HTTP Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 20, 2017

M. Thomson
Mozilla

April 18, 2017

Encrypted Content-Encoding for HTTP
draft-ietf-httpbis-encryption-encoding-09

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 October 20, 2017.

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 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 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. In particular, a key management system is not described.

1.1. Notational Conventions

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

2. The "aes128gcm" HTTP Content Coding

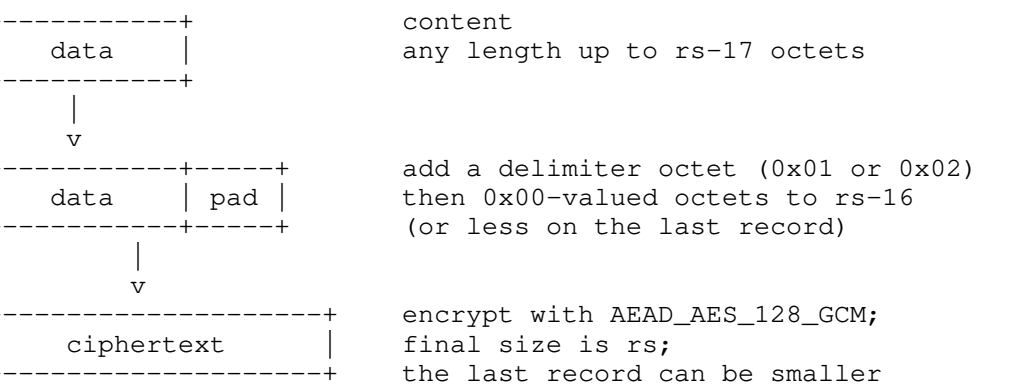
The "aes128gcm" HTTP content coding indicates that a payload has been encrypted using Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) as identified as AEAD_AES_128_GCM in [RFC5116], Section 5.1. The AEAD_AES_128_GCM algorithm uses a 128 bit content encryption key.

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 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 "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).



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.

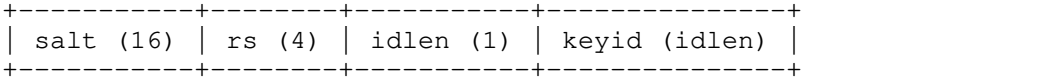
A consequence of this record structure is that range requests [RFC7233] and random access to encrypted payload bodies are possible at the granularity of the record size. Partial records at the ends of a range cannot be decrypted. Thus, it is best if range requests start and end on record boundaries. Note however 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 depending 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: 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 [RFC3629] encoded 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 HKDF function. 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.

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", terminated by a 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 Base 64 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 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 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 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.

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 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 (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 a 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 Codings Registry, as detailed in Section 2.

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

6. References

6.1. Normative References

[FIPS180-4] National Institute of Standards and Technology, U.S. Department of Commerce, "NIST FIPS 180-4, Secure Hash Standard", DOI 10.6028/NIST.FIPS.180-4, 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>.

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", 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., Roessler, T., Imamura, T., Dillaway, B., Simon, E., Yiu, K., and M. Nystroem, "XML Encryption Syntax and Processing", W3C Recommendation REC-xmlenc-core1-20130411, January 2013, <https://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) objects [RFC7516], 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.

Appendix B. 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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 21, 2016

M. Thomson
Mozilla

March 20, 2016

Message Encryption for Web Push
draft-ietf-webpush-encryption-02

Abstract

A message encryption scheme is described for the Web Push protocol. This scheme provides confidentiality and integrity for messages sent from an Application Server to a User Agent.

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."

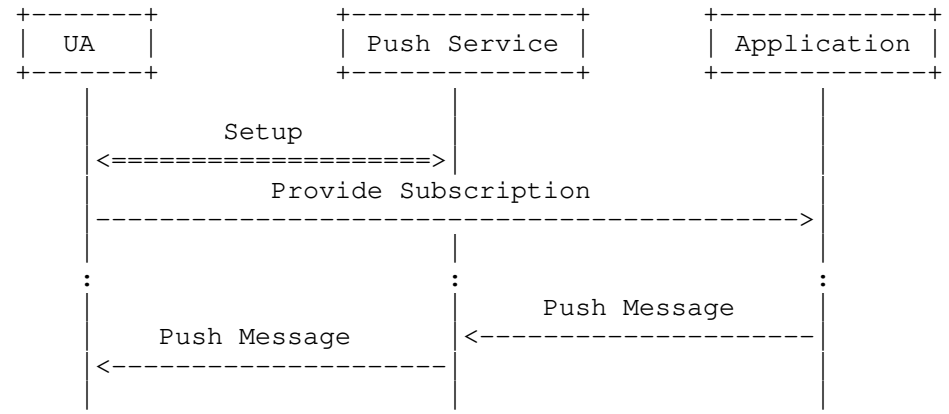
This Internet-Draft will expire on September 21, 2016.

Copyright Notice

Table of Contents

1. Introduction

The Web Push protocol [I-D.ietf-webpush-protocol] is an intermediated protocol by necessity. Messages from an Application Server are delivered to a User Agent via a Push Service.



This document describes how messages sent using this protocol can be secured against inspection, modification and falsification by a Push Service.

Web Push messages are the payload of an HTTP message [RFC7230]. These messages are encrypted using an encrypted content encoding [I-D.ietf-httpbis-encryption-encoding]. This document describes how

this content encoding is applied and describes a recommended key management scheme.

For efficiency reasons, multiple users of Web Push often share a central agent that aggregates push functionality. This agent can enforce the use of this encryption scheme by applications that use push messaging. An agent that only delivers messages that are properly encrypted strongly encourages the end-to-end protection of messages.

A web browser that implements the Web Push API [API] can enforce the use of encryption by forwarding only those messages that were properly encrypted.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting, when they are capitalized, they have the special meaning described in [RFC2119].

2. Key Generation and Agreement

For each new subscription that the User Agent generates for an application, it also generates an asymmetric key pair for use in Diffie-Hellman (DH) [DH] or elliptic-curve Diffie-Hellman (ECDH) [ECDH]. The public key for this key pair can then be distributed by the application to the Application Server along with the URI of the subscription. The private key MUST remain secret.

This key pair is used with the Diffie-Hellman key exchange as described in Section 4.2 of [I-D.ietf-httpbis-encryption-encoding].

A User Agent MUST generate and provide a public key for the scheme described in Section 5.

The public key MUST be accompanied by a key identifier that can be used in the "keyid" parameter to identify which key is in use. Key identifiers need only be unique within the context of a subscription.

2.1. Diffie-Hellman Group Information

As described in [I-D.ietf-httpbis-encryption-encoding], use of Diffie-Hellman for key agreement requires that the receiver provide clear information about its chosen group and the format for the "dh" parameter with each potential sender.

This document only describes a single ECDH group and point format, described in Section 5. A specification that defines alternative

groups or formats MUST provide a means of indicating precisely which group and format is in use for every public key that is provided.

2.2. Key Distribution

The application using the subscription distributes the key identifier and public key along with other subscription information, such as the subscription URI and expiration time.

The communication medium by which an application distributes the key identifier and public key MUST be confidentiality protected for the reasons described in [I-D.ietf-webpush-protocol]. Most applications that use push messaging have a pre-existing relationship with an Application Server. Any existing communication mechanism that is authenticated and provides confidentiality and integrity, such as HTTPS [RFC2818], is sufficient.

2.3. Push Message Authentication

To ensure that push messages are correctly authenticated, a symmetric authentication secret is added to the information generated by a User Agent. The authentication secret is mixed into the key derivation process described in [I-D.ietf-httpbis-encryption-encoding].

The authentication secret ensures that exposure or leakage of the DH public key - which, as a public key, is not necessarily treated as a secret - does not enable an adversary to generate valid push messages.

A User Agent MUST generate and provide a hard to guess sequence of octets that is used for authentication of push messages. This SHOULD be generated by a cryptographically strong random number generator [RFC4086] and be at least 16 octets long.

3. Message Encryption

An Application Server that has the public key, group and format information plus the authentication secret can encrypt a message for the User Agent.

3.1. Key Derivation

The Application Server generates a new DH or ECDH key pair in the same group as the value generated by the User Agent.

From the newly generated key pair, the Application Server performs a DH or ECDH computation with the public key provided by the User Agent to find the input keying material for key derivation. The

Application Server then generates 16 octets of salt that is unique to the message. A random [RFC4086] salt is acceptable.

Web push uses the authentication secret defined in Section 4.3 of [I-D.ietf-httpbis-encryption-encoding]. This authentication secret (see Section 2.3) is generated by the user agent and shared with the application server.

3.2. Push Message Content Encryption

The Application Server then encrypts the payload. Header fields are populated with base64url encoded [RFC7515] values:

- the salt is added to the "salt" parameter of the Encryption header field; and
- the public key for its DH or ECDH key pair is placed in the "dh" parameter of the Crypto-Key header field.

An application server MUST encrypt a push message with a single record. This allows for a minimal receiver implementation that handles a single record. If the message is 4096 octets or longer, the "rs" parameter MUST be set to a value that is longer than the encrypted push message length.

Note that a push service is not required to support more than 4096 octets of payload body, which equates to 4080 octets of cleartext, so the "rs" parameter can be omitted for messages that fit within this limit.

An application server MUST NOT use other content encodings for push messages. In particular, content encodings that compress could result in leaking of push message contents. The Content-Encoding header field therefore has exactly one value, which is "aesgcm128". Multiple "aesgcm128" values are not permitted.

An application server MUST include exactly one entry in each of the Encryption and Crypto-Key header fields. This allows the "keyid" parameter to be omitted from both header fields.

An application server MUST NOT include an "aesgcm128" parameter in the Encryption header field.

4. Message Decryption

A User Agent decrypts messages are decrypted as described in [I-D.ietf-httpbis-encryption-encoding]. The authentication secret described in Section 3.1 is used in key derivation.

Note that the value of the "keyid" parameter is used to identify the correct share, if there are multiple values for the Crypto-Key header field.

A receiver is not required to support multiple records. Such a receiver MUST check that the record size is large enough to contain the entire payload body in a single record. The "rs" parameter MUST NOT be exactly equal to the length of the payload body minus the length of the authentication tag (16 octets); that length indicates that the message has been truncated.

5. Mandatory Group and Public Key Format

User Agents MUST expose an elliptic curve Diffie-Hellman share on the P-256 curve [FIPS186].

Public keys, such as are encoded into the "dh" parameter, MUST be in the form of an uncompressed point as described in [X.692] (that is, a 65 octet sequence that starts with a 0x04 octet).

The label for this curve is the string "P-256" encoded in ASCII (that is, the octet sequence 0x50, 0x2d, 0x32, 0x35, 0x36).

6. IANA Considerations

This document has no IANA actions.

7. Security Considerations

The security considerations of [I-D.ietf-httpbis-encryption-encoding] describe the limitations of the content encoding. In particular, any HTTP header fields are not protected by the content encoding scheme. A User Agent MUST consider HTTP header fields to have come from the Push Service. An application on the User Agent that uses information from header fields to alter their processing of a push message is exposed to a risk of attack by the Push Service.

The timing and length of communication cannot be hidden from the Push Service. While an outside observer might see individual messages intermixed with each other, the Push Service will see what Application Server is talking to which User Agent, and the subscription that is used. Additionally, the length of messages could be revealed unless the padding provided by the content encoding scheme is used to obscure length.

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.

[ECDH] SECG, "Elliptic Curve Cryptography", SEC 1 , 2000, <http://www.secg.org/>.

[FIPS186] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", NIST PUB 186-4 , July 2013.

[I-D.ietf-httpbis-encryption-encoding] Thomson, M., "Encrypted Content-Encoding for HTTP", draft-ietf-httpbis-encryption-encoding-01 (work in progress), March 2016.

[I-D.ietf-webpush-protocol] Thomson, M., Damaggio, E., and B. Raymor, "Generic Event Delivery Using HTTP Push", draft-ietf-webpush-protocol-03 (work in progress), February 2016.

[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>.

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

[X.692] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62 , 1998.

8.2. Informative References

[API] van Ouwerkerk, M. and M. Thomson, "Web Push API", 2015, <https://w3c.github.io/push-api/>.

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

[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>.

[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>.

Author's Address

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

WEBPUSH
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2016

M. Thomson
Mozilla
E. Damaggio
B. Raymor, Ed.
Microsoft
March 21, 2016

Generic Event Delivery Using HTTP Push
draft-ietf-webpush-protocol-04

Abstract

A simple protocol for the delivery of realtime events to user agents is described. This scheme uses HTTP/2 server push.

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 22, 2016.

Copyright Notice

Table of Contents

1. Introduction

Many applications on mobile and embedded devices require continuous access to network communications so that real-time events - such as incoming calls or messages - can be delivered (or "pushed") in a timely fashion. These devices typically have limited power reserves, so finding more efficient ways to serve application requirements greatly benefits the application ecosystem.

One significant contributor to power usage is the radio. Radio communications consume a significant portion of the energy budget on a wireless device.

Uncoordinated use of persistent connections or sessions from multiple applications can contribute to unnecessary use of the device radio, since each independent session independently incurs overheads. In particular, keep alive traffic used to ensure that middleboxes do not prematurely time out sessions, can result in significant waste. Maintenance traffic tends to dominate over the long term, since events are relatively rare.

Consolidating all real-time events into a single session ensures more efficient use of network and radio resources. A single service consolidates all events, distributing those events to applications as they arrive. This requires just one session, avoiding duplicated overhead costs.

The W3C Web Push API [API] describes an API that enables the use of a consolidated push service from web applications. This expands on that work by describing a protocol that can be used to:

- request the delivery of a push message to a user agent,
- create new push message delivery subscriptions, and
- monitor for new push messages.

Requesting the delivery of events is particularly important for the Web Push API. The subscription, management and monitoring functions are currently fulfilled by proprietary protocols; these are adequate, but do not offer any of the advantages that standardization affords.

This document intentionally does not describe how a push service is discovered. Discovery of push services is left for future efforts, if it turns out to be necessary at all. User agents are expected to be configured with a URL for a push service.

1.1. Conventions and Terminology

In cases where normative language needs to be emphasized, this document falls back on established shorthands for expressing interoperability requirements on implementations: the capitalized words "MUST", "MUST NOT", "SHOULD" and "MAY". The meaning of these is described in [RFC2119].

This document defines the following terms:

application: Both the sender and ultimate consumer of push messages. Many applications have components that are run on a user agent and other components that run on servers.

application server: The component of an application that runs on a server and requests the delivery of a push message.

push message subscription: A message delivery context that is established between the user agent and the push service and shared with the application server. All push messages are associated with a push message subscription.

push message subscription set: A message delivery context that is established between the user agent and the push service that collects multiple push message subscriptions into a set.

push message: A message sent from an application server to a user agent via a push service.

push message receipt: A message delivery confirmation sent from the push service to the application server.

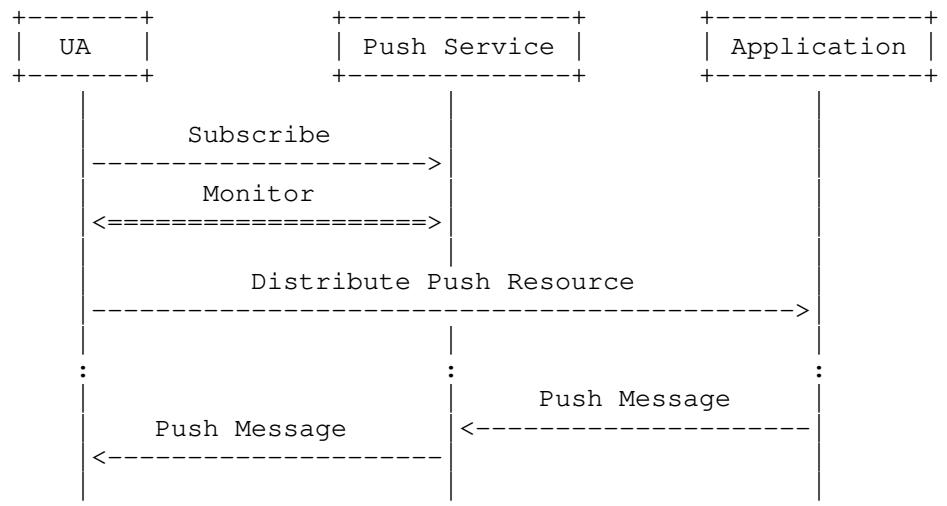
push service: A service that delivers push messages to user agents.

user agent: A device and software that is the recipient of push messages.

Examples in this document use the HTTP/1.1 message format [RFC7230]. Many of the exchanges can be completed using HTTP/1.1, where HTTP/2 is necessary, the more verbose frame format from [RFC7540] is used.

2. Overview

A general model for push services includes three basic actors: a user agent, a push service, and an application (server).



At the very beginning of the process, a new message subscription is created by the user agent and then distributed to its application server. This subscription is the basis of all future interactions between the actors.

To offer more control for authorization, a message subscription is modeled as two resources with different capabilities:

- A subscription resource is used to receive messages from a subscription and to delete a subscription. It is private to the user agent.

- A push resource is used to send messages to a subscription. It is public and shared by the user agent with its application server.

It is expected that a unique subscription will be distributed to each application; however, there are no inherent cardinality constraints in the protocol. Multiple subscriptions might be created for the same application, or multiple applications could use the same subscription. Note however that sharing subscriptions has security and privacy implications.

Subscriptions have a limited lifetime. They can also be terminated by either the push service or user agent at any time. User agents and application servers must be prepared to manage changes in subscription state.

2.1. HTTP Resources

This protocol uses HTTP resources [RFC7230] and link relations [RFC5988]. The following resources are defined:

push service: This resource is used to create push message subscriptions (see Section 4). A URL for the push service is configured into user agents.

push message subscription: This resource provides read and delete access for a message subscription. A user agent receives push messages (Section 7) using a push message subscription. Every push message subscription has exactly one push resource associated with it.

push message subscription set: This resource provides read and delete access for a collection of push message subscriptions. A user agent receives push messages (Section 7.1) for all the push message subscriptions in the set. A link relation of type "urn:ietf:params:push:set" identifies a push message subscription set.

push: A push resource is used by the application server to request the delivery of a push message (see Section 6). A link relation of type "urn:ietf:params:push" identifies a push resource.

push message: A push message resource is created to identify push messages that have been accepted by the push service. The push message resource is also used to acknowledge receipt of a push message.

receipt subscribe: A receipt subscribe resource is used by an application server to create a receipt subscription (see Section 5). A link relation of type "urn:ietf:params:push:receipts" identifies a receipt subscribe resource.

receipt subscription: An application server receives delivery confirmations (Section 6.1) for push messages using a receipt subscription. A link relation of type "urn:ietf:params:push:receipt" is identifies a receipt subscription.

3. Connecting to the Push Service

The push service shares the same default port number (443/TCP) with HTTPS, but MAY also advertise the IANA allocated TCP System Port 1001 using HTTP alternative services [I-D.ietf-httpbis-alt-svc]:

While the default port (443) offers broad reachability characteristics, it is most often used for web browsing scenarios with a lower idle timeout than other ports configured in middleboxes. For webpush scenarios, this would contribute to unnecessary radio communications to maintain the connection on battery-powered devices.

Advertising the alternate port (1001) allows middleboxes to optimize idle timeouts for connections specific to push scenarios with the expectation that data exchange will be infrequent.

Middleboxes SHOULD comply with REQ-5 in [RFC5382] which requires that "the value of the 'established connection idle-timeout' MUST NOT be less than 2 hours 4 minutes".

4. Subscribing for Push Messages

A user agent sends a POST request to its configured push service resource to create a new subscription.

```
POST /subscribe HTTP/1.1
Host: push.example.net
```

A response with a 201 (Created) status code includes a URI for a new push message subscription resource in the Location header field.

The push service MUST provide a URI for the push resource corresponding to the push message subscription using a link relation of type "urn:ietf:params:push".

The push service MUST provide a URI for a receipt subscribe resource in a link relation of type "urn:ietf:params:push:receipts".

An application-specific method is used to distribute the push and receipt subscribe URIs to the application server. Confidentiality protection and application server authentication MUST be used to ensure that these URIs are not disclosed to unauthorized recipients (see Section 9.3).

The push service MAY provide a URI for a subscription set resource in a link relation of type "urn:ietf:params:push:set". If provided, the push service supports subscription sets. If available, the user agent SHOULD use the subscription set to receive push messages rather than individual push message subscriptions.

The push service MAY return new subscription sets in response to different subscription requests from the same user agent. This

allows the push service to control the grouping of the push message subscriptions.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
        rel="urn:ietf:params:push"
Link: </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
        rel="urn:ietf:params:push:receipts"
Link: </set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
        rel="urn:ietf:params:push:set"
Location: https://push.example.net/s/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
```

4.1. Correlating Subscriptions

Collecting multiple push message subscriptions into a subscription set can represent a significant efficiency improvement for a push service. For that reason, if a subscription set is returned in a push message subscription response, the user agent SHOULD include this subscription set in subsequent push message subscription requests to the push service.

A user agent MAY omit the subscription set if it is unable to receive push messages that are aggregated for the lifetime of the subscription. This might be necessary if the user agent is forwarding requests from other clients.

The user agent adds a subscription set link relation to the request to create a push message subscription. This gives the push service the option to create the new subscription within that subscription set.

```
POST /subscribe HTTP/1.1
Host: push.example.net
Link: </set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
        rel="urn:ietf:params:push:set"
```

The push service SHOULD return the same subscription set in its response, although it MAY return a new subscription set if it is unable to reuse the one provided by the user agent.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </p/YBJNBIMwwA_Ag8EtD47J4A>;
        rel="urn:ietf:params:push"
Link: </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
        rel="urn:ietf:params:push:receipts"
Link: </set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
        rel="urn:ietf:params:push:set"
Location: https://push.example.net/s/i-nQ3A9Zm4kgSWg8_ZijVQ
```

A push service MAY return a 429 (Too Many Requests) status code [RFC6585] to reject requests which omit a subscription set or contain an invalid subscription set.

How a push service detects that requests originate from the same user agent is implementation-specific but could take ambient information into consideration, such as the TLS connection, source IP address and port. Implementers are reminded that some heuristics can produce false positives and cause requests to be rejected incorrectly.

5. Subscribing for Push Message Receipts

An application server requests the creation of a receipt subscription by sending a HTTP POST request to the receipt subscribe resource distributed to the application server by a user agent.

```
POST /receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ HTTP/1.1
Host: push.example.net
```

A successful response with a 201 (Created) status code includes a URI for the receipt subscription resource in the Location header field.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Location: https://push.example.net/r/3ZtI4YVNBnUUZhuoChl6omUvG4ZM
```

An application server that sends push messages to a large population of user agents incurs a significant load if it has to monitor a receipt subscription for each user agent. Reuse of receipt subscriptions is critical in reducing load on application servers. A receipt subscription can be used for all resources that have the same receipt subscribe URI.

A push service SHOULD provide the same receipt subscribe URI to all user agents. Application servers SHOULD reuse receipt subscription URIs if the receipt subscribe URI provided with the push resource is identical to the one used to create the receipt subscription. Checking that the receipt subscribe URI is identical allows the application server to avoid creating unnecessary receipt subscriptions.

6. Requesting Push Message Delivery

An application server requests the delivery of a push message by sending a HTTP POST request to a push resource distributed to the application server by a user agent. The push message is included in the body of the request.

```
POST /p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
Link: </r/3ZtI4YVNBnUUZhuoChl6omUvG4ZM>;
        rel="urn:ietf:params:push:receipt"
TTL: 15
Content-Type: text/plain;charset=utf8
Content-Length: 36

iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

A 201 (Created) response indicates that the push message was accepted. A URI for the push message resource that was created in response to the request MUST be returned in the Location header field. This does not indicate that the message was delivered to the user agent.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:55 GMT
Location: https://push.example.net/d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
```

A push service MAY return a 429 (Too Many Requests) status code [RFC6585] when an application server has exceeded its rate limit for push message delivery to a push resource. The push service SHOULD also include a Retry-After header [RFC7231] to indicate how long the application server is requested to wait before it makes another request to the push resource.

A push service MAY return a 413 (Payload Too Large) status code [RFC7231] in response to requests that include an entity body that is too large. Push services MUST NOT return a 413 status code in responses to an entity body that is 4k (4096 bytes) or less in size.

6.1. Requesting Push Message Receipts

An application server can use the "urn:ietf:params:push:receipt" link relation type to request a confirmation from the push service when a push message is delivered and acknowledged by the user agent.

The application sets the link relation value to a receipt subscription URI. This receipt subscription resource MUST be created from the same receipt subscribe resource which was returned with the push message subscription response (see Section 4).

6.2. Push Message Time-To-Live

A push service can improve the reliability of push message delivery considerably by storing push messages for a period. User agents are often only intermittently connected, and so benefit from having short term message storage at the push service.

Delaying delivery might also be used to batch communication with the user agent, thereby conserving radio resources.

Some push messages are not useful once a certain period of time elapses. Delivery of messages after they have ceased to be relevant is wasteful. For example, if the push message contains a call notification, receiving a message after the caller has abandoned the call is of no value; the application at the user agent is forced to suppress the message so that it does not generate a useless alert.

An application server MUST include the TTL (Time-To-Live) header field in its request for push message delivery. The TTL header field contains a value in seconds that suggests how long a push message is retained by the push service.

TTL = 1*DIGIT

A push service MUST return a 400 (Bad Request) status code in response to requests that omit the TTL header field.

A push service MAY retain a push message for a shorter duration than requested. It indicates this by returning a TTL header field in its response with the actual TTL. This TTL value MUST be less than or equal to the value provided by the application server.

Once the TTL period elapses, the push service MUST NOT attempt to deliver the push message to the user agent. A push service might adjust the TTL value to account for time accounting errors in processing. For instance, distributing a push message within a

server cluster might accrue errors due to clock skew or propagation delays.

A push service is not obligated to account for time spent by the application server in sending a push message to the push service, or delays incurred while sending a push message to the user agent. An application server needs to account for transit delays in selecting a TTL header field value.

A Push message with a zero TTL is immediately delivered if the user agent is available to receive the message. After delivery, the push service is permitted to immediately remove a push message with a zero TTL. This might occur before the user agent acknowledges receipt of the message by performing a HTTP DELETE on the push message resource. Consequently, an application server cannot rely on receiving acknowledgement receipts for zero TTL push messages.

If the user agent is unavailable, a push message with a zero TTL expires and is never delivered.

6.3. Push Message Urgency

For a device that is battery-powered, it is often critical that it remains dormant for extended periods. Radio communication in particular consumes significant power and limits the length of time that the device can operate.

To avoid consuming resources to receive trivial messages, it is helpful if an application server can communicate the urgency of a message and if the user agent can request that the push server only forward messages of a specific urgency.

An application server MAY include an Urgency header field in its request for push message delivery. This header field indicates the message urgency. The push service MUST not forward the Urgency header field to the user agent. A user agent MAY include the Urgency header field when requesting push messages to indicate that it only wants to receive messages at an equal or greater urgency.

```
Urgency = 1#(urgency-option)
urgency-option = ("very-low" / "low" / "normal" / "high")
```

In order of increasing urgency:

Urgency	Device State	Application Scenario
very-low	On power and wifi	Advertisements
low	On either power or wifi	Topic updates
normal	On neither power nor wifi	Chat or Calendar Message
high	Low battery	Incoming phone call or time-sensitive alert

Table 1: Table of Urgency Values

An absent Urgency header field indicates that the request is to be forwarded at "normal" urgency. Multiple values for the header field MUST NOT be included in requests; otherwise, the push service MUST return a 400 (Bad Request) status code.

6.4. Updating Push Messages

A push message that has been stored by the push service can be replaced with new content. If the user agent is offline during the time that the push messages are sent, updating a push message avoids the situation where outdated or redundant messages are sent to the user agent.

Only push messages that have been assigned a topic can be updated. A push message with a topic replaces any outstanding push message with an identical topic.

A push message topic is a string carried in a Topic header field. A topic is used to correlate push messages sent to the same subscription and does not convey any other semantics.

The grammar for the Topic header field uses the "token" and "quoted-string" rules defined in [RFC7230].

```
Topic = token / quoted-string
```

Any double quotes from the "quoted-string" form are removed before comparing topics for equality.

For use with this protocol, the Topic header field MUST be restricted to no more than 32 characters from the URL and filename safe Base 64 alphabet [RFC4648]. A push service that receives a request with a

Topic header field that does not meet these constraints MUST return a 400 (Bad Request) status code to the application server.

A push message update request creates a new push message resource and simultaneously deletes any existing message resource that has a matching topic. In effect, the information that is stored for the push message is updated, but a new resource is created to avoid problems with in flight acknowledgments for the old message. The push service MAY suppress acknowledgement receipts for the replaced message.

A push message with a topic that is not shared by an outstanding message to the same subscription is stored or delivered as normal.

For example, the following message could cause an existing message to be updated:

```
POST /p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
TTL: 600
Topic: "upd"
Content-Type: text/plain;charset=utf8
Content-Length: 36

ZuHSZPKa2b1jtOKLGpWrcrn8cNqt0iVQyroF
```

If the push service identifies an outstanding push message with a topic of "upd", then that message resource is deleted. A 201 (Created) response indicates that the push message update was accepted. A URI for the new push message resource that was created in response to the request is included in the Location header field.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:57:02 GMT
Location: https://push.example.net/d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
```

The value of the Topic header field MUST NOT be forwarded to user agents. Its value is neither encrypted nor authenticated.

7. Receiving Push Messages for a Subscription

A user agent requests the delivery of new push messages by making a GET request to a push message subscription resource. The push service does not respond to this request, it instead uses HTTP/2 server push [RFC7540] to send the contents of push messages as they are sent by application servers.

A user agent MAY include a Urgency header field in its request. The push service MUST only deliver messages with an urgency greater than or equal to the value of the header field as defined in the Table of Urgency Values.

Each push message is pushed as the response to a synthesized GET request in a PUSH_PROMISE. This GET request is made to the push message resource that was created by the push service when the application server requested message delivery. The response headers SHOULD provide a URI for the push resource corresponding to the push message subscription in a link relation of type "urn:ietf:params:push". The response body is the entity body from the most recent request sent to the push resource by the application server.

The following example request is made over HTTP/2.

```
HEADERS       [stream 7] +END_STREAM +END_HEADERS
  :method         = GET
  :path           = /s/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
  :authority      = push.example.net
```

The push service permits the request to remain outstanding. When a push message is sent by an application server, a server push is associated with the initial request. The response includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method         = GET
  :path           = /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
  :authority      = push.example.net

HEADERS       [stream 4] +END_HEADERS
  :status         = 200
  date            = Thu, 11 Dec 2014 23:56:56 GMT
  last-modified   = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control   = private
  :link           = </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                     rel="urn:ietf:params:push"
  content-type    = text/plain;charset=utf8
  content-length = 36

DATA          [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

The push service response for the GET request to the push message subscription resource SHOULD provide a URI for the receipt subscribe resource in a link relation of type "urn:ietf:params:push:receipts".

```
HEADERS       [stream 7] +END_STREAM +END_HEADERS
  :status         = 200
  :link           = </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
                       rel="urn:ietf:params:push:receipts"
```

A user agent can also request the contents of the push message subscription resource immediately by including a Prefer header field [RFC7240] with a "wait" parameter set to "0". In response to this request, the push service MUST generate a server push for all push messages that have not yet been delivered.

A 204 (No Content) status code with no associated server pushes indicates that no messages are presently available. This could be because push messages have expired.

7.1. Receiving Push Messages for a Subscription Set

There are minor differences between receiving push messages for a subscription and a subscripion set.

A user agent requests the delivery of new push messages for a collection of push message subscriptions by making a GET request to a push message subscription set resource. The push service does not respond to this request, it instead uses HTTP/2 server push [RFC7540] to send the contents of push messages as they are sent by application servers.

A user agent MAY include a Urgency header field in its request. The push service MUST only deliver messages with an urgency greater than or equal to the value of the header field as defined in the Table of Urgency Values.

Each push message is pushed as the response to a synthesized GET request sent in a PUSH_PROMISE. This GET request is made to the push message resource that was created by the push service when the application server requested message delivery. The synthetic request MUST provide a URI for the push resource corresponding to the push message subscription in a link relation of type "urn:ietf:params:push". This enables the user agent to differentiate the source of the message. The response body is the entity body from the most recent request sent to the push resource by an application server.

The following example request is made over HTTP/2.

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When a push message is sent by an application server, a server push is associated with the initial request. The response includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method        = GET
  :path          = /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
  :authority     = push.example.net
  :link          = </p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                     rel="urn:ietf:params:push"

HEADERS      [stream 4] +END_HEADERS
  :status        = 200
  date           = Thu, 11 Dec 2014 23:56:56 GMT
  last-modified  = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control  = private
  content-type   = text/plain;charset=utf8
  content-length = 36

DATA         [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

The push service response for the GET request to the push message subscription set resource SHOULD provide a URI for the receipt subscribe resource in a link relation of type "urn:ietf:params:push:receipts".

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :status        = 200
  :link          = </receipts/xjTG79I3VuptNWS0DsFu4ihT97aE6UQJ>;
                     rel="urn:ietf:params:push:receipts"
```

A user agent can request the contents of the push message subscription set resource immediately by including a Prefer header

field [RFC7240] with a "wait" parameter set to "0". In response to this request, the push service MUST generate a server push for all push messages that have not yet been delivered.

A 204 (No Content) status code with no associated server pushes indicates that no messages are presently available. This could be because push messages have expired.

7.2. Acknowledging Push Messages

To ensure that a push message is properly delivered to the user agent at least once, the user agent MUST acknowledge receipt of the message by performing a HTTP DELETE on the push message resource.

```
DELETE /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk HTTP/1.1
Host: push.example.net
```

If the push service receives the acknowledgement and the application has requested a delivery receipt, the push service MUST deliver a success response to the application server monitoring the receipt subscription resource.

If the push service does not receive the acknowledgement within a reasonable amount of time, then the message is considered to be not yet delivered. The push service SHOULD continue to retry delivery of the message until its advertised expiration.

The push service MAY cease to retry delivery of the message prior to its advertised expiration due to scenarios such as an unresponsive user agent or operational constraints. If the application has requested a delivery receipt, then the push service MUST push a failure response with a status code of 410 (Gone) to the application server monitoring the receipt subscription resource.

7.3. Receiving Push Message Receipts

The application server requests the delivery of receipts from the push service by making a HTTP GET request to the receipt subscription resource. The push service does not respond to this request, it instead uses HTTP/2 server push [RFC7540] to send push receipts when messages are acknowledged (Section 7.2) by the user agent.

Each receipt is pushed as the response to a synthesized GET request sent in a PUSH_PROMISE. This GET request is made to the same push message resource that was created by the push service when the

application server requested message delivery. A successful response includes a 204 (No Content) status code with no data.

The following example request is made over HTTP/2.

```
HEADERS      [stream 13] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /r/3ZtI4YVNBnUUZhuoChl6omUvG4ZM
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When the user agent acknowledges the message, the push service pushes a delivery receipt to the application server. A 204 (No Content) status code confirms that the message was delivered and acknowledged.

```
PUSH_PROMISE [stream 13; promised stream 82] +END_HEADERS
  :method        = GET
  :path          = /d/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
  :authority     = push.example.net

HEADERS      [stream 82] +END_STREAM
                         +END_HEADERS
  :status        = 204
  date           = Thu, 11 Dec 2014 23:56:56 GMT
```

If the user agent fails to acknowledge the receipt of the push message and the push service ceases to retry delivery of the message prior to its advertised expiration, then the push service MUST push a failure response with a status code of 410 (Gone).

8. Operational Considerations

8.1. Load Management

A push service is likely to have to maintain a very large number of open TCP connections. Effective management of those connections can depend on being able to move connections between server instances.

A user agent MUST support the 307 (Temporary Redirect) status code [RFC7231], which can be used by a push service to redistribute load at the time that a new subscription is requested.

A server that wishes to redistribute load can do so using alternative services [I-D.ietf-httpbis-alt-svc]. Alternative services allows for

redistribution of load whilst maintaining the same URIs for various resources. User agents can ensure a graceful transition by using the GOAWAY frame once it has established a replacement connection.

8.2. Push Message Expiration

Storage of push messages based on the TTL header field comprises a potentially significant amount of storage for a push service. A push service is not obligated to store messages indefinitely. A push service is able to indicate how long it intends to retain a message to an application server using the TTL header field (see Section 6.2).

A user agent that does not actively monitor for push messages will not receive messages that expire during that interval.

Push messages that are stored and not delivered to a user agent are delivered when the user agent recommences monitoring. Stored push messages SHOULD include a Last-Modified header field (see Section 2.2 of [RFC7232]) indicating when delivery was requested by an application server.

A GET request to a push message subscription resource that has only expired messages results in response as though no push message were ever sent.

Push services might need to limit the size and number of stored push messages to avoid overloading. To limit the size of messages, the push service MAY return the 413 (Payload Too Large) status code for messages that are too large. To limit the number of stored push messages, the push service MAY either expire messages prior to their advertised Time-To-Live or reduce their advertised Time-To-Live.

8.3. Subscription Expiration

In some cases, it may be necessary to terminate subscriptions so that they can be refreshed. This applies to both push message subscriptions and receipt subscriptions.

A push service can remove a subscription at any time. If a user agent or application server has an outstanding request to a subscription resource (see Section 7), this can be signaled by returning a 400-series status code, such as 410 (Gone).

A user agent or application server can request that a subscription be removed by sending a DELETE request to the push message subscription or receipt subscription URI.

A push service MUST return a 400-series status code, such as 404 (Not Found) or 410 (Gone) if an application server attempts to send a push message to a removed or expired push message subscription.

8.3.1. Subscription Set Expiration

A push service MAY expire a subscription set at any time which MUST also expire all push message subscriptions in the set. If a user agent has an outstanding request to a push subscription set (see Section 7.1) this can be signaled by returning a 400-series status code, such as 410 (Gone).

A user agent can request that a subscription set be removed by sending a DELETE request to the subscription set URI. This MUST also remove all push message subscriptions in the set.

If a specific push message subscription that is a member of a subscription set is expired or removed, then it MUST also be removed from its subscription set.

8.4. Implications for Application Reliability

A push service that does not support reliable delivery over intermittent network connections or failing applications on devices, forces the device to acknowledge receipt directly to the application server, incurring additional power drain in order to establish (usually secure) connections to the individual application servers.

Push message reliability can be important if messages contain information critical to the state of an application. Repairing state can be costly, particularly for devices with limited communications capacity. Knowing that a push message has been correctly received avoids costly retransmissions, polling and state resynchronization.

The availability of push message delivery receipts ensures that the application developer is not tempted to create alternative mechanisms for message delivery in case the push service fails to deliver a critical message. Setting up a polling mechanism or a backup messaging channel in order to compensate for these shortcomings negates almost all of the advantages a push service provides.

However, reliability might not be necessary for messages that are transient (e.g. an incoming call) or messages that are quickly superceded (e.g. the current number of unread emails).

8.5. Subscription Sets and Concurrent HTTP/2 streams

If the push service requires that the user agent use push message subscription sets, then it MAY limit the number of concurrently active streams with the SETTINGS_MAX_CONCURRENT_STREAMS parameter within a HTTP/2 SETTINGS frame [RFC7540]. The user agent MAY be limited to one concurrent stream to manage push message subscriptions and one concurrent stream for each subscription set returned by the push service. This could force the user agent to serialize subscription requests to the push service.

9. Security Considerations

This protocol MUST use HTTP over TLS [RFC2818]. This includes any communications between user agent and push service, plus communications between the application and the push service. All URIs therefore use the "https" scheme. This provides confidentiality and integrity protection for subscriptions and push messages from external parties.

9.1. Confidentiality from Push Service Access

The protection afforded by TLS does not protect content from the push service. Without additional safeguards, a push service is able to see and modify the content of the messages.

Applications are able to provide additional confidentiality, integrity or authentication mechanisms within the push message itself. The application server sending the push message and the application on the user agent that receives it are frequently just different instances of the same application, so no standardized protocol is needed to establish a proper security context. The process of providing the application server with subscription information provides a convenient medium for key agreement.

The Web Push API codifies this practice by requiring that each push subscription created by the browser be bound to a browser generated encryption key. Pushed messages are authenticated and decrypted by the browser before delivery to applications. This scheme ensures that the push service is unable to examine the contents of push messages.

The public key for a subscription ensures that applications using that subscription can identify messages from unknown sources and discard them. This depends on the public key only being disclosed to entities that are authorized to send messages on the channel. The push service does not require access to this public key.

The Topic header field exposes information that allows more granular correlation of push messages on the same subject. This might be used to aid traffic analysis of push messages by the push service.

9.2. Privacy Considerations

Push message confidentiality does not ensure that the identity of who is communicating and when they are communicating is protected. However, the amount of information that is exposed can be limited.

The URIs provided for push resources MUST NOT provide any basis to correlate communications for a given user agent. It MUST NOT be possible to correlate any two push resource URIs based solely on their contents. This allows a user agent to control correlation across different applications, or over time.

Similarly, the URIs provided by the push service to identify a push message MUST NOT provide any information that allows for correlation across subscriptions. Push message URIs for the same subscription MAY contain information that would allow correlation with the associated subscription or other push messages for that subscription.

User and device information MUST NOT be exposed through a push or push message URI.

In addition, push URIs established by the same user agent or push message URIs for the same subscription MUST NOT include any information that allows them to be correlated with the user agent.

Note: This need not be perfect as long as the resulting anonymity set (see [RFC6973], Section 6.1.1) is sufficiently large. A push URI necessarily identifies a push service or a single server instance. It is also possible that traffic analysis could be used to correlate subscriptions.

A user agent MUST be able to create new subscriptions with new identifiers at any time.

9.3. Authorization

This protocol does not define how a push service establishes whether a user agent is permitted to create a subscription, or whether push messages can be delivered to the user agent. A push service MAY choose to authorize requests based on any HTTP-compatible authorization method available, of which there are numerous options. The authorization process and any associated credentials are expected to be configured in the user agent along with the URI for the push service.

Authorization is managed using capability URLs for the push message subscription, push, and receipt subscription resources (see [CAP-URI]). A capability URL grants access to a resource based solely on knowledge of the URL.

Capability URLs are used for their "easy onward sharing" and "easy client API" properties. These make it possible to avoid relying on relationships between push services and application servers, with the protocols necessary to build and support those relationships.

Capability URLs act as bearer tokens. Knowledge of a push message subscription URI implies authorization to either receive push messages or delete the subscription. Knowledge of a push URI implies authorization to send push messages. Knowledge of a push message URI allows for reading and acknowledging that specific message. Knowledge of a receipt subscription URI implies authorization to receive push receipts. Knowledge of a receipt subscribe URI implies authorization to create subscriptions for receipts.

Note that the same receipt subscribe URI could be returned for multiple push message subscriptions. Using the same value for a large number of subscriptions allows application servers to reuse receipt subscriptions, which can provide a significant efficiency advantage. A push service that uses a common receipt subscribe URI loses control over the creation of receipt subscriptions. This can result in a potential exposure to denial of service; stateless resource creation can be used to mitigate the effects of this exposure.

Encoding a large amount of random entropy (at least 120 bits) in the path component ensures that it is difficult to successfully guess a valid capability URL.

9.4. Denial of Service Considerations

Discarding unwanted messages at the user agent based on message authentication doesn’t protect against a denial of service attack on the user agent. Even a relatively small volume of push messages can cause battery-powered devices to exhaust power reserves.

An application can limit where valid push messages can originate by limiting the distribution of push URIs to authorized entities. Ensuring that push URIs are hard to guess ensures that only application servers that have been given a push URI can use it.

A malicious application with a valid push URI could use the greater resources of a push service to mount a denial of service attack on a user agent. Push services SHOULD limit the rate at which push

messages are sent to individual user agents. A push service or user agent MAY terminate subscriptions (Section 8.3) that receive too many push messages.

End-to-end confidentiality mechanisms, such as those in [API], prevent an entity with a valid push message subscription URI from learning the contents of push messages. Push messages that are not successfully authenticated will not be delivered by the API, but this can present a denial of service risk.

Conversely, a push service is also able to deny service to user agents. Intentional failure to deliver messages is difficult to distinguish from faults, which might occur due to transient network errors, interruptions in user agent availability, or genuine service outages.

9.5. Logging Risks

Server request logs can reveal subscription-related URIs. Acquiring a push message subscription URI enables the receipt of messages or deletion of the subscription. Acquiring a push URI permits the sending of push messages. Logging could also reveal relationships between different subscription-related URIs for the same user agent. Encrypted message contents are not revealed to the push service.

Limitations on log retention and strong access control mechanisms can ensure that URIs are not learned by unauthorized entities.

10. IANA Considerations

This protocol defines new HTTP header fields in Section 10.1. New link relation types are identified using the URNs defined in Section 10.2. Port registration is defined in Section 10.3

10.1. Header Field Registrations

HTTP header fields are registered within the "Message Headers" registry maintained at <https://www.iana.org/assignments/message-headers/>.

This document defines the following HTTP header fields, so their associated registry entries shall be added according to the permanent registrations below (see [RFC3864]):

Header Field Name	Protocol	Status	Reference
TTL	http	standard	Section 6.2
Urgency	http	standard	Section 6.3
Topic	http	standard	Section 6.4

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

10.2. Link Relation URNs

This document registers URNs for use in identifying link relation types. These are added to a new "Web Push Identifiers" registry according to the procedures in Section 4 of [RFC3553]; the corresponding "push" sub-namespace is entered in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry.

The "Web Push Identifiers" registry operates under the IETF Review policy [RFC5226].

Registry name: Web Push Identifiers

URN Prefix: urn:ietf:params:push

Specification: (this document)

Repository: [Editor/IANA note: please include a link to the final registry location.]

Index value: Values in this registry are URNs or URN prefixes that start with the prefix "urn:ietf:params:push". Each is registered independently.

New registrations in the "Web Push Identifiers" are encouraged to include the following information:

URN: A complete URN or URN prefix.

Description: A summary description.

Specification: A reference to a specification describing the semantics of the URN or URN prefix.

Contact: Email for the person or group making the registration.

Index value: As described in [RFC3553], URN prefixes that are registered include a description of how the URN is constructed. This is not applicable for specific URNs.

These values are entered as the initial content of the "Web Push Identifiers" registry.

URN: urn:ietf:params:push

Description: This link relation type is used to identify a resource for sending push messages.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:set

Description: This link relation type is used to identify a collection of push message subscriptions.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:receipt

Description: This link relation type is used to identify a resource for receiving delivery confirmations for push messages.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:receipts

Description: This link relation type is used to identify a resource for subscribing to delivery confirmations for push messages.

Specification: (this document)

Contact: The Web Push WG (webpush@ietf.org)

10.3. Service Name and Port Number Registration

Service names and port numbers are registered within the "Service Name and Transport Protocol Port Number Registry" maintained at

<https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>.

IANA is requested to assign the System Port number 1001 and the service name "webpush" in accordance with [RFC6335].

Service Name.
webpush

Transport Protocol.
tcp

Assignee.
IESG (iesg@ietf.org)

Contact.
The Web Push WG (webpush@ietf.org)

Description.
HTTP Web Push

Reference.
[RFCthis]

Port Number.
1001

11. Acknowledgements

Significant technical input to this document has been provided by Ben Bangert, Kit Cambridge, JR Conlin, Matthew Kaufman, Costin Manolache, Mark Nottingham, Robert Sparks, Darshak Thakore and many others.

12. References

12.1. Normative References

[CAP-URI] Tennison, J., "Good Practices for Capability URLs", FPWD capability-urls, February 2014, <http://www.w3.org/TR/capability-urls/>.

[I-D.ietf-httpbis-alt-svc] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-14 (work in progress), March 2016.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, June 2003.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5382] Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", RFC 5382, October 2008.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", RFC 6335, August 2011.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, April 2012.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7232] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

[RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240, June 2014.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2", RFC 7540, May 2015.

12.2. Informative References

[API] van Ouwerkerk, M., Thomson, M., Sullivan, B., and E. Fullea, "Web Push API", ED push-api, January 2016, <https://w3c.github.io/push-api/>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, July 2013.

Appendix A. Change Log

[[The RFC Editor is requested to remove this section at publication.]]

A.1. Since draft-ietf-webpush-protocol-00

Editorial changes for Push Message Time-To-Live

Editorial changes for Push Acknowledgements

Removed subscription expiration based on HTTP cache headers

A.2. Since draft-ietf-webpush-protocol-01

Added Subscription Sets

Added System Port as an alternate service with guidance for idle timeouts

Finalized status codes for acknowledgements

Editorial changes for Rate Limits

A.3. Since draft-ietf-webpush-protocol-02

Added explicit correlation for Subscription Sets

Added Push Message Updates (message collapsing)

Renamed the push:receipt link relation to push:receipts and transitioned the Push-Receipt header field to the push:receipt link relation type

A.4. Since draft-ietf-webpush-protocol-03

An application server MUST include the TTL (Time-To-Live) header field in its request for push message delivery.

Added Push Message Urgency header field

Authors' Addresses

Martin Thomson
Mozilla
331 E Evelyn Street
Mountain View, CA 94041
US

Email: martin.thomson@gmail.com

Elio Damaggio
Microsoft
One Microsoft Way
Redmond, WA 98052
US

Email: elioda@microsoft.com

Brian Raymor (editor)
Microsoft
One Microsoft Way
Redmond, WA 98052
US

Email: brian.raymor@microsoft.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 3, 2016

M. Thomson
Mozilla
P. Beverloo
Google

January 31, 2016

Voluntary Application Server Identification for Web Push
draft-thomson-webpush-vapid-02

Abstract

An application server can voluntarily identify itself to a push service using the described technique. This identification information can be used by the push service to attribute requests that are made by the same application server to a single entity. This can used to reduce the secrecy for push subscription URLs by being able to restrict subscriptions to a specific application server. An application server is further able include additional information the operator of a push service can use to contact the operator of the application server.

Table of Contents

1. Introduction

The Web Push protocol [I-D.ietf-webpush-protocol] describes how an application server is able to request that a push service deliver a push message to a user agent.

As a consequence of the expected deployment architecture, there is no basis for an application server to be known to a push service prior to requesting delivery of a push message. By the same measure, requesting the creation of a subscription for push message receipts has no prior authentication. Requiring that the push service be able to authenticate application servers places an unwanted constraint on the interactions between user agents and application servers, who are the ultimate users of a push service. That constraint would also degrade the privacy-preserving properties the protocol provides. For these reasons, [I-D.ietf-webpush-protocol] does not define a mandatory system for authentication of application servers.

An unfortunate consequence of this design is that a push service is exposed to a greater risk of denial of service attack. While requests from application servers can be indirectly attributed to user agents, this is not always efficient or even sufficient. Providing more information about the application server directly to a push service allows the push service to better distinguish between legitimate and bogus requests.

Additionally, this design also relies on endpoint secrecy as any application server in possession of the endpoint is able to send messages, albeit without payloads. In situations where usage of a subscription can be limited to a single application server, the ability to associate a subscription with the application server could reduce the impact of a data leak.

1.1. Voluntary Identification

This document describes a system whereby an application server can volunteer information about itself to a push service. At a minimum, this provides a stable identity for the application server, though this could also include contact information, such as an email address.

A consistent identity can be used by a push service to establish behavioral expectations for an application server. Significant deviations from an established norm can then be used to trigger exception handling procedures.

Voluntarily-provided contact information can be used to contact an application server operator in the case of exceptional situations.

Experience with push service deployment has shown that software errors or unusual circumstances can cause large increases in push message volume. Contacting the operator of the application server has proven to be valuable.

Even in the absence of usable contact information, an application server that has a well-established reputation might be given preference over an unidentified application server when choosing whether to discard a push message.

1.2. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting, when they are capitalized, they have the special meaning described in [RFC2119].

The terms "push message", "push service", "push subscription", "application server", and "user agent" are used as defined in [I-D.ietf-webpush-protocol].

2. Application Server Self-Identification

Application servers SHOULD generate and maintain a signing key pair usable with elliptic curve digital signature (ECDSA) over the P-256 curve [FIPS186]. Use of this key when sending push messages establishes a continuous identity for the application server.

When requesting delivery of a push message, the application includes a JSON Web Token (JWT) [RFC7519], signed using its signing key. The token includes a number of claims as follows:

- An "aud" (Audience) claim in the token MUST include the unicode serialization of the origin (Section 6.1 of [RFC6454]) of the push resource URL. This binds the token to a specific push service. This ensures that the token is reusable for all push resource URLs that share the same origin.

- An "exp" (Expiry) claim MUST be included with the time after which the token expires. This limits the time that a token over which a token is valid. An "exp" claim MUST NOT be more than 24 hours from the time of the request.

This JWT is included in an Authorization header field, using an auth-scheme of "WebPush". A push service MAY reject a request with a 403 (Forbidden) status code [RFC7235] if the JWT signature or its claims are invalid.

The JWT MUST use a JSON Web Signature (JWS) [RFC7515]. The signature MUST use ECDSA on the NIST P-256 curve [FIPS186], that is "ES256" [RFC7518].

2.1. Application Server Contact Information

If the application server wishes to provide the JWT MAY include an "sub" (Subject) claim. The "sub" claim SHOULD include a contact URI for the application server as either a "mailto:" (email) [RFC6068] or an "https:" [RFC2818] URI.

2.2. Example

An application server requests the delivery of a push message as described in [I-D.ietf-webpush-protocol]. If the application server wishes to self-identify, it includes an Authorization header field with credentials that use the "WebPush" authentication scheme

Section 3 and a Crypto-Key header field that includes its public key Section 4.

```
POST /p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
Push-Receipt: https://push.example.net/r/3ZtI4YVNBnUUZhuoChl6omU
Content-Type: text/plain;charset=utf8
Content-Length: 36
Authorization: Bearer
    eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJhdWQiOiJodHRwczovL3B
    1c2guZXhhbXBsZS5uZXQiLCJleHAiOjE0NTM1MjM3NjgsInN1YiI6Im1haWx
    0bzpwdXNoQGV4YW1wbGUuY29tIn0.i3CYb7t4xfxCDquptFOepC9GAu_HLGk
    MlMuCGSK2rpiUfnK9ojFwDXb1JrErtmysazNjjvW2L9OkSSHzvoD1oA
Crypto-Key: p256ecdsa=BA1Hxzyi1RUM1b5wjxsn7nGxAszw2u61m164i3MrAIxH
                      F6YK5h4SDYic-dRuU_RCPCfA5aq9ojSwk5Y2EmClBPs

iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

Figure 1: Requesting Push Message Delivery with JWT

Note that the header fields shown in Figure 1 don't include line wrapping. Extra whitespace is added to meet formatting constraints.

This equates to a JWT with the header and body shown in Figure 2. This JWT would be valid until 2016-01-21T01:53:25Z [RFC3339].

```
header = {"typ":"JWT","alg":"ES256"}
body = { "aud":"https://push.example.net",
         "exp":1453341205,
         "sub":"mailto:push@example.com" }
```

Figure 2: Example JWT Header and Body

Issue: The first part of the JWT is effectively fixed. Would be it acceptable to require that that segment is omitted from the header field?

3. WebPush Authentication Scheme

A new "WebPush" HTTP authentication scheme [RFC7235] is defined. This authentication scheme carries a signed JWT, as described in Section 2.

This authentication scheme is for origin-server authentication only. Therefore, this authentication scheme MUST NOT be used with The Proxy-Authenticate or Proxy-Authorization header fields.

This authentication scheme does not require a challenge. Clients are able to generate the Authorization header field without any additional information from a server. Therefore, a challenge for this authentication scheme MUST NOT be sent in a WWW-Authenticate header field.

All unknown or unsupported parameters to "WebPush" authentication credentials MUST be ignored. The "realm" parameter is ignored for this authentication scheme.

4. Public Key Representation

In order for the push service to be able to validate the JWT, it needs to learn the public key of the application server. A "p256ecdsa" parameter is defined for the Crypto-Key header field [I-D.ietf-httpbis-encryption-encoding] to carry this information.

The "p256ecdsa" parameter includes an elliptic curve digital signature algorithm (ECDSA) public key [FIPS186] in uncompressed form [X9.62] that is encoded using the URL- and filename-safe variant of base-64 [RFC4648] with padding removed.

Note that with push message encryption [I-D.ietf-webpush-encryption], this results in two values in the Crypto-Key header field, one with the a "p256dh" key and another with a "p256ecdsa" key.

Editor's Note: JWK [RFC7517] seems like the obvious choice here. However, JWK doesn't define a compact representation for public keys, which complicates the representation of JWK in a header field.

5. Subscription Restriction

The public key of the application server serves as a stable identifier for the server. This key can be used to restrict a push subscription to a specific application server.

Subscription restriction reduces the reliance on endpoint secrecy by requiring proof of possession to be demonstrated by an application server when requesting delivery of a push message. This provides an additional level of protection against leaking of the details of the push subscription.

5.1. Creating a Restricted Push Subscription

The user agent includes the public key of the application server when requesting that a push subscription. This restricts use of the

resulting push subscription to application servers that are able to provide proof of possession for the corresponding private key.

This public key is then added to the request to create a push subscription as described in Section 4. The Crypto-Key header field includes exactly one public key. For example:

```
POST /subscribe/ HTTP/1.1
Host: push.example.net
Crypto-Key: p256ecdsa=BBa22H8qaZ-iDMH9izb4qE72puwyvfjH2RxoQr5oiS4b
                      KImoRwJm5xK9hLrbfIik20g31z8MpLFMCMr8y2cu6gY
```

Figure 3: Example Subscribe Request

An application might use the Web Push API [API] to include this information. For example, the API might permit an application to provide a public key as part of a new field on the "PushSubscriptionOptions" dictionary.

Editor's Note: Allowing the inclusion of multiple keys when creating a subscription would allow a subscription to be associated with multiple application servers or application server instances. This might be more flexible, but it also would require more state to be maintained by the push service for each subscription.

5.2. Using Restricted Subscriptions

When a push subscription has been associated with an application server, the request for push message delivery MUST include proof of possession for the associated private key or token that was used when creating the push subscription.

A push service MUST reject a message that includes omits mandatory credentials with a 401 (Unauthorized) status code. A push service MAY reject a message that includes invalid credentials with a 403 (Forbidden) status code. Credentials are invalid if:

- either the authentication credentials or public key are not included in the request,
- the signature on the JWT cannot be successfully verified using the included public key,
- the current time is later than the time identified in the "exp" (Expiry) claim or more than 24 hours before the expiry time,
- the origin of the push resource is not included in the "aud" (Audience) claim, or

- the public key used to sign the doesn't match the one that was included in the creation of the push message.

A push subscription that is not restricted to a particular key MAY still validate a token that is present, except for the last check. A push service MAY then reject a request if the token is found to be invalid.

Editor's Note: In theory, since the push service was given a public key, the push message request could omit the public key. On balance, this keeps things simple and it allows push services to compress the public key (by hashing it, for example). In any case, the relatively minor space savings aren't particularly important on the connection between the application server and push service.

A push service does not need to forward the JWT or public key to the user agent when delivering the push message.

6. Security Considerations

This authentication scheme is vulnerable to replay attacks if an attacker can acquire a valid JWT. Applying narrow limits to the period over which a replayable token can be reused limits the potential value of a stolen token to an attacker and can increase the difficulty of stealing a token.

An application server might offer falsified contact information. A push service operator therefore cannot use the presence of unvalidated contact information as input to any security-critical decision-making process.

Validation of a signature on the JWT requires a non-trivial amount of computation. For something that might be used to identify legitimate requests under denial of service attack conditions, this is not ideal. Application servers are therefore encouraged to reuse a JWT, which permits the push service to cache the results of signature validation.

7. IANA Considerations

7.1. WebPush Authentication Scheme

This registers the "WebPush" authentication scheme in the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" established in [RFC7235].

Authentication Scheme Name: WebPush

Pointer to specification text: Section 3 of this document

Notes: This scheme is origin-server only and does not define a challenge.

7.2. p256ecdsa Parameter for Crypto-Key Header Field

This registers a "p256ecdsa" parameter for the Crypto-Key header field in the "Hypertext Transfer Protocol (HTTP) Crypto-Key Parameters" established in [I-D.ietf-httpbis-encryption-encoding].

Parameter Name: p256ecdsa

Purpose: Conveys a public key for that is used to generate an ECDSA signature.

Reference: Section 4 of this document

8. Acknowledgements

This document would have been much worse than it currently is if not for the contributions of Benjamin Bangert, Chris Karlof, Costin Manolache, and others.

9. References

9.1. Normative References

[FIPS186] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", NIST PUB 186-4 , July 2013.

[I-D.ietf-httpbis-encryption-encoding] Thomson, M., "Encrypted Content-Encoding for HTTP", draft-ietf-httpbis-encryption-encoding-00 (work in progress), December 2015.

[I-D.ietf-webpush-protocol] Thomson, M., Damaggio, E., and B. Raymor, "Generic Event Delivery Using HTTP Push", draft-ietf-webpush-protocol-02 (work in progress), November 2015.

[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>.

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

[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>.

[RFC6068] Duerst, M., Masinter, L., and J. Zawinski, "The 'mailto' URI Scheme", RFC 6068, DOI 10.17487/RFC6068, October 2010, <http://www.rfc-editor.org/info/rfc6068>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <http://www.rfc-editor.org/info/rfc6454>.

[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>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <http://www.rfc-editor.org/info/rfc7518>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <http://www.rfc-editor.org/info/rfc7519>.

[X9.62] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62 , 1998.

9.2. Informative References

[API] van Ouwerkerk, M. and M. Thomson, "Web Push API", 2015, <https://w3c.github.io/push-api/>.

[I-D.ietf-webpush-encryption] Thomson, M., "Message Encryption for Web Push", draft-ietf-webpush-encryption-01 (work in progress), October 2015.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <http://www.rfc-editor.org/info/rfc3339>.

[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>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <http://www.rfc-editor.org/info/rfc7517>.

Authors' Addresses

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

Peter Beverloo
Google

Email: beverloo@google.com