Internet Engineering Task Force (IETF)
Request for Comments: 8291
Category: Standards Track
ISSN: 2070-1721

M. Thomson
Mozilla

November 2017

Message Encryption for Web Push

Abstract

This document describes a message encryption scheme 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 is an Internet Standards Track document.

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

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8291.

Table of Contents

1. Introduction

The Web Push protocol [RFC8030] is an intermediated protocol by necessity. Messages from an application server are delivered to a user agent (UA) via a push service, as shown in Figure 1.

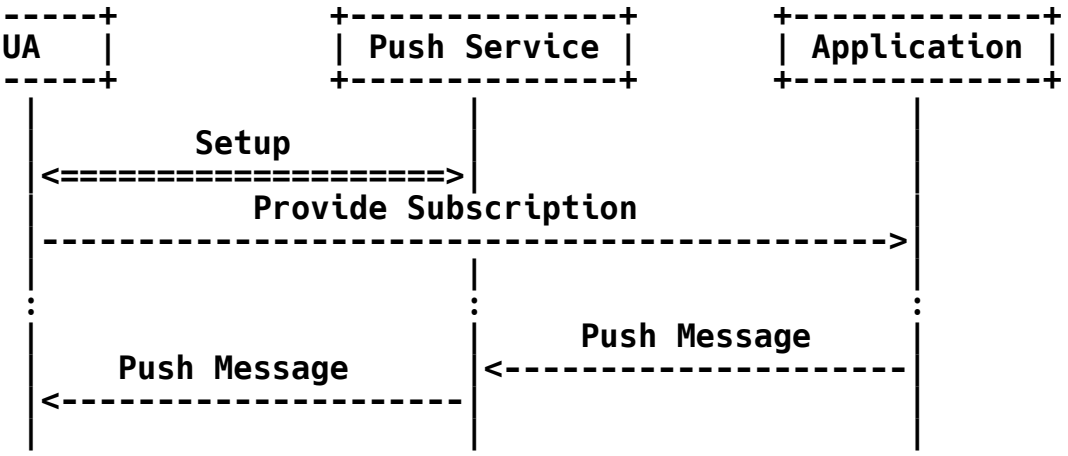

Figure 1

This document describes how messages sent using this protocol can be secured against inspection, modification, and forgery by a push service.

Web Push messages are the payload of an HTTP message [RFC7230]. These messages are encrypted using an encrypted content encoding [RFC8188]. This document describes how this content encoding is applied and describes a recommended key management scheme.

Multiple users of Web Push at the same user agent 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 Push API [API] can enforce the use of encryption by forwarding only those messages that were properly encrypted.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the terminology from [RFC8030], primarily "user agent", "push service", and "application server".

2. Push Message Encryption Overview

Encrypting a push message uses Elliptic Curve Diffie-Hellman (ECDH) [ECDH] on the P-256 curve [FIPS186] to establish a shared secret (see Section 3.1) and a symmetric secret for authentication (see Section 3.2).

A user agent generates an ECDH key pair and authentication secret that it associates with each subscription it creates. The ECDH public key and the authentication secret are sent to the application server with other details of the push subscription.

When sending a message, an application server generates an ECDH key pair and a random salt. The ECDH public key is encoded into the "keyid" parameter of the encrypted content coding header, and the salt is encoded into the "salt" parameter of that same header (see Section 2.1 of [RFC8188]). The ECDH key pair can be discarded after encrypting the message.

The content of the push message is encrypted or decrypted using a content encryption key and nonce. These values are derived by taking the "keyid" and "salt" as input to the process described in Section 3.

2.1. Key and Secret Distribution

The application using the subscription distributes the subscription public key and authentication secret to an authorized application server. This could be sent along with other subscription information that is provided by the user agent, such as the push subscription URI.

An application MUST use an authenticated, confidentiality-protected communications medium for this purpose. In addition to the reasons described in [RFC8030], this use ensures that the authentication secret is not revealed to unauthorized entities, which would allow those entities to generate push messages that will be accepted by the user agent.

Most applications that use push messaging have a preexisting relationship with an application server that can be used for distribution of subscription data. An authenticated communication mechanism that provides adequate confidentiality and integrity protection, such as HTTPS [RFC2818], is sufficient.

3. Push Message Encryption

Push message encryption happens in four phases:

- A shared secret is derived using ECDH [ECDH] (see Section 3.1 of this document).
- The shared secret is then combined with the authentication secret to produce the input keying material (IKM) used in [RFC8188] (see Section 3.3 of this document).
- A content encryption key and nonce are derived using the process in [RFC8188].
- Encryption or decryption follows according to [RFC8188].

The key derivation process is summarized in Section 3.4. Restrictions on the use of the encrypted content coding are described in Section 4.

3.1. Diffie-Hellman Key Agreement

For each new subscription that the user agent generates for an application, it also generates a P-256 [FIPS186] key pair for use in ECDH [ECDH].

When sending a push message, the application server also generates a new ECDH key pair on the same P-256 curve.

The ECDH public key for the application server is included as the "keyid" parameter in the encrypted content coding header (see Section 2.1 of [RFC8188]).

An application server combines its ECDH private key with the public key provided by the user agent using the process described in [ECDH]; on receipt of the push message, a user agent combines its private key with the public key provided by the application server in the "keyid" parameter in the same way. These operations produce the same value for the ECDH shared secret.

3.2. 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 Section 3.3.

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

3.3. Combining Shared and Authentication Secrets

The shared secret produced by ECDH is combined with the authentication secret using the HMAC-based key derivation function (HKDF) [RFC5869]. This produces the input keying material used by [RFC8188].

The HKDF function uses the SHA-256 hash algorithm [FIPS180-4] with the following inputs:

salt: the authentication secret

IKM: the shared secret derived using ECDH

info: the concatenation of the ASCII-encoded string "WebPush: info" (this string is not NUL-terminated), a zero octet, the user agent ECDH public key, and the application server ECDH public key, (both ECDH public keys are in the uncompressed point form defined in [X9.62]. That is:

key_info = "WebPush: info" || 0x00 || ua_public || as_public

L: 32 octets (i.e., the output is the length of the underlying SHA-256 HMAC function output)

3.4. Encryption Summary

This results in a final content encryption key and nonce generation using the following sequence, which is shown here in pseudocode with HKDF expanded into separate discrete steps using HMAC with SHA-256:

```
-- For a user agent:
ecdh_secret = ECDH(ua_private, as_public)
auth_secret = random(16)
salt = <from content coding header>

-- For an application server:
ecdh_secret = ECDH(as_private, ua_public)
auth_secret = <from user agent>
salt = random(16)

-- For both:

## Use HKDF to combine the ECDH and authentication secrets
# HKDF-Extract(salt=auth_secret, IKM=ecdh_secret)
PRK_key = HMAC-SHA-256(auth_secret, ecdh_secret)
# HKDF-Expand(PRK_key, key_info, L_key=32)
key_info = "WebPush: info" || 0x00 || ua_public || as_public
IKM = HMAC-SHA-256(PRK_key, key_info || 0x01)

## HKDF calculations from RFC 8188
# HKDF-Extract(salt, IKM)
PRK = HMAC-SHA-256(salt, IKM)
# HKDF-Expand(PRK, cek_info, L_cek=16)
cek_info = "Content-Encoding: aes128gcm" || 0x00
CEK = HMAC-SHA-256(PRK, cek_info || 0x01)[0..15]
# HKDF-Expand(PRK, nonce_info, L_nonce=12)
nonce_info = "Content-Encoding: nonce" || 0x00
NONCE = HMAC-SHA-256(PRK, nonce_info || 0x01)[0..11]
```

Note that this omits the exclusive-OR of the final nonce with the record sequence number, since push messages contain only a single record (see Section 4) and the sequence number of the first record is zero.

4. Restrictions on Use of "aes128gcm" Content Coding

An application server MUST encrypt a push message with a single record. This allows for a minimal receiver implementation that handles a single record. An application server MUST set the "rs" parameter in the "aes128gcm" content coding header to a size that is greater than the sum of the lengths of the plaintext, the padding delimiter (1 octet), any padding, and the authentication tag (16 octets).

A push message MUST include the application server ECDH public key in the "keyid" parameter of the encrypted content coding header. The uncompressed point form defined in [X9.62] (that is, a 65-octet sequence that starts with a 0x04 octet) forms the entirety of the "keyid". Note that this means that the "keyid" parameter will not be valid UTF-8 as recommended in [RFC8188].

A push service is not required to support more than 4096 octets of payload body (see Section 7.2 of [RFC8030]). Absent header (86 octets), padding (minimum 1 octet), and expansion for AEAD_AES_128_GCM (16 octets), this equates to, at most, 3993 octets of plaintext.

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 "aes128gcm". Multiple "aes128gcm" values are not permitted.

A user agent is not required to support multiple records. A user agent MAY ignore the "rs" parameter. If a record size is unchecked, decryption will fail with high probability for all valid cases. The padding delimiter octet MUST be checked; values other than 0x02 MUST cause the message to be discarded.

5. Push Message Encryption Example

The following example shows a push message being sent to a push service.

```
POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
TTL: 10
Content-Length: 145
Content-Encoding: aes128gcm

DGv6ra1nlYgDCS1FRnbzlwAAEABBBP4z9KsN6nGRTbVYI_c7VJSPQTBtkgcy27ml
mlMoZIIgDll6e3vCYLocInmYWAmS6TlzAC8wEqKK6PBru3jl7A_yl95bQpu6cVPT
pK4Mqgkf1CXztLVBSt2Ks3oZwbuwXPXLWyouBWLVWGNWQexSgSxsj_Qulcy4a-fN
```

This example shows the ASCII-encoded string, "When I grow up, I want to be a watermelon". The content body is shown here with line wrapping and URL-safe base64url [RFC4648] encoding to meet presentation constraints.

The keys used are shown below using the uncompressed form [X9.62] encoded using base64url.

```
Authentication Secret: BTBZMqHH6r4Tts7J_aSIgg
Receiver:
   private key: q1dXpw3UpT5VOmu_cf_v6ih07Aems3njxI-JWgLcM94
   public key: BCVxsr7N_eNgVRqvHtD0zTZsEc6-VV-JvLexhqUzORcx
               aOzi6-AYWXvTBHm4bjyPjs7Vd8pZGH6SRpkNtoIAiw4
Sender:
   private key: yfWPiYE-n46HLnH0KqZOF1fJJU3MYrct3AELtAQ-oRw
   public key: BP4z9KsN6nGRTbVYI_c7VJSPQTBtkgcy27mlmlMoZIIg
               Dll6e3vCYLocInmYWAmS6TlzAC8wEqKK6PBru3jl7A8
```

Intermediate values for this example are included in Appendix A.

6. IANA Considerations

This document does not require any IANA actions.

7. Security Considerations

The privacy and security considerations of [RFC8030] all apply to the use of this mechanism.

The Security Considerations section of [RFC8188] describes the limitations of the content encoding. In particular, no HTTP header fields are protected by the content encoding scheme. A user agent MUST consider HTTP header fields to have come from the push service.

Though header fields might be necessary for processing an HTTP response correctly, they are not needed for correct operation of the protocol. 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 which 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.

The user agent and application MUST verify that the public key they receive is on the P-256 curve. Failure to validate a public key can allow an attacker to extract a private key. The appropriate validation procedures are defined in Section 4.3.7 of [X9.62] and, alternatively, in Section 5.6.2.3 of [KEYAGREEMENT]. This process consists of three steps:

1. Verify that Y is not the point at infinity (O),

2. Verify that for Y = (x, y), both integers are in the correct interval,

3. Ensure that (x, y) is a correct solution to the elliptic curve equation.

For these curves, implementers do not need to verify membership in the correct subgroup.

In the event that this encryption scheme would need to be replaced, a new content coding scheme could be defined. In order to manage progressive deployment of the new scheme, the user agent can expose information on the content coding schemes that it supports. The "supportedContentEncodings" parameter of the Push API [API] is an example of how this might be done.

8. References

8.1. Normative References

[ECDH] SECG, "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009, <http://www.secg.org/>.

[FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015.

[FIPS186] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013.

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

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

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

[RFC8030] Thomson, M., Damaggio, E., and B. Raymor, Ed., "Generic Event Delivery Using HTTP Push", RFC 8030, DOI 10.17487/RFC8030, December 2016, <https://www.rfc-editor.org/info/rfc8030>.

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

[RFC8188] Thomson, M., "Encrypted Content-Encoding for HTTP", RFC 8188, DOI 10.17487/RFC8188, June 2017, <https://www.rfc-editor.org/info/rfc8188>.

[X9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 2005.

8.2. Informative References

[API] Beverloo, P., Thomson, M., van Ouwerkerk, M., Sullivan, B., and E. Fullea, "Push API", October 2017, <https://www.w3.org/TR/push-api/>.

[KEYAGREEMENT] Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A, Revision 2, DOI 10.6028/NIST.SP.800-56Ar2, May 2013.

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

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

[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, <https://www.rfc-editor.org/info/rfc7230>.

Appendix A. Intermediate Values for Encryption

The intermediate values calculated for the example in Section 5 are shown here. The base64url values in these examples include whitespace that can be removed.

The following are inputs to the calculation:

Plaintext: V2hlbiBJIGdyb3cgdXAsIEkgd2FudCB0byBiZSBhIHdhdGVybWVsb24

Application server public key (as_public):
BP4z9KsN6nGRTbVYI_c7VJSPQTBtkgcy27mlmlMoZIIg
Dll6e3vCYLocInmYWAmS6TlzAC8wEqKK6PBru3jl7A8

Application server private key (as_private):
yfWPiYE-n46HLnH0KqZOF1fJJU3MYrct3AELtAQ-oRw

User agent public key (ua_public): BCVxsr7N_eNgVRqvHtD0zTZsEc6-VV-
JvLexhqUzORcx aOzi6-AYWXvTBHm4bjyPjs7Vd8pZGH6SRpkNtoIAiw4

User agent private key (ua_private):
q1dXpw3UpT5VOmu_cf_v6ih07Aems3njxI-JWgLcM94

Salt: DGv6ra1nlYgDCS1FRnbzlw

Authentication secret (auth_secret): BTBZMqHH6r4Tts7J_aSIgg

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

This produces the following intermediate values:

Shared ECDH secret (ecdh_secret):
kyrL1jIIOHEzg3sM2ZWRHDRB62YACZhhSlknJ672kSs

Pseudorandom key (PRK) for key combining (PRK_key):
Snr3JMxaHVDXHWJn5wdC52WjpCtd2EIEGBykDcZW32k

Info for key combining (key_info): V2ViUHVzaDogaW5mbwAEJXGyvs3942BVG
q8e0PTNNmwR zr5VX4m8t7GGpTM5FzFo7OLr4BhZe9MEebhuPI-OztV3
ylkYfpJGmQ22ggCLDgT-M_SrDepxkU21WCP3O1SUj0Ew
bZIHMtu5pZpTKGSCIA5Zent7wmC6HCJ5mFgJkuk5cwAv MBKiiujwa7t45ewP

Input keying material for content encryption key derivation (IKM):
S4lYMb_L0FxCeq0WhDx813KgSYqU26kOyzWUdsXYyrg

PRK for content encryption (PRK):
09_eUZGrsvxChDCGRCdkLiDXrReGOEVeSCdCcPBSJSc

Info for content encryption key derivation (cek_info):
Q29udGVudC1FbmNvZGluZzogYWVzMTI4Z2NtAA

Content encryption key (CEK): oIhVW04MRdy2XN9CiKLxTg

Info for content encryption nonce derivation (nonce_info):
Q29udGVudC1FbmNvZGluZzogbm9uY2UA

Nonce (NONCE): 4h_95klXJ5E_qnoN

The salt, record size of 4096, and application server public key produce an 86-octet header of:

DGv6ra1nlYgDCS1FRnbzlwAAEABBBP4z 9KsN6nGRTbVYI_c7VJSPQTBtkgcy27ml
mlMoZIIgDll6e3vCYLocInmYWAmS6Tlz AC8wEqKK6PBru3jl7A8

The push message plaintext has the padding delimiter octet (0x02) appended to produce:

V2hlbiBJIGdyb3cgdXAsIEkgd2FudCB0 byBiZSBhIHdhdGVybWVsb24C

The plaintext is then encrypted with AES-GCM, which emits ciphertext of:

8pfeW0KbunFT06SuDKoJH9Ql87S1QUrd irN6GcG7sFz1y1sqLgVi1VhjVkHsUoEs
bI_0LpXMuGvnzQ

The header and ciphertext are concatenated and produce the result shown in Section 5.

Author's Address

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com