Internet Engineering Task Force (IETF)
Request for Comments: 8188
Category: Standards Track
ISSN: 2070-1721

M. Thomson
Mozilla

June 2017

Encrypted Content-Encoding for HTTP

Abstract

This memo introduces a content coding for HTTP that allows message payloads to be encrypted.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc8188.

Copyright Notice

Copyright (c) 2017 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 (http://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

It is sometimes desirable to encrypt the contents of an HTTP message (request or response) so that when the payload is stored (e.g., with an HTTP PUT), only someone with the appropriate key can read it.

For example, it might be necessary to store a file on a server without exposing its contents to that server. Furthermore, that same file could be replicated to other servers (to make it more resistant to server or network failure), downloaded by clients (to make it available offline), etc., without exposing its contents.

These uses are not met by the use of Transport Layer Security (TLS) [RFC5246], since it only encrypts the channel between the client and server.

This document specifies a content coding (see Section 3.1.2 of [RFC7231]) for HTTP to serve these and other use cases.

This content coding is not a direct adaptation of message-based encryption formats -- such as those that are described by [RFC4880], [RFC5652], [RFC7516], and [XMLENC]. Those formats are not suited to stream processing, which is necessary for HTTP. The format described here follows more closely to the lower-level constructs described in [RFC5116].

To the extent that message-based encryption formats use the same primitives, the format can be considered to be a sequence of encrypted messages with a particular profile. For instance, Appendix A explains how the format is congruent with a sequence of JSON Web Encryption [RFC7516] values with a fixed header.

This mechanism is likely only a small part of a larger design that uses content encryption. How clients and servers acquire and identify keys will depend on the use case. In particular, a key management system is not described.

1.1. Requirements Language

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.

2. The "aes128gcm" HTTP Content Coding

The "aes128gcm" HTTP content coding indicates that a payload has been encrypted using Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) as identified as AEAD_AES_128_GCM in [RFC5116], Section 5.1. The AEAD_AES_128_GCM algorithm uses a 128-bit content-encryption key.

Using this content coding requires knowledge of a key. How this key is acquired is not defined in this document.

The "aes128gcm" content coding uses a single fixed set of encryption primitives. Cipher agility is achieved by defining a new content-coding scheme. This ensures that only the HTTP Accept-Encoding header field is necessary to negotiate the use of encryption.

The "aes128gcm" content coding uses a fixed record size. The final encoding consists of a header (see Section 2.1) and zero or more fixed-size encrypted records; the final record can be smaller than the record size.

The record size determines the length of each portion of plaintext that is enciphered. The record size ("rs") is included in the content-coding header (see Section 2.1).

```
+-----------+             content
|   data    |             any length up to rs-17 octets
+-----------+
     |
     v
+-----------+-----+       add a delimiter octet (0x01 or 0x02)
|   data    | pad |       then 0x00-valued octets to rs-16
+-----------+-----+       (or less on the last record)
         |
         v
+--------------------+    encrypt with AEAD_AES_128_GCM;
|    ciphertext      |    final size is rs;
+--------------------+    the last record can be smaller
```

AEAD_AES_128_GCM produces ciphertext 16 octets longer than its input plaintext. Therefore, the unencrypted content of each record is shorter than the record size by 16 octets. Valid records always contain at least a padding delimiter octet and a 16-octet authentication tag.

Each record contains a single padding delimiter octet followed by any number of zero octets. The last record uses a padding delimiter octet set to the value 2, all other records have a padding delimiter octet value of 1.

On decryption, the padding delimiter is the last non-zero-valued octet of the record. A decrypter MUST fail if the record contains no non-zero octet. A decrypter MUST fail if the last record contains a padding delimiter with a value other than 2 or if any record other than the last contains a padding delimiter with a value other than 1.

The nonce for each record is a 96-bit value constructed from the record sequence number and the input-keying material. Nonce derivation is covered in Section 2.3.

The additional data passed to each invocation of AEAD_AES_128_GCM is a zero-length octet sequence.

A consequence of this record structure is that range requests [RFC7233] and random access to encrypted payload bodies are possible at the granularity of the record size. Partial records at the ends of a range cannot be decrypted. Thus, it is best if range requests start and end on record boundaries. However, note that random access

to specific parts of encrypted data could be confounded by the presence of padding.

Selecting the record size most appropriate for a given situation requires a trade-off. A smaller record size allows decrypted octets to be released more rapidly, which can be appropriate for applications that depend on responsiveness. Smaller records also reduce the additional data required if random access into the ciphertext is needed.

Applications that don’t depend on streaming, random access, or arbitrary padding can use larger records, or even a single record. A larger record size reduces processing and data overheads.

2.1. Encryption Content-Coding Header

The content coding uses a header block that includes all parameters needed to decrypt the content (other than the key). The header block is placed in the body of a message ahead of the sequence of records.

```
+-----------+--------+-----------+---------------+
| salt (16) | rs (4) | idlen (1) | keyid (idlen) |
+-----------+--------+-----------+---------------+
```

salt: The "salt" parameter comprises the first 16 octets of the "aes128gcm" content-coding header. The same "salt" parameter value MUST NOT be reused for two different payload bodies that have the same input-keying material; generating a random salt for every application of the content coding ensures that content-encryption key reuse is highly unlikely.

rs: The "rs" or record size parameter contains an unsigned 32-bit integer in network byte order that describes the record size in octets. Note that it is, therefore, impossible to exceed the 2^36-31 limit on plaintext input to AEAD_AES_128_GCM. Values smaller than 18 are invalid.

idlen: The "idlen" parameter is an unsigned 8-bit integer that defines the length of the "keyid" parameter.

keyid: The "keyid" parameter can be used to identify the keying material that is used. This field is the length determined by the "idlen" parameter. Recipients that receive a message are expected to know how to retrieve keys; the "keyid" parameter might be input to that process. A "keyid" parameter SHOULD be a UTF-8-encoded [RFC3629] string, particularly where the identifier might need to be rendered in a textual form.

2.2. Content-Encryption Key Derivation

In order to allow the reuse of keying material for multiple different HTTP messages, a content-encryption key is derived for each message. The content-encryption key is derived from the "salt" parameter using the HMAC-based key derivation function (HKDF) described in [RFC5869] using the SHA-256 hash algorithm [FIPS180-4].

The value of the "salt" parameter is the salt input to the HKDF. The keying material identified by the "keyid" parameter is the input-keying material (IKM) to HKDF. Input-keying material is expected to be provided to recipients separately. The extract phase of HKDF, therefore, produces a pseudorandom key (PRK) as follows:

```
PRK = HMAC-SHA-256 (salt, IKM)
```

The info parameter to HKDF is set to the ASCII-encoded string "Content-Encoding: aes128gcm" and a single zero octet:

```
cek_info = "Content-Encoding: aes128gcm" || 0x00
```

Note(1): Concatenation of octet sequences is represented by the "||" operator.

Note(2): The strings used here and in Section 2.3 do not include a terminating 0x00 octet, as is used in some programming languages.

AEAD_AES_128_GCM requires a 16-octet (128-bit) content-encryption key (CEK), so the length (L) parameter to HKDF is 16. The second step of HKDF can, therefore, be simplified to the first 16 octets of a single HMAC:

```
CEK = HMAC-SHA-256(PRK, cek_info || 0x01)
```

2.3. Nonce Derivation

The nonce input to AEAD_AES_128_GCM is constructed for each record. The nonce for each record is a 12-octet (96-bit) value that is derived from the record sequence number, input-keying material, and "salt" parameter.

The input-keying material and "salt" parameter are input to HKDF with different info and length (L) parameters.

The length (L) parameter is 12 octets. The info parameter for the nonce is the ASCII-encoded string "Content-Encoding: nonce", terminated by a single zero octet:

```
nonce_info = "Content-Encoding: nonce" || 0x00
```

The result is combined with the record sequence number -- using exclusive or -- to produce the nonce. The record sequence number (SEQ) is a 96-bit unsigned integer in network byte order that starts at zero.

Thus, the final nonce for each record is a 12-octet value:

```
NONCE = HMAC-SHA-256(PRK, nonce_info || 0x01) XOR SEQ
```

This nonce construction prevents removal or reordering of records.

3. Examples

This section shows a few examples of the encrypted-content coding.

Note: All binary values in the examples in this section use base64 encoding with URL and filename safe alphabet [RFC4648]. This includes the bodies of requests. Whitespace and line wrapping is added to fit formatting constraints.

3.1. Encryption of a Response

Here, a successful HTTP GET response has been encrypted. This uses a record size of 4096 octets and no padding (just the single-octet padding delimiter), so only a partial record is present. The input-keying material is identified by an empty string (that is, the "keyid" field in the header is zero octets in length).

The encrypted data in this example is the UTF-8-encoded string "I am the walrus". The input-keying material is the value "yqdlZ-tYemfogSmv7Ws5PQ" (in base64url). The 54-octet content body contains a single record and is shown here using 71 base64url characters for presentation reasons.

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 54
Content-Encoding: aes128gcm

I1BsxtFttlv3u_Oo94xnmwAAEAAA-NAVub2qFgBEuQKRapoZu-IxkIva3MEB1PD-
ly8Thjg
```

Note that the media type has been changed to "application/octet-stream" to avoid exposing information about the content. Alternatively (and equivalently), the Content-Type header field can be omitted.

Intermediate values for this example (all shown using base64url):

```
salt (from header) = I1BsxtFttlv3u_Oo94xnmw
PRK = zyeH5phsIsgUyd4oiSEIy35x-gIi4aM7y0hCF8mwn9g
CEK = _wniytB-ofscZDh4tbSjHw
NONCE = Bcs8gkIRKLI8GeI8
unencrypted data = SSBhbSB0aGUgd2FscnVzAg
```

3.2. Encryption with Multiple Records

This example shows the same message with input-keying material of "BO3ZVPxUlnLORbVGMpbT1Q". In this example, the plaintext is split into records of 25 octets each (that is, the "rs" field in the header is 25). The first record includes one 0x00 padding octet. This means that there are 7 octets of message in the first record and 8 in the second. A key identifier of the UTF-8-encoded string "a1" is also included in the header.

```
HTTP/1.1 200 OK
Content-Length: 73
Content-Encoding: aes128gcm

uNCkWiNYzKTnBN9ji3-qWAAAABkCYTHOG8chz_gnvgOqdGYovxyjuqRyJFjEDyoF
1Fvkj6hQPdPHI51OEUKEpgz3SsLWIqS_uA
```

4. Security Considerations

This mechanism assumes the presence of a key management framework that is used to manage the distribution of keys between valid senders and receivers. Defining key management is part of composing this mechanism into a larger application, protocol, or framework.

Implementation of cryptography -- and key management in particular -- can be difficult. For instance, implementations need to account for the potential for exposing keying material on side channels, such as might be exposed by the time it takes to perform a given operation. The requirements for a good implementation of cryptographic algorithms can change over time.

4.1. Automatic Decryption

As a content coding, a "aes128gcm" content coding might be automatically removed by a receiver in a way that is not obvious to the ultimate consumer of a message. Recipients that depend on content-origin authentication using this mechanism MUST reject messages that don’t include the "aes128gcm" content coding.

4.2. Message Truncation

This content encoding is designed to permit the incremental processing of large messages. It also permits random access to plaintext in a limited fashion. The content encoding permits a receiver to detect when a message is truncated.

A partially delivered message MUST NOT be processed as though the entire message was successfully delivered. For instance, a partially delivered message cannot be cached as though it were complete.

An attacker might exploit willingness to process partial messages to cause a receiver to remain in a specific intermediate state. Implementations performing processing on partial messages need to ensure that any intermediate processing states don’t advantage an attacker.

4.3. Key and Nonce Reuse

Encrypting different plaintext with the same content-encryption key and nonce in AES-GCM is not safe [RFC5116]. The scheme defined here uses a fixed progression of nonce values. Thus, a new content-encryption key is needed for every application of the content coding. Since input-keying material can be reused, a unique "salt" parameter is needed to ensure that a content-encryption key is not reused.

If a content-encryption key is reused -- that is, if input-keying material and "salt" parameter are reused -- this could expose the plaintext and the authentication key, nullifying the protection offered by encryption. Thus, if the same input-keying material is reused, then the "salt" parameter MUST be unique each time. This ensures that the content-encryption key is not reused. An implementation SHOULD generate a random "salt" parameter for every message.

4.4. Data Encryption Limits

There are limits to the data that AEAD_AES_128_GCM can encipher. The maximum value for the record size is limited by the size of the "rs" field in the header (see Section 2.1), which ensures that the 2^36-31 limit for a single application of AEAD_AES_128_GCM is not reached [RFC5116]. In order to preserve a 2^-40 probability of indistinguishability under chosen plaintext attack (IND-CPA), the total amount of plaintext that can be enciphered with the key derived from the same input-keying material and salt MUST be less than 2^44.5 blocks of 16 octets [AEBounds].

If the record size is a multiple of 16 octets, this means that 398 terabytes can be encrypted safely, including padding and overhead. However, if the record size is not a multiple of 16 octets, the total amount of data that can be safely encrypted is reduced because partial AES blocks are encrypted. The worst case is a record size of 18 octets, for which at most 74 terabytes of plaintext can be encrypted, of which at least half is padding.

4.5. Content Integrity

This mechanism only provides content-origin authentication. The authentication tag only ensures that an entity with access to the content-encryption key produced the encrypted data.

Any entity with the content-encryption key can, therefore, produce content that will be accepted as valid. This includes all recipients of the same HTTP message.

Furthermore, any entity that is able to modify both the Content-Encoding header field and the HTTP message body can replace the contents. Without the content-encryption key or the input-keying material, modifications to, or replacement of, parts of a payload body are not possible.

4.6. Leaking Information in Header Fields

Because only the payload body is encrypted, information exposed in header fields is visible to anyone who can read the HTTP message. This could expose side-channel information.

For example, the Content-Type header field can leak information about the payload body.

There are a number of strategies available to mitigate this threat, depending upon the application's threat model and the users' tolerance for leaked information:

1. Determine that it is not an issue. For example, if it is expected that all content stored will be "application/json", or another very common media type, exposing the Content-Type header field could be an acceptable risk.

2. If it is considered sensitive information and it is possible to determine it through other means (e.g., out of band, using hints in other representations, etc.), omit the relevant headers, and/or normalize them. In the case of Content-Type, this could be accomplished by always sending Content-Type: application/octet-stream (the most generic media type), or no Content-Type at all.

3. If it is considered sensitive information and it is not possible to convey it elsewhere, encapsulate the HTTP message using the application/http media type (see Section 8.3.2 of [RFC7230]), encrypting that as the payload of the "outer" message.

4.7. Poisoning Storage

This mechanism only offers data-origin authentication; it does not perform authentication or authorization of the message creator, which could still need to be performed (e.g., by HTTP authentication [RFC7235]).

This is especially relevant when an HTTP PUT request is accepted by a server without decrypting the payload; if the request is unauthenticated, it becomes possible for a third party to deny service and/or poison the store.

4.8. Sizing and Timing Attacks

Applications using this mechanism need to be aware that the size of encrypted messages, as well as their timing, HTTP methods, URIs and so on, may leak sensitive information. See, for example, [NETFLIX] or [CLINIC].

This risk can be mitigated through the use of the padding that this mechanism provides. Alternatively, splitting up content into segments and storing them separately might reduce exposure. HTTP/2 [RFC7540] combined with TLS [RFC5246] might be used to hide the size of individual messages.

Developing a padding strategy is difficult. A good padding strategy can depend on context. Common strategies include padding to a small set of fixed lengths, padding to multiples of a value, or padding to powers of 2. Even a good strategy can still cause size information to leak if processing activity of a recipient can be observed. This is especially true if the trailing records of a message contain only padding. Distributing non-padding data across records is recommended to avoid leaking size information.

5. IANA Considerations

5.1. The "aes128gcm" HTTP Content Coding

This memo registers the "aes128gcm" HTTP content coding in the "HTTP Content Coding Registry", as detailed in Section 2.

- o Name: aes128gcm
- o Description: AES-GCM encryption with a 128-bit content-encryption key
- o Reference: this specification

6. References

6.1. Normative References

[FIPS180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS180-4, August 2015, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

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

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[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, <http://www.rfc-editor.org/info/rfc7231>.

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

6.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", DOI 10.1007/978-3-319-08506-7_8, March 2014, <https://arxiv.org/abs/1403.0297>.

[NETFLIX] Reed, A. and M. Kranch, "Identifying HTTPS-Protected Netflix Videos in Real-Time", Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy CODASPY ’17, DOI 10.1145/3029806.3029821, 2017.

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

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <http://www.rfc-editor.org/info/rfc4880>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[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, <http://www.rfc-editor.org/info/rfc7233>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <http://www.rfc-editor.org/info/rfc7235>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <http://www.rfc-editor.org/info/rfc7516>.

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

[XMLENC] Eastlake, D., Reagle, J., Hirsch, F., and T. Roessler, "XML Encryption Syntax and Processing Version 1.1", World Wide Web Consortium Recommendation REC-xmlenc-core1-20130411, April 2013, <http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411>.

Appendix A. JWE Mapping

The "aes128gcm" content coding can be considered as a sequence of JSON Web Encryption (JWE) [RFC7516] objects, each corresponding to a single fixed-size record that includes trailing padding. The following transformations are applied to a JWE object that might be expressed using the JWE Compact Serialization:

- The JWE Protected Header is fixed to the value { "alg": "dir", "enc": "A128GCM" }, describing direct encryption using AES-GCM with a 128-bit content-encryption key. This header is not transmitted, it is instead implied by the value of the Content-Encoding header field.

- The JWE Encrypted Key is empty, as stipulated by the direct encryption algorithm.

- The JWE Initialization Vector ("iv") for each record is set to the exclusive-or of the 96-bit record sequence number, starting at zero, and a value derived from the input-keying material (see Section 2.3). This value is also not transmitted.

- The final value is the concatenated header, JWE Ciphertext, and JWE Authentication Tag, all expressed without base64url encoding. The "." separator is omitted, since the length of these fields is known.

Thus, the example in Section 3.1 can be rendered using the JWE Compact Serialization as:

eyAiYWxnIjogImRpciIsICJlbmMiOiAiQTEyOEdDTSIgfQ..Bcs8gkIRKLI8GeI8.
-NAVub2qFgBEuQKRapoZuw.4jGQi9rcwQHU8P6XLxOGOA

Where the first line represents the fixed JWE Protected Header, an empty JWE Encrypted Key, and the algorithmically determined JWE Initialization Vector. The second line contains the encoded body, split into JWE Ciphertext and JWE Authentication Tag.

Acknowledgements

Mark Nottingham was an original author of this document.

The following people provided valuable input: Richard Barnes, David Benjamin, Peter Beverloo, JR Conlin, Mike Jones, Stephen Farrell, Adam Langley, James Manger, John Mattsson, Julian Reschke, Eric Rescorla, Jim Schaad, and Magnus Westerlund.

Author’s Address

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com