

Encrypted Content-Encoding for HTTP
draft-ietf-httpbis-encryption-encoding-01

Abstract

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

Note to Readers

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/> .

Working Group information can be found at <http://httpwg.github.io/> ; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/encryption> .

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 <http://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 September 21, 2016.

Copyright Notice

Copyright (c) 2016 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	3
1.1.	Notational Conventions	3
2.	The "aesgcm" HTTP Content Encoding	4
3.	The Encryption HTTP Header Field	5
3.1.	Encryption Header Field Parameters	6
3.2.	Content Encryption Key Derivation	7
3.3.	Nonce Derivation	7
4.	Crypto-Key Header Field	8
4.1.	Explicit Key	9
4.2.	Diffie-Hellman	9
4.3.	Pre-shared Authentication Secrets	10
5.	Examples	11
5.1.	Successful GET Response	11
5.2.	Encryption and Compression	12
5.3.	Encryption with More Than One Key	12
5.4.	Encryption with Explicit Key	12
5.5.	Encryption with Multiple Records	13
5.6.	Diffie-Hellman Encryption	13
5.7.	Diffie-Hellman with Authentication Secret	14
6.	Security Considerations	14
6.1.	Key and Nonce Reuse	15
6.2.	Content Integrity	15
6.3.	Leaking Information in Headers	15
6.4.	Poisoning Storage	16
6.5.	Sizing and Timing Attacks	16
7.	IANA Considerations	16
7.1.	The "aesgcm" HTTP Content Encoding	16
7.2.	Encryption Header Fields	17
7.3.	The HTTP Encryption Parameter Registry	17
7.3.1.	keyid	18
7.3.2.	salt	18
7.3.3.	rs	18
7.4.	The HTTP Crypto-Key Parameter Registry	18
7.4.1.	keyid	19
7.4.2.	aesgcm	19
7.4.3.	dh	19
8.	References	19
8.1.	Normative References	19

Thomson

Expires September 21, 2016

[Page 2]

8.2. Informative References	20
Appendix A. JWE Mapping	21
Appendix B. Intermediate Values for Encryption	22
Appendix C. Acknowledgements	23
Author's Address	23

[1. Introduction](#)

It is sometimes desirable to encrypt the contents of a HTTP message (request or response) so that when the payload is stored (e.g., with a 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 TLS [[RFC5246](#)], since it only encrypts the channel between the client and server.

This document specifies a content-coding ([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](#)] - which are not suited to stream processing, which is necessary for HTTP. The format described here cleaves 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 as 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. Though a complete key management system is not described, this document defines an Crypto-Key header field that can be used to convey keying material.

[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](#)].

Base64url encoding is defined in [Section 2 of \[RFC7515\]](#).

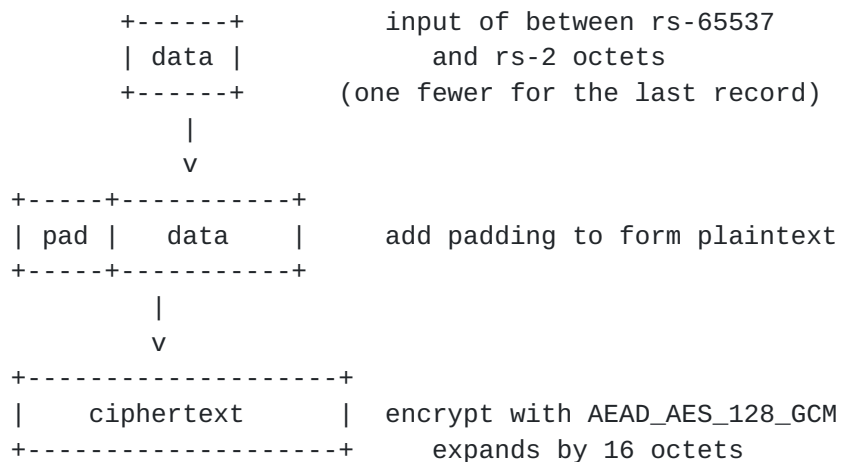
2. The "aesgcm" HTTP Content Encoding

The "aesgcm" 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.

When this content-coding is in use, the Encryption header field ([Section 3](#)) describes how encryption has been applied. The Crypto-Key header field ([Section 4](#)) can be included to describe how the content encryption key is derived or retrieved.

The "aesgcm" content-coding uses a single fixed set of encryption primitives. Cipher suite 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 "aesgcm" content-coding uses a fixed record size. The resulting encoding is a series of fixed-size records, with a final record that is one or more octets shorter than a fixed sized record.



The record size determines the length of each portion of plaintext that is enciphered, with the exception of the final record, which is necessarily smaller. The record size defaults to 4096 octets, but can be changed using the "rs" parameter on the Encryption header field.

AEAD_AES_128_GCM expands ciphertext to be 16 octets longer than its input plaintext. Therefore, the length of each enciphered record

other than the last is equal to the value of the "rs" parameter plus 16 octets. A receiver MUST fail to decrypt if the final record ciphertext is 16 octets or less in size. Valid records always contain at least one byte of padding and a 16 octet authentication tag.

Each record contains between 2 and 65537 octets of padding, inserted into a record before the enciphered content. Padding consists of a two octet unsigned integer in network byte order, followed that number of zero-valued octets. A receiver MUST fail to decrypt if any padding octet other than the first two are non-zero, or a record has more padding than the record size can accommodate.

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 3.3](#).

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

A sequence of full-sized records can be truncated to produce a shorter sequence of records with valid authentication tags. To prevent an attacker from truncating a stream, an encoder MUST append a record that contains only padding and is smaller than the full record size if the final record ends on a record boundary. A receiver MUST treat the stream as failed due to truncation if the final record is the full record size.

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. However, without data from adjacent ranges, partial records cannot be used. Thus, it is best if records start and end on multiples of the record size, plus the 16 octet authentication tag size.

3. The Encryption HTTP Header Field

The "Encryption" HTTP header field describes the encrypted content encoding(s) that have been applied to a payload body, and therefore how those content encoding(s) can be removed.

The "Encryption" header field uses the extended ABNF syntax defined in [Section 1.2 of \[RFC7230\]](#) and the "parameter" rule from [\[RFC7231\]](#)

```
Encryption = #encryption_params  
encryption_params = [ parameter *( ";" parameter ) ]
```


If the payload is encrypted more than once (as reflected by having multiple content-codings that imply encryption), each application of the content encoding is reflected in the Encryption header field, in the order in which they were applied.

Encryption header field values with multiple instances of the same parameter name are invalid.

The Encryption header MAY be omitted if the sender does not intend for the immediate recipient to be able to decrypt the payload body. Alternatively, the Encryption header field MAY be omitted if the sender intends for the recipient to acquire the header field by other means.

Servers processing PUT requests MUST persist the value of the Encryption header field, unless they remove the content-coding by decrypting the payload.

3.1. Encryption Header Field Parameters

The following parameters are used in determining the content encryption key that is used for encryption:

keyid: The "keyid" parameter contains a string that identifies the keying material that is used. The "keyid" parameter SHOULD be included, unless key identification is guaranteed by other means. The "keyid" parameter MUST be used if keying material included in an Crypto-Key header field is needed to derive the content encryption key.

salt: The "salt" parameter contains a base64url-encoded octets [[RFC7515](#)] that is used as salt in deriving a unique content encryption key (see [Section 3.2](#)). The "salt" parameter MUST be present, and MUST be exactly 16 octets long when decoded. The "salt" parameter 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 encoding ensures that content encryption key reuse is highly unlikely.

rs: The "rs" parameter contains a positive decimal integer that describes the record size in octets. This value MUST be greater than 1. If the "rs" parameter is absent, the record size defaults to 4096 octets.

3.2. 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 decoded value of the "salt" parameter using the HMAC-based key derivation function (HKDF) described in [RFC5869] using the SHA-256 hash algorithm [FIPS180-4].

The decoded value of the "salt" parameter is the salt input to HKDF function. The keying material identified by the "keyid" parameter is the input keying material (IKM) to HKDF. Input keying material can either be prearranged, or can be described using the Crypto-Key header field ([Section 4](#)). The first step of HKDF is therefore:

$$\text{PRK} = \text{HMAC-SHA-256}(\text{salt}, \text{IKM})$$

The info parameter to HKDF is set to the ASCII-encoded string "Content-Encoding: aesgcm", a single zero octet and an optional context string:

$$\text{cek_info} = \text{"Content-Encoding: aesgcm"} \parallel 0x00 \parallel \text{context}$$

Unless otherwise specified, the context is a zero length octet sequence. Specifications that use this content encoding MAY specify the use of an expanded context to cover additional inputs in the key derivation.

AEAD_AES_128_GCM requires a 16 octet (128 bit) content encryption key, 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:

$$\text{CEK} = \text{HMAC-SHA-256}(\text{PRK}, \text{cek_info} \parallel 0x01)$$

3.3. 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 is produced from the record sequence number and a value derived from the input keying material.

The input keying material and salt values are input to HKDF with different info and length parameters.

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

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

The context for nonce derivation SHOULD be the same as is used for content encryption key derivation.

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
```

4. Crypto-Key Header Field

An Crypto-Key header field can be used to describe the input keying material used in the Encryption header field.

The Crypto-Key header field uses the extended ABNF syntax defined in [Section 1.2 of \[RFC7230\]](#) and the "parameter" rule from [\[RFC7231\]](#).

```
Crypto-Key = #crypto_key_params  
crypto_key_params = [ parameter *( ";" parameter ) ]
```

keyid: The "keyid" parameter corresponds to the "keyid" parameter in the Encryption header field.

aesgcm: The "aesgcm" parameter contains the base64url-encoded octets [\[RFC7515\]](#) of the input keying material.

dh: The "dh" parameter contains an ephemeral Diffie-Hellman share. This form of the header field can be used to encrypt content for a specific recipient.

Crypto-Key header field values with multiple instances of the same parameter name are invalid.

The input keying material used by the key derivation (see [Section 3.2](#)) can be determined based on the information in the

Crypto-Key header field. The method for key derivation depends on the parameters that are present in the header field.

The value or values provided in the Crypto-Key header field is valid only for the current HTTP message unless additional information indicates a greater scope.

Note that different methods for determining input keying material will produce different amounts of data. The HKDF process ensures that the final content encryption key is the necessary size.

Alternative methods for determining input keying material MAY be defined by specifications that use this content-encoding.

4.1. Explicit Key

The "aesgcm" parameter is decoded and used as the input keying material for the "aesgcm" content encoding. The "aesgcm" parameter MUST decode to at least 16 octets in order to be used as input keying material for "aesgcm" content encoding.

Other key determination parameters can be ignored if the "aesgcm" parameter is present.

4.2. Diffie-Hellman

The "dh" parameter is included to describe a Diffie-Hellman share, either modp (or finite field) Diffie-Hellman [[DH](#)] or elliptic curve Diffie-Hellman (ECDH) [[RFC4492](#)].

This share is combined with other information at the recipient to determine the HKDF input keying material. In order for the exchange to be successful, the following information MUST be established out of band:

- o Which Diffie-Hellman form is used.
- o The modp group or elliptic curve that will be used.
- o A label that uniquely identifies the group. This label will be expressed as a sequence of octets and MUST NOT include a zero-valued octet.
- o The format of the ephemeral public share that is included in the "dh" parameter. This encoding MUST result in a single, canonical sequence of octets. For instance, using ECDH both parties need to agree whether this is an uncompressed or compressed point.

In addition to identifying which content-encoding this input keying material is used for, the "keyid" parameter is used to identify this additional information at the receiver.

The intended recipient recovers their private key and are then able to generate a shared secret using the designated Diffie-Hellman process.

The context for content encryption key and nonce derivation (see [Section 3.2](#)) is set to include the means by which the keys were derived. The context is formed from the concatenation of group label, a single zero octet, the length of the public key of the recipient, the public key of the recipient, the length of the public key of the sender, and the public key of the sender. The public keys are encoded into octets as defined for the group when determining the context string.

```
context = label || 0x00 ||  
          length(recipient_public) || recipient_public ||  
          length(sender_public) || sender_public
```

The two length fields are encoded as a two octet unsigned integer in network byte order.

Specifications that rely on an Diffie-Hellman exchange for determining input keying material MUST either specify the parameters for Diffie-Hellman (label, group parameters, or curves and point format) that are used, or describe how those parameters are negotiated between sender and receiver.

4.3. Pre-shared Authentication Secrets

Key derivation MAY be extended to include an additional authentication secret. Such a secret is shared between the sender and receiver of a message using other means.

A pre-shared authentication secret is not explicitly signaled in either the Encryption or Crypto-Key header fields. Use of this additional step depends on prior agreement.

When a shared authentication secret is used, the keying material produced by the key agreement method (e.g., Diffie-Hellman, explicit key, or otherwise) is combined with the authentication secret using HKDF. The output of HKDF is the input keying material used to derive the content encryption key and nonce [Section 3.2](#).

The authentication secret is used as the "salt" parameter to HKDF, the raw keying material (e.g., Diffie-Hellman output) is used as the "IKM" parameter, the ASCII-encoded string "Content-Encoding: auth" with a terminal zero octet is used as the "info" parameter, and the length of the output is 32 octets (i.e., the entire output of the underlying SHA-256 HMAC function):

```
auth_info = "Content-Encoding: auth" || 0x00
IKM = HKDF(authentication, raw_key, auth_info, 32)
```

This invocation of HKDF does not take the same context that is provided to the final key derivation stages. Alternatively, this phase can be viewed as always having a zero-length context.

Note that in the absence of an authentication secret, the input keying material is simply the raw keying material:

```
IKM = raw_key
```

5. Examples

This section shows a few examples of the content encoding.

Note: All binary values in the examples in this section use the URL and filename safe variant of base64 [[RFC4648](#)]. This includes the bodies of requests. Whitespace in these values is added to fit formatting constraints.

5.1. Successful GET Response

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Encoding: aesgcm
Connection: close
Encryption: keyid="http://example.org/bob/keys/123";
           salt="XZwpw6o37R-6qoZjw6KwAw"
```

[encrypted payload]

Here, a successful HTTP GET response has been encrypted using input keying material that is identified by a URI.

Note that the media type has been changed to "application/octet-stream" to avoid exposing information about the content.

5.2. Encryption and Compression

In this example, a response is first compressed, then encrypted. Note that this particular encoding might compromise confidentiality if the contents of the response could be influenced by an attacker.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Encoding: gzip, aesgcm
Transfer-Encoding: chunked
Encryption: keyid="mailto:me@example.com";
           salt="m2hJ_NttRtFyUiMRPwfpHA"
```

[encrypted payload]

5.3. Encryption with More Than One Key

Here, a PUT request has been encrypted twice with different input keying material; decrypting twice is necessary to read the content. The outer layer of encryption uses a 1200 octet record size.

```
PUT /thing HTTP/1.1
Host: storage.example.com
Content-Type: application/http
Content-Encoding: aesgcm, aesgcm
Content-Length: 1235
Encryption: keyid="mailto:me@example.com";
           salt="Nfz0euV5USPRA-n_9s1Lag",
           keyid="http://example.org/bob/keys/123";
           salt="bDMSGoc2uobK_IhavSHsHA"; rs=1200
```

[encrypted payload]

5.4. Encryption with Explicit Key

This example shows the UTF-8 encoded string "I am the walrus" encrypted using an directly provided value for the input keying material. The content body contains a single record only and is shown here using base64url encoding for presentation reasons.


```
HTTP/1.1 200 OK
Content-Length: 33
Content-Encoding: aesgcm
Encryption: keyid="a1"; salt="vr0o6Uq3w_KDWeatc27mUg"
Crypto-Key: keyid="a1"; aesgcm="csPJEXBYA5U-Tal9EdJi-w"

VDeU0XxaJk0JDAxPl7h9JD5V8N43RorP7PfpPdZZQuwF
```

5.5. Encryption with Multiple Records

This example shows the same encrypted message, but split into records of 10 octets each. The first record includes a single additional octet of padding, which causes the end of the content to align with a record boundary, forcing the creation of a third record that contains only padding.

```
HTTP/1.1 200 OK
Content-Length: 70
Content-Encoding: aesgcm
Encryption: keyid="a1"; salt="4pdat984KmT9BWsU3np0nw"; rs=10
Crypto-Key: keyid="a1"; aesgcm="B03ZVPxUlnLORbVGMpbT1Q"

uzLfrZ4cbMTC6hlUqHz4NvwZshF1TN3o2RLr6FrIu0KEfl2VrM_jYgoiIyEo
Zvc-ZGwV-RMJejG4M6ZfGysBAdhpPqrLzw
```

5.6. Diffie-Hellman Encryption

```
HTTP/1.1 200 OK
Content-Length: 33
Content-Encoding: aesgcm
Encryption: keyid="dhkey"; salt="Qg61ZJRva_XBE9IEUelU3A"
Crypto-Key: keyid="dhkey";
            dh="BDgpRKok2GZZDmS4r63vbJSUtcQx4Fq1V58-6-3NbZzS
            TlZsQiCEDTQy3CZ0ZMsqeqsEb7qw2blQHA4S48fynTk"

yqD2bapcx14XxUbtwjiGx69eHE3Yd6AqXcwBpT2Kd1uy
```

This example shows the same string, "I am the walrus", encrypted using ECDH over the P-256 curve [[FIPS186](#)], which is identified with the label "P-256" encoded in ASCII. The content body is shown here encoded in URL-safe base64url for presentation reasons only.

The receiver (in this case, the HTTP client) uses a key pair that is identified by the string "dhkey" and the sender (the server) uses a key pair for which the public share is included in the "dh" parameter above. The keys shown below use uncompressed points [[X9.62](#)] encoded

using base64url. Line wrapping is added for presentation purposes only.

Receiver:

private key: 9FWl15_QUQAWDaD3k3l50ZBZQJ4au27F1V4F0uLSD_M
 public key: BCEkBjzL8Z3C-oi2Q7oE5t2Np-p7osjGLg93qUP0wvqR
 T21EEWyf0cQDQcakQMqz4hQKY0Q3il2nNZct4HgAUQU

Sender:

private key: vG7TmzUX9NfVR4XUGBkLAFu8iDyQe-q_165JkkN0Vlw
 public key: <the value of the "dh" parameter>

5.7. Diffie-Hellman with Authentication Secret

This example shows the same receiver key pair from [Section 5.6](#), but with a shared authentication secret of "R29vIGdvbyBnJyBqb29iIQ".

```
HTTP/1.1 200 OK
Content-Length: 33
Content-Encoding: aesgcm
Encryption: keyid="dhkey"; salt="lngarbyKfMoi9Z75xYXmkg"
Crypto-Key: keyid="dhkey";
             dh="BNoRDbb84JGm8g5Z5CFxurSqsXWJ11ItfXEWYVLE85Y7
             CYkDjXsIEc4aqxYaQ1G8BqkXCJ6DPpDrWtdWj_mugHU"
```

6nqAQUME8hNqw5J3kl8cpVVJylXKYqZ0eseZG8UueKpA

The sender's private key used in this example is "nCScck-QpEjm00lT-rQ38nZzvdPlqa00Zy0i6m20JvY". Intermediate values for this example are included in [Appendix B](#).

6. 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.

6.1. 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 encoding. Since input keying material can be reused, a unique "salt" parameter is needed to ensure a content encryption key is not reused.

If a content encryption key is reused - that is, if input keying material and salt 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; a counter could achieve the same result.

6.2. 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 Encryption 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.

6.3. Leaking Information in Headers

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 ([Section 8.3.2 of \[RFC7230\]](#)), encrypting that as the payload of the "outer" message.

[6.4.](#) Poisoning Storage

This mechanism only offers encryption of content; it does not perform authentication or authorization, which still needs to be performed (e.g., by HTTP authentication [[RFC7235](#)]).

This is especially relevant when a HTTP PUT request is accepted by a server; if the request is unauthenticated, it becomes possible for a third party to deny service and/or poison the store.

[6.5.](#) 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.

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

[7.](#) IANA Considerations

[7.1.](#) The "aesgcm" HTTP Content Encoding

This memo registers the "encrypted" HTTP content-coding in the HTTP Content Codings Registry, as detailed in [Section 2](#).

- o Name: aesgcm

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

7.2. Encryption Header Fields

This memo registers the "Encryption" HTTP header field in the Permanent Message Header Registry, as detailed in [Section 3](#).

- o Field name: Encryption
- o Protocol: HTTP
- o Status: Standard
- o Reference: this specification
- o Notes:

This memo registers the "Crypto-Key" HTTP header field in the Permanent Message Header Registry, as detailed in [Section 4](#).

- o Field name: Crypto-Key
- o Protocol: HTTP
- o Status: Standard
- o Reference: this specification
- o Notes:

7.3. The HTTP Encryption Parameter Registry

This memo establishes a registry for parameters used by the "Encryption" header field under the "Hypertext Transfer Protocol (HTTP) Parameters" grouping. The "Hypertext Transfer Protocol (HTTP) Encryption Parameters" registry operates under an "Specification Required" policy [[RFC5226](#)].

Entries in this registry are expected to include the following information:

- o Parameter Name: The name of the parameter.
- o Purpose: A brief description of the purpose of the parameter.

- o Reference: A reference to a specification that defines the semantics of the parameter.

The initial contents of this registry are:

7.3.1. keyid

- o Parameter Name: keyid
- o Purpose: Identify the key that is in use.
- o Reference: this document

7.3.2. salt

- o Parameter Name: salt
- o Purpose: Provide a source of entropy for derivation of a content encryption key. This value is mandatory.
- o Reference: this document

7.3.3. rs

- o Parameter Name: rs
- o Purpose: The size of the encrypted records.
- o Reference: this document

7.4. The HTTP Crypto-Key Parameter Registry

This memo establishes a registry for parameters used by the "Crypto-Key" header field under the "Hypertext Transfer Protocol (HTTP) Parameters" grouping. The "Hypertext Transfer Protocol (HTTP) Crypto-Key Parameters" operates under an "Specification Required" policy [[RFC5226](#)].

Entries in this registry are expected to include the following information:

- o Parameter Name: The name of the parameter.
- o Purpose: A brief description of the purpose of the parameter.
- o Reference: A reference to a specification that defines the semantics of the parameter.

The initial contents of this registry are:

7.4.1. keyid

- o Parameter Name: keyid
- o Purpose: Identify the key that is in use.
- o Reference: this document

7.4.2. aesgcm

- o Parameter Name: aesgcm
- o Purpose: Provide an explicit input keying material value for the aesgcm content encoding.
- o Reference: this document

7.4.3. dh

- o Parameter Name: dh
- o Purpose: Carry a modp or elliptic curve Diffie-Hellman share used to derive input keying material.
- o Reference: this document

8. References

8.1. Normative References

- [DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, V.IT-22 n.6 , June 1977.
- [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>>.
- [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>>.

- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", [RFC 4492](#), DOI 10.17487/RFC4492, May 2006, <<http://www.rfc-editor.org/info/rfc4492>>.
- [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>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", [RFC 7515](#), DOI 10.17487/RFC7515, May 2015, <<http://www.rfc-editor.org/info/rfc7515>>.

[8.2.](#) Informative References

- [FIPS186] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", NIST PUB 186-4 , July 2013.
- [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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", [BCP 26](#), [RFC 5226](#), DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.

- [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>>.
- [X9.62] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62 , 1998.
- [XMLENC] Eastlake, D., Reagle, J., Imamura, T., Dillaway, B., and E. Simon, "XML Encryption Syntax and Processing", W3C REC , December 2002, <<http://www.w3.org/TR/xmlenc-core/>>.

[Appendix A.](#) JWE Mapping

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

- o The JWE Protected Header is fixed to a 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.

- o The JWE Encrypted Key is empty, as stipulated by the direct encryption algorithm.
- o 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 3.3](#)). This value is also not transmitted.
- o The final value is the concatenated JWE Ciphertext and the JWE Authentication Tag, both expressed without URL-safe Base 64 encoding. The "." separator is omitted, since the length of these fields is known.

Thus, the example in [Section 5.4](#) can be rendered using the JWE Compact Serialization as:

```
eyJYXnIjogImRpciIsICJlbmMiOiAiQTEyOEdDTSIgfQ..31iQYc1v4a36EgyJ.  
VDeU0XxaJk0JDAXPl7h9JD4.VfDeN0aKz-z36T3WWULsBQ
```

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.

[Appendix B](#). Intermediate Values for Encryption

The intermediate values calculated for the example in [Section 5.7](#) are shown here. The following are inputs to the calculation:

Plaintext: SSBhbSB0aGUgd2FscnVz

Sender public key: BNoRDbb84JGm8g5Z5CFxurSqsXWJ11ItfXEWYVLE85Y7
CYkdjXsIEc4aqxYaQ1G8BqkXCJ6DPpDrWtdWj_mugHU

Sender private key: nCScek-QpEjm001T-rQ38nZzvdPlqa00Zy0i6m20JvY

Receiver public key: BCEkBjzL8Z3C-oi2Q7oE5t2Np-p7osjGLg93qUP0wvqR
T21EEwyf0cQDQcakQMqz4hQKY0Q3il2nNZct4HgAUQU

Receiver private key: 9FWl15_QUQAWDaD3k3l50ZBZQJ4au27F1V4F0uLSD_M

Salt: lngarbyKfMoi9Z75xYXmkg

Note that knowledge of just one of the private keys is necessary. The sender randomly generates the salt value, whereas salt is input to the receiver.

This produces the following intermediate values:

Shared secret (raw_key): RNjC-NVW4BGJbxWPW7G2mowsLeDa53LYKYm4-N0Q6Y

Input keying material (IKM): EhpZec37Ptm4IRD5-jtZ0q6r1iK5vYmY1tZwtN8fbZY

Context for content encryption key derivation:

Q29udGVudC1FbmNvZGluZzogYWVzZ2NtAFAtMjU2AABB BCEkBjzL8Z3C-
oi2Q7oE5t2Np-p7osjGLg93qUP0wvqR
T21EEWyf0cQDQcakQMqz4hQKY0Q3il2nNZct4HgAUQUA
QQTaEQ22_OCRpvIOWeQhcbq0qrF1iddSLX1xFmFSxPOW
OwmJA417CBHOGqsWGkNRvAapFwiegz6Q61rXVo_5roB1

Content encryption key (CEK): AN2-xhvFWeYh5z0fcDu0Ww

Context for nonce derivation: Q29udGVudC1FbmNvZGluZzogbm9uY2UAUC0yNT

YAAEEE ISQGPMvxncL6iLZDugTm3Y2n6nuiyMYuD3epQ_TC-pFP
bUQRbJ_RxANBxqRAYrPiFApg5DeKXac1ly3geABRBQBB
BNoRDbb84JGm8g5Z5CFxurSqsXWJ11ItfXEWYVLE85Y7
CYkDjXsIEc4aqxYaQ1G8BqkXCJ6DPpDrWtdWj_mugHU

Base nonce: JY10kw5rw1Drkg9J

When the CEK and nonce are used with AES GCM and the padded plaintext of AABJIGFtIHRoZSB3YWxydXM, the final ciphertext is 6nqAQUME8hNqw5J3kl8cpVVJylXKYqZ0eseZG8UueKpA, as shown in the example.

[Appendix C](#). Acknowledgements

Mark Nottingham was an original author of this document.

The following people provided valuable input: Richard Barnes, David Benjamin, Peter Beverloo, Mike Jones, Stephen Farrell, Adam Langley, John Mattsson, Eric Rescorla, and Jim Schaad.

Author's Address

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

