Web Push Payload Encryption

By Mat Scales
Mat is a contributor to Web**Fundamentals**

Prior to Chrome 50, push messages could not contain any payload data. When the 'push' event fired in your service worker, all you knew was that the server was trying to tell you something, but not what it might be. You then had to make a follow up request to the server and obtain the details of the notification to show, which might fail in poor network conditions.

Now in Chrome 50 (and in the current version of Firefox on desktop) you can send some arbitrary data along with the push so that the client can avoid making the extra request. However, with great power comes great responsibility, so all payload data must be encrypted.

Encryption of payloads is an important part of the security story for web push. HTTPS gives you security when communicating between the browser and your own server, because you trust the server. However, the browser chooses which push provider will be used to actually deliver the payload, so you, as the app developer, have no control over it.

Here, HTTPS can only guarantee that no one can snoop on the message in transit to the push service provider. Once they receive it, they are free to do what they like, including re-transmitting the payload to third-parties or maliciously altering it to something else. To protect against this we use encryption to ensure that push services can't read or tamper with the payloads in transit.

Client-side changes

If you have already implemented push notifications without payloads then there are only two small changes that you need to make on the client-side.

This first is that when you send the subscription information to your backend server you need to gather some extra information. If you already use `JSON.stringify()` on the PushSubscription object to serialize it for sending to your server then you don't need to change anything. The subscription will now have some extra data in the keys property.

```
> JSON.stringify(subscription)
{"endpoint":"https://android.googleapis.com/gcm/send/f1LsxkKphfQ:APA91bFUx7ja4BK4
"keys":{"p256dh":"BLc4xRzKlKORKWlbdgFaBrrPK3ydWAHo4M0gs0i1oEKgPpWC5cW8OCzVrOQRv-1
"auth":"5I2Bu2oKdyy9CwL8QVF0NQ=="}}
```

The two values **p256dh** and **auth** are encoded in a variant of Base64 that I'll call URL-Safe Base64.

If you want to get right at the bytes instead, you can use the new **getKey()** method on the subscription that returns a parameter as an **ArrayBuffer**. The two parameters that you need are **auth** and **p256dh**.

```
> new Uint8Array(subscription.getKey('auth'));
[228, 141, 129, ...] (16 bytes)

> new Uint8Array(subscription.getKey('p256dh'));
[4, 183, 56, ...] (65 bytes)
```

The second change is a new data property when the **push** event fires. It has various synchronous methods for parsing the received data, such as **.text()**, **.json()**, **.arrayBuffer()** and **.blob()**.

```
self.addEventListener('push', function(event) {
  if (event.data) {
    console.log(event.data.json());
  }
});
```

Server-side changes

On the server side, things change a bit more. The basic process is that you use the encryption key information you got from the client to encrypt the payload and then send that as the body of a POST request to the endpoint in the subscription, adding some extra HTTP headers.

The details are relatively complex, and as with anything related to encryption it's better to use an actively developed library than to roll your own. The Chrome team has published a library for Node.js, with more languages and platforms coming soon. This handles both encryption and the web push protocol, so that sending a push message from a Node.js server is as easy as **webpush.sendWebPush(message, subscription)**.

While we definitely recommend using a library, this is a new feature and there are many popular languages that don't yet have any libraries. If you do need to implement this for yourself, here are the details.

I'll be illustrating the algorithms using Node-flavored JavaScript, but the basic principles should be the same in any language.

Inputs

In order to encrypt a message, we first need to get two things from the subscription object that we received from the client. If you used `JSON.stringify()` on the client and transmitted that to your server then the client's public key is stored in the `keys.p256dh` field, while the shared authentication secret is in the `keys.auth` field. Both of these will be URL-safe Base64 encoded, as mentioned above. The binary format of the client public key is an uncompressed P-256 elliptic curve point.

```
const clientPublicKey = new Buffer(subscription.keys.p256dh, 'base64');
const clientAuthSecret = new Buffer(subscription.keys.auth, 'base64');
```

The public key allows us to encrypt the message such that it can only be decrypted using the client's private key.

Public keys are usually considered to be, well, public, so to allow the client to authenticate that the message was sent by a trusted server we also use the authentication secret. Unsurprisingly, this should be kept secret, shared only with the application server that you want to send you messages, and treated like a password.

We also need to generate some new data. We need a 16-byte cryptographically secure random salt and a public/private pair of elliptic curve keys. The particular curve used by the push encryption spec is called P-256, or `prime256v1`. For the best security the key pair should be generated from scratch every time you encrypt a message, and you should never reuse a salt.

ECDH

Let's take a little aside to talk about a neat property of elliptic curve cryptography. There is relatively simple process which combines *your* private key with *someone else's* public key to derive a value. So what? Well, if the other party takes *their* private key and *your* public key it will derive the exact same value!

This is the basis of the elliptic curve Diffie-Hellman (ECDH) key agreement protocol, which allows both parties to have the same *shared secret* even though they only exchanged public keys. We'll use this shared secret as the basis for our actual encryption key.

```
const crypto = require('crypto');

const salt = crypto.randomBytes(16);

// Node has ECDH built-in to the standard crypto library. For some languages
// you may need to use a third-party library.
const serverECDH = crypto.createECDH('prime256v1');
const serverPublicKey = serverECDH.generateKeys();
const sharedSecret = serverECDH.computeSecret(clientPublicKey);
```

HKDF

Already time for another aside. Let's say that you have some secret data that you want to use as an encryption key, but it isn't cryptographically secure enough. You can use the HMAC-based Key Derivation Function (HKDF) to turn a secret with low security into one with high security.

One consequence of the way that it works is that it allows you to take a secret of any number of bits and produce another secret of any size up to 255 times as long as a hash produced by whatever hashing algorithm you use. For push, the spec requires us to use SHA-256, which has a hash length of 32 bytes (256 bits).

As it happens, we know that we only need to generate keys up to 32 bytes in size. This means that we can use a simplified version of the algorithm that can't handle larger output sizes.

I've included the code for a Node version below, but you can find out how it actually works in RFC 5869.

The inputs to HKDF are a salt, some initial keying material (ikm), an optional piece of structured data specific to the current use-case (info) and the length in bytes of the desired output key.

```
// Simplified HKDF, returning keys up to 32 bytes long
function hkdf(salt, ikm, info, length) {
  if (length > 32) {
    throw new Error('Cannot return keys of more than 32 bytes, ${length} requeste
  }

  // Extract
```

```
  const keyHmac = crypto.createHmac('sha256', salt);
  keyHmac.update(ikm);
  const key = keyHmac.digest();

  // Expand
  const infoHmac = crypto.createHmac('sha256', key);
  infoHmac.update(info);
  // A one byte long buffer containing only 0x01
  const ONE_BUFFER = new Buffer(1).fill(1);
  infoHmac.update(ONE_BUFFER);
  return infoHmac.digest().slice(0, length);
}
```

Deriving the encryption parameters

We now use HKDF to turn the data we have into the parameters for the actual encryption.

The first thing we do is use HKDF to mix the client auth secret and the shared secret into a longer, more cryptographically secure secret. In the spec this is referred to as a Pseudo-Random Key (PRK) so that's what I'll call it here, though cryptography purists may note that this isn't strictly a PRK.

Now we create the final content encryption key and a nonce that will be passed to the cipher. These are created by making a simple data structure for each, referred to in the spec as an info, that contains information specific to the elliptic curve, sender and receiver of the information in order to further verify the message's source. Then we use HKDF with the PRK, our salt and the info to derive the key and nonce of the correct size.

The info type for the content encryption is 'aesgcm' which is the name of the cipher used for push encryption.

```
const authInfo = new Buffer('Content-Encoding: auth\0', 'utf8');
const prk = hkdf(clientAuthSecret, sharedSecret, authInfo, 32);

function createInfo(type, clientPublicKey, serverPublicKey) {
  const len = type.length;

  // The start index for each element within the buffer is:
  // value               | length | start    |
  // ----------------------------------------
  // 'Content-Encoding: '| 18     | 0        |
  // type                | len    | 18       |
  // nul byte            | 1      | 18 + len |
  // 'P-256'             | 5      | 19 + len |
  // nul byte            | 1      | 24 + len |
```

```
  // client key length   | 2      | 25 + len |
  // client key          | 65     | 27 + len |
  // server key length   | 2      | 92 + len |
  // server key          | 65     | 94 + len |
  // For the purposes of push encryption the length of the keys will
  // always be 65 bytes.
  const info = new Buffer(18 + len + 1 + 5 + 1 + 2 + 65 + 2 + 65);

  // The string 'Content-Encoding: ', as utf-8
  info.write('Content-Encoding: ');
  // The 'type' of the record, a utf-8 string
  info.write(type, 18);
  // A single null-byte
  info.write('\0', 18 + len);
  // The string 'P-256', declaring the elliptic curve being used
  info.write('P-256', 19 + len);
  // A single null-byte
  info.write('\0', 24 + len);
  // The length of the client's public key as a 16-bit integer
  info.writeUInt16BE(clientPublicKey.length, 25 + len);
  // Now the actual client public key
  clientPublicKey.copy(info, 27 + len);
  // Length of our public key
  info.writeUInt16BE(serverPublicKey.length, 92 + len);
  // The key itself
  serverPublicKey.copy(info, 94 + len);

  return info;
}

// Derive the Content Encryption Key
const contentEncryptionKeyInfo = createInfo('aesgcm', clientPublicKey, serverPubl
const contentEncryptionKey = hkdf(salt, prk, contentEncryptionKeyInfo, 16);

// Derive the Nonce
const nonceInfo = createInfo('nonce', clientPublicKey, serverPublicKey);
const nonce = hkdf(salt, prk, nonceInfo, 12);
```

Padding

Another aside, and time for a silly and contrived example. Let's say that your boss has a server that sends her a push message every few minutes with the company stock price. The plain message for this will always be a 32-bit integer with the value in cents. She also has a sneaky deal with the catering staff which means that they can send her the string "doughnuts in the break room" 5 minutes before they are actually delivered so that she can “coincidentally” be there when they arrive and grab the best one.

The cipher used by Web Push creates encrypted values that are exactly 16 bytes longer than the unencrypted input. Since "doughnuts in the break room" is longer that a 32-bit stock price, any snooping employee will be able to tell when the doughnuts are arriving without decrypting the messages, just from the length of the data.

For this reason, the web push protocol allows you to add padding to the beginning of the data. How you use this is up to your application, but in the above example you could pad all messages to be exactly 32 bytes, making it impossible to distinguish the messages based only on length.

The padding value is a 16-bit big-endian integer specifying the padding length followed by that number of `NUL` bytes of padding. So the minimum padding is two bytes - the number zero encoded into 16 bits.

```
const padding = new Buffer(2 + paddingLength);
// The buffer must be only zeroes, except the length
padding.fill(0);
padding.writeUInt16BE(paddingLength, 0);
```

When your push message arrives at the client, the browser will be able to automatically strip out any padding, so your client code only receives the unpadded message.

Encryption

Now we finally have all of the things to do the encryption. The cipher required for Web Push is AES128 using GCM. We use our content encryption key as the key and the nonce as the initialization vector (IV).

In this example our data is a string, but it could be any binary data. You can send payloads up to a size of 4078 bytes - 4096 bytes maximum per post, with 16-bytes for encryption information and at least 2 bytes for padding.

```
// Create a buffer from our data, in this case a UTF-8 encoded string
const plaintext = new Buffer('Push notification payload!', 'utf8');
const cipher = crypto.createCipheriv('id-aes128-GCM', contentEncryptionKey,
nonce);

const result = cipher.update(Buffer.concat(padding, plaintext));
cipher.final();

// Append the auth tag to the result - https://nodejs.org/api/crypto.html#crypto_
return Buffer.concat([result, cipher.getAuthTag()]);
```

Web push

Phew! Now that you have an encrypted payload, you just need to make a relatively simple HTTP POST request to the endpoint specified by the user's subscription.

You need to set three headers.

```
Encryption: salt=<SALT>
Crypto-Key: dh=<PUBLICKEY>
Content-Encoding: aesgcm
```

<SALT> and **<PUBLICKEY>** are the salt and server public key used in the encryption, encoded as URL-safe Base64.

When using the Web Push protocol, the body of the POST is then just the raw bytes of the encrypted message. However, until Chrome and Firebase Cloud Messaging support the protocol, you can easily include the data in your existing JSON payload as follows.

```
{
    "registration_ids": [ "…" ],
    "raw_data": "BIXzEKOFquzVlr/1tS1bhmobZ…"
}
```

The value of the **rawData** property must be the base64 encoded representation of the encrypted message.

Debugging / verifier

Peter Beverloo, one of the Chrome engineers who implemented the feature (as well as being one of the people who worked on the spec), has created a verifier.

By getting your code to output each of the intermediate values of the encryption you can paste them into the verifier and check that you are on the right track.

Note: Be sure to check out the full documentation including best practices for using Web Push Notifications

Last updated July 2, 2018.