Passkeys: the future of user authentication

Dr. Philippe De Ryck

https://PragmaticWebSecurity.com

What is the current state of user authentication?

News > Privacy

Gates predicts death of the password

Traditional password-based security is headed for extinction, says Microsoft's chairman, because it cannot "meet the challenge" of keeping critical information secure.

Munir Kotadia

Feb. 25, 2004 1:27 p.m. PT

3 min read

SAN FRANCISCO--Microsoft Chairman Bill Gates predicted the demise of the traditional password because it cannot "meet the challenge" of keeping critical information secure.

Gates, speaking at the RSA Security conference here on Tuesday, said: "There is no doubt that over time, people are going to rely less and less on passwords. People use the same password on different systems, they write them down and they just don't meet the challenge for anything you really want to secure."

https://www.cnet.com/news/privacy/gates-predicts-death-of-the-password/

Will passwords become obsolete soon?

Password Manager Pro | December 12, 2013 | 3 min read

Will passwords soon become a thing of the past? Have they already become obsolete? This is perhaps one of the most prominent topics under discussion in the technical media these days.

A couple of weeks ago, Forbes.com published a story about the probable public launch of U2F (Universal Second Factor) – a new form of authentication by Google in alliance with Yubico. Through U2F, Google wants "to help move the web towards easier and stronger authentication, where web users can own a single easy-to-use secure authentication device built on open standards, which works across the entire web." Media reports following the story have fuelled wild speculations that traditional passwords will soon die.

https://blogs.manageengine.com/it-security/passwordmanagerpro/2013/12/12/will-passwords-become-obsolete-soon.html

SAFETY & SECURITY

The beginning of the end of the password

May 03, 2023

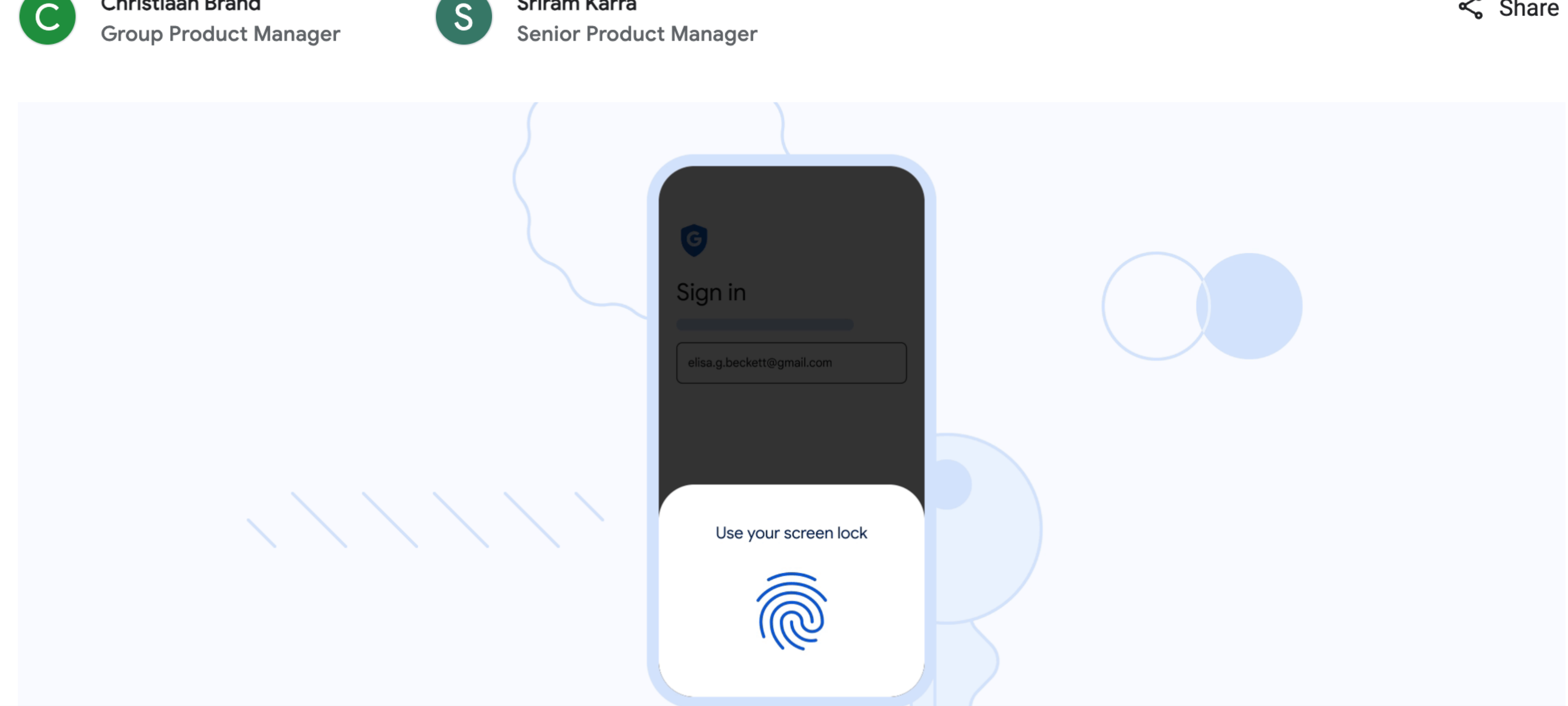
1 min read

For the first time, we've begun rolling out passkeys, the easiest and most secure way to sign in to apps and websites and a major step toward a "passwordless future."

Christiaan Brand
Group Product Manager

Sriram Karra
Senior Product Manager

Share



https://blog.google/technology/safety-security/the-beginning-of-the-end-of-the-password/

I am *Dr. Philippe De Ryck*

Founder of Pragmatic Web Security

Google Developer Expert

SecAppDev organizer

I help developers with security

Hands-on in-depth security training

Advanced online security courses

Security advisory services

https://pdr.online

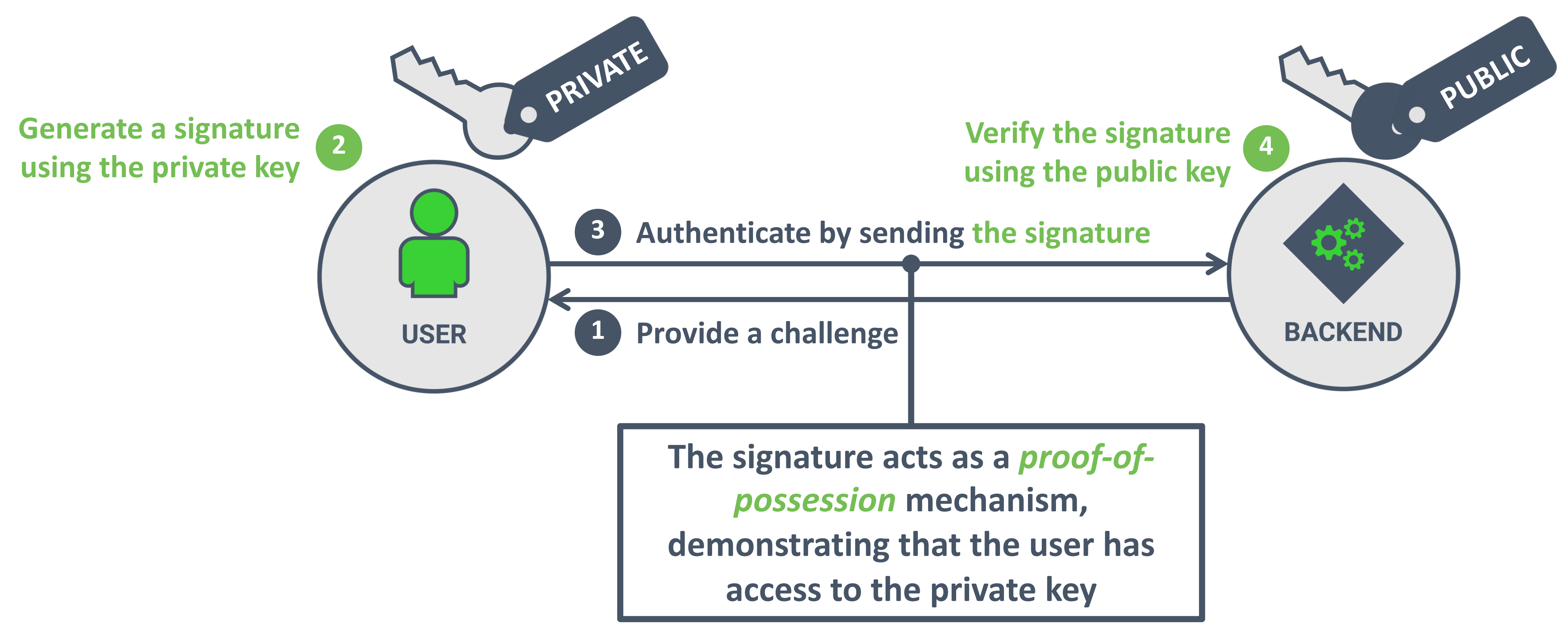
What are requirements for a good and secure user authentication system?

REQUIREMENTS FOR SECURE USER AUTHENTICATION

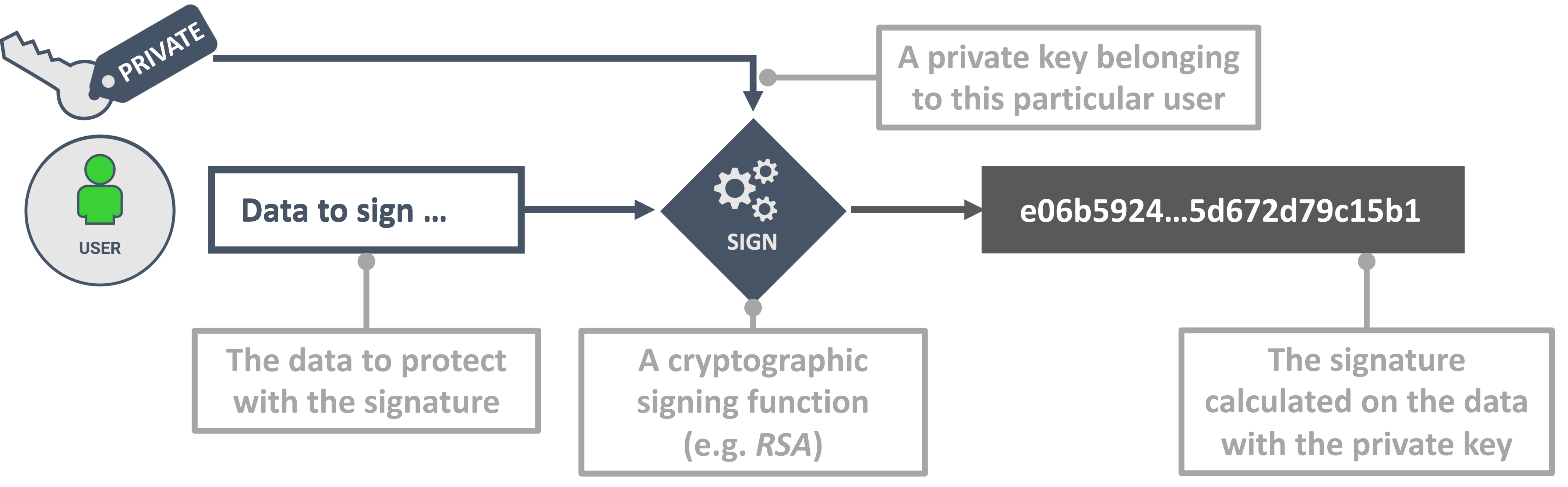
- Simple user experience that inspires secure use
  - Easy to select and use
  - Nothing to remember to avoid re-use across different applications
- Resistant against common attacks against authentication mechanisms
  - Not subject to guessing or brute force attacks
  - Protected against phishing attacks
- Easily portable across different devices
  - Users authenticate on computers, phones, tablets, etc.
  - Ideally, an authentication factor is portable across devices, even on less-trusted devices

Most current authentication mechanisms (e.g., passwords, TOTP codes, ...) do not meet these security requirements

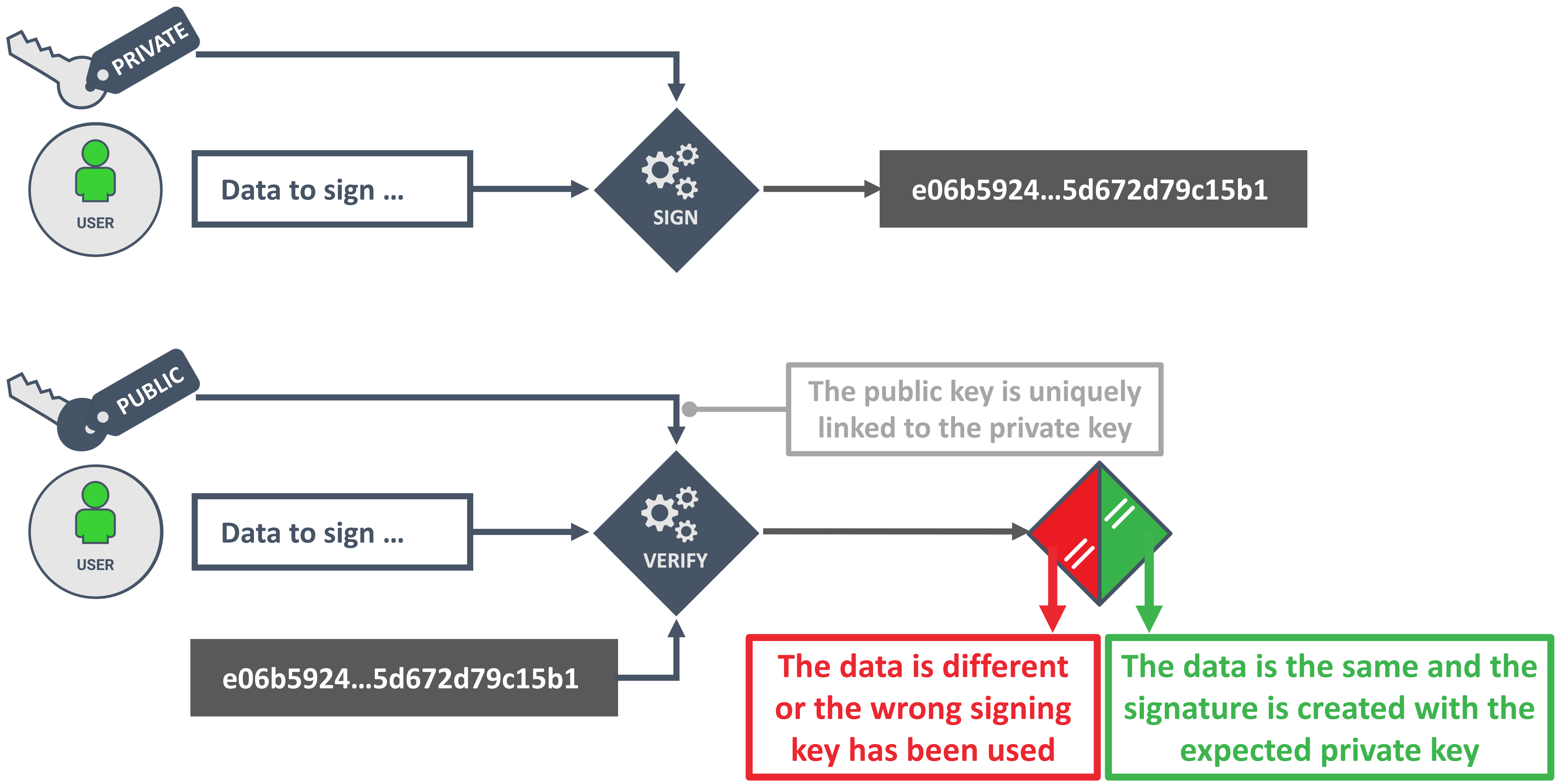
KEY-BASED USER AUTHENTICATION



INTERMEZZO: DIGITAL SIGNATURES



INTERMEZZO: DIGITAL SIGNATURES



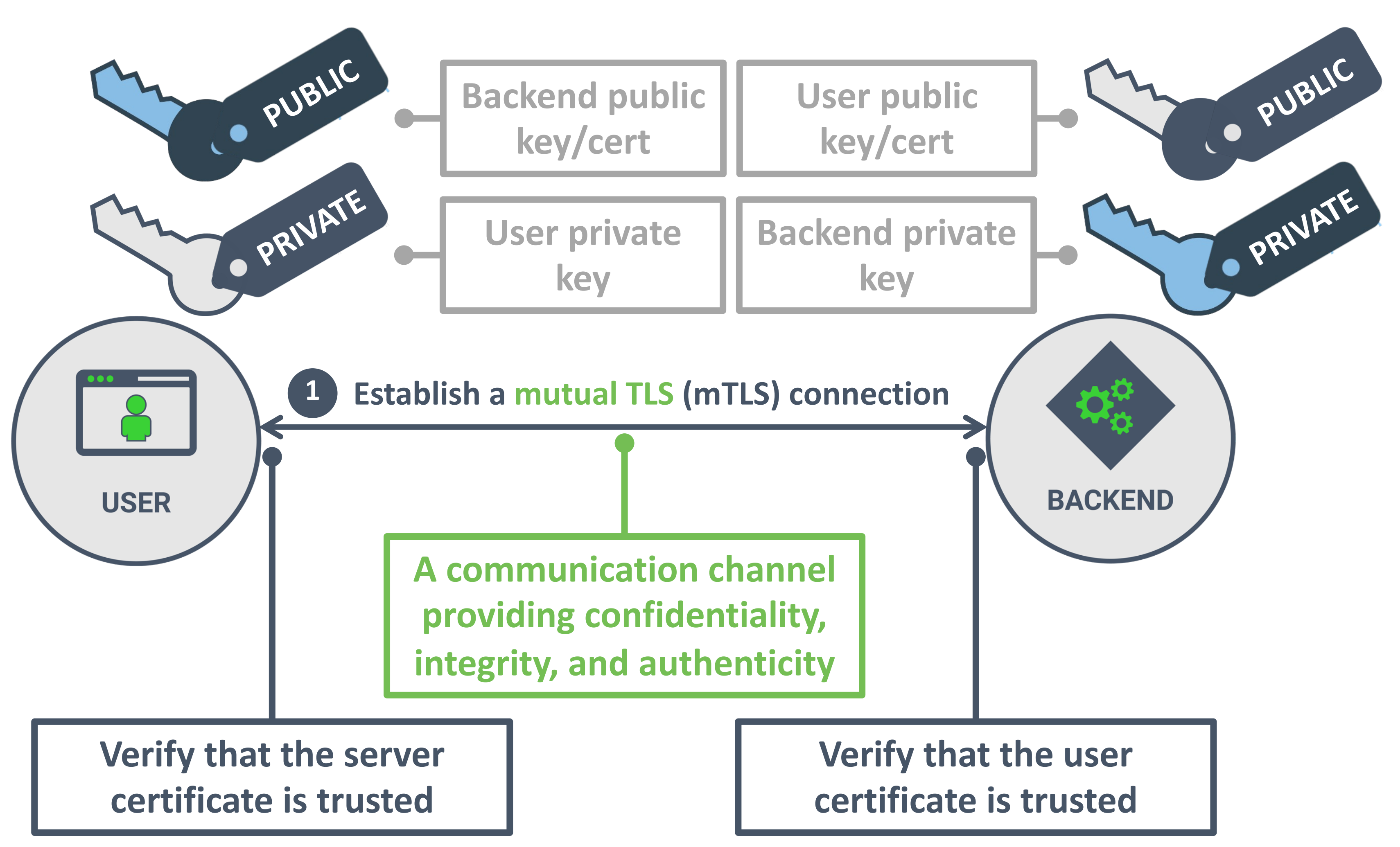
Key-based authentication in practice

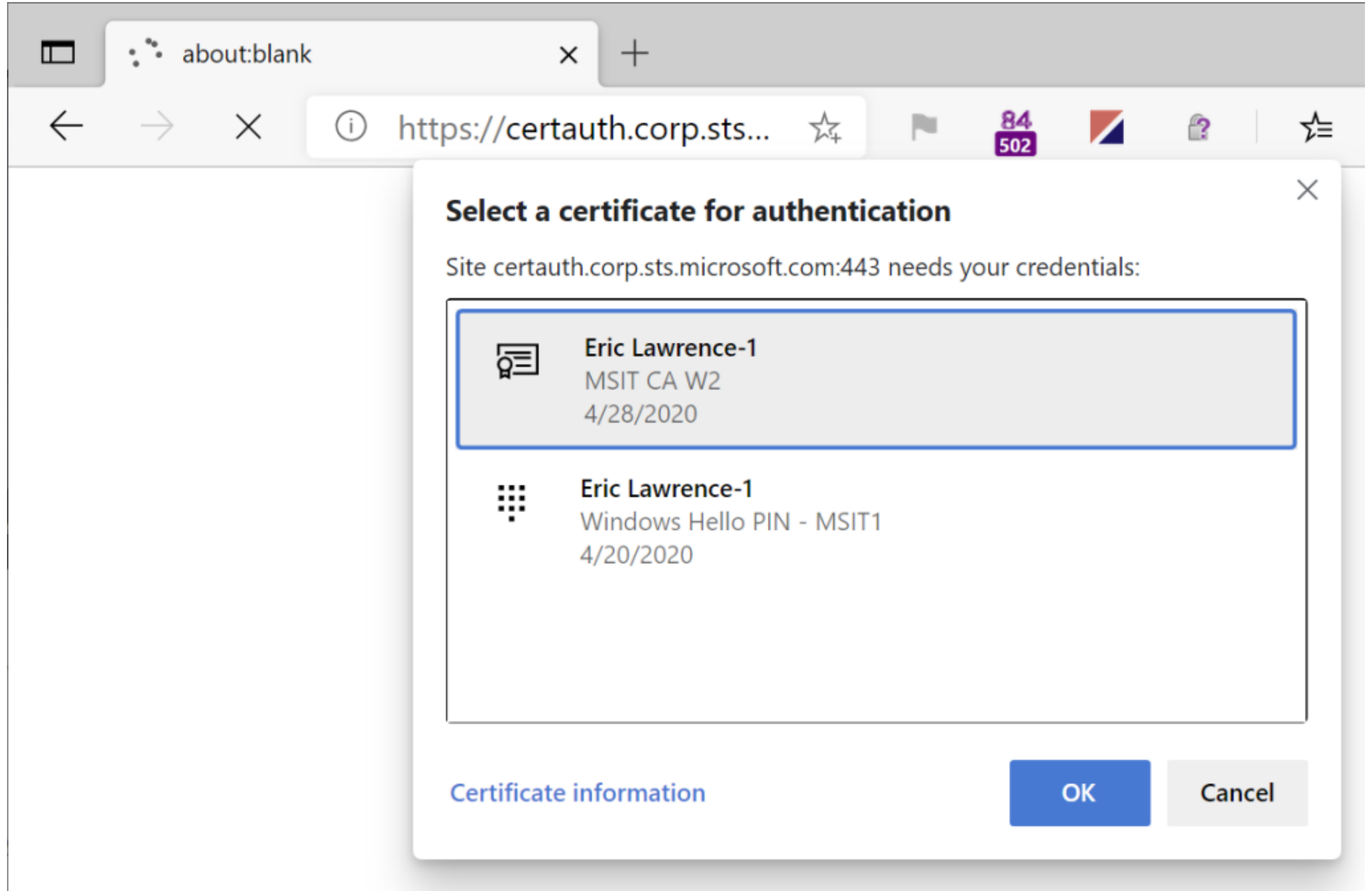
- The user has a private key with an associated public key
  - Possession of the private key is used for authentication
    - Typically by signing a challenge with the private key
  - The service requiring authentication verifies the signature with the public key
    - A valid signature means that the other party possesses the private key

- Key-based authentication does not rely on shared secrets
  - Only the legitimate party is supposed to have this private key
  - Best practices require secure storage of the private key (E.g., in an OS-backed keychain)

- Implementing key-based authentication requires client support
  - The use of mutual TLS is a common pattern, but not in browser-based applications
  - Out-of-band mobile applications can be used to manage authentication keys
  - The new Web Authentication API supports various key-based mechanisms

Key-based authentication with mTLS





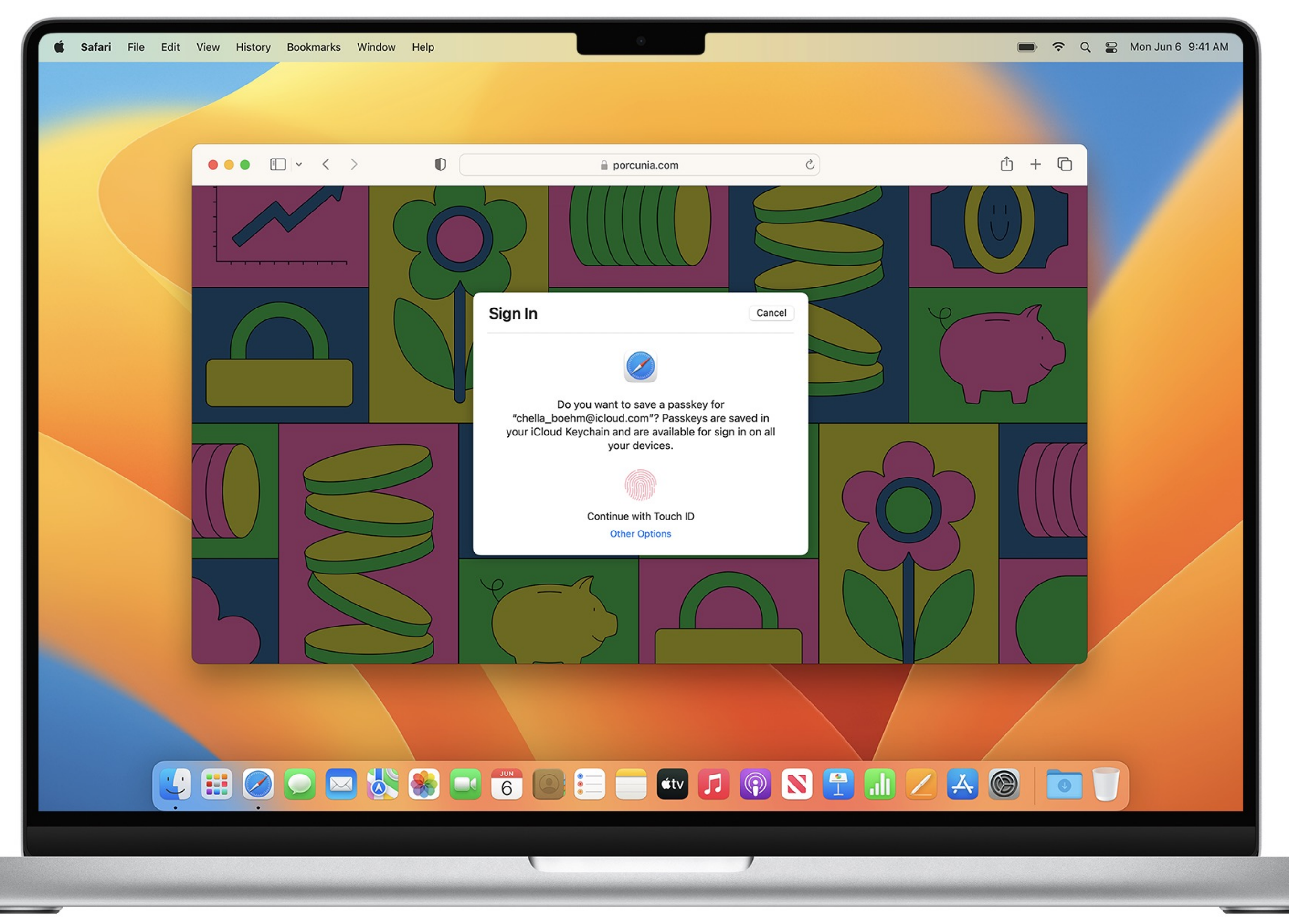
https://textslashplain.com/2020/05/04/client-certificate-authentication/

Challenges with mTLS

- mTLS is great for building machine-to-machine authentication
  - Certificate generation and deployment can be automated
  - mTLS is supported by virtually all TLS libraries, so small implementation effort
  - Supported by default in service meshes like Istio
- mTLS is horrible for building user authentication
  - Users do not understand certificates
  - Installing certificates in a browser is quite challenging
  - Authentication failures result in protocol errors, making error handling difficult
  - Unless you have a certificate on a dedicated device, the certificate is not portable
- Key-based authentication is great, but mTLS is not the way to implement it

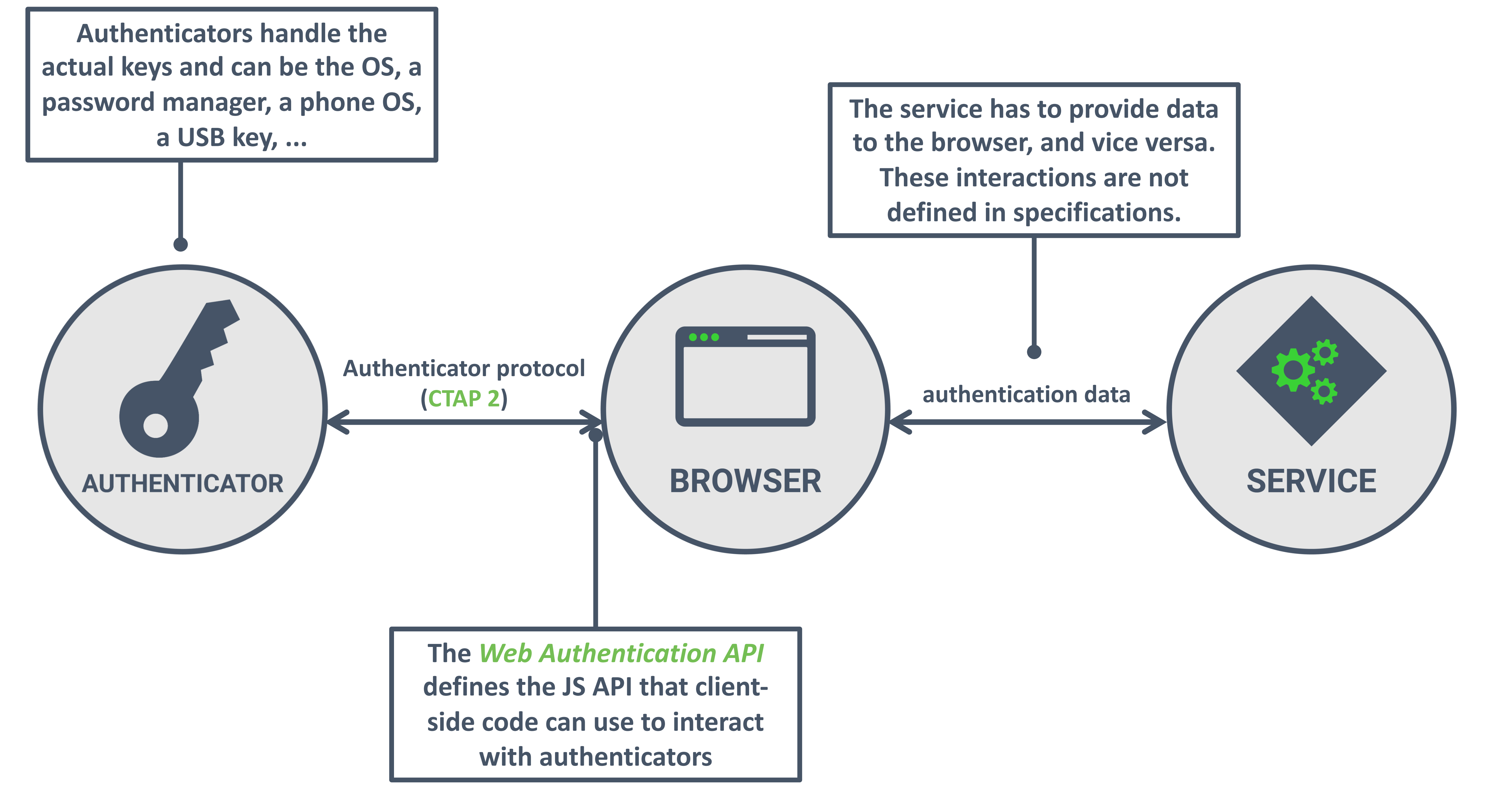
Passkeys

Passkeys can now be synced using external providers, and you can create groups to share passwords and passkeys. In managed environments, passkeys support Managed Apple IDs, including syncing via iCloud Keychain, and access controls let people easily restrict how passkeys are shared and synced.

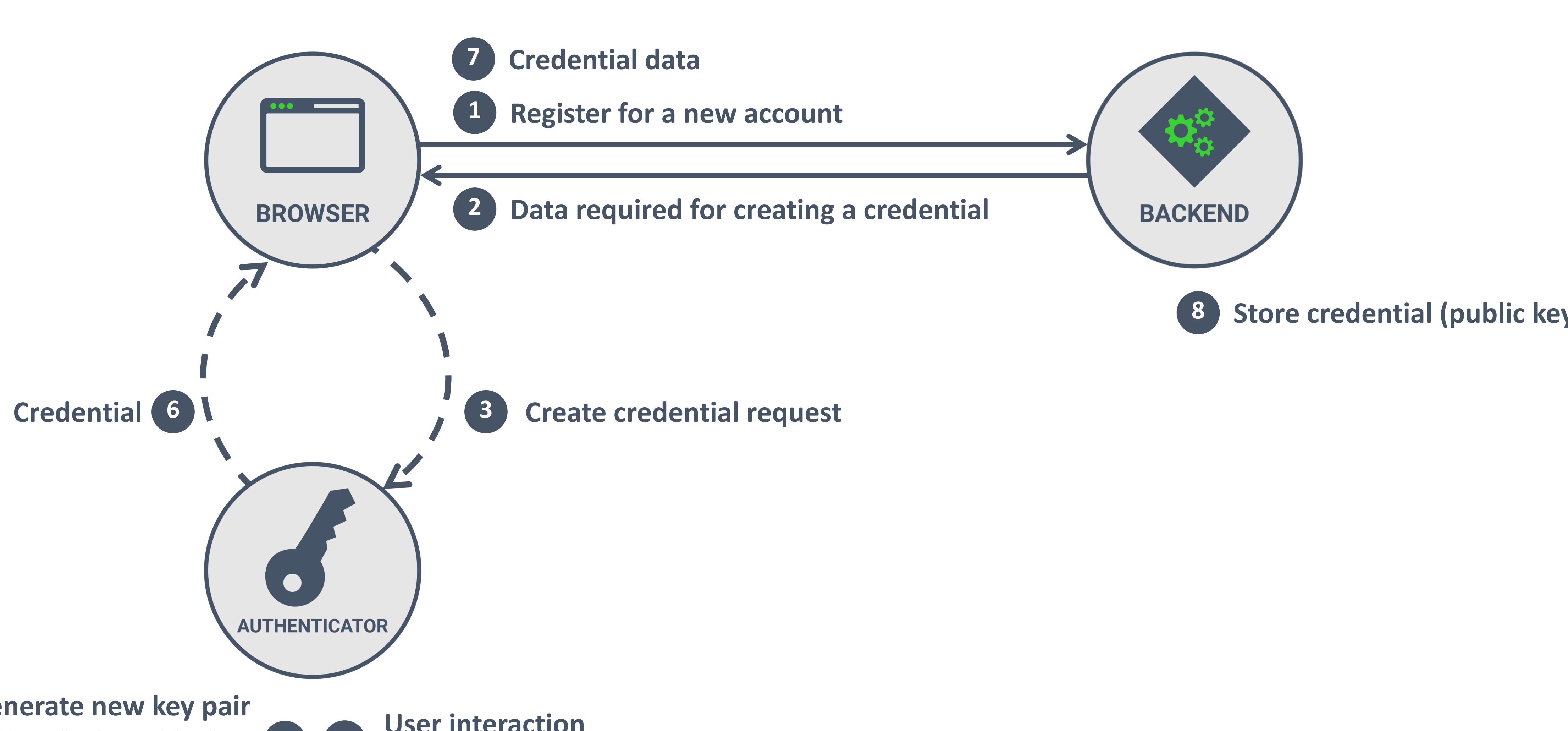


https://developer.apple.com/passkeys/

Passkeys from the user's perspective

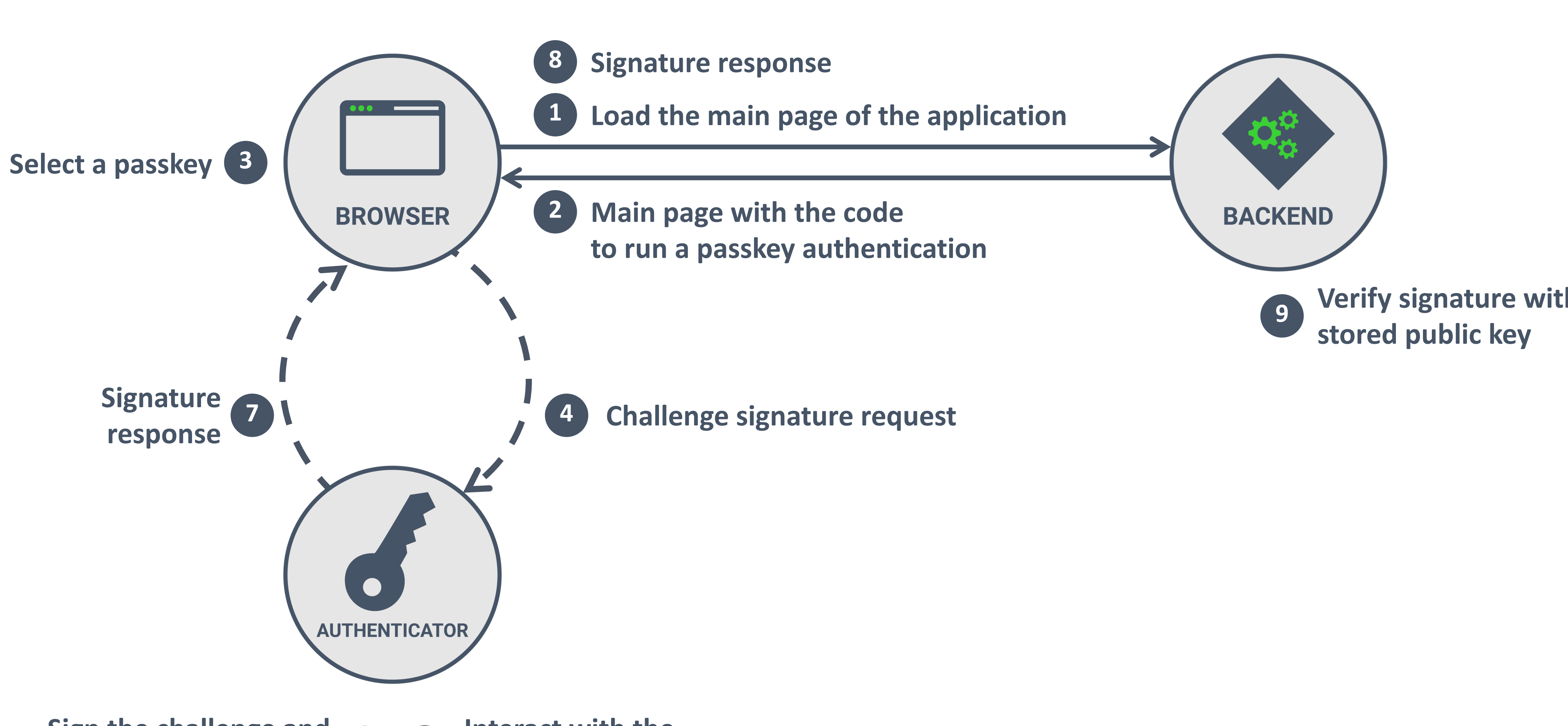

Authenticators handle the actual keys and can be the OS, a password manager, a phone OS, a USB key, ...
The service has to provide data to the browser, and vice versa. These interactions are not defined in specifications.
Authenticator protocol
(CTAP 2)
authentication data
AUTHENTICATOR
BROWSER
SERVICE
The Web Authentication API defines the JS API that client-side code can use to interact with authenticators

SETTING UP A PASSKEY



Creating a passkey on passkeys.io

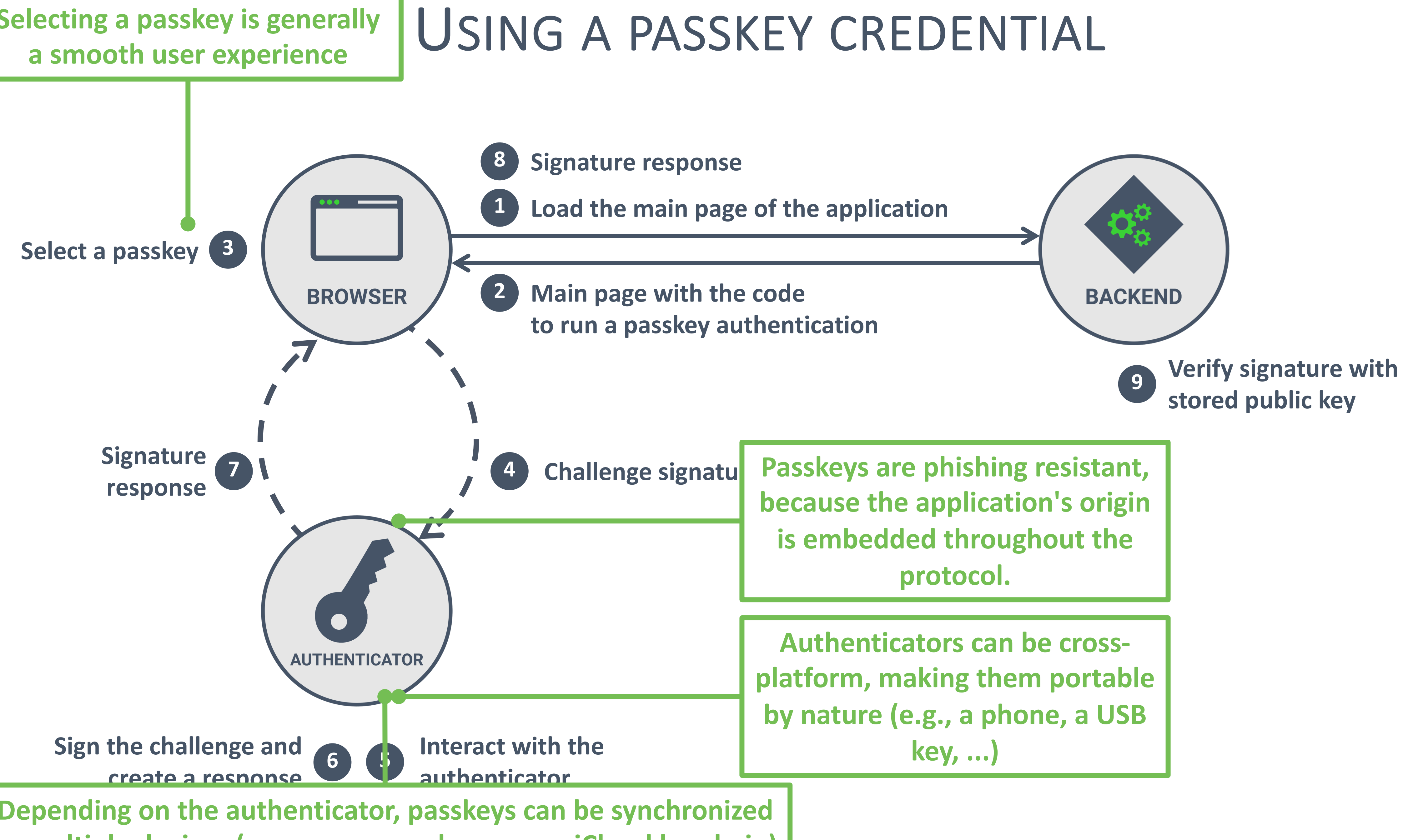
Using a passkey credential



Using a passkey on passkeys.io

Does a passkey meet our initial requirements?

USING A PASSKEY CREDENTIAL



The full passkey experience

Passkeys - OTHER

Usage % of all users ?

Global 89.33%

Passkeys, also known as Multi-device FIDO Credentials, provide users with an alternative to passwords that is much easier to use and far more secure.

Current aligned | Usage relative | Date relative | Filtered | All

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile
								4-19.0		
4-107	12-107	3.1-16.0	2-121	10-96			3.2-15.8	20		
108-124	108-124	16.1-17.4	122-125	97-108	6-10		16.0-17.4	21-23		12-12.1
125	125	17.5	126	109	11	124	17.5	24	all	80
126-128		17.6-TP	127-129				17.6			

caniuse.com

Passkeys from a developer's perspective

Using the Web Authentication API to create a new passkey

```
1  const credential = await navigator.credentials.create(
2    createCredentialOptions
3  );
```

The *credentials API* supports various use cases. The *options* provided here determine the type of credential.

Browsers offer a *credentials API* to support two crucial operations: *create* and *get*

For the use of *passkeys*, the type of credential is *publicKey*

```
1   const createCredentialOptions = {
2     publicKey: {
3       rp: {
4         id: 'restograde.com',
5         name: 'Restograde',
6       },
7       user: {
8         id: Uint8Array.from(serverData.userId, (c) => c.charCodeAt(0)),
9         name: serverData.userEmail,
10        displayName: serverData.userDisplayName,
11      },
12      pubKeyCredParams: [
13        { type: 'public-key', alg: -7 },
14        { type: 'public-key', alg: -257 },
15      ],
16      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
17      authenticatorSelection: {
18        residentKey: 'required',
19        requireResidentKey: true
20      }
21  }}
```

```
1   const createCredentialOptions = {
2     publicKey: {
3       rp: {
4         id: 'restograde.com',
5         name: 'Restograde',
6       },
7       user: {
8         id: Uin                    .userId, (c) => c.charCodeAt(0)),
9         name: s
10        displayName: serverData.userDisplayName,
11
12
13
14                                   },
15
16                                ata.challenge, (c) => c.charCodeAt(0)),
17      authenticatorSelection: {
18        residentKey: 'required',
19        requireResidentKey: true
20      }
21  }}
```

Information about the *relying party* (i.e., the service asking for authentication)

A human-readable name for the relying party

The ID of the relying party. This value must correspond to the match the relying party's origin or domain.

For example, for *https://app.restograde.com*, the ID can be *app.restograde.com* or *restograde.com*

```
6        },
7        user: {
8          id: Uint8Array.from(serverData.userId, (c) => c.charCodeAt(0)),
9          name: serverData.userEmail,
10         displayName: serverData.userDisplayName,
11       },
12                      ams: [
13                      blic-key', alg: -7 },
14         { type: 'public-key', alg: -257 },
15       ],
16       challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
17       authenticatorSelection: {
18         residentKey: 'required',
19         requireResidentKey: true
20       }
21   }}
```

A unique ID to identify the user with their passkey credential. This ID should not contain PII (i.e., no email) and is preferably indepdent from the user's primary ID in the application.

Also referred to as the user handle.

A *username* and a *display name* intended for use in UX

The user handle is embedded in the credential and is provided to the server during authentication

Using the Web Authentication API to create a new passkey

```
1   const createCredentialOptions = {
2     publicKey: {
3       rp: {
4         id: 'restograde.com',
5         name: 'Restograde',
6       },
7       user: {
8         id: Uint8Array.from(serverData.userId, (c) => c.charCodeAt(0)),
9         name: serverData.userEmail,
10                    ta.userDisplayName,
13          { type: 'public-key', alg: -7 },
14          { type: 'public-key', alg: -257 },
15        ],
16      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
17      authenticatorSelection: {
18        residentKey: 'required',
19        requireResidentKey: true
20      }
21  }}
```

A challenge that should be signed by the credential. The challenge is provided by the server.

Using the Web Authentication API to create a new passkey

```
1   const createCredentialOptions = {
2     publicKey: {
3       rp: {
4         id: 'restograde.com',
5         name: 'Restograde',
6       },
7       user: {
8         id: Uint8Array.from(serverData.userId, (c) => c.charCodeAt(0)),
9         name: serverData.userEmail,
10        displayName:
11      },
12      pubKeyCredPara
13        { type: 'pub
14        { type: 'pub
15      ],
16      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
17      authenticatorSelection: {
18        residentKey: 'required',
19        requireResidentKey: true
20      }
21  }}
```

A *resident key* is the indicator of a discoverable credential, which allows the user to select this credential for authentication, even when using it the first time in a specific browser.

This is an important requirement for passkeys.

THE *AUTHENTICATORSELECTION* OBJECT

- The ***authenticatorAttachment*** indicates where the key can be stored
  - ***Platform***: the key will be stored locally (e.g., keychain with password or touch ID)
  - ***Cross-platform***: they must be portable across different machines (e.g., USB)

- The ***userVerification*** property indicates if the user identity should be verified
  - Fingerprint, password, or biometrics counts as user verification
  - Touching a yubikey (without fingerprint scan) is not considered user verification
    - This is known as *user presence,* but not *user verification*
  - Browsers can allow verifiable authenticators, even when user verification is discouraged

- The ***residentKey*** properties indicate “discoverable credentials”, aka ***passkeys***
  - These are credentials that can be used without the server explicitly asking for them

```
1   const createCredentialOptions = {
2     publicKey: {
3       rp: {
4         id: 'restograde.com',
8   erverData.userId, (c) => c.charCodeAt(0)),
9         name: serverData.userEmail,
10        displayName: serverData.userDisplayName,
11      }
12      pubKeyCredParams: [
13        { type: 'public-key', alg: -7 },
14        { type: 'public-key', alg: -257 },
15      ],
16      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
17      authenticatorSelection: {
18        residentKey: 'required',
19        requireResidentKey: true
20      }
21  }}
```

Indicates which algorithms can be used to generate signatures. This allows a relying party to indicate what types of signatures the backend service can verify.

Self-explanatory, right?

Which "pubKeyCredParams" to use? #1757

dagnelies opened this issue on Jun 28, 2022 · 32 comments · Fixed by #1843

dagnelies commented on Jun 28, 2022

Hi,

I noticed that during `credentials.create(...)`, if the list does not contain what the authenticator can provide, the authenticator will not be included in the list of authenticators to choose from. For example, if you don't include `"alg":-257`, Windows Hello won't work.

Now, as a relying party this all sounds a bit like unknown mysteries.

- the specification says "pick your algorithms" from a huge list!
- no idea which algos the authenticators support
- no idea which algos you really have to support as an RP

In practice, using this list restricts your choice to a subset of authenticators available... if you manage to find out which algo is needed. Also, most RPs are not deeply knowledgeable about which crypto algorithms is better suited or not.

So ...are all common authenticators covered by RS256 and ES256? Or should you as an RP add some more to cover most authenticators? Which ones?

As it turns out, there's no clear understanding of which authenticator supports what ...

https://github.com/w3c/webauthn/issues/1757

Which "pubKeyCredParams" to use? #1757

dagnelies opened this issue on Jun 28, 2022 · 32 comments · Fixed by #1843

dagnelies commented on Jun 28, 2022 ···

MasterKale commented on Jun 28, 2022 Contributor ···

And for anyone interested: based on some extensive testing I did a few months back of in-the-wild authenticators, most everything I tested **only** supported `-7` ("ES256"), with the exception of Windows Hello which was only `-257` ("RS256"). Only the YubiKey 5C, 5Ci, and Bio **also** supported Ed25519 (`-8`, "EdDSA").

suited or not.

So ...are all common authenticators covered by RS256 and ES256? Or should you as an RP add some more to cover most authenticators? Which ones?

https://github.com/w3c/webauthn/issues/1757

Using the Web Authentication API to create a new passkey

```
1   const createCredentialOptions = {
2     publicKey: {
3       rp: {
4         id: 'restograde.com',
8   erverData.userId, (c) => c.charCodeAt(0)),
9         name: serverData.userEmail,
10        displayName: serverData.userDisplayName,
11      }
12      pubKeyCredParams: [
13        { type: 'public-key', alg: -7 },
14        { type: 'public-key', alg: -257 },
15      ],
16      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
17      authenticatorSelection: {
18        residentKey: 'required',
19        requireResidentKey: true
20      }
21  }}
```

Indicates which algorithms can be used to generate signatures. This allows a relying party to indicate what types of signatures the backend service can verify.

-7 (ES256) **and** ***-257 (RS256)*** **cover the main authenticators, but it's a good idea to also support** ***-8 (EdDSA)*** **if your backend can handle it**

The result of creating a credential

- Creating a credential yields a promise that resolves to a ***PublicKeyCredential***
  - This object holds a bunch of data about the newly created credential (e.g., an ID)
  - The important property is the ***response***, which is an ***AuthenticatorAttestationResponse***
- In the response, there's an encoded JSON value called ***clientDataJSON***
  - This value is the JSON data that was passed to the authenticator at creation time
    - Values include the origin of the context that created the credential
  - The client can use the JSON data to do a sanity check on the generated credential
- The client sends the following data to the backend for registration
  - The public key of the credential
  - The ***authenticatorData***, a binary format providing the flags and ID of the authenticator

https://developer.mozilla.org/en-US/docs/Web/API/PublicKeyCredential
https://w3c.github.io/webauthn/#authenticator-data

Digging into passkey creation on learnpasskeys.io

Using the Web Authentication API to use an existing passkey

```
1  const credential = await navigator.credentials.get(
2    getCredentialOptions
3  );
```

The *credentials API* supports various use cases. The *options* provided here determine the type of credential.

Browsers offer a new *credentials API* to support two crucial operations: *create* and *get*

For *passkeys*, the type of credential is *publicKey*

The ID of the relying party, used to identify which existing credentials can be used. Must be an exact match for the value used during registration.

Using the Web Authentication API to use an existing passkey

```
1  const getCredentialOptions = {
2    publicKey: {
3      rpId: 'restograde.com',
4      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
5    }
6  }
```

This **rpId** offers phishing protection. A phishing website would have to use ***restograde.com***, but the browser will refuse to use that on ***rest0grade.com***

The *challenge* provided by the server to sign with the private key.

It is critical to avoid replay attacks that this value is not empty, and generated from a secure random source.

THE RESULT OF USING A CREDENTIAL

- Using a credential yields a promise that resolves to a ***PublicKeyCredential***
  - This object holds the data of using the credential (e.g., its ID, the generated signature)
  - The important property is the ***response***, which is an ***AuthenticatorAssertionResponse***

- The client sends all the relevant ArrayBuffers to the backend for verification
  - The ID of the credential (***rawId***)
  - JSON data from creating the credential (***response.clientDataJSON***)
  - Authenticator data, e.g., flags indicating user verification (***response.authenticatorData***)
  - The signature (***response.signature***)

- The client does not handle this data, it just forwards it to the backend

https://www.w3.org/TR/webauthn-2/#authenticatorassertionresponse

Digging into passkey usage on learnpasskeys.io

Attestations represent the authenticator acknowledging the creation of a new keypair

Assertions represent the authenticator acknowledging that a specific key was used

Conditional mediation is critical for a seamless user experience

Passwordless sign-in on forms with WebAuthn passkey autofill

WebAuthn conditional UI leverages browser's form autofill functionality to let users sign in with a passkey seamlessly in the traditional password based flow.

Published on Wednesday, November 30, 2022

Eiji Kitamura
Developer Advocate for identity, security, privacy and payment on the web.
Twitter GitHub Glitch Mastodon

Chrome 108 supports passkeys, including autofill suggestions. This allows sites to build easy sign-in experiences that are more secure.

Table of contents

What is a passkey?
Conditional UI
- How it works
- How to use conditional UI

https://developer.chrome.com/blog/webauthn-conditional-ui/

Modern browsers support a conditional UI for various authentication mechanisms

```
1  <input type="text" name="username" autocomplete="username webauthn" ...>
```

This is optional, as the application can always explicitly start the selection of a passkey.

However, when there is no passkey available, this will result in a disruption of the flow with an error.

This input field accepts either a username, or triggers the selection of a passkey when available.

It is designed to offer a seamless user experience regardless of the authentication mechanism the user wants to use.

Modern browsers support a conditional UI for various authentication mechanisms

```
1  <input type="text" name="username" autocomplete="username webauthn" ...>
```

Trigger the passkey UI on the input field if passkeys are discovered

```
1  const credential = await navigator.credentials.get({
2    publicKey: {
3      rpId: 'app.example.com',
4      challenge: Uint8Array.from(serverData.challenge, (c) => c.charCodeAt(0)),
5    },
6    mediation: 'conditional'
7  });
```

This API call starts a conditional passkey authentication dialog, which only suggests passkey authentication as an *autocomplete* option if a passkey is discovered.

Conditional mediation in action

PublicKeyCredential API: `isConditionalMediationAvailable()` static method

Usage % of all users ?

Global 77.2%

Current aligned | Usage relative | Date relative | Filtered All

Chrome	Edge *	Safari	Firefox	Opera	IE ⚠ *	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile
4 - 107	12 - 107	3.1 - 15.6	2 - 118	10 - 93				4 - 20		
108 - 124	108 - 124	16.0 - 17.4	119 - 125	94 - 108	6 - 10		3.2 - 17.4	21 - 23		12 - 12.1
125	125	17.5	126	109	11	124	17.5	24	all	80
126 - 128		17.6 - TP	127 - 129				17.6			

caniuse.com

What about the server?

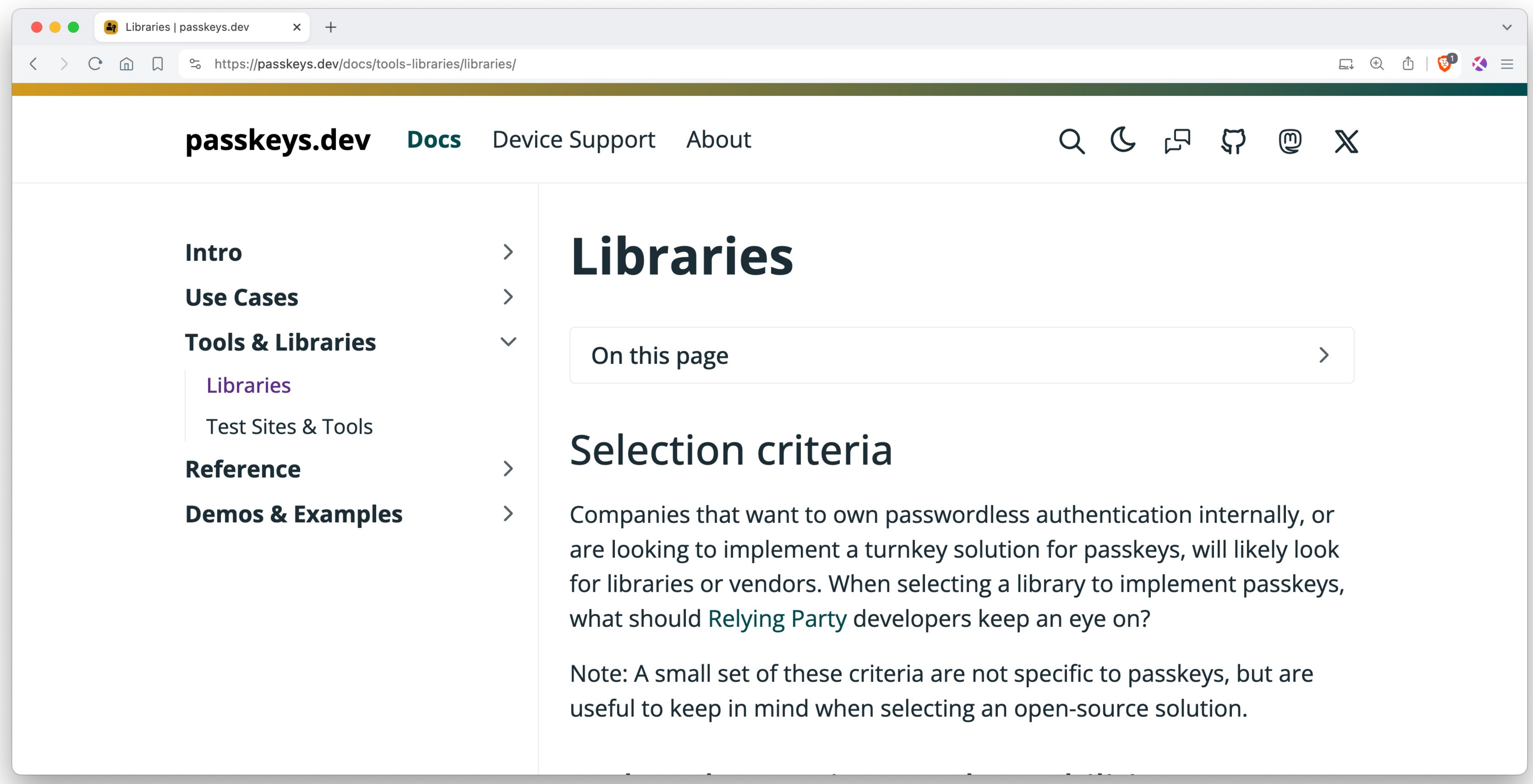
Responsibilities of the backend

- The specification does not define how to send credential data to the backend
  - Most server-side libraries and frameworks define the format/data they expect
  - Some of the data is binary, so they must be base64-urlencoded for safe transport
- During registration, the backend is responsible for
  - Providing the browser with a user handle and challenge for credential creation
  - Verifying the incoming credential data
    - Make sure the origin matches the expected origin
    - Verify the feature flags of the authenticator (e.g., user verification, user presence, …)
    - Verify additional metadata if included
  - Storing the authenticator data and public key to use during authentication

Responsibilities of the backend

- The specification does not define how to send credential data to the backend
  - Most server-side libraries and frameworks define the format/data they expect
  - Some of the data is binary, so they must be base64-urlencoded for safe transport

- During authentication, the backend is responsible for
  - Providing the browser with the challenge to sign
  - Verifying the incoming data
    - Lookup the authenticator using the provided authenticator ID
    - Ensure that origin included in the response data matches the expected origin
  - Verifying the signature using the public key of the authenticator



https://passkeys.dev/docs/tools-libraries/libraries/

java-webauthn-server

Server-side Web Authentication library for Java. Provides implementations of the Relying Party operations required for a server to support Web Authentication, including passkey authentication.

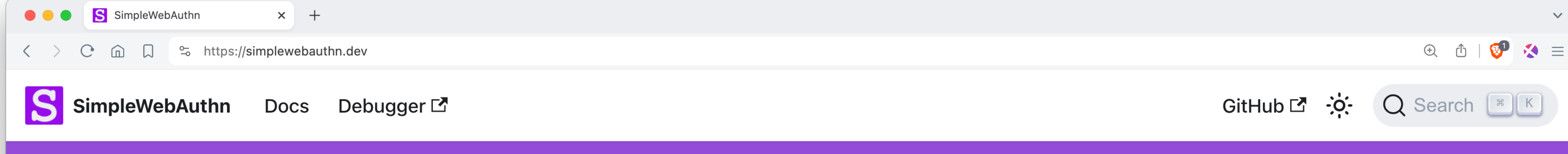
> WARNING
>
> ***Psychic signatures in Java***
>
> In April 2022, CVE-2022-21449 was disclosed in Oracle's OpenJDK (and other JVMs derived from it) which can impact applications using java-webauthn-server. The impact is that for the most common type of WebAuthn credential, invalid signatures are accepted as valid, allowing authentication bypass for users with such a credential. Please read Oracle's advisory and make sure you are not using one of the impacted OpenJDK versions. If you are, we urge you to upgrade your Java deployment to a version that is safe.

Table of contents

Features

- Generates request objects suitable as parameters to `navigator.credentials.create()` and `.get()`
- Performs all necessary validation logic on the response from the client
- No mutable state or side effects - everything (except builders) is thread safe
- Optionally integrates with an "attestation trust source" to verify authenticator attestations
- Reproducible builds: release signatures match fresh builds from source. See [reproducible-builds] below.

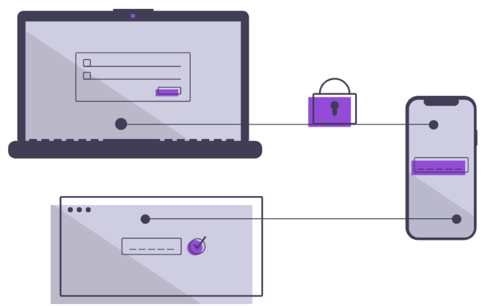
https://developers.yubico.com/java-webauthn-server/



SimpleWebAuthn

A collection of TypeScript-first libraries for simpler WebAuthn integration. Supports modern browsers and Node.

Get Started



Simple to Use

It's in the title! SimpleWebAuthn makes it as easy as possible to add WebAuthn-powered passkeys to your websites so that you can move on to the fun stuff.

First-Class TypeScript Support

Everything is authored in 100% TypeScript! And a dedicated package for type declarations makes it even simpler to use SimpleWebAuthn in your own TypeScript projects.

FIDO® Conformant

SimpleWebAuthn passes FIDO® Conformance Server Tests with flying colors! You can rest easy knowing that when you need to take things to the next level, SimpleWebAuthn will grow

https://simplewebauthn.dev/

Authenticating with passkeys

- Passkeys are key-based credentials with embedded user information
  - This information is typically a unique identifier pointing to a specific user
  - The authenticator stores this data along with the private key

- During authentication, the authentication data includes the unique identifier
  - The service can verify the signature using the authenticator's public key
  - If the signature matches, the service uses the unique identifier to authenticate the user

- Passkey authentication relies on two core building blocks in the browser
  - The ***Web Authentication API*** handles the interactions between JS and authenticators
  - ***Conditional mediation*** enables the UX to allow seamless passkey selection

Integrating passkeys into your applications

PASSKEY ADOPTION

- Conditional mediation allows passkeys to co-exist with passwords
  - Give users the option to create passkeys to gradually move away from passwords
  - Consider allowing users to disable password-based authentication for enhanced security
- Keep in mind that users will want to use multiple passkeys
  - Ideally, a user authenticates once with a ***cross-platform*** passkey
  - The application then prompts the user to register a platform-specific passkey
    - This passkey will be added to the user's account, along with the cross-platform passkey
  - The platform-specific passkey improves the user experience
- Adding multiple passkeys to an account is straightforward
  - Multiple credentials with their IDs/public keys map to a specific user account
  - During authentication, the application matches the used credential to a specific user

Passkeys are linked to an origin

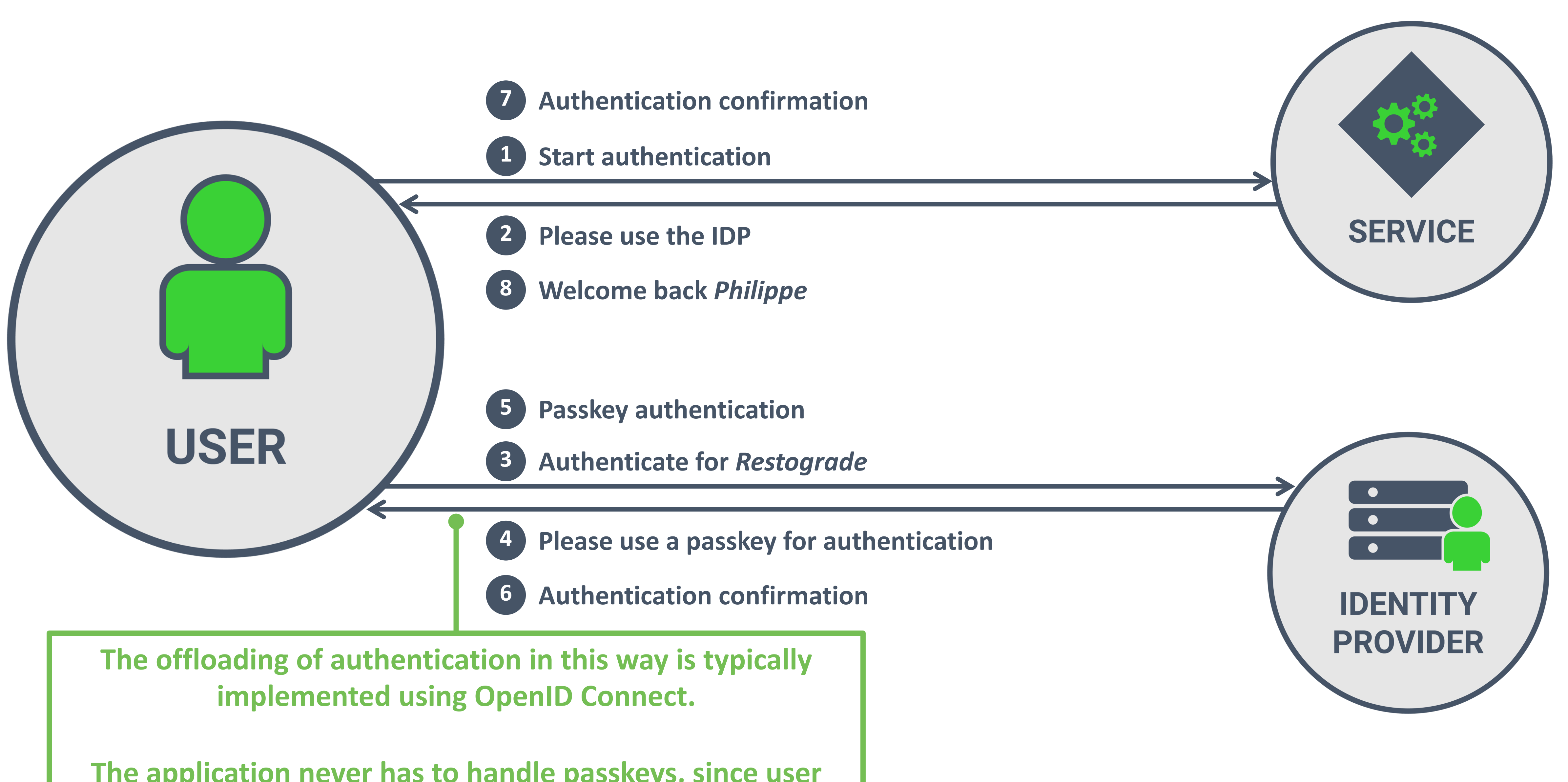
Relaxing the passkey origin

- Typically, passkeys are associated with the origin of the page creating them
  - The origin is embedded in the protocol and cannot be changed for a particular passkey
  - This behavior offers strong phishing protection
  - This behavior can become a drawback when changing the origin of an application
    - E.g., moving the application from ***www.restograde.com*** to ***app.restograde.com***

- Browsers allow creating passkeys where the relying party is the parent domain
  - E.g., ***app.restograde.com*** can create a passkey for ***restograde.com***
  - This behavior is still resistant against phishing attacks, but offers some flexibility

- Relaxing the passkey's relying party is not intended for sharing passkeys
  - Subdomain-based attacks illustrate the danger of sharing resources with subdomains
  - Only use this mechanism for flexibility in (re)deploying applications within the parent domain

When possible, consider offloading passkey usage to an (internal) identity provider


7 Authentication confirmation
1 Start authentication
2 Please use the IDP
8 Welcome back *Philippe*
SERVICE
USER
5 Passkey authentication
3 Authenticate for *Restograde*
4 Please use a passkey for authentication
6 Authentication confirmation
IDENTITY PROVIDER
The offloading of authentication in this way is typically implemented using OpenID Connect.
The application never has to handle passkeys, since user authentication is the responsibility of the identity provider.

Welcome

Log in to Auth0 Demo to continue to Example App.

Email address

philippe@pragmaticwebsecurity.com

Can't login to

philippe@pragmaticwebsecurity.com
Passkey from your Chrome profile

Use a different passkey

Manage passwords and passkeys...

Don't have an a

OR

Continue with a passkey

When an Identity Provider supports passkeys, enabling it is typically straightforward, as the IDP handles all the heavy lifting.

KEY TAKEAWAYS

1. Passkeys offer key-based authentication with a great UX
2. Passkeys are widely adopted by browsers, password managers, etc.
3. Consider offering users passkey support to eradicate passwords

Thank you!

Need training or security guidance?
Reach out to discuss how I can help

https://pragmaticwebsecurity.com