTP response. This is a binary format that is defined in Section 5.2.

Type name: message

Subtype name: ohhttp-res

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 8

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8. Security Considerations

In this design, a client wishes to make a request of a server that is authoritative for the oblivious target resource. The client wishes to make this request without linking that request with either:

1. The identity at the network and transport layer of the client (that is, the client IP address and TCP or UDP port number the client uses to create a connection).
2. Any other request the client might have made in the past or might make in the future.

In order to ensure this, the client selects a proxy (that serves the oblivious proxy resource) that it trusts will protect this information by forwarding the encapsulated request and response without passing the server (that serves the oblivious request resource).

In this section, a deployment where there are three entities is considered:

- * A client makes requests and receives responses
- * A proxy operates the oblivious proxy resource
- * A server operates both the oblivious request resource and the oblivious target resource

To achieve the stated privacy goals, the oblivious proxy resource cannot be operated by the same entity as the oblivious request resource. However, colocation of the oblivious request resource and oblivious target resource simplifies the interactions between those resources without affecting client privacy.

As a consequence of this configuration, Oblivious HTTP prevents linkability described above. Informally, this means:

1. Requests and responses are known only to clients and targets in possession of the corresponding response encapsulation key and HPKE keying material. In particular, the oblivious proxy knows the origin and destination of an encapsulated request and response, yet does not know the decapsulated contents. Likewise, targets know only the oblivious request origin, i.e., the proxy, and the decapsulated request. Only the client knows both the plaintext request and response.
2. Targets cannot link requests from the same client in the absence of unique per-client keys.

Traffic analysis that might affect these properties are outside the scope of this document; see Section 8.2.2.

A formal analysis of Oblivious HTTP is in [OHTTP-ANALYSIS].

8.1. Client Responsibilities

Clients MUST ensure that the key configuration they select for generating encapsulated requests is integrity protected and authenticated so that it can be attributed to the oblivious request resource; see Section 4.

Since clients connect directly to the proxy instead of the target, application configurations wherein clients make policy decisions about target connections, e.g., to apply certificate pinning, are incompatible with Oblivious HTTP. In such cases, alternative technologies such as HTTP CONNECT (Section 9.3.6 of [HTTP]) can be used. Applications could implement related policies on key configurations and proxy connections, though these might not provide the same properties as policies enforced directly on target connections. When this difference is relevant, applications can instead connect directly to the target at the cost of either privacy or performance.

Clients MUST NOT include identifying information in the request that is encapsulated. Identifying information includes cookies [COOKIES], authentication credentials or tokens, and any information that might reveal client-specific information such as account credentials.

Clients cannot carry connection-level state between requests as they only establish direct connections to the proxy responsible for the oblivious proxy resource. However, clients need to ensure that they construct requests without any information gained from previous requests. Otherwise, the server might be able to use that information to link requests. Cookies [COOKIES] are the most obvious feature that **MUST NOT** be used by clients. However, clients need to include all information learned from requests, which could include the identity of resources.

Clients **MUST** generate a new HPKE context for every request, using a good source of entropy ([RANDOM]) for generating keys. Key reuse not only risks requests being linked, reuse could expose request and response contents to the proxy.

The request the client sends to the oblivious proxy resource only requires minimal information; see Section 6. The request that carries the encapsulated request and is sent to the oblivious proxy resource **MUST NOT** include identifying information unless the client ensures that this information is removed by the proxy. A client **MAY** include information only for the oblivious proxy resource in header fields identified by the Connection header field if it trusts the proxy to remove these as required by Section 7.6.1 of [HTTP]. The client needs to trust that the proxy does not replicate the source addressing information in the request it forwards.

Clients rely on the oblivious proxy resource to forward encapsulated requests and responses. However, the proxy can only refuse to forward messages, it cannot inspect or modify the contents of encapsulated requests or responses.

8.2. Proxy Responsibilities

The proxy that serves the oblivious proxy resource has a very simple function to perform. For each request it receives, it makes a request of the oblivious request resource that includes the same content. When it receives a response, it sends a response to the client that includes the content of the response from the oblivious request resource. When generating a request, the proxy **MUST** follow the forwarding rules in Section 7.6 of [HTTP].

A proxy can also generate responses, though it assumed to not be able to examine the content of a request (other than to observe the choice of key identifier, KDF, and AEAD), so it is also assumed that it cannot generate an encapsulated response.

A proxy MUST NOT add information about the client identity when forwarding requests. This includes the Via field, the Forwarded field [FORWARDED], and any similar information. A client does not depend on the proxy using an authenticated and encrypted connection to the oblivious request resource, only that information about the client not be attached to forwarded requests.

8.2.1. Denial of Service

As there are privacy benefits from having a large rate of requests forwarded by the same proxy (see Section 8.2.2), servers that operate the oblivious request resource might need an arrangement with proxies. This arrangement might be necessary to prevent having the large volume of requests being classified as an attack by the server.

If a server accepts a larger volume of requests from a proxy, it needs to trust that the proxy does not allow abusive levels of request volumes from clients. That is, if a server allows requests from the proxy to be exempt from rate limits, the server might want to ensure that the proxy applies a rate limiting policy that is acceptable to the server.

Servers that enter into an agreement with a proxy that enables a higher request rate might choose to authenticate the proxy to enable the higher rate.

8.2.2. Linkability Through Traffic Analysis

As the time at which encapsulated request or response messages are sent can reveal information to a network observer. Though messages exchanged between the oblivious proxy resource and the oblivious request resource might be sent in a single connection, traffic analysis could be used to match messages that are forwarded by the proxy.

A proxy could, as part of its function, add delays in order to increase the anonymity set into which each message is attributed. This could latency to the overall time clients take to receive a response, which might not be what some clients want.

A proxy can use padding to reduce the effectiveness of traffic analysis.

A proxy that forwards large volumes of exchanges can provide better privacy by providing larger sets of messages that need to be matched.

8.3. Server Responsibilities

A server that operates both oblivious request and oblivious target resources is responsible for removing request encapsulation, generating a response the encapsulated request, and encapsulating the response.

Servers should account for traffic analysis based on response size or generation time. Techniques such as padding or timing delays can help protect against such attacks; see Section 8.2.2.

If separate entities provide the oblivious request resource and oblivious target resource, these entities might need an arrangement similar to that between server and proxy for managing denial of service; see Section 8.2.1. It is also necessary to provide confidentiality protection for the unprotected requests and responses, plus protections for traffic analysis; see Section 8.2.2.

An oblivious request resource needs to have a plan for replacing keys. This might include regular replacement of keys, which can be assigned new key identifiers. If an oblivious request resource receives a request that contains a key identifier that it does not understand or that corresponds to a key that has been replaced, the server can respond with an HTTP 422 (Unprocessable Content) status code.

A server can also use a 422 status code if the server has a key that corresponds to the key identifier, but the encapsulated request cannot be successfully decrypted using the key.

8.4. Replay Attacks

Encapsulated requests can be copied and replayed by the oblivious proxy resource. The design of oblivious HTTP does not assume that the oblivious proxy resource will not replay requests. In addition, if a client sends an encapsulated request in TLS early data (see Section 8 of [TLS] and [RFC8470]), a network-based adversary might be able to cause the request to be replayed. In both cases, the effect of a replay attack and the mitigations that might be employed are similar to TLS early data.

A client or oblivious proxy resource MUST NOT automatically attempt to retry a failed request unless it receives a positive signal indicating that the request was not processed or forwarded. The HTTP/2 REFUSED_STREAM error code (Section 8.1.4 of [RFC7540]), the HTTP/3 H3_REQUEST_REJECTED error code (Section 8.1 of [QUIC-HTTP]), or a GOAWAY frame with a low enough identifier (in either protocol version) are all sufficient signals that no processing occurred. Connection failures or interruptions are not sufficient signals that no processing occurred.

The anti-replay mechanisms described in Section 8 of [TLS] are generally applicable to oblivious HTTP requests. Servers can use the encapsulated keying material as a unique key for identifying potential replays. This depends on clients generating a new HPKE context for every request.

The mechanism used in TLS for managing differences in client and server clocks cannot be used as it depends on being able to observe previous interactions. Oblivious HTTP explicitly prevents such linkability. Applications can still include an explicit indication of time to limit the span of time over which a server might need to track accepted requests. Clock information could be used for client identification, so reduction in precision or obfuscation might be necessary.

The considerations in [RFC8470] as they relate to managing the risk of replay also apply, though there is no option to delay the processing of a request.

Limiting requests to those with safe methods might not be satisfactory for some applications, particularly those that involve the submission of data to a server. The use of idempotent methods might be of some use in managing replay risk, though it is important to recognize that different idempotent requests can be combined to be not idempotent.

Idempotent actions with a narrow scope based on the value of a protected nonce could enable data submission with limited replay exposure. A nonce might be added as an explicit part of a request, or, if the oblivious request and target resources are co-located, the encapsulated keying material can be used to produce a nonce.

The server-chosen response_nonce field ensures that responses have unique AEAD keys and nonces even when requests are replayed.

8.5. Post-Compromise Security

This design does not provide post-compromise security for responses. A client only needs to retain keying material that might be used compromise the confidentiality and integrity of a response until that response is consumed, so there is negligible risk associated with a client compromise.

A server retains a secret key that might be used to remove protection from messages over much longer periods. A server compromise that provided access to the oblivious request resource secret key could allow an attacker to recover the plaintext of all requests sent toward affected keys and all of the responses that were generated.

Even if server keys are compromised, an adversary cannot access messages exchanged by the client with the oblivious proxy resource as messages are protected by TLS. Use of a compromised key also requires that the oblivious proxy resource cooperate with the attacker or that the attacker is able to compromise these TLS connections.

The total number of affected messages affected by server key compromise can be limited by regular rotation of server keys.

9. Privacy Considerations

One goal of this design is that independent client requests are only linkable by the chosen key configuration. The oblivious proxy and request resources can link requests using the same key configuration by matching `KeyConfig.key_id`, or, if the oblivious target resource is willing to use trial decryption, a limited set of key configurations that share an identifier. An oblivious proxy can link requests using the public key corresponding to `KeyConfig.key_id`.

Request resources are capable of linking requests depending on how `KeyConfigs` are produced by servers and discovered by clients. Specifically, servers can maliciously construct key configurations to track individual clients. A specific method for a client to acquire key configurations is not included in this specification. Clients need to consider these tracking vectors when choosing a discovery method. Applications using this design should provide accommodations to mitigate tracking using key configurations.

10. Operational and Deployment Considerations

Using Oblivious HTTP adds both cryptographic and latency to requests relative to a simple HTTP request-response exchange. Deploying proxy services that are on path between clients and servers avoids adding significant additional delay due to network topology. A study of a similar system [ODoH] found that deploying proxies close to servers was most effective in minimizing additional latency.

Oblivious HTTP might be incompatible with network interception regimes, such as those that rely on configuring clients with trust anchors and intercepting TLS connections. While TLS might be intercepted successfully, interception middleboxes devices might not receive updates that would allow Oblivious HTTP to be correctly identified using the media types defined in Section 7.

Oblivious HTTP has a simple key management design that is not trivially altered to enable interception by intermediaries. Clients that are configured to enable interception might choose to disable Oblivious HTTP in order to ensure that content is accessible to middleboxes.

11. IANA Considerations

Please update the "Media Types" registry at <https://www.iana.org/assignments/media-types> (<https://www.iana.org/assignments/media-types>) with the registration information in Section 7 for the media types "message/ohttp-req", "message/ohttp-res", and "application/ohttp-keys".

12. References

12.1. Normative References

- [BINARY] Thomson, M., "Binary Representation of HTTP Messages", Work in Progress, Internet-Draft, draft-thomson-http-binary-message-00, 26 October 2021, <<https://datatracker.ietf.org/doc/html/draft-thomson-http-binary-message-00>>.
- [HPKE] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-12>>.

- [HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-19, 12 September 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-19>>.
- [QUIC] 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>>.
- [QUIC-HTTP] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.
- [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>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/rfc/rfc7540>>.
- [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>>.
- [RFC8470] Thomson, M., Nottingham, M., and W. Tareau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/rfc/rfc8470>>.
- [TLS] 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>>.

12.2. Informative References

- [ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.

- [COOKIES] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.
- [Dingledine2004] Dingledine, R., Mathewson, N., and P. Syverson, "Tor: The Second-Generation Onion Router", August 2004, <<https://svn.torproject.org/svn/projects/design-paper/tor-design.html>>.
- [FORWARDED] Petersson, A. and M. Nilsson, "Forwarded HTTP Extension", RFC 7239, DOI 10.17487/RFC7239, June 2014, <<https://www.rfc-editor.org/rfc/rfc7239>>.
- [ODOH] Singanamalla, S., Chunhapanya, S., Vavrusa, M., Verma, T., Wu, P., Fayed, M., Heimerl, K., Sullivan, N., and C. A. Wood, "Oblivious DNS over HTTPS (ODOH): A Practical Privacy Enhancement to DNS", 7 January 2021, <<https://www.petsymposium.org/2021/files/papers/issue4/popets-2021-0085.pdf>>.
- [ODOH] Kinnear, E., McManus, P., Pauly, T., Verma, T., and C. A. Wood, "Oblivious DNS Over HTTPS", Work in Progress, Internet-Draft, draft-pauly-dprive-oblivious-doh-07, 2 September 2021, <<https://datatracker.ietf.org/doc/html/draft-pauly-dprive-oblivious-doh-07>>.
- [OHTTP-ANALYSIS] Hoyland, J., "Tamarin Model of Oblivious HTTP", 23 August 2021, <<https://github.com/cloudflare/ohttp-analysis>>.
- [PRIO] Corrigan-Gibbs, H. and D. Boneh, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics", 14 March 2017, <<https://crypto.stanford.edu/prio/paper.pdf>>.
- [RANDOM] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/rfc/rfc4086>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.
- [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.

[X25519] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/rfc/rfc7748>>.

Appendix A. Complete Example of a Request and Response

A single request and response exchange is shown here. Binary values (key configuration, secret keys, the content of messages, and intermediate values) are shown in hexadecimal. The request and response here are absolutely minimal; the purpose of this example is to show the cryptographic operations.

The oblivious request resource generates a key pair. In this example the server chooses DHKEM(X25519, HKDF-SHA256) and generates an X25519 key pair [X25519]. The X25519 secret key is:

```
cb14d538a70d8a74d47fb7e3ac5052a086da127c678d3585dcad72f98e3bfff83
```

The oblivious request resource constructs a key configuration that includes the corresponding public key as follows:

```
01002012a45279412ea6ef11e9f839bb5a422fc1262b5c023d787e4e636e70ae  
d3d56e00080001000100010003
```

This key configuration is somehow obtained by the client. Then when a client wishes to send an HTTP request of a GET request to <https://example.com>, it constructs the following binary HTTP message:

```
00034745540568747470730b6578616d706c652e636f6d012f
```

The client then reads the oblivious request resource key configuration and selects a mutually supported KDF and AEAD. In this example, the client selects HKDF-SHA256 and AES-128-GCM. The client then generates an HPKE context that uses the server public key. This results in the following encapsulated key:

```
cd7786fd75143f12e03398dbe2bcfa8e01a8132e7b66050674db72730623ca3b
```

The corresponding private key is:

```
c20afd33a2f2663faf023acf5d56fc08fddd38aada29b21b3b96e16f4326ccf7
```

Applying the Seal operation from the HPKE context produces an encrypted message, allowing the client to construct the following encapsulated request:

```
01002000010001cd7786fd75143f12e03398dbe2bcfa8e01a8132e7b66050674
db72730623ca3b68b9e75a0576745da12c4fa5053b7ec06d7f625197564a6087
ec299f8d6fffa2a8addfc1c0f64b4b05
```

The client then sends this to the oblivious proxy resource in a POST request, which might look like the following HTTP/1.1 request:

```
POST /request.example.net/proxy HTTP/1.1
Host: proxy.example.org
Content-Type: message/ohttp-req
Content-Length: 78
```

<content is the encapsulated request above>

The oblivious proxy resource receives this request and forwards it to the oblivious request resource, which might look like:

```
POST /oblivious/request HTTP/1.1
Host: example.com
Content-Type: message/ohttp-req
Content-Length: 78
```

<content is the encapsulated request above>

The oblivious request resource receives this request, selects the key it generated previously using the key identifier from the message, and decrypts the message. As this request is directed to the same server, the oblivious request resource does not need to initiate an HTTP request to the oblivious target resource. The request can be served directly by the oblivious target resource, which generates a minimal response (consisting of just a 200 status code) as follows:

```
0140c8
```

The response is constructed by extracting a secret from the HPKE context:

```
9c0b96b577b9fc7a5beef536e0ff3a64
```

The key derivation for the encapsulated response uses both the encapsulated KEM key from the request and a randomly selected nonce. This produces a salt of:

```
cd7786fd75143f12e03398dbe2bcfa8e01a8132e7b66050674db72730623ca3b
061d62d5df5832c6c9fa4617ceb848a7
```

The salt and secret are both passed to the Extract function of the selected KDF (HKDF-SHA256) to produce a pseudorandom key of:

a0ab55d3b1811694943bb72c386f59bd030e1278692a3db2f30d8aac2f89a5fc

The pseudorandom key is used with the Expand function of the KDF and an info field of "key" to produce a 16-byte key for the selected AEAD (AES-128-GCM):

1dae9d7fe263d23e51a768bcaf310aa5

With the same KDF and pseudorandom key, an info field of "nonce" is used to generate a 12-byte nonce:

e520beec147740e4f8a3b553

The AEAD Seal function is then used to encrypt the response, which is added to the randomized nonce value to produce the encapsulated response:

061d62d5df5832c6c9fa4617ceb848a7a6f694da45accc3c32ad576cb204f7cd3bf23e

The oblivious request resource then constructs a response:

HTTP/1.1 200 OK
Date: Wed, 27 Jan 2021 04:45:07 GMT
Cache-Control: private, no-store
Content-Type: message/ohhttp-res
Content-Length: 38

<content is the encapsulated response>

The same response might then be generated by the oblivious proxy resource which might change as little as the Date header. The client is then able to use the HPKE context it created and the nonce from the encapsulated response to construct the AEAD key and nonce and decrypt the response.

Acknowledgments

This design is based on a design for oblivious DoH, described in [ODOH]. David Benjamin and Eric Rescorla made technical contributions.

Authors' Addresses

Martin Thomson
Mozilla

Email: mt@lowentropy.net

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net