Web Push Encryption

Peter Beverloo <peter@chromium.org>, last updated on Thursday, February 25th, 2016.

This document describes the steps required for creating an encrypted data payload for push messages, either using the Web Push protocol or using the proprietary GCM protocol.

The steps in this document combine the encryption information in the W3C Push API, IETF Web Push Encryption and IETF Encrypted Content-Encoding For HTTP.

A flow-chart is available in this document, and an encryption validator exists on my website.

1. Subscribe for Push Messaging on the client-side, and send all information for the PushSubscription to the server. (example gist)

   The data received from the client is as follows:

   `endpoint` - endpoint of the subscription. Includes the subscription id for GCM.
   `p256dh` - public key, in uncompressed form per SEC1 2.3.3 (base64url encoded).
   `auth` - 16-byte authentication secret (base64url encoded).

2. Here we need to decide whether to use ephemeral keys for each message (better, recommended), or compute the shared secret once (computationally cheaper).

   When using ephemeral keys:

   a. Store the client information in the database.

   ```
   { client endpoint, client p256dh, client auth }
   ```

   When calculating it once:

   a. Create a P-256 public/private key-pair for the server.

   b. Compute the shared P-256 secret with the server's private key and the client's decoded public key (|`p256dh`|).

   c. Store the created public key together with the shared secret in the database, on top of the information received from the client.

   ```
   { client endpoint, client p256dh, client auth
   ```

`server public_key, server shared_key }`

3. When an event occurs for which you want to send a message.

    a. Store the to-be-encrypted payload in |`plaintext`|.

    Note that push services are not required to support more than 4,078 bytes of |`plaintext`|, assuming a message that does not contain padding.

    b. Store the decoded client's public key (p256dh) in |`client_public`|.

    c. Store the client's authentication secret in |`auth`|.

    d. Generate 16 cryptographically secure random bytes, store them in |`salt`|.

    e. When using ephemeral keys:

        i. Create a P-256 public/private key-pair for the server.

        ii. Calculate the P-256 secret with the server's private key, and the client's decoded public key (|`client_public`|).

        iii. Store the ephemeral public key to |`server_public`|, the shared secret secret to |`shared_secret`|.

    When you calculated it once:

        i. Store the server's public key in |`server_public`|, and the precalculated shared secret to |`shared_secret`|.

    f. Determine the |`content_encryption_key_info`| per the Steps for creating Info with |`type`| being the string "aesgcm".

    g. Determine the |`nonce_info`| per the Steps for creating Info with |`type`| being the string "nonce".

    h. Run the steps for encrypting the payload, store it to |`ciphertext`|.

    i. Determine the |`encryption_header`| by running the steps for creating the Encryption header with |`salt`|.

j. Determine the |`crypto_key_header`| by running the steps for creating the Crypto-Key header with |`server_public`|.

k. Add an header named `Encryption` with value |`encryption_header`|.

l. Add an header named `Crypto-Key` with value |`crypto_key_header`|..

m. When using the Web Push protocol:

   i. Add an header named `Content-Encoding` with value “aesgcm”

   ii. Have the body of the message be the raw |`ciphertext`|.

   When using the proprietary GCM protocol:

   i. Encode the |`ciphertext`| using the base64 encoding, store it in a field named “`raw_data`” in the JSON message.

n. Send the message as you normally would.

Steps for encrypting the payload

Given the |`plaintext`|, |`shared_secret`|, |`auth`|, |`salt`|, |`content_encryption_key_info`| and |`nonce_info`|, do…

1. Create an HKDF over |`shared_secret`| and |`auth`|. Extract 32 bytes using “`Content-Encoding: auth\0`” as the info-parameter, and set the result to |`prk`|.

2. Set |`hkdf`| to an HKDF over |`prk`| and |`salt`|.

   a. Extract 16 bytes using |`content_encryption_key_info`| as the info-parameter, and set the result to |`content_encryption_key`|.

   b. Extract 12 bytes using |`nonce_info`| as the info-parameter, and set the result to |`nonce`|.

3. Set |`record`| to the concatenation of (NULL-byte, NULL-byte, |`plaintext`|).

   Optional padding may be included here to hide the size of the payload. When desired, given |`pad_bytes`| as a uint16 noting the number of padding bytes to add ([0, 65535]), write the two byte value of |`pad_bytes`| in network byte order, followed by |`pad_bytes`| NULL-bytes.

4. Set |`ciphertext`| to the result of encrypting |`record`| with AEAD_AES_128_GCM, using the |`content_encryption_key`| as the key, the |`nonce`| as the nonce/IV, and an authentication tag of 16 bytes.

5. Return |`ciphertext`|.

Steps for creating Info

Given |`type`|, |`client_public`| and |`server_public`|, do…

1. Create a string |`info`|, and...

    a. Append the string “Content-Encoding: “
    b. Append the |`type`|
    c. Append a NULL-byte
    d. Append the string “P-256”
    e. Append a NULL-byte
    f. Append the length of the recipient’s public key (here |`client_public`|) as a two-byte integer in network byte order.
    g. Append the raw bytes (65) of the recipient’s public key.
    h. Append the length of the sender’s public key (here |`server_public`|) as a two-byte integer in network byte order.
    i. Append the raw bytes (65) of the sender’s public key.

2. The length of |`info`| should be 159 bytes, plus the length of |`type`|.

3. Return |`info`|.

Steps for creating the Encryption header

Given |`salt`|, do…

1. Encode |`salt`| using the URL-safe base64 encoding, store it in |`encoded_salt`|.

2. Return the result of concatenating (“salt=”, |`encoded_salt`|).

Steps for creating the Crypto-Key header

Given |`server_public`|, do…

1. Encode |`server_public`| using the URL-safe base64 encoding, store it in |`encoded_server_public`|.

2. Return the result of concatenating (“dh=”, |`encoded_server_public`|).

Flow-chart of where all the data goes