Passkeys: What the Heck and Why?

Neal Fennimore on Apr 12, 2023

These things called **passkeys** sure are making the rounds these days. They were a main attraction at W3C TPAC 2022 (https://www.w3.org/2022/09/TPAC/demos/passkeys.html) , gained support in Safari 16 (https://developer.apple.com/documentation/safari-release-notes/safari-16_1-release-notes/) , are finding their way into macOS and iOS (https://developer.apple.com/passkeys/) , and are slated to be the future for password managers like 1Password (https://www.future.1password.com/passkeys/) . They are already supported (https://passkeys.dev/device-support/) in Android, and will soon find their way into Chrome OS and Windows in future releases.

Geeky OS security enhancements don't exactly make big headlines in the front-end community, but it stands to reason that passkeys are going to be a "thing". And considering how passwords and password apps affect the user experience of things like authentication and form processing, we might want to at least wrap our minds around them, so we know what's coming.

That's the point of this article. I've been studying and experimenting with passkeys — and the WebAuthn API they are built on top of — for some time now. Let me share what I've learned.

(#aa-table-of-contents) Table of contents

(#aa-terminology) Terminology

Here's the obligatory section of the terminology you're going to want to know as we dig in. Like most tech, passkeys are wrought with esoteric verbiage and acronyms that are often roadblocks to understanding. I'll try to de-mystify several for you here.

- **Relying Party:** the server you will be authenticating against. We'll use "server" to imply the Relying Party in this article.
- **Client:** in our case, the web browser or operating system.
- **Authenticator:** Software and/or hardware devices that allow generation and storage for public key pairs.
- **FIDO**: An open standards body that also creates specifications around FIDO credentials.
- **WebAuthn**: The underlying protocol for passkeys, Also known as a FIDO2 (https://fidoalliance.org/fido2/) credential or single-device FIDO credentials.
- **Passkeys**: WebAuthn, but with cloud syncing (also called multi-device FIDO credentials, discoverable credentials, or resident credentials).
- **Public Key Cryptography:** A generated key pair that includes a private and public key. Depending on the algorithm, it should either be used for signing and verification or encrypting and decrypting. This is also known as *asymmetric cryptography*.
- **RSA:** An acronym of the creators' names, Rivest Shamir and Adel. RSA is an older, but still useful, family of public key cryptography based on factoring primes.
- **Elliptic Curve Cryptography (ECC):** A newer family of cryptography based on elliptic curves (https://csrc.nist.gov/Projects/Elliptic-Curve-Cryptography) .
- **ES256:** An elliptic curve public key that uses an ECDSA signing algorithm (PDF (https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf)) with SHA256 (https://en.wikipedia.org/wiki/SHA-2) for hashing.

- **RS256:** Like ES256, but it uses RSA with RSASSA-PKCS1-v1.5 (https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/sign#rsassa-pkcs1-v1_5_2) and SHA256.

(#aa-what-are-passkeys) What are passkeys?

Before we can talk specifically about passkeys, we need to talk about another protocol called WebAuthn (https://webauthn.guide/) (also known as FIDO2). Passkeys are a specification that is built on top of WebAuthn. WebAuthn allows for public key cryptography to replace passwords. We use some sort of security device, such as a hardware key or Trusted Platform Module (https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-top-node) (TPM), to create private and public keys.

The public key is for anyone to use. The private key, however, cannot be removed from the device that generated it. This was one of the issues with WebAuthn; if you lose the device, you lose access.

Passkeys solves this by providing a cloud sync of your credentials. In other words, what you generate on your computer can now also be used on your phone (though confusingly, there are single-device credentials too).

Currently, at the time of writing, only iOS, macOS, and Android provide full support for cloud-synced passkeys, and even then, they are limited by the browser being used. Google and Apple provide an interface for syncing via their Google Password Manager (https://passwords.google.com/) and Apple iCloud Keychain (https://support.apple.com/en-us/HT204085) services, respectively.

(#aa-how-do-passkeys-replace-passwords) How do passkeys replace passwords?

In public key cryptography, you can perform what is known as *signing*. Signing takes a piece of data and then runs it through a signing algorithm with the private key, where it can then be verified with the public key.

Anyone can generate a public key pair, and it's not attributable to any person since any person could have generated it in the first place. What makes it useful is that only data signed with the private key can be verified with the public key. That's the portion that replaces a password — a server stores the public key, and we sign in by verifying that we have the other half (e.g. private key), by signing a random challenge.

As an added benefit, since we're storing the user's public keys within a database, there is no longer concern with password breaches affecting millions of users. This reduces phishing, breaches, and a slew of other security issues that our password-dependent world currently faces. If a database is breached, all that's stored in the user's public keys, making it virtually useless to an attacker.

No more forgotten emails and their associated passwords, either! The browser will remember which credentials you used for which website — all you need to do is make a couple of clicks, and you're logged in. You can provide a secondary means of verification to use the passkey, such as biometrics or a pin, but those are still much faster than the passwords of yesteryear.

(#aa-more-about-cryptography) More about cryptography

Public key cryptography involves having a private and a public key (known as a key pair). The keys are generated together and have separate uses. For example, the private key is intended to be kept secret, and the public key is intended for whomever you want to exchange messages with.

When it comes to encrypting and decrypting a message, the recipient's public key is used to encrypt a message so that only the recipient's private key can decrypt the message. In security parlance, this is known as "providing confidentiality". However, this doesn't provide proof that the sender is who they say they are, as anyone can potentially use a public key to send someone an encrypted message.

There are cases where we need to verify that a message did indeed come from its sender. In these cases, we use signing and signature verification to ensure that the sender is who they say they are (also known as *authenticity*). In public key (also called *asymmetric*) cryptography, this is generally done by signing the hash of a message, so that only the public key can correctly verify it. The hash and the sender's private key produce a signature after running it through an algorithm, and then anyone can verify the message came from the sender with the sender's public key.

(#aa-how-do-we-access-passkeys) How do we access passkeys?

To access passkeys, we first need to generate and store them somewhere. Some of this functionality can be provided with an authenticator. An *authenticator* is any hardware or software-backed device that provides the ability for cryptographic key generation. Think of those one-time passwords you get from Google Authenticator (https://support.google.com/accounts/answer/1066447?hl=en&co=GENIE.Platform%3DAndroid) , 1Password (https://1password.com/) , or LastPass (https://www.lastpass.com/) , among others.

For example, a software authenticator can use the Trusted Platform Module (TPM) or secure enclave of a device to create credentials. The credentials can be then stored remotely and synced across devices e.g. passkeys. A hardware authenticator would be something like a YubiKey (https://www.yubico.com/) , which can generate and store keys on the device itself.

To access the authenticator, the browser needs to have access to hardware, and for that, we need an interface. The interface we use here is the Client to Authenticator Protocol (CTAP). It allows access to different authenticators over different mechanisms. For example, we can access an authenticator over NFC, USB, and Bluetooth by utilizing CTAP.

One of the more interesting ways to use passkeys is by connecting your phone over Bluetooth to another device that might not support passkeys. When the devices are paired over Bluetooth, I can log into the browser on my computer using my phone as an intermediary!

(#aa-the-difference-between-passkeys-and-webauthn) The difference between passkeys and WebAuthn

Passkeys and WebAuthn keys differ in several ways. First, passkeys are considered multi-device credentials and can be synced across devices. By contrast, WebAuthn keys are single-device credentials — a fancy way of saying you're bound to one device for verification.

Second, to authenticate to a server, WebAuthn keys need to provide the user handle for login, after which an `allowCredentials` list is returned to the client from the server, which informs what credentials can be used to log in. **Passkeys skip this step and use the server's domain name to show which keys are already bound to that site.** You're able to select the passkey that is associated with that server, as it's already known by your system.

Otherwise, the keys are cryptographically the same; they only differ in how they're stored and what information they use to start the login process.

(#aa-the-process-in-a-nutshell) The process… in a nutshell

The process for generating a WebAuthn or a passkey is very similar: get a challenge from the server and then use the `navigator.credentials.create` web API to generate a public key pair. Then, send the challenge and the public key back to the server to be stored.

Upon receiving the public key and challenge, the server validates the challenge and the session from which it was created. If that checks out, the public key is stored, as well as any other relevant information like the user identifier or attestation data, in the database.

The user has one more step — retrieve another challenge from the server and use the `navigator.credentials.get` API to sign the challenge. We send back the signed challenge to the server, and the server verifies the challenge, then logs us in if the signature passes.

There is, of course, quite a bit more to each step. But that is generally how we'd log into a website using WebAuthn or passkeys.

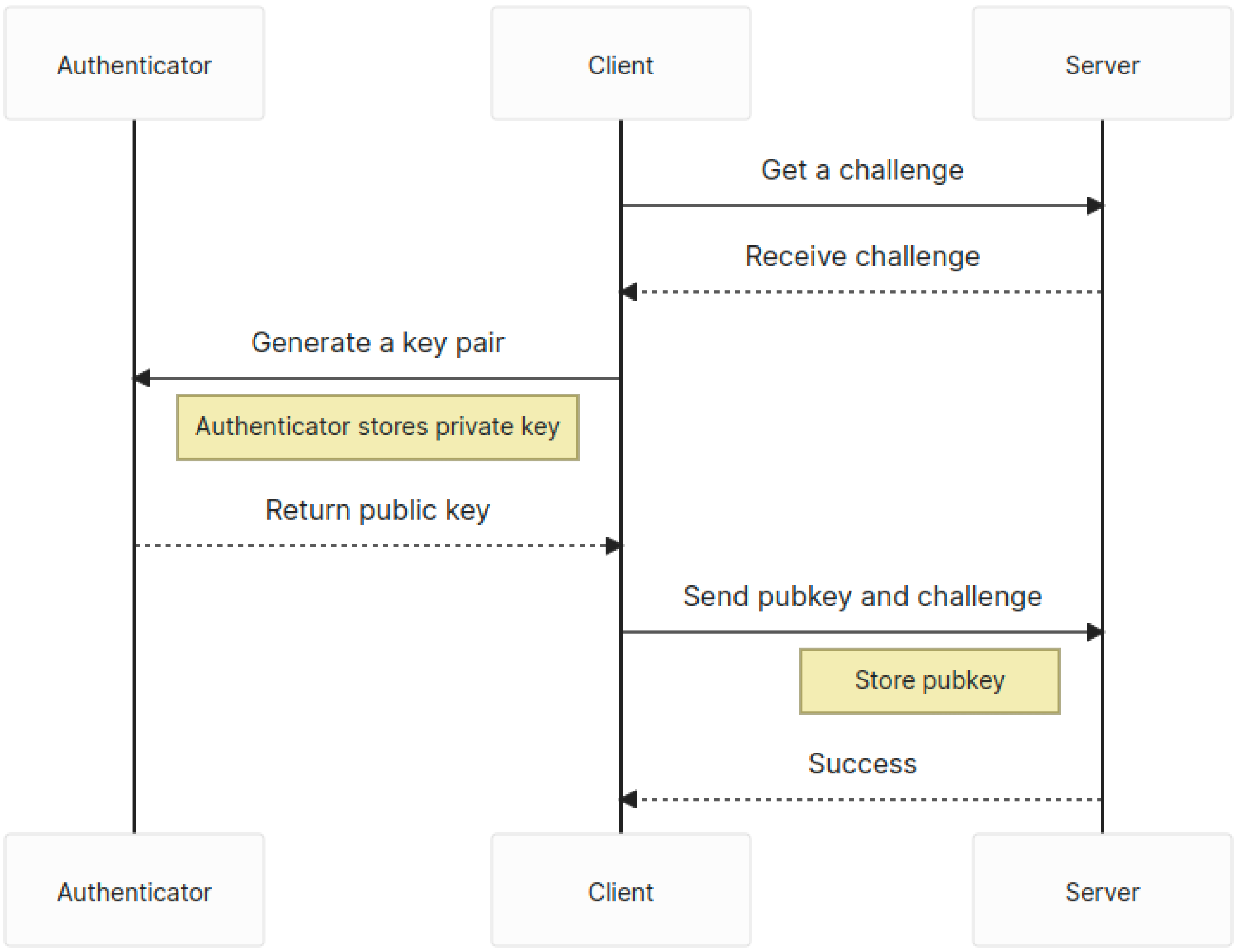
(#aa-the-meat-and-potatoes) The meat and potatoes

Passkeys are used in two distinct phases: the **attestation** and **assertion** phases.

The attestation phase can also be thought of as the registration phase. You'd sign up with an email and password for a new website, however, in this case, we'd be using our passkey.

The assertion phase is similar to how you'd log in to a website after signing up.

(#aa-attestation) Attestation



(https://i0.wp.com/css-tricks.com/wp-content/uploads/2023/02/attestation.png?ssl=1)

View full size (https://css-tricks.com/wp-content/uploads/2023/02/attestation.png)

The `navigator.credentials.create` API is the focus of our attestation phase. We're registered as a new user in the system and need to generate a new public key pair. However, we need to specify what kind of key pair we want to generate. That means we need to provide options to `navigator.credentials.create`.

```
// The `challenge` is random and has to come from the server
const publicKey: PublicKeyCredentialCreationOptions = {
  challenge: safeEncode(challenge),
  rp: {
    id: window.location.host,
    name: document.title,
  },
  user: {
    id: new TextEncoder().encode(crypto.randomUUID()), // Why not make it random?
    name: 'Your username',
    displayName: 'Display name in browser',
  },
  pubKeyCredParams: [
    {
      type: 'public-key',
      alg: -7, // ES256
    },
    {
      type: 'public-key',
      alg: -256, // RS256
    },
  ],
  authenticatorSelection: {
    userVerification: 'preferred', // Do you want to use biometrics or a pin?
    residentKey: 'required', // Create a resident key e.g. passkey
  },
  attestation: 'indirect', // indirect, direct, or none
  timeout: 60_000,
};
```

```
const pubKeyCredential: PublicKeyCredential = await navigator.credentials.create({
  publicKey
});
const {
  id // the key id a.k.a. kid
} = pubKeyCredential;
const pubKey = pubKeyCredential.response.getPublicKey();
const { clientDataJSON, attestationObject } = pubKeyCredential.response;
const { type, challenge, origin } = JSON.parse(new TextDecoder().decode(clientDataJSON));
// Send data off to the server for registration
```

We'll get `PublicKeyCredential` (https://developer.mozilla.org/en-US/docs/Web/API/PublicKeyCredential) which contains an `AuthenticatorAttestationResponse` (https://developer.mozilla.org/en-US/docs/Web/API/AuthenticatorAttestationResponse) that comes back after creation. The credential has the generated key pair's ID.

The response provides a couple of bits of useful information. First, we have our public key in this response, and we need to send that to the server to be stored. Second, we also get back the `clientDataJSON` property which we can decode, and from there, get back the `type`, `challenge`, and `origin` of the passkey.

For attestation, we want to validate the `type`, `challenge`, and `origin` on the server, as well as store the public key with its identifier, e.g. kid. We can also optionally store the `attestationObject` if we wish. Another useful property to store is the COSE (https://www.iana.org/assignments/cose/cose.xhtml#algorithms) algorithm, which is defined above in our `PublicKeyCredentialCreationOptions` with `alg: -7` or `alg: -256`, in order to easily verify any signed challenges in the assertion phase.

(#aa-assertion) Assertion

(https://i0.wp.com/css-tricks.com/wp-content/uploads/2023/02/assertion.png?ssl=1)

View full size (https://css-tricks.com/wp-content/uploads/2023/02/assertion.png)

The `navigator.credentials.get` API will be the focus of the assertion phase.
Conceptually, this would be where the user logs in to the web application after signing up.

```
// The `challenge` is random and has to come from the server
const publicKey: PublicKeyCredentialRequestOptions = {
  challenge: new TextEncoder().encode(challenge),
  rpId: window.location.host,
  timeout: 60_000,
};
```

```
const publicKeyCredential: PublicKeyCredential = await navigator.credentials.get({
  publicKey,
  mediation: 'optional',
});
const {
  id // the key id, aka kid
} = pubKeyCredential;
const { clientDataJSON, attestationObject, signature, userHandle } = pubKeyCredential.response;
const { type, challenge, origin } = JSON.parse(new TextDecoder().decode(clientDataJSON));
// Send data off to the server for verification
```

We'll again get a `PublicKeyCredential` (https://developer.mozilla.org/en-US/docs/Web/API/PublicKeyCredential) with an `AuthenticatorAssertionResponse` (https://developer.mozilla.org/en-US/docs/Web/API/AuthenticatorAssertionResponse) this time. The credential again includes the key identifier.

We also get the `type`, `challenge`, and `origin` from the `clientDataJSON` again.
The `signature` is now included in the response, as well as the `authenticatorData`. We'll need those and the `clientDataJSON` to verify that this was signed with the private key.

The `authenticatorData` (https://developer.mozilla.org/en-US/docs/Web/API/AuthenticatorAssertionResponse/authenticatorData) includes some properties that are worth tracking First is the SHA256 hash of the origin you're using, located within the first 32 bytes, which is useful for verifying that request comes from the same origin server. Second is the `signCount`, which is from byte 33 to 37. This is generated from the authenticator and should be compared to its previous value to ensure that nothing fishy is going on with the key. The value should always 0 when it's a multi-device passkey and should be randomly larger than the previous signCount when it's a single-device passkey.

Once you've asserted your login, you should be logged in — *congratulations*! Passkeys is a pretty great protocol, but it does come with some caveats.

(#aa-some-downsides) Some downsides

There's a lot of upside to Passkeys, however, there are some issues with it at the time of this writing. For one thing, passkeys is somewhat still early support-wise, with only single-device credentials allowed on Windows and very little support for Linux systems. Passkeys.dev (http://passkeys.dev/) provides a nice table that's sort of like the Caniuse of this protocol (https://passkeys.dev/device-support/) .

Also, Google's and Apple's passkeys platforms do not communicate with each other. If you want to get your credentials from your Android phone over to your iPhone… well, you're out of luck for now. That's not to say there is no interoperability! You can log in to your computer by using your phone as an authenticator. But it would be much cleaner just to have it built into the operating system and synced without it being locked at the vendor level.

(#aa-where-are-things-going) Where are things going?

What does the passkeys protocol of the future look like? It looks pretty good! Once it gains support from more operating systems, there should be an uptake in usage, and you'll start seeing it used more and more in the wild. Some password managers (https://www.future.1password.com/passkeys) are even going to support them first-hand.

Passkeys are by no means only supported on the web. Android (https://developer.android.com/training/sign-in/passkeys) and iOS (https://developer.apple.com/documentation/authenticationservices/public-private_key_authentication/supporting_passkeys) will both support native passkeys as first-class citizens. We're still in the early days of all this, but expect to see it mentioned more and more.

After all, we eliminate the need for passwords, and by doing so, make the world safer for it!

(#aa-resources) Resources

Here are some more resources if you want to learn more about Passkeys. There's also a repository and demo I put together for this article.

- Live Demo (https://passkeys.neal.codes/) (no actual information is collected by the form)
- Demo GitHub Repository (https://github.com/nealfennimore/passkeys)
- YubiKey Documentation (https://developers.yubico.com/Passkeys/)
- Passkeys.dev (https://passkeys.dev/)
- Passkeys.io (https://www.passkeys.io/)
- Webauthn.io (https://webauthn.io/)