HTTP Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 12, 2017

M. Thomson
Mozilla

October 9, 2016

Encrypted Content-Encoding for HTTP
draft-ietf-httpbis-encryption-encoding-03

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

Copyright Notice

Table of Contents

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 Coding

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 either a single record, or a series of fixed-size records. The final record, or a lone record, MUST be shorter than the fixed record size.

```
      +-----------+       content is rs octets minus padding
      |   data    |       of between 2 and 65537 octets;
      +-----------+       the last record is smaller
           |
           v
+-----+-----------+       add padding to get rs octets;
| pad |   data    |       the last record contains
+-----+-----------+       up to rs minus 1 octets
         |
         v
+--------------------+    encrypt with AEAD_AES_128_GCM;
|    ciphertext      |    final size is rs plus 16 octets
+--------------------+    the last record is smaller
```

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 produces ciphertext 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 less than 18 octets in size. Valid records always contain at least two octets 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 range requests start and end on multiples of the record size, plus the 16 octet authentication tag size.

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 depend on being able to pad by arbitrary amounts cannot increase the record size beyond 65537 octets.

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

3. The Encryption HTTP Header Field

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

The "Encryption" header field uses the extended ABNF syntax defined in Section 1.2 of [RFC7230] and the "parameter" and "OWS" rules from [RFC7231].

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

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

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

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 identifies the keying material that is used. When the Crypto-Key header field is used, the "keyid" identifies a matching value in that field. The "keyid" parameter MUST be used if keying material included in an Crypto-Key header field is needed to derive the content encryption key. The "keyid" parameter can also be used to identify keys in an application-specific fashion.

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 coding 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. For the "aesgcm" content coding, this value MUST NOT be greater than 2^36-31 (see Section 6.2). The "rs" parameter is optional. 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 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: aesgcm", a single zero octet and an optional context string:

```
cek_info = "Content-Encoding: aesgcm" || 0x00 || context
```

Unless otherwise specified, the context is a zero length octet sequence. Specifications that use this content coding 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 (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)
```

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 an context:

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

The context for nonce derivation is 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
```

This nonce construction prevents removal or reordering of records. However, it permits truncation of the tail of the sequence (see Section 2 for how this is avoided).

4. Crypto-Key Header Field

A 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" and "OWS" rules from [RFC7231].

```
Crypto-Key = #crypto_key_params
crypto_key_params = [ parameter *( OWS ";" OWS 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 for the "aesgcm" content coding.

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 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.

Alternative methods for determining input keying material MAY be defined by specifications that use this content coding. This document only defines the use of the "aesgcm" parameter which describes an explicit key.

The "aesgcm" parameter MUST decode to at least 16 octets in order to be used as input keying material for "aesgcm" content coding.

5. Examples

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

Note: All binary values in the examples in this section use base64url encoding [RFC7515]. This includes the bodies of requests. Whitespace and line wrapping is added to fit formatting constraints.

5.1. Encryption of a Response

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

The encrypted data in this example is the UTF-8 encoded string "I am the walrus". The input keying material is included in the Crypto-Key header field. The content body contains a single record only and is shown here using base64url encoding for presentation reasons.

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

VDeU0XxaJkOJDAxPl7h9JD5V8N43RorP7PfpPdZZQuwF
```

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

5.2. 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="BO3ZVPxUlnLORbVGMpbT1Q"

uzLfrZ4cbMTC6hlUqHz4NvWZshFlTN3o2RLr6FrIuOKEfl2VrM_jYgoiIyEo
Zvc-ZGwV-RMJejG4M6ZfGysBAdhpPqrLzw
```

5.3. Encryption and Compression

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

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

[encrypted payload]
```

5.4. 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="NfzOeuV5USPRA-n_9s1Lag",
            keyid="bob/keys/123";
            salt="bDMSGoc2uobK_IhavSHsHA"; rs=1200

[encrypted payload]
```

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 coding. 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. Data Encryption Limits

There are limits to the data that AEAD_AES_128_GCM can encipher. The maximum record size is 2^36-31 [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 MUST be less than 2^44.5 blocks [AEBounds].

If rs is a multiple of 16 octets, this means 398 terabytes can be encrypted safely, including padding. However, if the record size is a multiple of 16 octets, the total amount of data that can be safely encrypted is reduced. The worst case is a record size of 3 octets, for which at most 74 terabytes of plaintext can be encrypted, of which at least two-thirds is padding.

6.3. 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.

6.4. 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.

6.5. 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.

6.6. 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 Coding

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

o Name: aesgcm

- Description: AES-GCM encryption with a 128-bit content encryption key

- 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.

- Field name: Encryption

- Protocol: HTTP

- Status: Standard

- Reference: this specification

- Notes:

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

- Field name: Crypto-Key

- Protocol: HTTP

- Status: Standard

- Reference: this specification

- 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:

- Parameter Name: The name of the parameter.

- 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 coding.
- o Reference: this document

8. References

8.1. 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>.

[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>.

[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>.

[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>.

[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

[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>.

[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., 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 coding 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:

- 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 3.3). This value is also not transmitted.

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

Thus, the example in Section 5.1 can be rendered using the JWE Compact Serialization as:


```
eyAiYWxnIjogImRpciIsICJlbmMiOiAiQTEyOEdDTSIgfQ..31iQYc1v4a36EgyJ.
VDeU0XxaJkOJDAxPl7h9JD4.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. 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