Biometrics (fingerprint) auth in your web apps

Vasyl Boroviak Follow
Jan 6, 2020 · 12 min read

Here is the minimum necessary information and all the code you need to implement the fingerprint (re)sign-in, aka (re)login, aka (re)authentication feature in your Web
a]

Click to add this story to a list.

Got it

U ry bottom. Last update **2020–08–02**.

On the video below you'll see how the biometrics login experience looks.

All the below is tested and confirmed fully working in the Chrome browser.

Before diving into the code you'd need to learn some theoretical knowledge. Otherwise, you might leave your web app exposed to hackers.

Theory

Per device setup

You would need to track every device (browser) of the user. To reach that goal I would recommend your web app to have a long lived cookie (a randomly generated string ID, e.g. a UUID) to uniquely identify every device (and browser) a user has. Your backend would need to create a database record for each of the cookie used by a signed-in user.

As a result a single user would have multiple database records in your `devices` table (collection, namespace, etc). A typical user would have 1 or 2 device records. I'd recommend the cookie string to be a unique key in the table or the primary key itself.

WebAuthn

WebAuthn (aka Web Authentication) is the browser standard and **API for authenticating users** to web-based applications and services using public-key cryptography.

Public-key cryptography

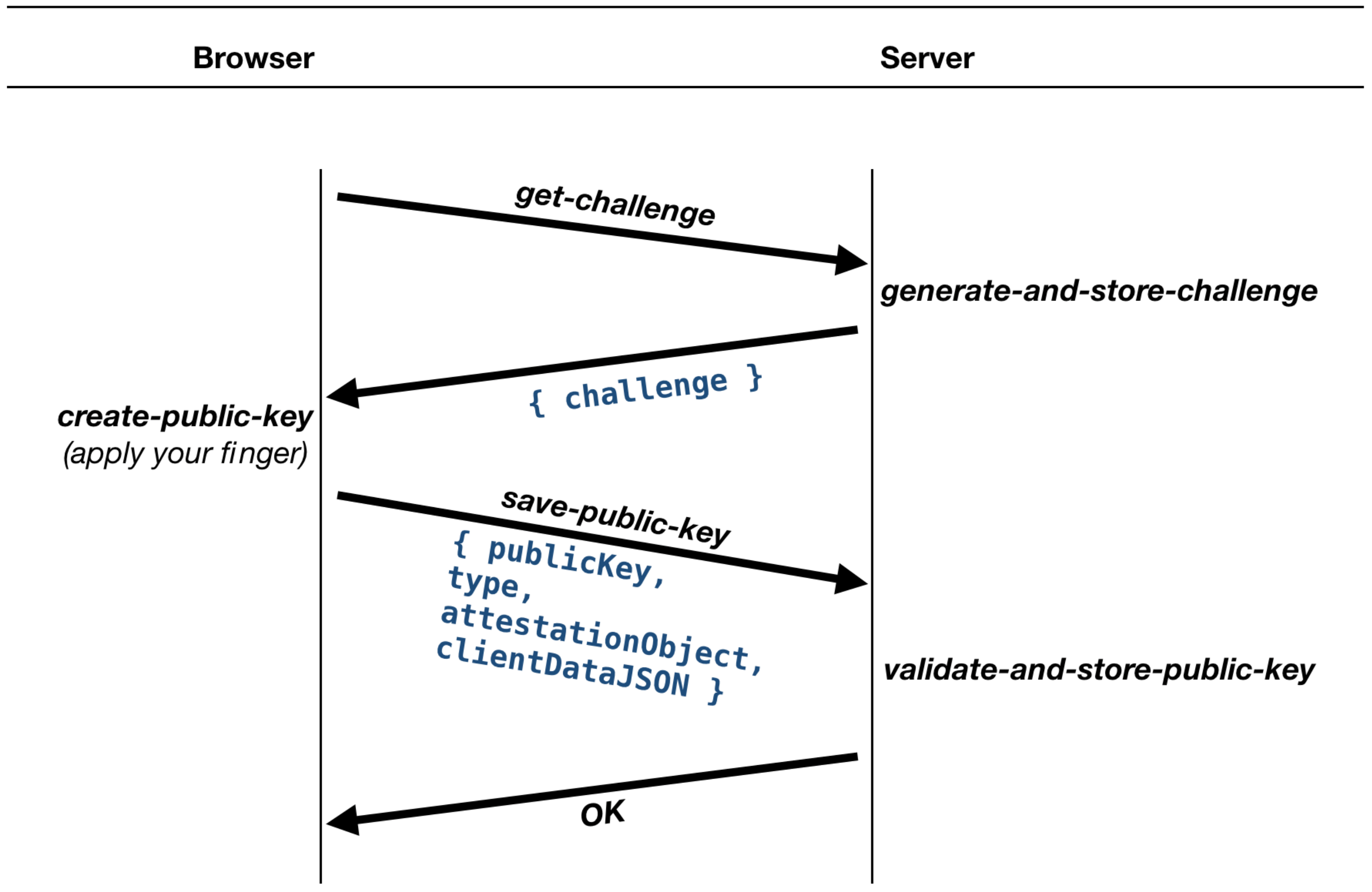
The public-key cryptography is when you can verify a signature (it's just a long string) created by a private key (it is also a string) using the previously issued public key (also a string). In short: private key encodes data, public key verifies its identity and consistency.

Authenticators

Browsers, operating systems, or other hardware devices (USB, Bluetooth, NFC, etc) serve as private key storages. They issue public keys via the above mentioned API and securely *store* a corresponding private key on the device/OS itself. These storages are called **authenticators**.

We are going to ignore the so-called "roaming" authenticators (typically USB, NFC, Bluetooth detachable devices) because they complicate the code significantly and are not the topic of this article. We will use only the so-called "platform" authenticators, which are typically the biometric scanners built into your phone/laptop.

Setup Fingerprint Flow

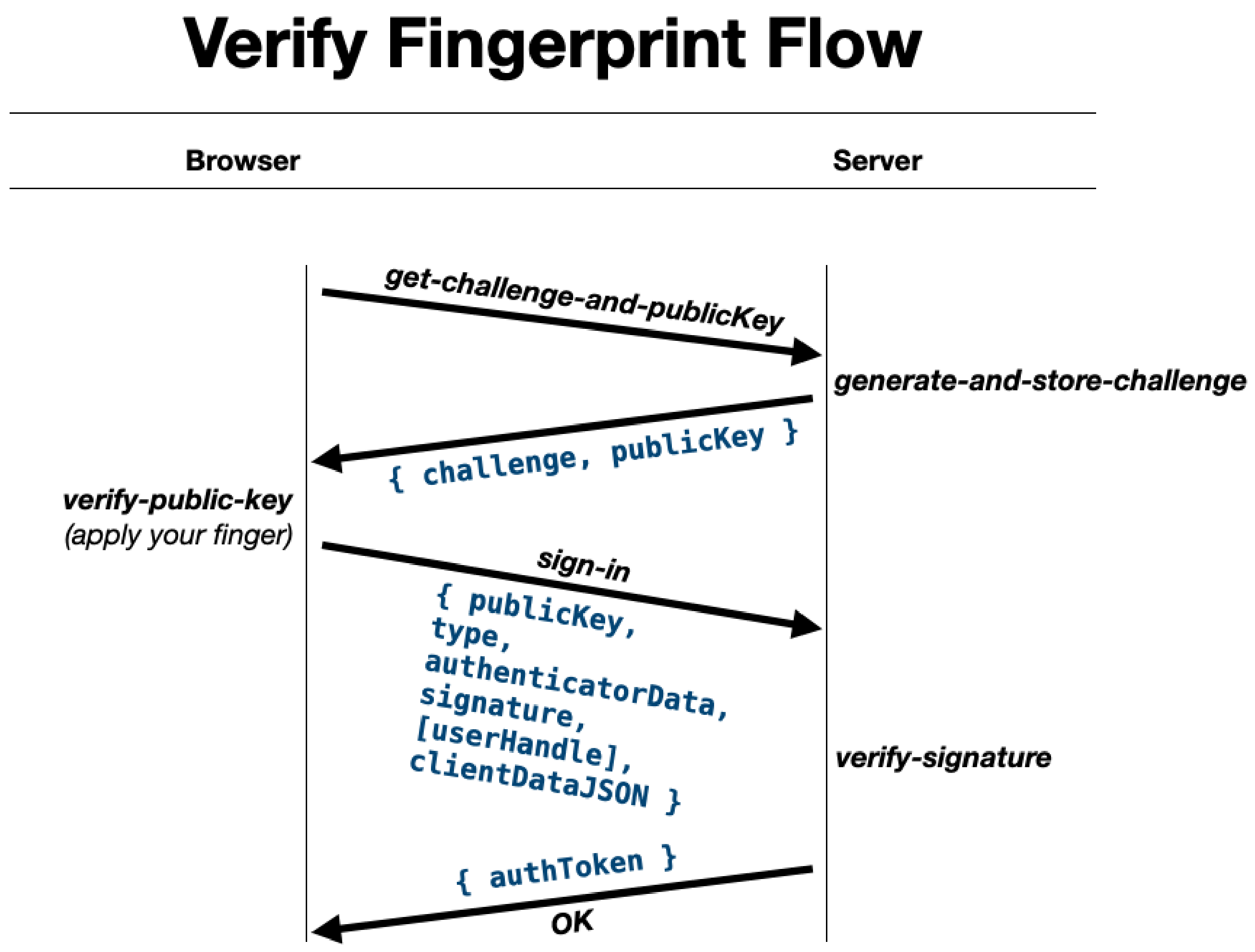


The data flow between browser and server while setting up the fingerprint re-login feature.

1. `GET get-challenge` — browser sends an empty request containing the long lived cookie (the ID of the device).
2. `generate-and-store-challenge` — server generates a new `challenge` string for that cookie (device ID) and saves it to the `devices` table.
3. `create-public-key` — browser invokes WebAuth API to create a new public key. User applies their finger.
4. `POST save-public-key` — the new public key data is sent to the server for validation and storage. Note the `attestationObject`, it’s important.
5. `validate-and-store-public-key` — validate and store the public key data inside the same `devices` record.

Verifying a fingerprint

This flow assumes that your user is *not signed in*. That's where we desperately need that unique long lived cookie (device ID) with `challenge` and `publicKey` values linked to it.



The data flow between browser and server while verifying the fingerprint and signing in the user.

1. `GET get-challenge-and-publicKey` — browser sends an empty request containing the long lived cookie (device ID).
2. The server generates a **new** `challenge` string for that cookie (device ID) and saves it instead of the existing challenge to the `devices` table. Returns both the new `challenge` and the `publicKey`.
3. `verify-public-key` — browser invokes WebAuth API to create public key verification data. User applies their finger.

4. `POST sign-in` — the assertion data is sent to the server for verification.
5. `verify-signature` — server verifies the signature using the previously stored `attestationObject` and issues authorisation token.

Let's dive deep into each of the steps.

Setup Fingerprint flow

get-challenge

First of all, your server needs an API endpoint (GraphQL query, gRPC method, etc) to return a randomly generated string called "challenge". The server must memorise this challenge value in the database into the corresponding `devices` record. It is assumed that the record was created long ago when user signed in (signed up) for the first time.

Here is a server-side pseudo code:

```
const deviceId = req.cookies["my-long-live-cookie"];
const device = await devices.findById(deviceId);
if (!device) return {};
device.challenge = require("crypto").randomBytes(16).toString("hex")
await device.save();
return { challenge: device.challenge };
```

The `devices` table minimum schema looks like this:

```
_id: String, // long lived cookie
challenge: String, // server side random generated string
counter: Number, // protects from the so called replay attacks
publicKey: String, // public key created by the fingerprint scanner
attestationObject: String, // low level device data (BASE64 binary)
clientDataJSON: String, // BASE64 encoded JSON info of the website
userAgent: String, // last seen user-agent HTTP header
user: ObjectId, // the link back to the user, aka the foreign key
```

create-public-key

This is the point when your browser invokes the WebAuthn API to scan user’s finger for the first time. (See explanation below.)

```
import { decode: base64urlDecode } from "base64url";

const attestation = await navigator.credentials.create({
    publicKey: {
        authenticatorSelection: {
            authenticatorAttachment: "platform",
            userVerification: "required"
        },
        challenge: base64urlDecode(challenge),
        rp: { id: document.domain, name: "My Acme Inc" },
        user: {
            id: base64urlDecode(user.id),
            name: user.email,
            displayName: user.fullName
        },
        pubKeyCredParams: [
            { type: "public-key", alg: -7 },
            { type: "public-key", alg: -257 }
        ]
    }
});

navigator.credentials.preventSilentAccess();

import { encode: base64urlEncode } from "base64url";

function publicKeyCredentialToJSON(pubKeyCred) {
    if (pubKeyCred instanceof ArrayBuffer) {
        return base64urlEncode(pubKeyCred);
    } else if (pubKeyCred instanceof Array) {
        return pubKeyCred.map(publicKeyCredentialToJSON);
    } else if (pubKeyCred instanceof Object) {
        const obj = {};
        for (let key in pubKeyCred) {
            obj[key] = publicKeyCredentialToJSON(pubKeyCred[key]);
        }
        return obj;
    } else return pubKeyCred;
}

const webAuthnAttestation = publicKeyCredentialToJSON(attestation);

fetch("example.com/save-public-key", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: webAuthnAttestation
});
```

The *`navigator.credentials.create`* can accept a few more various options. Although, I'm showing you the list I believe is best for fingerprint re-login purposes.

The **`authenticatorAttachment`**`:` *`"platform"`* means that you do not want people to use their, so called, *roaming* authenticators. These are not built into your device, but separate devices you attach to it (USB, NFC, etc). The "platform" means that you want to use only the authenticators built into your computer/mobile. These are mostly various biometric scanners like fingerprint, FaceID, etc.

The **`userVerification`**`:` *`"required"`* is needed to make sure the fingerprint scanner always pops up when you call the API. Otherwise, in some cases, the browser might skip the scanning. Although, I'm not aware of what are the real life scenarios of it being skipped.

The **`challenge`**`:` *`base64urlDecode`*`(challenge)` is the `challenge` value you have just read from the server.

The **`rp`**`: {` **`id`**`:` *`document.domain`*`,` **`name`**`:` *`"My Acme Inc"`* `}` stands for **R**elying **P**arty. Long story short — it's your website. The `name` might be showed to the user at the fingerprint verification stage. Btw, providing anything other than `document.domain` throws exceptions.

The **`user:`** is self descriptive I believe. These are mostly optional values. But I recommend to pass them for better UX and security.

The **`pubKeyCredParams:`** are always the same. I'm not aware how these impact.

The *`navigator.credentials.preventSilentAccess`*`()` ensures auto-sign-in never happens. However, I do not know if this call is necessary at all. See MDN docs.

The returned `webAuthnAttestation` looks like this:

```
{
    id: "a_very_very_log_string",
    type: "public-key",
    response: {
        attestationObject: "even_longer_string",
        clientDataJSON: "another_very_long_string"
    }
}
```

Where `id` is the public key.

The `attestationObject` would need to be decoded on the server side in a special way. It contains an important auth data which will be used during the fingerprint verification process. We will discuss it later.

The `clientDataJSON` is just a base64 encoded JSON (find the decoding source code below). Contains at least 3 things:

- The server generated `challenge` you are already familiar with.
- The `origin` — your website hostname.
- The `type` — it must be the following string: “webauthn.create”.

validate-and-store-public-key

This is the server side code. Firstly, you would need to validate the incoming data.

The `type` must be “public-key” string.

```
assert(type === "public-key");
```

Decoding the `clientDataJSON` string.

```
const base64url = require("base64url");
clientDataJSON = JSON.parse(base64url.decode(clientDataJSON));
```

The `clientDataJSON` checks: `challenge` must be the same as the one we sent to the browser. `origin` must be our website hostname. `type` must be “webauthn.create”.

```
assert(clientDataJSON.challenge === device.challenge)

assert(clientDataJSON.origin === "example.com");

assert(clientDataJSON.type === "webauthn.create");
```

The `attestationObject` is a bit more complex though.

Firstly, you need to parse it.

```
const base64url = require("base64url");
const cbor = require("cbor");

function parseAttestationObject(attestationObject) {
    const buffer = base64url.toBuffer(attestationObject);
    return cbor.decodeAllSync(buffer)[0];
}

const makeCredsReponse = parseAttestationObject(attestationObject);
```

The `makeCredsReponse` object contains few properties, however we are interested in only two of them: `fmt` and `authData`. I will not dive deep into this object, but you can read more about it in this excellent article by Ackermann Yuriy.

The `makeCredsReponse.fmt` must be `"none"`.

```
assert(makeCredsReponse.fmt === "none");
```

Long story short, this means you do not want to track your users’ hardware. Also, this significantly simplifies the public key verification code. And also, I found that browsers ask somewhat frightening questions if you attempt to use something different here. Like:

> Do you want this website to get access to your mobile phone's private security keys?

My instincts tell me to click "No", which I did when I saw it for the first time while playing around with the WebAuthn API.

The `makeCredsReponse.authData` will be used on the fingerprint verification stage. Just make sure it's present.

```
assert(makeCredsReponse.authData);
```

Of course, **you would need to save** `publicKey` , `type` , `attestationObject` , and `clientDataJSON` to your most secure database storage in the corresponding `devices` record.

Pseudo code:

```
device.counter = 0; // we must reset the counter

device.publicKey = publicKey;
device.type = type;
device.attestationObject = attestationObject;
device.clientDataJSON = clientDataJSON;

device.userAgent = req.headers["user-agent"];

await device.save();
```

That's all. Your user have just setup fingerprint scanner re-login.

Verify Fingerprint flow

get-challenge-and-publicKey

This is a server side endpoint (GraphQL query, gRPC method, etc). Similar to the `get-challenge` request above, your web app would need to do an empty `get-challenge-and-publicKey` request. Server would grab the long lived cookie, lookup the database by it, generate and save a new `challenge` , and return both the `challenge` and the `publicKey` .

Pseudo code:

```
const deviceId = req.cookies["my-long-live-cookie"];
const device = await devices.findById(deviceId);
if (!device) return {};
device.challenge = require("crypto").randomBytes(16).toString("hex")
await device.save();
return { challenge: device.challenge, publicKey: device.publicKey };
```

verify-public-key

Your browser invokes the WebAuthn API to scan user's finger to validate the key and the server challenge.

```
const assertionObj = await navigator.credentials.get({
    publicKey: {
        challenge: base64urlDecode(challenge),
        rpId: document.domain,
        allowCredentials: [
            {
                type: "public-key",
                id: base64urlDecode(publicKey)
            }
        ],
        userVerification: "required"
    }
});

const webAuthnAssertion = publicKeyCredentialToJSON(assertionObj);

fetch("example.com/sign-in", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: webAuthnAssertion
});
```

The `navigator.credentials.get` will ask our user to scan their finger.

The `challenge`: is the value we have just received from the server.

The `rpId`: must be the same as before. Again, values other than the website domain name didn't work in my tests.

The **`allowCredentials`**: is the list of public keys the user is validating. Yeah, there can be multiple. I would not recommend more than one here for simplicity reasons.

The **`type`**: *`"public-key"`* is a must.

The **`id`**: *`base64urlDecode`*`(publicKey)` is the `publicKey` value we've just received from the server.

The **`userVerification`**: *`"required"`* is the same as earlier. It makes sure the user is always involved (applies their finger).

The returned `webAuthnAssertion` looks like this:

```
{
    id: "same_very_very_log_string",
    type: "public-key",
    response: {
        authenticatorData: "long_string",
        clientDataJSON: "similar_long_string",
        signature: "not_long_but_still_string",
        userHandle: "your_user_id"
    }
}
```

Where `id` is the same public key.

The `authenticatorData` would need to be decoded on the server side in a special way. It contains an important auth data which will be used during the fingerprint verification process on the server.

The `clientDataJSON` is the same `clientDataJSON` as above, — a base64 encoded JSON:

- The good old server generated `challenge` string.
- The `origin` — your website hostname.
- The `type` — it must be the following string: "webauthn.get".

The `signature` is the main cryptographic value our server is going to verify.

The `userHandle` should be the `user.id` value we provided during the setup phase. Although, the specification says that `userHandle` is **optional** and can be missing.

verify-signature

Here goes the most important part of the whole thing. Please, implement it on your server-side without shortcuts.

Here is the list of checks your server would need to perform one by one.

1. Make sure the `user-agent` (the browser) of the `sign-in` request is the same as during the **validate-and-store-public-key** server call (the setup). Make sure you cater for the version change. For example, the Chrome browser updates the version every 6 weeks.

2. Make sure the `id` (aka the `publicKey`) and the `type` are exactly the same as during the setup. These should be stored in your `devices` table.

   ```
   assert(publicKey === device.publicKey);
   ```

3. If `userHandle` was provided then compare it too.

   ```
   if (userHandle) assert(userHandle === device.user);
   ```

4. Parse the `clientDataJSON` (same as before) and check:
   * `challenge` is exactly the same as during the setup (stored in your `devices` table).

   ```
   assert(clientDataJSON.challenge === device.challenge);
   ```

   * `origin` is exactly as your website's domain name.

   ```
   assert(clientDataJSON.origin === "example.com");
   ```

   * `type` is exactly this string — "*webauthn.get*".

   ```
   assert(clientDataJSON.type === "webauthn.get");
   ```

5. Cryptographic signature verification. This is the most difficult part of the puzzle. The code is long and can be confusing. I copied (and modified) the logic from webauthn for Express.js project.

Full source code of the server-side signature verification

The `verifyAssertion` function requires a bit of explanation. It verifies the received data using the received signature and the previously stored public key.

The arguments are:

- `counter` — the `counter` value of the assertion during previous re-login (or `0` if re-login is happening for the first time) stored in the `device.counter`. Browser increases this number by a random value every time the *`navigator.credentials.get`* is called. **This `counter` is one of the reasons why fingerprint re-login feature is as secure as the traditional 2FA solutions.**
  It is used to avoid the so called "replay attacks". When the authentication data (like login with password, or in our case `assertionObject`) is captured by a bad person,

and then resent to our server over and over again to re-login at will from any device out there. Thanks to this `counter` the `signature` value is never the same!

- `attestationObject` — the previously provided string stored in the `device.attestationObject`. See the **Setup Fingerprint flow** above. (If you look closely at the code you probably won't need to store whole string, maybe just the `COSEPublicKey` part of it.)
- `clientDataJSON`, `authenticatorData`, `signature` — are the values sent from the browser.

In case of success the `verifyAssertion` function returns the new `counter` value. You must save it to your database next to the `attestationObject` value. Then you need to issue new `authToken` (aka create a new login session for the user linked to this device) and return it back to the browser.

The function throws if something goes wrong. I'd recommend sending security alert (emails?) every time this function throws. This would most likely mean you are getting hacked.

Server pseudo code:

```
const base64url = require("base64url");

const deviceId = req.cookies["my-long-live-cookie"];
const device = await devices.findById(deviceId);
try {
    const { publicKey, type, authenticatorData, clientDataJSON,
signature, userHandle } = req.body;

    assert(type === "public-key");
    assert(publicKey === device.publicKey);
    if (userHandle) assert(userHandle === device.user);

    const clientData = JSON.parse(base64url.decode(clientDataJSON));

    assert(clientData.challenge === device.challenge);
    assert(clientData.origin === "example.com");
    assert(clientData.type === "webauthn.get");
```

```
        const newCounter = verifyAssertion({
            counter: device.counter,
            attestationObject: device.attestationObject,
            clientDataJSON,
            authenticatorData,
            signature
        });
        device.counter = newCounter;
        await device.save();

        res.status(200).send({
            authToken: CREATE_NEW_AUTH_TOKEN(device.user)
        });
    } catch (err) {
        res.sendStatus(400);
        console.error(err);
        NOTIFY_SECURITY_OF_A_HACK_ATTEMPT();
    }
```

That's all folks!

Disclaimers

As far as I understand, even if someday someone would steal the `devices` database table (it contains the `challenge` , the `publicKey` , and the `attestationObject` secure values), your users will stay protected! Because the private key is still stored on the user's device. In other words, no one can login with that stolen data.

> *Please note, I'm not a high grade cryptography and security expert. You are invited to find flaws in this article. It would be really appreciated.*

I intentionally kept this blog post as minimalistic as possible. I'm not diving into much details of the WebAuthn API and cryptography. The purpose is to give non-expert developers just enough information to quickly implement the fingerprint (or other biometrics) re-login function in a secure way.

The UI on the video above was developed by Alexander — https://twitter.com/AlexanderKon. He's available for web development projects.

The application demonstrated is https://flash-fx.com, we are an Australian company doing international money transfers using blockchain technology (Ripple).

Updates

2020–08–02:

> *1. A new* `challenge` *must be generated every time a browser requests for it.*
> *Previously, the article showed that challenge is never re-generated.*
>
> *2. Generate* `challenge`*s using* `Buffer.randomBytes`*.*
> *Previously, the article used UUID as a challenge generator, which is considered not random enough in the InfoSec world.*

Thanks to Kristian Dupont and Kiarash Irandoust.

JavaScript Biometrics Security Programming Web Development