Network Working Group M. Thomson
Internet-Draft Mozilla
Intended status: Standards Track J. Yasskin
Expires: February 15, 2019 Google
August 14, 2018

Merkle Integrity Content Encoding
draft-thomson-http-mice-03

Abstract

This memo introduces a content-coding for HTTP that provides progressive integrity for message contents. This integrity protection can be evaluated on a partial representation, allowing a recipient to process a message as it is delivered while retaining strong integrity protection.

Note to Readers

RFC EDITOR: please remove this section before publication

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

The source code and issues list for this draft can be found at https://github.com/martinthomson/http-mice [2].

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 February 15, 2019.

Copyright Notice

Copyright (c) 2018 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
  1.1. Notational Conventions 3
2. The "mi-sha256" HTTP Content Encoding 3
  2.1. Content Encoding Structure 5
  2.2. Validating Integrity Proofs 5
3. The "mi-sha256" Digest Algorithm 6
4. Examples 7
  4.1. Simple Example 7
  4.2. Example with Multiple Records 7
5. Security Considerations 8
  5.1. Message Truncation 8
  5.2. Algorithm Agility 9
6. IANA Considerations 9
  6.1. The "mi-sha256" HTTP Content Encoding 9
  6.2. The "mi-sha256" Digest Algorithm 9
7. References 9
  7.1. Normative References 9
  7.2. Informative References 10
  7.3. URIs 10
Appendix A. Acknowledgements 11
Appendix B. FAQ 11
Authors' Addresses 11

1. Introduction

Integrity protection for HTTP content is highly valuable. HTTPS [RFC2818] is the most common form of integrity protection deployed, but that requires a direct TLS [RFC8446] connection to a host. However, additional integrity protection might be desirable for some use cases. This might be for additional protection against failures

or attack (see [SRI]) or because content needs to remain unmodified throughout multiple HTTPS-protected exchanges.

This document describes a "mi-sha256" content-encoding (see Section 2) that is a progressive, hash-based integrity check based on Merkle Hash Trees [MERKLE].

The means of conveying the root integrity proof used by this content encoding will depend on deployment requirements. This document defines a digest algorithm (see Section 3) that can carry an integrity proof.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. The "mi-sha256" HTTP Content Encoding

A Merkle Hash Tree [MERKLE] is a structured integrity mechanism that collates multiple integrity checks into a tree. The leaf nodes of the tree contain data (or hashes of data) and non-leaf nodes contain hashes of the nodes below them.

A balanced Merkle Hash Tree is used to efficiently prove membership in large sets (such as in [RFC6962]). However, in this case, a right-skewed tree is used to provide a progressive integrity proof. This integrity proof is used to establish that a given record is part of a message.

The hash function used for "mi-sha256" content encoding is SHA-256 [FIPS180-4]. The integrity proof for all records other than the last is the hash of the concatenation of the record, the integrity proof of all subsequent records, and a single octet with a value of 0x1:

```
proof(r[i]) = SHA-256(r[i] || proof(r[i+1]) || 0x1)
```

The integrity proof for the final record is the hash of the record with a single octet with a value 0x0 appended:

```
proof(r[last]) = SHA-256(r[last] || 0x0)
```

Figure 1 shows the structure of the integrity proofs for a message that is split into 4 blocks: A, B, C, D). As shown, the integrity proof for the entire message (that is, "proof(A)") is derived from the content of the first block (A), plus the value of the proof for the second and subsequent blocks.

```
    proof(A)
      /\
     /  \
    /    \
   A    proof(B)
         /\
        /  \
       /    \
      B    proof(C)
             /\
            /  \
           /    \
          C    proof(D)
                 |
                 |
                 D
```

Figure 1: Proof structure for a message with 4 blocks

The final encoded message is formed from the record size and first record, followed by an arbitrary number of tuples of the integrity proof of the next record and then the record itself. Thus, in Figure 1, the body is:

rs || A || proof(B) || B || proof(C) || C || proof(D) || D

Note: The "||" operator is used to represent concatenation.

A message that has a content length less than or equal to the content size does not include any inline proofs. The proof for a message with a single record is simply the hash of the body plus a trailing zero octet.

As a special case, the encoding of an empty payload is itself an empty message (i.e. it omits the initial record size), and its integrity proof is SHA-256("\0").

RFC EDITOR: Please remove the next paragraph before publication.

Implementations of drafts of this specification MUST implement a content encoding named "mi-sha256-##" instead of the "mi-sha256" content encoding specified by the final RFC, with "##" replaced by the draft number being implemented. For example, implementations of draft-thomson-http-mice-03 would implement "mi-sha256-03".

2.1. Content Encoding Structure

In order to produce the final content encoding the content of the message is split into equal-sized records. The final record can contain less than the defined record size.

For non-empty payloads, the record size is included in the first 8 octets of the message as an unsigned 64-bit integer. This refers to the length of each data block.

The final encoded stream comprises of the record size ("rs"), plus a sequence of records, each "rs" octets in length. Each record, other than the last, is followed by a 32 octet proof for the record that follows. This allows a receiver to validate and act upon each record after receiving the proof that precedes it. The final record is not followed by a proof.

Note: This content encoding increases the size of a message by 8 plus 32 octets times the length of the message divided by the record size, rounded up, less one. That is, 8 + 32 * (ceil(length / rs) - 1).

Constructing a message with the "mi-sha256" content encoding requires processing of the records in reverse order, inserting the proof derived from each record before that record.

This structure permits the use of range requests [RFC7233]. However, to validate a given record, a contiguous sequence of records back to the start of the message is needed.

2.2. Validating Integrity Proofs

A receiver of a message with the "mi-sha256" content-encoding applied first attempts to acquire the integrity proof for the first record, "top-proof". If the Digest header field is present with the mi-sha256 parameter, a value might be included there.

The receiver attempts to read the first 8 octets as an unsigned 64-bit integer, "rs". If 8 octets aren't available then:

- If 0 octets are available, and "top-proof" is SHA-256("\0") (whose base64 encoding is "bjQLnP+zepicpUTmu3gKLHiQHT+zNzh2hRGjBhevoB0="), then return a 0-length decoded payload.

- Otherwise, validation fails.

The remainder of the message is read into records of size "rs" plus 32 octets. The last record is between 1 and "rs" octets in length, if not then validation fails. For each record:

1. Hash the record using SHA-256 with a single octet appended:
    a. All records other than the last have an octet with a value of 0x1 appended.
    b. The last record has an octet with a value of 0x0 appended.
2. Compare the hash with the expected value:
    a. For the first record, the expected value is "top-proof".
    b. For records after the first, the expected value is the last 32 octets of the previous record.
3. If the hash is different, then this record and all subsequent records do not have integrity protection and this process ends.
4. If a record is valid, up to "rs" octets is passed on for processing. In other words, the trailing 32 octets is removed from every record other than the last before being used.

If an integrity check fails, the message SHOULD be discarded and the exchange treated as an error unless explicitly configured otherwise. For clients, treat this as equivalent to a server error; servers SHOULD generate a 400 or other 4xx status code. However, if the integrity proof for the first record is not known, this check SHOULD NOT fail unless explicitly configured to do so.

3. The "mi-sha256" Digest Algorithm

[RFC3230] describes digests applying to "the entire instance associated with the message". The instance corresponds to the "representation" in Section 3 of [RFC7231], but unlike the existing digest algorithms, the "mi-sha256" digest algorithm specifies the top-level digest at the point when the "mi-sha256" content coding (Section 2) is applied or removed from the representation.

When the "mi-sha256" digest algorithm is specified for a representation, the recipient MUST use the base64-decoding (Section 4 of [RFC4648]) of the "mi-sha256" digest as the "top-proof" for the "mi-sha256" content encoding (Section 2.2).

The recipient MUST behave as described by Section 4.2.9 of [I-D.ietf-httpbis-header-structure] if it encounters improper

padding, non-zero padding bits, or non-alphabet characters, where rejecting the data means to reject the representation.

If different mechanisms specify different "top-proof" values for the "mi-sha256" content encoding, the recipient MUST reject the representation.

If "mi-sha256" content coding has not been applied to the representation exactly once (Section 3.1.2.2 of [RFC7231]), the recipient MUST reject the representation.

When rejecting the representation, clients SHOULD treat this as equivalent to a server error, and servers SHOULD generate a 400 or other 4xx status code.

RFC EDITOR: Please remove the next paragraph before publication.

Implementations of drafts of this specification MUST use a digest algorithm named the same as the "mi-sha256-##" content encoding they implement, with the meaning described for "mi-sha256" above.

4. Examples

4.1. Simple Example

The following example contains a short message. This contains just a single record, so there are no inline integrity proofs, just a single value in the mi-sha256 parameter of a Digest header field. The record size is prepended to the message body (shown here in angle brackets).

```
HTTP/1.1 200 OK
Digest: mi-sha256=dcRDgR2GM35DluAV13PzgnG6+pvQwPywfFvAu1UeFrs=
Content-Encoding: mi-sha256
Content-Length: 49

<0x0000000000000029>When I grow up, I want to be a watermelon
```

4.2. Example with Multiple Records

This example shows the same message as above, but with a smaller record size (16 octets). This results in two integrity proofs being included in the representation.

```
PUT /test HTTP/1.1
Host: example.com
Digest: mi-sha256=IVa9shfs0nyKEhHqtB3WVNANJ2Njm5KjQLjRtnbkYJ4=
Content-Encoding: mi-sha256
Content-Length: 113

<0x0000000000000010>When I grow up,
OElbplJlPK+Rv6JNK6p5/515IaoPoZo+2elWL7OQ60A=
I want to be a w
iPMpmgExHPrbEX3/RvwP4d16fWlK4l++p75PUu_KyN0=
atermelon
```

Since the inline integrity proofs contain non-printing characters, these are shown here using the base64 encoding [RFC4648] with new lines between the original text and integrity proofs. Note that there is a single trailing space (0x20) on the first line.

5. Security Considerations

The integrity of an entire message body depends on the means by which the integrity proof for the first record is protected. If this value comes from the same place as the message, then this provides only limited protection against transport-level errors (something that TLS provides adequate protection against).

Separate protection for header fields might be provided by other means if the first record retrieved is the first record in the message, but range requests do not allow for this option.

5.1. Message Truncation

This integrity scheme permits the detection of truncated messages. However, it enables and even encourages processing of messages prior to receiving an complete message. Actions taken on a partial message can produce incorrect results. For example, a message could say "I need some 2mm copper cable, please send 100mm for evaluation purposes" then be truncated to "I need some 2mm copper cable, please send 100m". A network-based attacker might be able to force this sort of truncation by delaying packets that contain the remainder of the message.

Whether it is safe to act on partial messages will depend on the nature of the message and the processing that is performed.

5.2. Algorithm Agility

A new content encoding type is needed in order to define the use of a hash function other than SHA-256.

6. IANA Considerations

6.1. The "mi-sha256" HTTP Content Encoding

This memo registers the "mi-sha256" HTTP content-coding in the HTTP Content Codings Registry, as detailed in Section 2.

- Name: mi-sha256

- Description: A Merkle Hash Tree based content encoding that provides progressive integrity.

- Reference: this specification

6.2. The "mi-sha256" Digest Algorithm

This memo registers the "mi-sha256" digest algorithm in the HTTP Digest Algorithm Values [3] registry:

- Digest Algorithm: mi-sha256

- Description: As specified in Section 3.

7. References

7.1. Normative References

[FIPS180-4]
Department of Commerce, National., "NIST FIPS 180-4, Secure Hash Standard", March 2012, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

[I-D.ietf-httpbis-header-structure]
Nottingham, M. and P. Kamp, "Structured Headers for HTTP", draft-ietf-httpbis-header-structure-07 (work in progress), July 2018.

[MERKLE] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", International Crytology Conference - CRYPTO , 1987.

[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/info/rfc2119>.

[RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002, <https://www.rfc-editor.org/info/rfc3230>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

7.2. Informative References

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <https://www.rfc-editor.org/info/rfc7233>.

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

[SRI] Akhawe, D., Braun, F., Marier, F., and J. Weinberger, "Subresource Integrity", W3C CR , November 2015, <https://w3c.github.io/webappsec-subresource-integrity/>.

7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] https://github.com/martinthomson/http-mice

[3] https://www.iana.org/assignments/http-dig-alg/http-dig-alg.xhtml

Appendix A. Acknowledgements

David Benjamin and Erik Nygren both separately suggested that something like this might be valuable. James Manger and Eric Rescorla provided useful feedback.

Appendix B. FAQ

1. Why not include the first proof in the encoding?

   The requirements for the integrity proof for the first record require a great deal more flexibility than this allows for. Transferring the proof separately is sometimes necessary. Separating the value out allows for that to happen more easily.

2. Why do messages have to be processed in reverse to construct them?

   The final integrity value, no matter how it is derived, has to depend on every bit of the message. That means that there are three choices: both sender and receiver have to process the whole message, the sender has to work backwards, or the receiver has to work backwards. The current form is the best option of the three. The expectation is that this will be useful for content that is generated once and sent multiple times, since the onerous backwards processing requirement can be amortized.

3. Why not just generate a table of hashes?

   An alternative design includes a header that comprises hashes of every block of the message. The final proof is a hash of that table. This has the advantage that the table can be built in any order. The disadvantage is that a receiver needs to store the table while processing content, whereas a chained hash can be processed with a single stored hash worth of state no matter how many blocks are present. The chained hash is also smaller by 32 octets.

Authors' Addresses

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org