A Tour of WebAuthn

Adam Langley

2024-12-23

Apart from the use of the Whisper medium model to transcribe dictation, and OpenAI's o1-preview model to proofread, no AI tools were used in the creation of this work.

The cover image is View of a Villa, by Lancelot-Théodore Turpin de Crissé. Part of the collection of the National Gallery of Art.

The book is typeset with Typst in Libertinus Serif and Roboto Mono.

Contents

CHAPTER 1
Introduction

Passwords are rubbish.

If you're reading this book then hopefully you're already on board with this idea, but let's recap anyway.

The typical practice with passwords is to remember a few different ones and re-use them widely. (Password managers support generating random passwords, but people mostly don't.) Sites must store hashes of these passwords to recognize them, but most passwords have too little entropy to resist brute-forcing when the hashes leak. (The website haveibeenpwned.com now has records of about 13.5 billion accounts that have been found in account database leaks from nearly 800 websites.)

When a password database leaks, not only can any successfully cracked passwords be used immediately to sign in to that site but, because of password re-use, those users' accounts on other sites may also be compromised.

Next, because users typically remember their passwords, they can be tricked into entering them on lookalike websites. These "phishing" attacks are common, effective, and can have global implications when used to interfere in elections[1].

Lastly, passwords can leak from many other parts of the software stack. Facebook inadvertently logged[2] hundreds of millions of passwords over many years, and Javascript-injection attacks[3] can exfiltrate anything entered on a site, including passwords.

This book is about using public key signature schemes to try and build a better system of authentication. These schemes have names like ECDSA, RSA, and ML-DSA. They vary in how large their outputs are, how fast they operate, and whether they're resistant to (still theoretical) large quantum computers. In this book we'll consider them only in the abstract—the construction of public key signature schemes is a deep and fascinating topic, but we will cover none of it here.

In the abstract, a public key signature scheme provides three operations:

1. A `generate` operation takes some random bits and returns two byte strings that we will call a *public key* and a *private key*.
2. A `sign` operation that takes a private key and an arbitrary byte string (called the "message"), and produces another byte string that we will call a "signature".

3. A `verify` operation that takes a public key, a message, and a claimed signature, and returns whether that signature was generated by the `sign` operation with the corresponding private key.

The names of these operations have obviously been chosen to suggest analogs in the real world, but that is a double-edged sword. These are not handwritten marks on a piece of paper; they are only defined by the operations above. For example, there can be multiple valid "signature" values by a public key for a given message. But while these subtleties are important in some contexts, like cryptocurrencies, they aren't for us.

To be useful, a public key signature scheme must have properties like the following:

1. It's not possible to compute the private key from the public key.
2. It's not possible to compute a signature without the private key.

Those properties are vaguely defined. (Indeed, the second implies the first.) Readers are welcome to research the formal definition of security ("EUF-CMA") to better understand the topic, but a rough understanding is all you need in this context.

Let's sketch out a toy example of an authentication scheme using public key signatures.

In this sketch, rather than registering with a website by creating a username and password, people will create a username and then their computer will run the `generate` operation, record the private key, and submit the public key and username to the site.

In order to sign in, people will enter their username. Then their computer will run the `sign` operation with their private key on the message "let me in". It'll send the username and signature to the website. Lastly, the website uses the public key that it recorded when the user signed up, and runs the `verify` operation on that, the message "let me in", and the submitted signature. If the signature is valid then the user is signed in.

We have immediately solved the problem of database leaks because now websites only need to record public keys, not password hashes. A public key can't be used to generate a signature, only validate one. So, unlike passwords, when they leak, they can't be used to sign in to that site (or any other).

But we haven't solved the phishing problem because people might submit a signature value to the wrong site. Also, if an attacker learns a single signature value, they can use it to sign in as the user. Lastly, we breezed over where private keys are stored and how many there are.

Don't worry, we'll solve all of those issues with more realistic designs. Firstly, let's deal with phishing.

Phishing involves an attacker proxying a user's sign-in information. The victim mistakenly signs in to a fake website, and then that information can be reused with the real website. The toy scheme above was vulnerable to phishing because the message, "let me in", was the same for all sites and so a signature was valid everywhere.

So our first design tweak is to change the signed message to name the intended destination. Let's start using JSON for the message too.

Now, when a user signs in, their computer runs the `sign` operation with the private key. But rather than using the message "let me in", we'll make it `{"origin": "https://example.com"}`. The site needs to run the `verify` operation on the resulting signature, and the `verify` operation requires the message as an input, so we'll also have the user's computer send the message along with the signature and username.

Now consider what happens when the user clicks a malicious link and tries to sign in to `exampl3.com` (an evil phishing website). The user's computer will sign the message `{"origin": "https://exampl3.com"}`. When the phishing site proxies the sign-in details to the real website, the signature will be valid, but the real website can notice that the user was attempting to sign in to a different website and reject it. (Computers, unlike humans, can reliably notice single-character differences in URLs.)

The phishing site can't change the message because the real site will reject the signature when `verify` checks it, and they can't update the signature because they don't have the private key.

So phishing has been solved with a small tweak, but we're still left with the problem that if a signature value for the real site leaks then that can be used to sign in as a user. While, unlike password hashes, signature values don't need to be stored, we've seen that similar values can be exfiltrated via Javascript-injection and inadvertent logging.

To solve phishing, we made the message specific to a given site. To solve this problem, we'll make the message specific to an authentication attempt.

So our next tweak to the design will be that, when a user tries to sign in, the site will send a large random *challenge* to the user's computer, to be included in the signed message. Each time a user attempts to sign in, the random challenge will be different (with extremely high probability).

So a signed message will now look like this: `{"origin": "https://example.com", "challenge": "8065afbaa4faee78123ad2061bc78df3"}`.

Now if a signature gets logged or exfiltrated by malicious JavaScript, it quickly becomes useless. A signature is only valid for a specific message, but the challenge (and thus the message) will be different in the future.

We still need to consider how many public keys a person has and where they are stored.

A simple answer would be that each person has a single public key and uses it across all sites and apps, but the obvious problem with that is that it is a unique tracking value for that person, and people don't want to be linked across all their sites and apps.

For now, we will say that each website or app gets its own set of keys. Things are more complicated than this in practice, but we will cover these complexities in future chapters.

Unlike a password, the value of a private key never needs to be sent anywhere to be used. So for maximum security, we'll start by generating and keeping private keys in dedicated hardware, usually connected via USB. This hardware can be designed to be resistant even to a degree of physical attack. Later in the book we'll see how this can be relaxed so that these authentication schemes can be usable in a consumer context.

The next chapter will dive into the concrete and cover the nitty-gritty of the first implementation of this design, but we should keep in mind the limits of any authentication system:

In a digital context, people are always acting through their computer. While we talk about authenticating a user, the thing that directly gains authority as a result of authenticating is that user's computer. So if that computer is controlled by an attacker, the authentication system is moot. Tackling the authentication problem does not solve all security issues, but many security issues *are* authentication problems, so better authentication systems are necessary part of fixing the world.

WebAuthn, the subject of this book, is such a system.

---

1 www.nytimes.com

2 krebsonsecurity.com

3 arstechnica.com

CHAPTER 2
Universal Second Factor

The first embodiment of the broad design that we sketched out in the previous chapter was the Universal Second Factor (U2F) system from the FIDO Alliance. The FIDO Alliance is a consortium of companies, all of whom care about the problems of online authentication, and they developed a pair of standards to try and solve phishing by adding a public-key authentication system as a second factor to sign-ins. (I.e. in addition to a password.)

The first of these standards, called CTAP1[1], defines a protocol between computers and dedicated devices called *security keys* that perform the `generate` and `sign` operations. The second defines a Javascript API so that websites can make use of them.

At this point, the U2F Javascript API is thoroughly obsolete and is not worth covering even for historical reasons. But millions of U2F security keys were produced over the years and, while security keys now use the more modern CTAP2 protocol, CTAP1 is very simple, is still supported, and is worth understanding because it contains the core elements of everything that follows. So we'll discuss CTAP1 in some detail in this chapter.

CTAP1

CTAP1 only includes two commands: one that implements the `generate` operation and another that implements `sign`. To cause a CTAP1 security key to run `generate`, a computer sends a command consisting of the following bytes, concatenated together: (There's quite a long list of fields here, but don't worry. Each will be explained.)

1. A byte with value 0.
2. A byte with value 1—the command code for `generate`.
3. A flags byte with two flags set: 0x1 for "user presence required" and 0x2 for "user presence consumed".
4. Another zero byte.
5. The length of the following data as a 24-bit, big-endian value. This is always 64.
6. The SHA-256 hash of the "client data".
7. The SHA-256 hash of the "relying party ID".
8. Two zero bytes which indicate that the maximum response length is supported.

(This message format comes from a smart card format called an application protocol data unit or APDU. This chapter will include the needed details without further reference to APDUs.)

The response consists of the following, concatenated together:

1. A byte with value 5.
2. A public key (X9.62 encoded P-256, see chapter 13. U2F only supports ECDSA with P-256).
3. An 8-bit credential ID length.
4. The credential ID.
5. A certificate (X.509 DER encoded).
6. A signature.

The number of values going back and forth should be somewhat surprising in light of the previous chapter. There, the `generate` operation was defined as requiring random bits as the sole input, and as producing a private key (which we expect the security key to store) plus a public key (which it should report to the computer). In U2F, the random bits come from within the security key so there are two unexpected inputs ("client data" and "relying party ID") and three unexpected outputs (the credential ID, a certificate, and a signature).

An ID is the most obvious addition: we need some way to refer to the generated keys. Credential IDs are generated by security keys for that purpose and can be assumed to be globally unique because they must be at least 16 bytes long and contain at least 100 bits of entropy. Given that the ID length in the response is only eight bits, a U2F credential ID is, at most, 255 bytes long.

We have already mentioned that we want to avoid reusing public keys too widely for privacy reasons. Thus, to stop sites from sharing public keys and tracking users with them, U2F allows an arbitrary value called the *relying party ID* (RP ID) to be associated with each public key. (A *relying party* is any entity that does authentication, i.e. a website in the examples that we've been using.) The same relying party ID must be sent whenever the `sign` operation is invoked, therefore public keys cannot be used outside the context in which they were intended.

Typically, an RP ID is a domain name, like `example.com`. So even if `anothersite.com` knows the ID for an `example.com` credential, it can't use it because the browser will specify that the RP ID is `anothersite.com` at signing time, and so the security key will reject the request. (See chapter 5 for more about RP IDs.)

The security key checks the RP ID, and not the browser, because security keys are assumed to move with the user, between different computers. So a browser on a specific computer or phone might not have been involved in creating the credential that is now being used.

User presence

The term "user presence" appears in the description of the request flags. This refers

to the idea that an operation should only be performed when a human physically touches the security key.

Most security keys will have a capacitive sensor: a metal band or disc that can recognize a touch by the change in capacitance it causes. Requiring user presence (a physical touch) for each operation stops any malware on the computer from making requests to the security key in the background.

Since security keys typically don't have screens, when users touch them they don't know what operation is being requested and have to trust that the computer is requesting a legitimate operation. So the benefit of requiring user presence is modest. But the principle of requiring a user interaction for every operation has become a core part of the WebAuthn ecosystem that developed from U2F, and thus of everything covered in this book.

U2F splits the concept into "user presence required" and "user presence consumed". The former requires that the security key have been recently touched and the latter resets that flag. But this split didn't survive into later versions of the standards which instead specify when the flag is reset rather than deferring to the request.

Attestation

To explain the remaining values, we'll have to briefly cover the topic of attestation.

Suppose you're a company worried about the security of your employees signing in to your corporate systems. In that case, you may want the private keys that they're using to do so to be stored in a specific type of security key. (Probably the type of security key you issued them for this purpose.) You may even want to ensure that the security key they use is the specific one that was inventory-tracked from the factory and assigned to them.

Attestation is designed to solve these problems.

The concept is that the security key has a private key installed in the factory. This private key is not used for signing in, but rather to prove that future `generate` operations were performed in a known model of security key, or in a specific security key.

The "client data" hash, and the certificate and signature outputs, are all part of this process. They are covered in detail in chapter 7, but most uses of security keys don't deal with attestation, and so we'll ignore these fields for now.

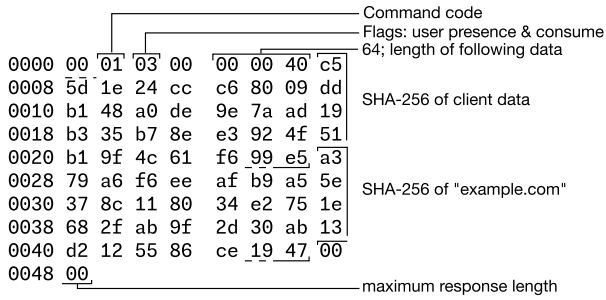
Invoking the `generate` operation

The easiest way to trigger a generate operation is in a web browser. We'll use the following snippet of JavaScript to ask the browser to generate a public and private key, and we'll have the browser do that with a U2F security key so that we can inspect the request and the response. (This will be our first example of using the

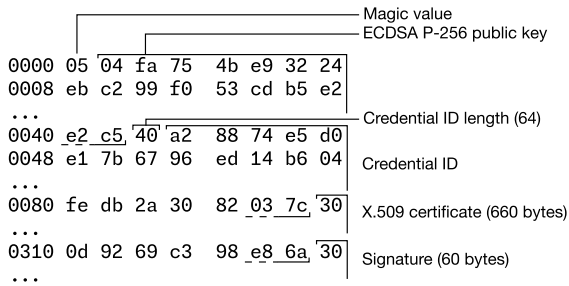
WebAuthn Javascript API, which will be covered extensively in chapter 4. We could also have used the APIs on Android or iOS, which are covered in chapter 11.)

```
navigator.credentials.create({
    publicKey: {
        // Unused in this example.
        challenge: new Uint8Array([0]).buffer,
        // Boilerplate required values.
        pubKeyCredParams: [{
            type: "public-key",
            alg: -7,
        }],
        // The relying party ID.
        rp: {name: window.location.host},
        // Required values that are inapplicable in U2F.
        user: {
            id: new Uint8Array(1),
            name: "user",
            displayName: ""
        },
    }
}).then(console.log, console.log);
```

Here is the message sent by the browser, to the U2F security key, broken down into the same parts as listed above.



And here's the response, similarly broken down:



(Note that the UP flag was set in the request so you know that I had to touch the security key before it would generate that response.)

Now that we have performed a `generate` operation, the next step is to perform a `sign` operation with the private key that has just been generated.

Invoking the `sign` operation

Again, we'll foreshadow future chapters by using a snippet of the WebAuthn Javascript API to ask a browser to perform a `sign` operation with a U2F security key.

```
navigator.credentials.get({
    publicKey: {
        allowCredentials: [{
          type: 'public-key',
          transports: ['usb'],
          id: hexStringToArrayBuffer(
              "a28874e5d0e17b6796ed14b60447278a" +
              "c544e6b8dec18e54ccb178afb797e21e" +
              "e54a0cf264741b6cf4f8f89f41d12fff" +
              "18aafeff82ee318225c5339fd3fedb2a"),
        }],
        challenge: new Uint8Array([0,1,2,3,4,5,6,7,
                                   8,9,10,11,12,13,14,15]).buffer,
    },
}).then(console.log, console.log);
```

Note the appearance of the credential ID from the previous response in this request, so that the security key knows which private key to sign with.

The request format is an APDU again and so follows a similar structure. This time we'll dive directly into the concrete request that the browser sent and explain the structure as we go:

- `00` (all CTAP1 commands start with a zero byte)
- `02` (the command code for `sign`)
- `03` (a flags byte with two flags set: 0x1 for "user presence required" and 0x2 for "user presence consumed".)
- `00` (an unused flags byte)
- `000081` (the 24-bit length of the following data: 129 bytes)
- `a438...c659` (the SHA-256 hash of the "client data", described below.)
- `a379...1947` (the SHA-256 hash of the relying party ID, `example.com`. Note that this is the same value that was sent when generating the credential, otherwise the request would be rejected.)
- `40` (the length of the credential ID; 64 bytes.)
- `a288...db2a` (the credential ID. The same value that was returned from the `generate` operation, and given in the Javascript request to the browser.)
- `0000` (indicating that the maximum response length is supported).

The response to a `sign` request is the simplest message so far:

- `01` (a flags byte with one flag set: 0x01 for "user presence"—covered later)
- `00000004` (a 32-bit signature counter, also covered later)
- `3045...0e2d` (the signature itself).

Recall that, in the previous chapter, we considered a series of shapes for the signed message. Firstly, we tried using a constant value ("let me in"). Next we realized that the message should include the site's origin to prevent phishing, so it became a JSON object: `{"origin": "https://example.com"}`. Then we realized that it should contain a random challenge value from the site to make it unique to an authentication attempt. This JSON is called the *client data* because it comes from the client (i.e. computer), rather than from the security key. It's hashed and combined with the *authenticator data* (which comes from the security key) to form the message that is ultimately signed.

The client data generated by a modern browser has evolved slightly and, for the Javascript request above, looks like this: (With whitespace and line breaks added for clarity.)

```
{
  "type": "webauthn.get",
  "challenge": "AAECAwQFBgcICQoLDA0ODw",
  "origin": "https://example.com",
  "crossOrigin": false
}
```

We've seen the `origin` and `challenge` fields before, although note that the challenge is encoded with base64url, rather than the more common base64 encoding. The `type` field is just good security hygiene: you never want to allow ambiguity, so it's good to be explicit about the intended meaning of all messages. The `crossOrigin` field communicates whether the authentication was done within an iframe inside another site.

(Browsers may add more fields in the future, so always parse such JSON rather than assuming that the challenge can be inserted into a template.)

The authenticator data is the concatenation of the following:

1. The SHA-256 of the relying party ID.
2. The flags byte from the `sign` response.
3. The signature counter from the `sign` response.

Then the signed message, i.e. the final value given to the `sign` operation, is the authenticator data followed by the SHA-256 hash of the client data.

We now have all the pieces to implement the public-key authentication scheme sketched out in the previous chapter:

When an account is first registered, we have a Javascript snippet that will cause the browser to send a request to a U2F security key to generate a public and private key. The server can store the credential ID and public key from the response and associate them with the new account. Then, when that account tries to sign in, the server generates a random challenge and makes a Javascript call with that challenge and the credential ID. The browser will send a request to a U2F security key to sign a message containing that challenge (and other values) and return the resulting client data, authenticator data, and signature.

The server can now:

- Run the `verify` operation with its stored public key and the response from the browser.
- Check that the client data's `type`, `origin`, `crossOrigin`, and `challenge` fields are as expected.
- Check that the authenticator data is well-structured and contains the expected relying party ID, and that the flags byte has "user presence" set.

This is still a simplistic example and there are more steps in a full implementation. (For example, unless the user provides a password as well then it would be possible to take a security key from someone's desk and sign in as them!) But you can now see the core ideas in action, all the way from a server, through a browser, to concrete messages sent over USB to a security key.

Statelessness

Nearly all U2F security keys are designed in a manner that avoids the need to actually store the private keys, thus they can generate an unlimited number of credentials with only a constant amount of onboard storage. The exact details are private to the security key but go something like this:

Either at the factory or on first use, the security key will generate and store a random symmetric key. We'll call this the "root secret". Then, when the security key is asked to generate a public and private key, it chooses a random seed value and encrypts it with the root secret. Next, it authenticates the encrypted seed, along with the hash of the relying party ID and any other pertinent values, using an algorithm like HMAC.

Together, the encrypted seed and the HMAC form the credential ID. Then the security key generates a public key and private key from the seed, discards the private key, and returns the credential ID and public key to the computer.

When asked to do a sign operation, the security key splits the credential ID into an encrypted seed and an HMAC value. It checks whether the HMAC is correct for the relying party ID hash and the encrypted seed. If not, then either this credential ID came from a different security key or the relying party ID is incorrect. In either case, it returns an error.

But, if the HMAC is correct, it decrypts the seed using its root secret and can then derive the same private key as it did before. Then it performs the `sign` operation with that private key and returns the resulting signature.

At no point did the security key need to store anything other than the root secret, and there is only a single root secret for all credentials, so the storage requirements are constant. The state necessary for a credential is kept in the credential ID, and that is stored on the server.

There are a couple of consequences of this style of design:

First, a credential can never be used without knowing the credential ID. In chapter 3 we'll discuss "discoverable" credentials that change this assumption but, in U2F, a credential ID is always needed.

Second, there's no way to delete an individual credential from a security key because the credential isn't stored on the security key, it's "stored" in the ID, on the server. Instead, U2F security keys will typically support a reset command that generates a fresh root secret. Once the root secret has been changed, all previous credential IDs are invalidated and will appear to be IDs generated by a different security key.

In U2F this reset functionality, if it exists for a particular model of security key, is a vendor-specific command. It gets standardized in CTAP2, see page 56.

Signature counters

The rest of this chapter contains details that can be skipped on a first reading.

The last unexplained field is the "signature counter" that is returned from `sign` operations. In the example above, it was four, but that doesn't mean that I discarded three responses while trying to capture that example.

Signature counters are optional for the security key to implement, although the majority of them do. If there's no counter then the value in the response is always zero. But once a security key has produced a non-zero value then it has to ensure that the counter, for all future signatures from that credential, strictly increases.

The motivation for having a counter is that it might allow websites to detect when a security key has been cloned. Cloning a security key is supposed to be very difficult but, if you assume someone managed to do it (probably destroying it in the process),

then one could create a working replica which could be slipped back into the possession of the legitimate user, leaving them unaware that anything has happened. At this point the attacker, who presumably also created a replica for themselves, can create signatures as easily as the legitimate user.

If all that has happened, then the signature counter might uncover it. Unless the attacker can know exactly when the legitimate user has used their security key, and thus incremented the counter, then eventually either they or the real user will create a signature where the counter didn't increase.

If the website noticed this, it could sound the alarm. At a minimum, the security key in question should be replaced. Ideally, the account and security key would be investigated carefully for signs of compromise.

This is a rather far-fetched scenario. Generally, when signature counters are checked by a site, any error is treated as a transient authentication failure. But that defeats the point: the user or attacker will simply try again after the signature counter has naturally incremented, and then it'll work. The user will simply come to think of their security key as a bit worn out and will learn that sometimes it requires a couple of attempts to work.

On the other hand, many security keys only have a single, global signature counter, and this allows different websites to correlate the use of the same security key between them. That is, the current counter value of your security key is somewhat identifying and can be combined with information about how often it increases. (Better security keys will implement more granular signature counters.)

Signature counters are also incompatible with syncing private keys between computers (see chapter 3) and thus are not implemented in an increasing fraction of cases.

So signature counters *might* be useful in the most extreme cases if carefully implemented and coupled with a robust incident response process, but they can otherwise be ignored.

Platform behavior

The computers (or phones) that security keys are used with are called *platforms* and U2F is a bit more complex for platforms than suggested in the sketch above. There can be multiple security keys plugged into a laptop, and a WebAuthn request can list many credential IDs. The browser has to find the right credential ID for the right security key.

Also, U2F was designed so that security keys could be implemented in a Java-based framework that did not allow requests to block. But requests cannot complete until user-presence is satisfied, i.e. until someone touches the sensor on the security key.

These factors mean that browsers have to poll U2F security keys. Two protocol features facilitate this:

First, whenever an operation cannot be completed because the security key is waiting for a touch, the request will immediately result in a special error code ("test of user presence required"). The security key will usually blink an indicator for a short while after this.

Second, the flags byte in `sign` requests can be set to 0x07 ("check only"). This causes the request to always fail but with one of two distinct error codes. If the credential ID and relying party ID hash are valid for the security key, then it returns the same "test of user presence required" error as before. (Whether the security key has been touched recently or not!) Otherwise it returns a different error ("bad key handle").

Browsers combine these two features with the following algorithm for handling U2F security keys:

For `generate` requests, the request is sent to all security keys. The security keys will fail the request because they haven't been touched, so the requests are repeated every few hundred milliseconds until the user touches one of the security keys to select it.

For `sign` requests, each credential ID is sent to each security key in "check only" mode until either a valid credential ID is found, or until all credential IDs have been tried with a given security key. If a valid credential ID is found for a security key then a stream of sign requests are sent, waiting for a touch. If a security key doesn't recognize any IDs then either a message is displayed on the screen, or else a stream of `generate` requests are sent, waiting for a touch. These generate requests are sent just to cause the security key to blink its indicator, and to see whether the security key is touched—any resulting public key is discarded. If the security key is touched, an error is displayed to the user because that security key isn't going to work.

The transport layer

A fully working spec also needs to define how security keys are discovered on the USB bus and how the U2F messages are encoded for transmission.

Security keys advertise themselves as USB Human Interface Devices. HID devices can advertise a number of *usage pages* that describe broadly what kind of device they are. For example, usage page one covers keyboards, mice, joysticks, and other common input devices. Security keys advertise the usage page 0xf1d0 ("FIDO", get it?) to identify themselves.

HID devices communicate using short *report* messages and FIDO defines[2] how longer messages are fragmented and reassembled so that they can be sent as a series of reports.

To deal with the issue of multiple applications trying to talk to the same security key at the same time, and potentially interleaving their streams of report messages, every report starts with a channel identifier. Applications get an identifier by sending a 64-bit nonce and watching for a channel allocation report from the device that echoes that nonce.

Thus, in theory, a security key can handle multiple concurrent communication streams with different applications on the computer. In practice, since security keys are embedded devices, they often only support a single active channel and so an application requesting a new channel will disable the previously active one. However, this mechanism still allows communications to be cleanly broken off rather than having fragments from different applications interleaved, producing unpredictable results.

Security keys often also support communication over NFC, so that a security key can simply be held near the top of a mobile phone to be used. While the underlying technology and framing differs completely between USB and NFC, it is ultimately just another way of exchanging messages and so the underlying transport can be switched out without affecting any of the higher-level protocol.

A Bluetooth Low Energy (BLE) transport was also defined, but this requires the security key to have its own power source. The primary motivation was to use security keys with iPhones before they had support for NFC security keys, but now that the issue has been resolved, BLE security keys are very rarely seen. (This transport is unrelated to the one used between computers and phones, which is covered in chapter 10.)

---

1 fidoalliance.org

2 fidoalliance.org

CHAPTER 3
FIDO2 and passkeys

The standards described in the previous chapter are sufficient to enable the classic 2nd-factor pattern of entering a username, then a password, and then tapping a security key. Many companies have deployed this sort of system to their employees to very great effect. Authenticating employees with just a password in this day and age is bordering on negligence and, unlike code-based second factors (whether delivered over SMS or from an app on a phone), security keys aren't phishable.

But a password is still necessary. Otherwise, the security key alone would be sufficient to authenticate, and mislaid security keys would be a significant concern.

U2F also doesn't solve the problem of usernames. Remembering usernames is a bother that password managers help solve today. So, if public key authentication is going to find a broader audience, it needs to solve this too. Thus we want to be able to ask a security key what accounts have been registered for a particular site, and for it to store the corresponding usernames.

The U2F protocol outlined in the previous chapter cannot support this. It assumes that the credential ID is already known before a `sign` operation can be performed, and thus the account must be known before the security key can be used. Indeed, the vast majority of U2F security keys do not store *any* per-credential data, as described on page 15, and so it's impossible for them to work with any other pattern of interaction.

So security keys, and the protocol for communicating with them, had to evolve. Within the FIDO Alliance this was broadly done under the umbrella term "FIDO2", which covered both a new security key protocol (CTAP2) and a new Javascript API (WebAuthn), which we've already seen examples of.

The first major new concept in FIDO2 is *discoverable credentials*.

The model of discoverable credentials

Credentials on a U2F security key, which we'll now call *non-discoverable credentials*, conceptually consist of three values: the credential ID, the private key, and the relying party ID. Since U2F security keys generally don't store any per-credential state, they can be thought of as an infinite bucket of such credentials.

A security key that stores *discoverable* credentials can be thought of as a database table. The first two columns of this table form the primary key, and those columns are the relying party ID and the *user ID*. A user ID is a new concept for discoverable

credentials: an identifier for an account chosen by the site. It is not (and should not be) the username, for reasons that will be explained on page 81. Instead, it is better to think of it as the "user UUID".

Since these two columns are a primary key, that implies that the security key stores at most one credential for any given pair of relying party ID and user ID. So if you create a second discoverable credential that has the same RP ID and user ID as an existing one, then the existing one is overwritten.

The third and fourth columns of this conceptual table are familiar: the credential ID and the private key. Discoverable credentials still have credential IDs and can be specified by them just like non-discoverable credentials. But the crux is that it's also possible to ask for *some* credential for a given RP ID, without specifying any credential IDs. This is the key distinction between discoverable and non-discoverable credentials, and the one that lets them act like a username as well as an authentication factor.

There are many more columns in the table. Of course there's a username (in fact, two), and an increasing number of extensions require more fields to be added to the credential row. These values will be covered in chapters 4 and 9.

Since discoverable credentials require per-credential storage on a security key, many security keys support creating both discoverable and non-discoverable credentials and have a limit on the number of discoverable credentials.

User verification

Discoverable credentials solve the problem of usernames, but we would still need a password to avoid risks from mislaid security keys. In fact, discoverable credentials make that problem even more acute: you wouldn't even have to guess whose security key it was, it would tell you!

U2F introduced the concept of user presence—that some human was physically present. Coupled with discoverable credentials comes the stronger notion of *user verification*—that the *correct* human is physically present. The mechanism for establishing this varies. Some security keys have fingerprint readers, a few security keys have a physical PIN pad on the device, but most security keys use a PIN that is entered on the computer.

The implied contract with the security key is that it will maintain a user verification chain: if user verification is performed with a PIN, then that PIN may be changed, but the old PIN has to be presented to do so. If the PIN is lost and needs to be reset, then all the credentials must be deleted to do so. Different security keys may differ in the details, but each should maintain such a chain of verification.

While the term "PIN" is used, these PINs can be alphanumeric, so you could also call them passwords. But since FIDO2 was supposed to be replacing passwords, calling them passwords would have been awkward, and so they were called PINs. But the difference is more than just a name: these PINs are never sent over the network, can never be disclosed in a database leak, and, if they're being used with a security key, then the security key can enforce hardware-based guess limits.

Platform authenticators

It also became apparent with U2F that many people wanted to keep their security keys plugged into their computer all the time. In response, a form factor of security key that mostly fits inside a USB port, and leaves only a small lump on the outside of the computer, became quite popular.

But this was driven by more than the realization that a USB port was a useful place to keep a security key where it wouldn't get lost.

Malware that steals cookies is a serious concern because, no matter how strong the initial authentication is, if malware can steal the results of that authentication, you still have a security problem. So enterprises often ask their users to re-authenticate with their security keys, perhaps daily. Since a hardware-bound private key cannot be stolen by malware, this establishes that the active session is still legitimate.

But many computers already have a Trusted Platform Module[1] or similar device within them that can store a hardware-bound private key and sign with it. So why not use those as opposed to having a security key permanently inserted? Since the computers or phones that security keys are used with are called *platforms*, these are called *platform authenticators*.

Traditional platform authenticators cannot help you with signing in on a different computer, but they can provide proof that an active session hasn't been stolen by malware and, when coupled with user verification, they can provide evidence that the correct human is still behind the keyboard. They are also *far* more common than security keys since no extra hardware has to be purchased.

(From this point, we'll start to use the more generic term *authenticator*, rather than security key, unless specifically speaking about the latter type of device.)

WebAuthn

In the next chapter we'll cover the Javascript API that was developed as part of the broader FIDO2 effort in order to expose these concepts: WebAuthn. The APIs for Android, iOS, macOS, and Windows are strongly shaped by it and, collectively, they form the *WebAuthn family* of APIs. So even if you never touch Javascript, you need to understand the core of WebAuthn. While the syntax of these different APIs varies, they'll generally use the same terminology, and they all produce compatible outputs.

Passkeys

Since (traditional) platform authenticators can't be used to sign in on other devices, FIDO2 originally assumed that all users had security keys. Otherwise losing or reinstalling a laptop would leave a user without any usable credentials.

Security keys are great and should be deployed with gusto in all enterprise and government environments that require strong authentication. But it's unlikely that regular people are going to adopt them and, even if they did, security keys work much better when there's a helpdesk that can be a backstop after the dog has chewed on one.

If the benefits of FIDO2 are going to be enjoyed more widely, credentials are going to have to be more usable, and that means syncing them. In this book, thus far, there has been a firm link between a credential and a single physical device that created and holds the private key. That device might be a security key, or it might be a platform authenticator, but either way you can point to where the private key is stored and it never moves. Syncing changes that, in good ways (people can recover from losing a device), and bad (a sync account could be compromised).

Wider deployment also needs to avoid overly technical terms; "WebAuthn credential" doesn't sound very friendly. Thus the term *passkey* was invented by Apple, but they nicely let everyone use it.

A passkey is a synced, discoverable WebAuthn credential. Or, when that's awkward, sometimes it's just a discoverable credential. But this book will stick to the former definition. In general, passkeys are an attempt to take WebAuthn outside the enterprise.

The passkey ecosystem consists of providers, provider APIs, passkey APIs, and the hybrid transport. We'll cover the hybrid transport in chapter 10. It's the fallback for when syncing doesn't bring a passkey to where you need it: you can pull out your phone, scan a QR code, and send a signature to a computer.

Passkey providers are services that store and sync passkeys (which are, remember, just another name for discoverable WebAuthn credentials). Within the Apple ecosystem, iCloud Keychain is the most common provider. On Android, you'll commonly find Google Password Manager or Samsung Pass. Most other password managers are also passkey providers, such as 1Password, Dashlane, and Bitwarden.

Provider APIs are the way that passkey providers register with an operating system. We won't be covering these APIs in detail in this book, but the Apple platforms and Android both provide APIs where providers can register. Then providers can offer passkeys to applications that want them.

The flip side of the provider APIs are the *passkey APIs*. These are covered in detail in chapter 11. They provide applications with access to registered providers so that they can request that passkeys be created, and can request signatures from them. These APIs are all based on the original FIDO2 API: WebAuthn.

When credentials are synced, the sync account (rather than a device) is considered to be the authenticator. So when the model of discoverable credentials says that only one credential with a given pair of RP ID and user ID exists within an authenticator, that applies to the sync account, not to each individual device that is syncing.

For example, if a credential is created in iCloud Keychain on a Mac for a given site and user ID, and then another credential is created on an iPhone, signed into the same iCloud account, with matching RP ID and user ID values, then the latter will overwrite the former because the sync account is the same, and the invariant applies to the whole account.

---

1 en.wikipedia.org

CHAPTER 4
WebAuthn

WebAuthn (short for "Web Authentication") is a web API that lets a site interact with U2F-era security keys (see chapter 2), but also take advantage of all the new concepts in FIDO2, as outlined in the previous chapter.

The data formats and structures in WebAuthn are also strongly reflected in the platform APIs that exist for apps running on the Apple family of platforms, on Android, and on Windows. So even non-Web developers will need to understand the concepts from this chapter.

Patterns in WebAuthn

WebAuthn is integrated with the W3C Credential Manager API[1], which lives in the `navigator.credentials` namespace. You don't need to know this API to use WebAuthn and it's not covered here. But you'll notice that the APIs are structured to support other credential types too. That's not an accident: both passwords and federated credentials can be used via Credential Management.

Creating a credential

Creating a credential looks like this:

```
const promise = navigator.credentials.create({
    publicKey: creationOptions,
});
```

The resulting promise either resolves with a representation of the newly created credential, or else an error. But first we'll look in detail at the options that control what is created and where it'll be stored.

The `creationOptions` above is a `PublicKeyCredentialCreationOptions`[2] dictionary and it contains the following members:

```
dictionary PublicKeyCredentialCreationOptions {
    // Selecting the public key signature scheme to use.
    required sequence<PublicKeyCredentialParameters> pubKeyCredParams;

    // Controlling the location and type of the new credential.
    AuthenticatorSelectionCriteria          authenticatorSelection;
    sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];

    // Metadata stored with the credential.
    required PublicKeyCredentialRpEntity    rp;
    required PublicKeyCredentialUserEntity  user;
```

```
    // Guiding the UI.
    unsigned long                           timeout;
    sequence<DOMString>                     hints = [];

    // Extension features, covered in a later chapter.
    AuthenticationExtensionsClientInputs    extensions;

    // Fields related to attestation; covered in a later chapter
    required BufferSource                   challenge;
    DOMString                               attestation = "none";
    sequence<DOMString>                     attestationFormats = [];
};
```

Selecting a signature scheme

Signature schemes are identified by integers assigned by the IETF's COSE group. A site specifies the set of schemes that it accepts (in preference order) by listing them in `pubKeyCredParams`. Since the values assigned by the COSE group leave some parameters unspecified, WebAuthn additionally defines those parameters for the common schemes.

ECDSA with P-256 and SHA-256 is the dominant scheme, by far, in the WebAuthn ecosystem. (If you recall from chapter 2, it's the only scheme that was supported in U2F.) It has been given identifier −7. (Yes, a negative number.)

Many PCs have a TPM[3] that only supports the older RSA PKCS#1 v1.5 standard and so, to be compatible with them, sites are advised to support that scheme too. RSA PKCS#1 v1.5 uses the identifier −257 and also specifies SHA-256 as the hash function.

So most sites should set `pubKeyCredParams` to the following:

```
[{type: "public-key", alg: -7}, {type: "public-key", alg: -257}]
```

Some other, less common schemes are −8 for Ed25519 and −35 for ECDSA with P-384 and SHA-384. Ed25519 is superior to ECDSA P-256 but the difference is often not sufficient to outweigh how common ECDSA is. ECDSA with P-384 can be used in order to meet certain regulatory requirements[4] but, given that the primary risks are implementation issues, it is likely less secure in practice. In the future ML-DSA (previously called Dilithium) is likely to replace ECDSA due to concerns about quantum attacks, but that transition is still some years in the future.

Controlling the location and type of the new credential

The `authenticatorSelection` field is, itself, another dictionary with the following fields:

```
dictionary AuthenticatorSelectionCriteria {
    // Controlling whether the credential is discoverable.
    DOMString    residentKey;
```

```
    boolean      requireResidentKey = false;

    // Controlling whether user verification is performed.
    DOMString    userVerification = "preferred";

    // Controlling which types of authenticators can store the
    // credential.
    DOMString    authenticatorAttachment;
};
```

The first two fields control whether the credential must be discoverable. But why are there two of them, and why are they named "resident", not "discoverable"?

Historically, discoverable credentials were first called *resident credentials* so this term will pop up in several places in the protocol. It is inaccurate, however. Resident credentials are those where the authenticator keeps state about the credential. Discoverable credentials are always resident, but non-discoverable credentials can be resident too. The important point is whether the credential follows the model of discoverable credentials and can be used without knowing the credential ID, not whether the authenticator operates statelessly.

So, despite the naming, these fields control whether the credential must be discoverable.

`requireResidentKey` was defined first but is just a boolean. In order to allow sites to express that they prefer, but do not require, a discoverable credential it was necessary to add a second field, `residentKey`. A site must pick one of the following scenarios in order to set these two fields:

1. If the site will always use the credential in the 2nd-factor pattern, i.e. the U2F style where credential IDs will always be provided when requesting a signature, set `requireResidentKey` to false. (The resulting credential may still be discoverable, but discoverable credentials work perfectly well with credential IDs. This is the default if neither field is set.)
2. If the site will use the credential without first collecting a username, thus no credential IDs can be given, then set `requireResidentKey` to true. The resulting credential will be discoverable. (But will still work when credential IDs are given.)
3. If the site will take advantage of discoverable credentials when possible, but also supports using credentials in a 2nd-factor flow with credential IDs, then set `residentKey` to `"preferred"`. (It's possible to optionally take advantage of discoverable credentials with conditional UI, see chapter 8.) Whether the credential is known to be discoverable is reported via the `credProps` extension (see page 75).

Note that there's no way to require a non-discoverable credential. Created credentials can always be discoverable and these controls only allow a site to insist that they must be.

User verification

See page 21 for a discussion of the concept of user verification. The `userVerification` field can be set to one of the strings `discouraged`, `preferred`, or `required`, and the default is `preferred`. Creating a credential with user verification ensures that the capability will be available when getting a signature. It also marks the start of a user verification chain: each future signature from the credential with the UV flag set must have collected some proof of user verification, and that proof must chain back to the proof collected at creation time. For example, if user verification is established with a PIN, then changing that PIN must require the old PIN. If a fingerprint is used, then enrolling a new fingerprint must require an old fingerprint or PIN, etc.

As with `residentKey`, setting `userVerification` to `preferred` means that the site would make use of user verification if performed, but will accept a credential without it. But how "preferred" is it? For example, should a PIN be set up on a security key in order to create a credential with user verification, or should a PIN only be used if it happens to already be set up? That's up to the platform. Windows tends towards a stronger interpretation of "preferred" than other platforms and will, indeed, set up a PIN on a security key in response to a request that prefers user verification.

When it comes to platform authenticators and passkeys, the interpretation of the `userVerification` value varies by provider, and might change over time. Here are the cases when user verification will be performed today, for three common passkey providers:

iCloud Keychain

	Discouraged	Preferred	Required
Biometrics available	✓	✓	✓
Biometrics not available			✓

Google password manager (desktop)

	Discouraged	Preferred	Required
Biometrics available		✓	✓
Biometrics not available			✓

Windows Hello

	Discouraged	Preferred	Required
Biometrics available	✓	✓	✓
Biometrics not available	✓	✓	✓

For simple sites, the presence or absence of user verification will often make no difference: if the WebAuthn signature is valid, the user is signed in and, if not, they aren't. If user verification makes no difference, the parameter should be set to `discouraged`.

More complex sites might base the sign-in decision on a risk analysis, or might subject sign-ins without user verification to additional challenges. In this case, the parameter should be set to `preferred` since doing user verification has value to the user.

Some sites might decide to only ever accept sign-ins that include user verification. In this case, set the parameter to `required`.

Applicable authenticators

The `authenticatorAttachment` field can be left undefined, set to `platform`, or set to `cross-platform`. This controls which types of authenticators can be used.

Setting this to `platform` means that no external authenticators can be used. So security keys and external mobile devices are not applicable. For example, if a site offers to register "this computer" in its UI then it might set this field to `platform`. The resulting platform UI will skip or remove other options.

Setting this to `cross-platform` means that only authenticators that can be moved between devices are applicable. I.e. security keys and external mobile devices. If a company website knows that users have been issued security keys to use with company resources, it might set this to `cross-platform`.

Leaving this field undefined, the default, permits all authenticators, but the platform UI will generally default to a platform authenticator if available. If a site offers to register a passkey in its account settings, it might thus leave this value undefined.

Excluded credentials

Another way in which applicable authenticators can be controlled is by setting `excludeCredentials` in the top-level dictionary. This should contain the credential IDs of all currently known credentials for the user. It communicates that those credentials should not be overwritten, and that any external authenticator with one of those credentials should not be registered again.

Consider a user who is attempting to register a second security key on an account that uses the 2nd-factor flow. Their security key will likely create a stateless, non-discoverable credential and thus no existing credential can be overwritten. But it's completely useless to register two credentials on the same security key: two credentials will appear in the user's account, but there's only a single security key that contains them both. It would be confusing.

By listing all existing credential IDs in `excludeCredentials`, the site can instruct the platform to forbid an existing security key to be registered. If the user tries to do so, they should see an error message and the platform will usually allow them to try again with a different security key.

Next, consider a platform authenticator that always creates discoverable credentials. Recall that the model of discoverable credentials only allows at most one credential with a given RP ID and user ID to exist in an authenticator. So, if the user were to register a second credential it would overwrite the first. There would be two credentials listed on the site, but one would no longer exist in the platform authenticator. This would also be confusing for the user.

In the case of a platform authenticator, however, the platform will do something different if a credential in `excludeCredentials` already exists. The creation process will appear to succeed from the point of view of the user, but the platform will return `InvalidStateError` to the website. This special error code is unique to this situation and the website can choose to show an error. (E.g. "You've already registered this computer".) The site can also choose to ignore this error: the user wanted the platform registered as an authenticator and it already is; no need to complicate things with an error message.

Note that the `excludeCredentials` field is not just a list of credential IDs. It's written like this:

```
[
  {type: "public-key", id: /* credential ID in ArrayBuffer */},
```

```
  /* more such elements ... */
]
```

There is no `type` other than "public-key"—that's extensibility that was never used. The `id` is binary and binary values in WebAuthn, of which there are a lot, are passed in `ArrayBuffer` and `BufferSource` objects. This is awkward because these objects don't convert to/from JSON automatically. There is light on the horizon in the form of native JSON support in WebAuthn (see page 71) but this cannot be relied on to be present in current browsers. So for now, when receiving messages from a server to make WebAuthn calls, any binary data has to be handled specially.

Metadata stored with the credential.
The `rp` (relying party) and `user` fields in the top-level options dictionary contain metadata that is stored with the credential. The `rp` field has this structure:

```
dictionary PublicKeyCredentialRpEntity {
             DOMString id;
    required DOMString name;
};
```

This `id` field specifies the relying party ID. This topic is complex enough that it's covered separately in chapter 5. If omitted, it defaults to the domain of the current origin.

The `name` field is required, but currently never used. It was intended to be a human-friendly name for the site, e.g. "ACME Corporation". If there's no obvious value to use, just pass the empty string as this field will probably be unused forever.

The `user` dictionary looks like this:

```
dictionary PublicKeyCredentialUserEntity {
    required BufferSource id;
    required DOMString    name;
    required DOMString    displayName;
};
```

The `id` field here is the user ID. Recall from the discussion of the discoverable credential model that an authenticator stores at most one credential with a given pair of RP ID and user ID values. The user ID is *not* the username and users will never see this value. It is an opaque, binary identifier for an account that can be up to 64 bytes long. However, since the user ID is treated as less sensitive than a username by security keys, it should not be equal to the username. It cannot be empty either. It's recommended to be a large random value (e.g. a UUID) that is stored by the site for each account.

(In this book we'll always call this value the user ID. But it's also sometimes called the "user handle", including in the WebAuthn API later on! These are two names for the same thing.)

The `name` field is the username. This is a human-readable string that usually uniquely identifies an account. It might be an email address. This value will be displayed in account selectors and management UIs so the user can understand which account a credential corresponds to.

The `displayName` field is for a more "friendly" name for the account. For example, a site may require an account to have a unique username, but allow users to configure a (potentially non-unique) name that they will appear as on the site. That second name would appear in this field. This field cannot be omitted but, if there is no obvious value to put in it, set it to the empty string.

Unlike the username, the display name is not always shown in account selectors and management UI, depending on the platform. Apple's platforms, for example, do not show display names.

Controlling the platform UI

The platform will show UI to help guide the user in creating the requested credential and there are a couple of ways in which the UI can be guided.

Firstly, there's the `timeout` parameter. If a site will reject a credential that took too long to create (because it's concerned that the user has walked away and that someone else could be behind the keyboard) it can set a timeout, in milliseconds. But it can take a while for people to create credentials! They may have to dig a security key out from the back of a drawer. Because of this, platforms are likely to silently increase small timeouts to some minimum value. So a timeout of less than five minutes (the recommended default) may be rounded up.

Secondly, there is the `hints` parameter. This is a catch-all for expressing non-binding requests to the platform. It is a list of strings and currently three are defined:

- `security-key` suggests that the platform should show a security key-focused UI. Enterprise cases where security keys have been issued to employees should set this.
- `client-device` suggests that the platform should focus the UI on using a platform authenticator.
- `hybrid` suggests that the platform should expect the user to use an external mobile device to complete the request. (See chapter 10.)

The first two of these somewhat duplicate the `authenticatorAttachment` field, described above. Unlike that field, they do not forbid the use of other types of authen-

ticators. However, hints are not yet supported by all platforms and so cannot be assumed to have any effect.

Attestation and extensions get their own chapters and are not covered here, but the `challenge` parameter, which is unused during creation except for attestation, is required. You can set it to `new Uint8Array([0]).buffer` in all non-attestation cases.

Common patterns of options

A consumer site, prompting to create a first passkey with the local device:

(A website would have checked if a local authenticator exists first. See page 67.)

```
let promise = navigator.credentials.create({
    publicKey: {
        pubKeyCredParams: [
            {type: "public-key", alg: -7},
            {type: "public-key", alg: -257},
        ],
        authenticatorSelection: {
            authenticatorAttachment: "platform",
            userVerification: "discouraged",
            requireResidentKey: true,
        },
        rp: {id: "example.com", name: ""},
        user: {
            id: userId,
            name: "jsmith",
            displayName: "John Smith",
        },
        hints: ["client-device"],
        challenge: new Uint8Array([0]).buffer,
    }
});
```

A consumer site, after a user clicks “Add passkey” in their account settings:

```
let promise = navigator.credentials.create({
    publicKey: {
        pubKeyCredParams: [
            {type: "public-key", alg: -7},
            {type: "public-key", alg: -257},
        ],
        authenticatorSelection: {
            userVerification: "discouraged",
            requireResidentKey: true,
        },
        rp: {id: "example.com", name: ""},
        user: {
            id: userId,
            name: "jsmith",
```

```
            displayName: "John Smith",
        },
        hints: ["client-device"],
        challenge: new Uint8Array([0]).buffer,
    }
});
```

Interpreting the response

When the `Promise` returned by `navigator.credentials.create` resolves it might result in an error. There are only three buckets of errors that need to be considered: `InvalidStateError`, a programming error, and everything else.

`InvalidStateError` arises when the user attempts to use a platform authenticator that already holds one of the credentials listed in `excludeCredentials`. Sites may wish to report an error to the user in this case ("This device is already registered") or they may validly conclude that the user wanted the local device registered, and it is registered, therefore that's not really an error.

If there's an error in the structure of the options (e.g. a required field is missing), or the site is attempting to use an RP ID that it cannot use (see chapter 5), then a descriptive error will be returned. But this is a bug in the Javascript that should be resolved during development.

For privacy reasons, all other errors are essentially indistinguishable. They will often be of type `NotAllowedError`, and the error message may contain more details. However the error messages are not stable and should not be used for anything other than logging and debugging.

Assuming that promise resolves successfully, it returns a `PublicKeyCredential`. Let's take a look at what that contains.

```
interface PublicKeyCredential {
    // The credential ID.
    USVString   id;
    ArrayBuffer rawId;

    // The type of authenticator used.
    DOMString?  authenticatorAttachment;

    // More about the credential.
    AuthenticatorAttestationResponse response;

    // See extensions chapter.
    AuthenticationExtensionsClientOutputs getClientExtensionResults();
}
```

Firstly, you get the credential ID. Twice! `id` contains the credential ID as a string. Then `rawId` contains it as an `ArrayBuffer`.

Where binary data is encoded in a string, WebAuthn always uses the base64url variant of base64. This replaces the `+` and `/` characters with `-` and `_` and doesn't include any `=` padding characters at the end. So the `id` field is the base64url encoding of `rawId`.

Next, `authenticatorAttachment` will probably contain either `platform` or `cross-platform` depending on the type of authenticator used. (The platform can omit this field if it doesn't know, and beware that it's always possible that future versions of WebAuthn will define new values.)

You should be expecting at least a public key too in order to be able to use this credential, so let's look inside `response`:

```
interface AuthenticatorAttestationResponse {
    // The public key
    COSEAlgorithmIdentifier  getPublicKeyAlgorithm();
    ArrayBuffer?             getPublicKey();

    // Information from the authenticator.
    sequence<DOMString>      getTransports();
    ArrayBuffer              getAuthenticatorData();

    // Only used for attestation.
    ArrayBuffer              clientDataJSON;
    ArrayBuffer              attestationObject;
};
```

`getTransports` returns a list of strings that the site needs to store if it'll ever request this credential by ID. This will be covered later, in the section about getting signatures from credentials.

Next there's `getAuthenticatorData`. This returns a binary blob that comes directly from the authenticator. It's returned both when creating a credential and when getting a signature from one and is covered in more detail later.

Getting the public key

The public key signature scheme of the new credential is returned by `getPublicKeyAlgorithm`. It uses the same identifiers as `pubKeyCredParams` in the options and the value must be one of the algorithms that was listed there. (See page 26.)

`getPublicKey` returns the public key itself in Subject Public Key Info (SPKI) format. This is a commonly used format for public keys, but there are so many public key formats that they get a chapter to themselves. (Chapter 13.) SPKI format can be passed to, at least:

- Java's `java.security.spec.X509EncodedKeySpec`.

- .NET's `System.Security.Cryptography.ECDsa.ImportSubjectPublicKeyInfo`.
- Go's `crypto/x509.ParsePKIXPublicKey`.

(For more obscure public key schemes, the platform might not know how to convert the public key and so `getPublicKey` can return `null`. That doesn't apply to any of the common formats discussed here but, if you're using such a format, you'll have to extract the public key from the authenticator data, see chapter 7.)

There are more options in WebAuthn than in the U2F protocol! But we have finally reached the core values that we expected to get when creating a credential: the credential ID and its public key. Think back to the introduction and recall that the other major operation is getting a signature, where we expect to provide some credential IDs (or not, with discoverable credentials) and a challenge value, and get back a signature and the message that was signed, which should contain the challenge and contextual information to prevent phishing.

Keep that big-picture view in mind as we dive into the second (and final) major operation in WebAuthn.

Getting signatures

Getting signatures from credentials is also performed through the Credential Management API and looks like this:

```
const promise = navigator.credentials.get({publicKey: options});
```

That's exactly the same as creating a credential except that, rather than `create`, the operation is `get`. Again, a dictionary of options is passed in:

```
dictionary PublicKeyCredentialRequestOptions {
    // Core parameters for getting a signature.
    required BufferSource                  challenge;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];

    // The relying party ID.
    USVString rpId;

    // Whether to perform user verification.
    DOMString userVerification = "preferred";

    // Controlling the platform UI.
    unsigned long       timeout;
    sequence<DOMString> hints = [];

    // Extension features.
    AuthenticationExtensionsClientInputs extensions;
};
```

The first option is hopefully familiar from the previous chapters. The `challenge` makes the signature specific to a request so that it can't be used again. It should be a large, random value generated at the server and stored there, temporarily, to check against when the signature is received.

The `allowCredentials` is the list of credential IDs or, if you're using discoverable credentials, it can be empty. If you do wish to pass a list of IDs, they're not passed directly but rather as a list of objects that look like this:

```
[{
   type: "public-key",
   transports: ["hybrid", "internal"],
   id: /* ArrayBuffer */,
}]
```

It's structured just like `excludeCredentials` except for the additional transports list. That list of transports should be copied verbatim from the result of calling `getTransports` when creating a credential. (See above.)

The `rpId` is the relying party ID. It must match the value used when the credential was created. See chapter 5 about picking a relying party ID.

The `userVerification` field is either `discouraged`, `preferred`, or `required`. This has the same meaning as when the credential was created, see page 28. Note that requiring user verification when it wasn't performed at registration time might not work: the credential might have been created on an authenticator that doesn't support it. So this value will usually be the same as when the credential was created.

The `timeout` and `hints` parameters also have the same meaning as during creation, and the `extensions` get their own chapter. And with that, we're done. Handling the response is more complex, however…

Interpreting the response

The result of the promise from `get` might, of course, be an error. There are no particular error cases that you need to handle, so all errors can be treated as generic failures.

Otherwise the response to a signature request is also a `PublicKeyCredential` object, but the type of the `response` field within it is different. Here's a recap of the structure, which was discussed on page 34.

```
interface PublicKeyCredential {
    // The credential ID.
    USVString   id;
    ArrayBuffer rawId;

    // The type of authenticator used.
    DOMString?  authenticatorAttachment;
```

```
    // More about the assertion.
    AuthenticatorAssertionResponse response;

    AuthenticationExtensionsClientOutputs getClientExtensionResults();
}
```

This structure contains the credential ID (twice, again) which lets you know which credential was used. If the `allowCredentials` list was empty, this tells you who the purported user was. But, even if it wasn't, `allowCredentials` can contain multiple credentials and so this tells you which to validate against.

The rest of the new values are contained in the `response` field:

```
interface AuthenticatorAssertionResponse {
    ArrayBuffer? userHandle;
    ArrayBuffer  signature;
    ArrayBuffer  authenticatorData;

    // Part of the signed message.
    ArrayBuffer clientDataJSON;
};
```

The name of this structure is the first time that we've come across the term *assertion*. Until now we've always called the thing generated from the private key a *signature*. The term assertion refers to the whole response, as defined here, and the rest of the text will only use the word signature when specifically referring to that field.

An assertion contains several fields. First there's a field called the `userHandle`. Above, it was mentioned that the user ID is sometimes called the user handle, and here it is! The value unfortunately has different names in the two places that it appears in WebAuthn, but it is the `user.id` value that was set at creation time.

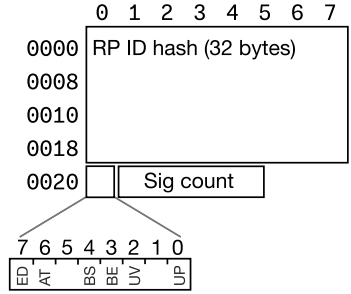
Note that it's optional. There was no user ID field in the U2F protocol when we covered it, so U2F security keys cannot possibly store this value. In fact, non-discoverable credentials aren't required to store it, only discoverable ones. In either case, this field should only be used in very particular situations:

The credential ID is the best identifier for a credential, but it's randomly generated by the authenticator. Some sites cannot support looking up an account based on an identifier like that and must generate the identifiers themselves. The user ID can serve that need since it's specified by the site but, in order to ensure that it's always returned, such a site must require discoverable credentials. (See also page 103.)

The next field is the `signature`. This must be validated by the public key signature scheme used by this credential. But, to do so, you need to construct the message that was signed. That's where the authenticator data comes in.

Authenticator data

The authenticator data is so named because it comes directly from the authenticator. Since it's included in the signed message, it must be exposed directly by WebAuthn: any transformation would change the bits and cause the signature verification to fail. So here you have to deal with a binary format; no more Javascript objects.



You might notice that looks very similar to the signed message format in the U2F protocol. That is no accident! Since this data cannot be modified, in order to be backwards compatible the format has only been added to since U2F. (Otherwise U2F security keys wouldn't be interoperable with WebAuthn.)

The flags are named with two-character abbreviations, as shown in the diagram above:

- `UP`: User Presence. (Discussed previously.)
- `UV`: User Verification. (Discussed previously.)
- `BE`: Backup Eligible. Indicates that the credential can be backed up (i.e. is a passkey).
- `BS`: Backup State. Indicates that the credential *has* been backed up. In practice, passkey providers just set both BE and BS flags at all times. In theory, if syncing was paused, then you could see BE set without BS, indicating that syncing was pending.
- `AT`: Attested credential data. Indicates that an "attested credential data" structure follows. This will always be set when creating a credential and never set when getting an assertion. (See chapter 7.)
- `ED`: Extension Data. Indicates that a CBOR map of extension data follows. If both AT and ED are set, then the extension data always comes second.

We are finally at the point where we can validate the `signature`! Since WebAuthn is backwards compatible with U2F security keys, the signed message must be compatible with what U2F did, so it's the concatenation of the authenticator data with the hash of the `clientDataJSON`.

There are many checks that the server should make on the signed response in order to be secure. These are covered in chapter 12. But congratulations, you now understand the core of WebAuthn.

Client Data

We saw an example of the client data in the U2F chapter. It appears as the `clientDataJSON` in the assertion response. Let's look at it again and discuss each value in more detail:

```
{
 "type": "webauthn.get",
 "challenge": "AAECAwQFBgcICQoLDA0ODw",
 "origin": "https://example.com",
 "crossOrigin": false
}
```

`type`

It's good security hygiene to ensure that any signed message is unambiguous about what it is. Many security issues have been caused by inducing two parties in a system to have divergent views about what is going on. So the client data uses the `type` field to specify the context in which it should be interpreted.

`challenge`

This field is the base64url encoding of the challenge specified in the request. As discussed in the introduction, this ensures that the signature is specific to an authentication request and doesn't devolve into what is effectively a password for the account that could be reused later.

`origin`

The origin field specifies the entity that requested the signature, here a URL. This prevents phishing by ensuring that the signature is specific to the requester and cannot be proxied by a malicious site. But the requester is not always a website. Mobile apps can also make WebAuthn requests using platform APIs covered in chapter 11.

For Android apps this field will contain `android:apk-key-hash:` followed by the base64url encoding of the SHA-256 hash of the APK signing certificate. For iOS/iPadOS, the origin will contain the requested RP ID with `https://` prefixed. This is a mistake! It's probably too late to change it now but, because of this, it's not possible to distinguish between requests from apps and requests from websites with the Apple APIs.

`crossOrigin`

This specifies whether the request came from an iframe that is not same-origin with all its parent frames. This is obviously specific to the web, and most sign-in pages

will use the `frame-ancestors` directive in a `Content-Security-Policy` header to prevent ever being shown in an iframe. But, if you need to make WebAuthn requests from an iframe, see page 71.

`topOrigin`
If `crossOrigin` is true, this contains the origin of the top-level frame, so that the server can see where the iframe was embedded.

It's important to note that these fields are not exclusive: more fields have been added to WebAuthn over time. So a server-side validator must be able to handle unknown fields in the JSON. But some validators do not want to have the complexity of a full JSON parser. For them, WebAuthn does guarantee some additional structure:

The fields `type`, `challenge`, `origin`, and `crossOrigin` are guaranteed to appear and guaranteed to appear in that order without any spaces or newlines between the JSON tokens. If `crossOrigin` is true then the next field is guaranteed to be `topOrigin`, again without any spaces or newlines between tokens. All strings in this prefix are guaranteed to be minimally escaped.

Any implementation taking advantage of this should carefully follow these steps[5].

Common patterns of options
A site using discoverable credentials:

```
let promise = navigator.credentials.get({
    publicKey: {
        challenge: new Uint8Array([
            // Must be a cryptographically-random number sent
            // by the server
            0x79, 0x50, 0x68, 0x71, 0xDA, 0xEE, 0xEE, 0xB9,
            0x94, 0xC3, 0xC2, 0x15, 0x67, 0x65, 0x26, 0x22,
            0xE3, 0xF3, 0xAB, 0x3B, 0x78, 0x2E, 0xD5, 0x6F,
            0x81, 0x26, 0xE2, 0xA6, 0x01, 0x7D, 0x74, 0x50,
        ]).buffer,
    },
});
```

An enterprise site, requesting that an employee authenticate with their issued security key:

```
let promise = navigator.credentials.get({
    publicKey: {
        allowCredentials: [{
          type: "public-key",
          id: new Uint8Array([
              0x94, 0x38, 0x2b, 0x37, 0xbf, 0x38, 0xc0, 0x05,
              0x9a, 0xbd, 0x16, 0x09, 0xdd, 0xf5, 0xd7, 0x0c,
          ]).buffer,
          transports: ["usb"],
```

```
        }],
        challenge: new Uint8Array([
            // Must be a cryptographically-random number sent
            // by the server
            0x79, 0x50, 0x68, 0x71, 0xDA, 0xEE, 0xEE, 0xB9,
            0x94, 0xC3, 0xC2, 0x15, 0x67, 0x65, 0x26, 0x22,
            0xE3, 0xF3, 0xAB, 0x3B, 0x78, 0x2E, 0xD5, 0x6F,
            0x81, 0x26, 0xE2, 0xA6, 0x01, 0x7D, 0x74, 0x50,
        ]).buffer,
    },
});
```

Threats

Keep in mind that WebAuthn is only about authentication. If you authenticate yourself on a machine that is controlled by malware, then the malware has all the same authority that you do. At best, some sites require frequent reauthentication with WebAuthn so that the malware can't exfiltrate long-lived cookies and has to remain active on the compromised machine. That increases the attacker's costs, but is no silver bullet.

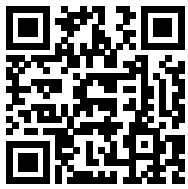
In the introduction we mentioned that Javascript injected into a site could steal passwords. WebAuthn certainly stops passwords from being stolen by removing the need to enter them. But, similar to the malware case, if the malicious Javascript controls the origin context in the browser, it can make authenticated HTTP requests just as if it were the user. It's certainly a lot less convenient for the attacker than exfiltrating a password, but an attack is still possible.

Lastly, many sites add WebAuthn as an authentication method alongside a password. But, as long as the password is still a valid way to sign in, a phishing site can try to get the user to enter it by pretending that WebAuthn isn't working. The ultimate goal must be to remove passwords from accounts. The "backup state" flag in the authenticator data reports whether a credential has been backed up and, if an account has a backed-up credential, and the user has a history of successfully using it, perhaps prompt them to disable their password.

It's still the case that passwords cause a lot of problems and we should try to address them. WebAuthn is by far the best prospect in that direction and the world would be much better off if it succeeds. But magical thinking only leads to disappointment, and WebAuthn isn't magic.

---

1 www.w3.org



2 www.w3.org

3 en.wikipedia.org

4 media.defense.gov

5 www.w3.org

CHAPTER 5
Relying party IDs

As discussed in the previous chapter, relying party IDs (RP IDs) identify sites and prevent the use of credentials between them.

This happens to result in some degree of phishing protection, but this mechanism is not designed to prevent phishing. Phishing is stopped by the `origin` field in the client data, which goes into the signed message. RP IDs, on the other hand, are:

1. A constraint to prevent credentials from being used too widely. WebAuthn deliberately does not want people to have a single credential that is used everywhere.
2. A way to filter candidate discoverable credentials so that a user is only shown a list of applicable accounts to choose from.
3. A way to prevent other sites from obtaining any secrets linked to the credential. (See page 76.)

RP IDs are "domain shaped" rather than "URL shaped". E.g. `example.com` is an RP ID, but `https://example.com` is not.

(The U2F Javascript API, which was mentioned in chapter 2, *did* use URL-shaped values as RP IDs, which it called AppIDs. That has an impact on WebAuthn in the form of the `appid` and `appidExclude` extensions, which are covered on page 84. But otherwise AppIDs are irrelevant now.)

Every operation has an RP ID associated with it, whether it's using the WebAuthn API or one of the platform APIs. While an API may pick a default RP ID if one isn't specified, it's always possible to request a specific RP ID, and so every API defines a way of validating whether a website or app is allowed to request a specific RP ID.

For WebAuthn, the default RP ID is the domain of the current origin. So for `https://example.com` the default RP ID is `example.com`. WebAuthn allows an origin to claim any RP ID that can be formed by discarding zero or more "labels" from the left of its domain name until it hits an *effective TLD*[1]. A label in a domain name is a substring delimited by periods. So for `www.example.com`, the labels are `www`, `example`, and `com`. (Plus the empty label at the end to represent the root, but we'll ignore that.)

An effective TLD is an effective top-level domain. That is either a top-level domain, like `com`, or a domain that acts like a top-level domain and is listed in the Public Suffix List[2], like `co.uk`.

So say that the current origin is `https://www.example.co.uk`: it can specify an RP ID of `www.example.co.uk` (discarding zero labels), `example.co.uk` (discarding one

label), but not `co.uk` because that's an effective TLD. It also couldn't specify an RP ID of `example.com` because that's a different site: that domain cannot be formed simply by discarding labels from the left of `www.example.co.uk`.

> There is no relationship between similar-looking RP IDs beyond WebAuthn's rules for which origins can claim which RP IDs. So `www.example.co.uk` is as different to `example.co.uk` as to `example.org`. Recall that RP IDs are hashed before being sent to U2F security keys, enforcing that all internal structure is irrelevant.

Android

Android checks whether a given app is allowed to use an RP ID by treating the RP ID as a domain name and fetching `https://example.com/.well-known/assetlinks.json`. This needs to be a JSON file in the Digital Asset Links format, which broadly attempts to describe the relations between different entities.

> In fact, it's incorrect to say that "Android" checks this because the check is actually performed by the passkey provider handling the request. Google Password Manager, which ships as part of Play Services, uses `assetlinks.json` and has defined the norm on that platform, but, technically, other password managers could decide to do something different.

The `assetlinks.json` file needs to contain JSON similar to the following:

```
[
  {
    "relation" : [
      "delegate_permission/common.handle_all_urls",
      "delegate_permission/common.get_login_creds"
    ],
    "target" : {
      "namespace" : "android_app",
      "package_name" : "PACKAGE_NAME",
      "sha256_cert_fingerprints" : [
        "APK_CERTIFICATE_FINGERPRINT"
      ]
    }
  }
]
```

The `PACKAGE_NAME` placeholder should be replaced with the name of the Android package, e.g. `com.example.myapp`. The `APK_CERTIFICATE_FINGERPRINT` should be replaced with a hex-with-colons encoded, SHA-256 hash of the APK signing certificate. You can get this value from an APK with:

```
keytool -list -printcert -jarfile app.apk
```

Or you can run `./gradlew signingReport` in your build directory. Either way, the fingerprint value must separate each pair of hex digits with a colon, like `ab:cd:12:34:...`.

This JSON object should be replicated for each app that needs access to the RP ID. When doing so, consider that this is declaring a significant level of trust for each listed app. Not only can they use the RP ID, but saved passwords may be shared between these apps and the website too.

Currently, the only relation needed is `delegate_permission/common.handle_all_urls`, but Google hopes to transition to the more specific `delegate_permission/common.get_login_creds` relation in the future and currently documents that apps should list both.

The `assetlinks.json` is currently not allowed to be behind a redirect. So, if `https://example.com` just serves redirects to `https://www.example.com` then that won't work. To debug issues, run:

```
curl 'https://digitalassetlinks.googleapis.com/v1/assetlinks:check?
source.web.site=https://example.com&relation=delegate_permission/
common.handle_all_urls&target.android_app.package_name=PACKAGE_NAME&
target.android_app.certificate.sha256_fingerprint=APK_CERTIFICATE_
FINGERPRINT'
```

Remember to replace `PACKAGE_NAME` and `APK_CERTIFICATE_FINGERPRINT` with the same values as in the JSON above.

To see an example of an `assetlinks.json` file, try fetching it for a commonly-known site. E.g. `https://amazon.com/.well-known/assetlinks.json`.

Apple platforms

The Apple platforms use a similar system of `.well-known` files in order to decide which apps are allowed to use a given RP ID. For Apple devices, the file is `https://example.com/.well-known/apple-app-site-association` and it should include something like this:

```
{
  "webcredentials": {
      "apps": [ "T7AYYU7S6A.example.com.YourApp" ]
   }
}
```

The first label of the app identifier (`T7AYYU7S6A` in the example) is your "Team ID". You can find this under "Membership details" on `https://developer.apple.com/account`. The rest is the "bundle identifier" for your app, which is set when the project is created.

Requests for this file will come from Apple's servers by default but, for testing, it's possible to enable "Associated Domains Development" in the Developer menu on iOS. (But only, it seems, on devices. The option doesn't appear in the simulator.) If you want to see what Apple's servers see for your domain, fetch `https://app-site-association.cdn-apple.com/a/v1/example.com`.

An app also needs to list `webcredentials:example.com` as a domain in the "Associated Domains" capability.

If you change your site association file, delete the app and reinstall to ensure that any caches are updated on the device.

Browsers and other privileged apps

Browsers have to be special. Consider what would happen if you browsed to `https://example.com` on a phone and tried to use a passkey to sign in. Using just the rules specified above, example.com would have to authorize every possible browser app to use its RP ID.

This would be untenable, so browsers have to be trusted to act as any RP ID. There are processes run by both Apple and Google to recognize trusted browser apps for this purpose.

> As with `assetlinks.json`, the set of recognized browsers on Android is just a norm established by Google Password Manager, which publishes its list of trusted browsers at `https://gstatic.com/gpm-passkeys-privileged-apps/apps.json`. Other password managers on Android tend to follow this norm.

Considerations when choosing an RP ID

It's important to carefully consider RP IDs from the outset. Let's take the example of `https://www.example.co.uk`. That site might happily be creating credentials with its default RP ID (`www.example.co.uk`) but later decide that it wants to move all sign-in activity to an isolated origin, `https://accounts.example.co.uk`. But none of its credentials could be used from that origin. It could discard a label from the left to form `example.co.uk`, but the rules don't allow any labels to be prepended, so `www.example.co.uk` isn't a valid RP ID for that origin. The site would have needed to create credentials with an RP ID of `example.co.uk` from the outset.

But the rule is not to always use the most general RP ID possible. Going back to our example, if `https://usercontent.example.co.uk` existed to host uploaded content, then pages on that origin could *create* credentials with an RP ID of `example.co.uk`. We can assume that `accounts.example.co.uk` is checking the origin of any assertions, so `usercontent.example.co.uk` can't use its ability to set an RP ID of `example.co.uk` to generate valid signatures, but it can still try to get the

user to create new credentials which could overwrite the legitimate ones. It can also get any secrets associated with the credentials because it can assert them. (See page 76.)

All this means that the choice of RP ID needs to be considered carefully at the beginning of any deployment.

Related origins

As described above, you have a lot more freedom with the native APIs than you do on the web. With the native APIs, you can nominate many apps to be able to use your RP ID, but the rules on the web don't allow any other website, even with permission, to use another site's RP ID.

Not all websites fit into that structure. Some are spread across country code top-level domains and exist as `example.com`, `example.de`, `example.in`, etc. But with the rules above, none of those country-specific instances of the site could share credentials. Sometimes two different brands are intimately linked, but have separate websites, like Hilton and DoubleTree.

Probably the best option in this case is to use a single origin to handle all sign-ins, and use a federation protocol like OpenID Connect[3] on the related sites. But not all websites can do that, thus the RP ID rules for the web were relaxed somewhat with the introduction of related origins. When a WebAuthn request specifies an RP ID that would not be permitted under the rules above, browsers that implement related origins will attempt to fetch a document hosted at the following URL.

```
https://example.com/.well-known/webauthn
```

If it exists, and has the MIME type `application/json`, then it is parsed as JSON and can contain something like the following.

```
{
    "origins": [
        "https://example.com",
        "https://example.co.uk",
        "https://example.in",
        "https://www.example.in",
        "https://otherbrand.com"
    ]
}
```

If the origin that made the WebAuthn request is listed as one of the permitted origins in that document, then the request will be allowed to continue. However, there are limits on how many origins can be listed like that:

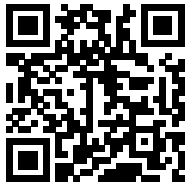
For each listed origin, its eTLD+1 label is extracted from its domain name. The eTLD+1 label is the rightmost label that is not part of the effective TLD. For the example origins above:

Origin	eTLD+1 label
`https://example.com`	`example`
`https://example.co.uk`	`example`
`https://example.in`	`example`
`https://www.example.in`	`example`
`https://otherbrand.com`	`otherbrand`

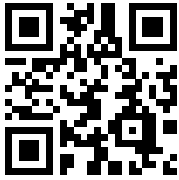
The maximum number of distinct eTLD+1 labels currently permitted by browsers is five. So for sites that are spread across many country-code top-level domains, all of those domains only count as one label. But if you have many different brands, then you could quickly hit the limit.

---

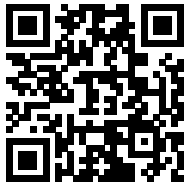
1 en.wikipedia.org



2 publicsuffix.org



3 openid.net



CHAPTER 6
CTAP2

Chapter 2 covered the protocol for communicating with U2F security keys, but that protocol doesn't support discoverable credentials, user verification, or many other features of WebAuthn. So a new protocol for communicating with security keys was needed: CTAP2. (Computer To Authenticator Protocol Two; the U2F protocol can be considered to be CTAP1.)

CTAP2 is a significantly more complex protocol which involves the computer and security key exchanging messages encoded in a format called CBOR. This use of CBOR breaks through into WebAuthn in several places so, even if you're going to skip most of this chapter, a familiarity with CBOR will be valuable.

CBOR

CBOR officially stands for "Concise Binary Object Representation", but the lead author's name is C. Bormann, which might have had more to do with it. It is described in RFC 8949[1] and is a MsgPack-inspired format in the family of "binary JSONs".

CTAP2 (and thus WebAuthn) only uses a subset of CBOR, which will be described here. CBOR used in CTAP2 and WebAuthn must conform to this subset, so beware of using generic CBOR libraries for encoding as they may not stay within it.

Each CBOR value (called a *data item*) starts with a leading byte. The most-significant three bits specify the *major type* from zero to seven. The value of the five least-significant bits specifies how to calculate the *argument*:

- ≤23: the argument is the value of the five least significant bits.
- 24: the argument is taken from the following byte.
- 25: the argument is taken from the following two bytes, in big-endian order.
- 26: the argument is taken from the following four bytes, in big-endian order.
- 27: the argument is taken from the following eight bytes, in big-endian order.
- ≥ 28: invalid in CTAP2.

The argument is then interpreted depending on the value of the major type:

- 0: the argument is an unsigned integer.
- 1: the argument is a negative integer formed by adding one to the value and negating. Thus an argument of 19 represents −20.
- 2: a byte string. The argument specifies the number of following bytes that make up the byte string.

- 3: a text string: The argument specifies the number of following bytes that are a UTF-8 encoded string. (But see below because text strings aren't always valid UTF-8 in CTAP2.)
- 4: an array. The argument specifies the number of following data items in the array.
- 5: a map. The argument specifies the number of pairs of data items that follow. The first value in each pair is a key, and the second is the corresponding value.
- 6: invalid in CTAP2
- 7: the argument is either 20 to represent the `false` value or 21 to represent `true`. All other values are invalid in CTAP2.

There are some additional rules applied to ensure that the CBOR is canonically encoded. I.e. that a given CBOR message has exactly one valid encoding:

1. Arguments must be encoded in the fewest possible bytes. So the value 10 is always encoded in a single byte and a byte string of length 10 never encodes that length with any additional bytes.
2. The pairs of data items in a map are sorted by key. Lower major types sort first with ties resolved in favor of shorter keys. If two keys have the same major type and length, then they are compared lexicographically. (Two identical keys cannot appear in a map.)

This canonical form is the same as the Core Deterministic Encoding Requirements[2] from the CBOR RFC, since only integers and strings appear as map keys in CTAP2, but it is *not* the same as the "Canonical CBOR" section of the older RFC 7049.

> Since security keys are either impossible or very difficult to update, and often under significant code-size pressure, using canonical encodings minimizes interoperability problems.

In the underlying CBOR data model, integers are signed, 65-bit values. However, since support for 65-bit integers is rare in programming languages, treating integers as signed 64-bit values is always sufficient in CTAP2.

One point about UTF-8 strings is worth keeping in mind. Security keys are embedded devices with limited storage and CPU. WebAuthn allows sites to specify arbitrary strings for things like user names, which security keys have to store (at least for discoverable credentials). Since storage is limited, security keys are allowed to truncate overly-long strings. However, when they do that truncation, they often do so after a fixed number of bytes, which may fall in the middle of a multi-byte UTF-8 sequence. Thus, when that string is returned in response to a future operation, the UTF-8 will be invalid. Platforms have to make accommodations for this because

that makes the CBOR from security keys technically invalid and standard CBOR parsers will reject it.

Commands and responses

Commands in CTAP2 consist of a single byte to identify the command, followed by a CBOR map with all the parameters. Responses consist of a single-byte response code which is either non-zero, representing an error, or zero for success. Successful responses are optionally followed by a CBOR map with details of the response.

Let's look at the first command sent to a CTAP2 security key: `authenticatorGetInfo`. It's command number four, and it doesn't have any parameters, so the whole command is a single byte with value 0x04. The response will consist of a zero byte (to indicate success) followed by a CBOR map. Let's look at the contents of that map (translated into CBOR's diagnostic notation) to get a feel for the protocol.

(There's a lot here, you don't need to understand everything. It's just to give a sketch of all that's contained in CTAP2.)

```
{
  # Supported protocol revisions
  1: ["FIDO_2_0", "FIDO_2_1_PRE", "FIDO_2_1"],
  # Supported extensions.
  2: [
       "credProtect",
       "hmac-secret",
       "largeBlobKey",
       "credBlob",
       "minPinLength"
  ],
  # The AAGUID of the security key. This identifies the make & model
  # of security key.
  3: h'D8522D9F575B486688A9BA99FA02F35B',
  4: {
    # Supports discoverable credentials.
    "rk": true,
    # Supports user presence tests.
    "up": true,
    # Supports user verification, but it's not configured.
    "uv": false,
    # Is not a platform authenticator.
    "plat": false,
    # Not actually an official option!
    "uvToken": true,
    # The security key will do user verification for all operations.
    "alwaysUv": true,
    # Supports listing and deleting discoverable credentials.
    "credMgmt": true,
    # Supports the `authenticatorConfig` CTAP2 command.
```

```
  "authnrCfg": true,
  # Has a biometric sensor, but it's not configured.
  "bioEnroll": false,
  # Supports PIN entry on the computer.
  "clientPin": true,
  # Supports the largeBlob extension.
  "largeBlobs": true,
  # Supports tokenized user verification.
  "pinUvAuthToken": true,
  # Supports configuring the minimum PIN length.
  "setMinPINLength": true,
  # Doesn't support creating credentials without user verification.
  "makeCredUvNotRqd": false,
  # Whether the authenticator supports older versions of a couple
  # of commands.
  "credentialMgmtPreview": true,
  "userVerificationMgmtPreview": false
},
# Maximum command size, in bytes.
5: 1200,
# User verification token protocol versions, in preference order.
6: [2, 1],
# Maximum number of credential IDs that can be included in a
# single command
7: 8,
# The maximum length of a credential ID from this security key.
8: 128,
# The list of supported transports.
9: ["usb"],
# The list of supported signature schemes: ECDSA with P-256
# and Ed25519.
10: [
   {"alg": -7, "type": "public-key"},
   {"alg": -8, "type": "public-key"}
],
# Maximum largeBlob array.
11: 1024,
# The current PIN doesn't have to be changed immediately.
12: false,
# The minimum PIN length.
13: 4,
# Firmware version.
14: 328966,
# Maximum length of a credBlob value.
15: 32,
# Number of RP IDs that can be configured to receive the minimum
# PIN length.
16: 1,
# Number of biometric attempts that can fail before falling back
# to using a PIN.
```

```
  17: 3,
  # The type of biometric sensor used: a fingerprint reader
  18: 2,
  # How many more discoverable credentials can the security key store.
  20: 25
}
```

A few patterns of the protocol are demonstrated here: the top-level keys in the CBOR maps are integers for compactness, but strings can be used to identify things too. Sometimes WebAuthn Javascript structures are transliterated into CBOR: key 10 clearly mirrors the `pubKeyCredParams` structure which should be familiar from chapter 4. Lastly, it's obviously a lot more complex than U2F!

We won't be covering every corner of CTAP2 in this book; instead we'll be focusing on some of the higher-level concepts. The FIDO Alliance publishes the CTAP2 specification[3] if you want all the details.

If you want to see the `authenticatorGetInfo` response for a given CTAP2 security key, do an operation with it in Chrome on macOS or Linux and then open `chrome://device-log`. It'll be logged there along with other details of the request.

User verification

User verification is one of the headline features of CTAP2. But performing a single WebAuthn operation can require many CTAP2 commands, in the same way that it required many U2F messages. So the platform has to ask the security key to verify the user, and the security key returns a secret value to the platform to represent that verification, which the platform uses as an HMAC key to authenticate all the commands to which that verification applies. The platform is then trusted to discard the token (although there are limits, as we'll see).

There are broadly two forms of user verification supported in CTAP2. Either the user verification is built into the security key itself (with a fingerprint reader or a PIN pad), or user verification is done by entering a PIN on the computer and sending it to the security key to be checked. The CTAP2 spec refers to the computer as the "client" and thus the latter pattern is called *client PIN*.

The former is more secure because it eliminates the risk of the PIN being captured while it is entered on the computer. But a client PIN allows for security keys to be simpler and cheaper.

PIN protocols

CTAP2 does not expose PINs and user verification tokens as plaintext in the protocol. While that's not a huge concern for USB-connected security keys, security keys can also work via NFC. So the computer and security key perform an elliptic-curve

Diffie–Hellman (ECDH) key agreement whenever UV is used in order to mutually calculate a secret key to encrypt any PINs and tokens with.

The *PIN protocol* (which, despite the name, is also used for built-in user verification methods to protect the resulting token) specifies the key agreement, encryption, and authentication primitives to use. There are only two defined by CTAP2 and the second is just a tweak of the first to make it easier to certify under NIST's FIPS 140 programs. The second version is now required to be implemented by security keys and so will slowly replace the first. That's what will be described here.

Client PIN

In the `authenticatorGetInfo` response above, both `clientPin` and `pinUvAuthToken` are true, which means that the security key has a PIN set and supports getting a token to represent a user verification. To get this token, the platform would do the following:

1. Ask the user to enter their PIN.
2. Hash it with SHA-256.
3. Send an `authenticatorClientPIN` command to get an ephemeral ECDH value from the security key.
4. Send another `authenticatorClientPIN` with its own ECDH value and 16 bytes of the hashed PIN, encrypted with the mutual shared secret from the ECDH calculation.

The security key decrypts the PIN hash and compares it against the correct value. The security key can enforce a maximum number of attempts but, if it's satisfied, it encrypts a random value, called a PIN/UV Authentication Token (PUAT), and returns it to the platform.

Now that the platform has the PUAT, it can use it as an HMAC key to authenticate future commands. When it requested the PUAT, the platform had to specify what type of commands it was planning to use it for, and the relying party ID for all those commands. Thus, even if the PUAT leaks, its scope is limited.

That doesn't make a lot of difference if UV is implemented with client PIN because a misbehaving platform could just save the PIN itself, and obtain as many PUATs as it needed. But when user verification is built into the security key, it does limit the abilities of misbehaving platforms a little.

Getting a PUAT from a security key that has a built-in method of user verification, like a fingerprint reader, looks very similar, but there's no encrypted PIN in the request. In contrast with U2F, CTAP2 security keys don't have to immediately respond to commands and so can take their time, flashing and waiting for the user to present their fingerprint.

However, security keys with fingerprint readers will often have a client PIN too because fingerprint readers can be temperamental.

Making credentials and getting assertions

Having covered these operations in detail in chapter 4, there is not much more that needs to be said about them here, because the CTAP2 commands for these operations simply transliterate most of the WebAuthn structures into CBOR.

The complexity of implementing these operations in CTAP2 comes from the fact that the size of the commands can exceed the message buffer of the security key. If you look at the example `authenticatorGetInfo` result above, key 5 specifies the maximum number of bytes in a command, but the `excludedCredentials` list in a `create()` operation, or the `allowCredentials` list in a `get()`, can be arbitrarily long. While it's unusual, it's valid to have 50 credential IDs, each of which are 100 bytes. That would exceed the message buffer of nearly any security key.

The `authenticatorGetInfo` result specifies the maximum length of a credential ID that the security key will generate (in key 8). So any credential IDs longer than this can be discarded by the platform when interacting with this security key, potentially reducing the problem but the core issue remains.

So, after doing that filtering, the platform needs to split up the list of credential IDs into batches such that no batch causes a command to be too long. It will then probe for credential IDs with these batches, very much like the U2F protocol, until it finds a batch where one of the credential IDs was recognized by the security key. Each of these probe messages sets the `up` (user presence) flag to false, so that no physical interaction with the security key is needed.

Things get particularly complicated when user verification interacts with the `appId` extension (see page 84) but these are details that only platform implementers need to concern themselves with, and so we won't cover them here.

Management commands

CTAP2 also includes a number of commands that aren't directly used for implementing WebAuthn operations, but it's still useful to be aware of them.

Reset

Security keys support being reset. This erases all credentials and configuration, and rotates any root secret for stateless credentials such that they are invalidated. For obvious reasons, this is a dangerous command! Thus, security keys don't always support this command over NFC if they also support USB and, over USB, this command is only valid within a few seconds of inserting the security key.

Fingerprint enrollment

CTAP2 allows the platform to drive fingerprint enrollment and to manage finger-

print templates. The enrollment process is like that for any fingerprint reader: you have to press your finger repeatedly on the sensor until it has gathered enough data to compute a template that it can later match against. The security key can return a series of status messages ("finger too far to the left", "finger wasn't pressed for long enough", etc) for the platform to report back to the user.

The `authenticatorBioEnrollment` command also allows the platform to list and delete any existing templates, each of which can have a "friendly name" set so that the user can identify them.

Credential management

The `authenticatorCredentialManagement` lets the platform list the RP IDs of all the discoverable credentials on a security key. Then, for a given RP ID, it lets the platform list each discoverable credential recorded, including the user information. Discoverable credentials can be deleted, and the user information can be updated.

Miscellaneous configuration

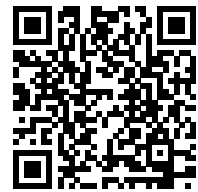
The `authenticatorConfig` command allows a number of miscellaneous features to be controlled:

- Enterprise attestation (see page 64) can be turned on and off.
- If the security key supports it, the policy to require user verification for every operation can be turned on and off.
- The set of RP IDs that can query for the minimum PIN length (see page 83) can be set.

---

1 www.rfc-editor.org

2 datatracker.ietf.org



3 fidoalliance.org

CHAPTER 7
Attestation

If a company has distributed security keys to its employees in order to protect authentication to its corporate resources, it might want to know if those employees are actually using them and not others. It might even want to know that they are using the precise security key that was inventory tracked and assigned to them. That is what attestation is for.

Nearly all security keys ship from the factory with an attestation private key included within them. Unlike every other private key that we have dealt with so far, it is not generated on demand and it is not specific to any given credential; it is global to the security key. When you create a credential in these security keys, they will sign over the resulting public key with their attestation private key to show that the newly generated key was generated within that security key.

This creates a tension: if the attestation private key were specific to the security key then all credentials generated in that security key could be correlated across RPs. But if the same private key were used in lots of security keys then the potential would exist for an attacker to extract it from another security key of the same model, and so the security of the attestation private key would be reduced.

FIDO balanced this tension by requiring that attestation private keys must be used over a batch of at least 100 000 security keys. So they do not uniquely identify an individual security key, they instead identify a large batch of them and so convey only the make and model of the security key. Correspondingly, this means that if you have the right (expensive) equipment, you could buy a security key from that batch and extract the attestation private key. (And some security keys don't sell well enough to ever have a full 100 000-unit batch and, for those, the privacy is proportionally limited.)

For cases where this level of attestation is insufficient, and the attestation private key really needs to be specific to a security key, the concept of "enterprise attestation" was created, which allows this in a restricted fashion.

Getting attestation

WebAuthn does not provide attestation by default; it's assumed that most sites will not want it. To request attestation when making a credential, set the `attestation` parameter in the `PublicKeyCredentialCreationOptions` dictionary to `direct`.

Just like a regular signature in WebAuthn, you want to bind the signature from the attestation private key to the specific creation, thus it needs a challenge. Therefore, when doing attestation, set the `challenge` parameter in that dictionary to a large random value from the server.

Once the credential has been successfully created, you *may* have received an attestation. To find out, you need to look at the `response.attestationObject` field of the resulting `PublicKeyCredential`.

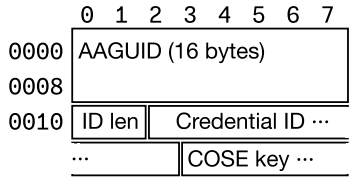
Since this field can contain data that comes directly from the security key, and is signed by the attestation private key, we are leaving the realm of JavaScript. It is a CBOR map in the CTAP2 subset of CBOR (see chapter 6). It contains the attestation information but also the credential public key and the authenticator data (see page 39).

Having the public key inside `attestationObject` may sound duplicative because chapter 4 already covered how to get the public key. But the server cannot use the `getPublicKey`, `getAuthenticatorData`, and `getPublicKeyAlgorithm` methods when checking attestation. Those methods are convenient because the platform will do the parsing and conversion for you, but the attestation private key doesn't sign the nicely parsed and converted values, it signs the raw data from the security key. And so an implementation that wants to check attestation must only trust the raw data in `attestationObject`.

The CBOR map will contain at least the following keys:

Key	Value
`fmt`	A string that specifies the type of attestation provided. If this is `none`, then no attestation has been provided.
`authData`	The authenticator data, a byte string.
`attStmt`	The attestation information itself.

The authenticator data here is the same authenticator data that is returned when getting a regular signature (see page 39) except that it will have the AT flag set, indicating that "ATtested credential data" follows the signature counter. This new data has the following format:



The AAGUID is a random, 16-byte value that identifies a model of security key and is used when validating the attestation. The variable-length credential ID is as explained in chapter 2, but the public key is in the obscure COSE format. (See chapter 13 for details of public key formats and how to convert between them.) There's no length prefix for the public key and the attested credential data may be followed by extension data if the ED flag is set. So, in order to find the end of the COSE key and thus the start of the extension data, you have to parse the CBOR map.

When checking an attestation, you also need to extract the `clientDataJSON` field from the `PublicKeyCredential`'s `response` field. This JSON is similar to what you have seen before except that the `type` will now be `webauthn.create`. When not checking attestation there's no point looking at this field because, if an attacker was trying to do something nefarious, they could just update it. But it is covered by the attestation signature so it's meaningful to check it in the same way as when processing a regular credential signature.

U2F attestation

Recall from chapter 2 that when a U2F security key creates a credential, it returns an X.509 certificate and a signature. That certificate contains the attestation public key and the signature is made by the attestation private key.

Any attestations from a U2F security key will appear in WebAuthn with a `fmt` of `fido-u2f` because the platform will convert them. Seeing this format thus indicates that you have to validate a U2F attestation. The `attStmt` in this case will be a CBOR map (in CTAP2 format, as always) with the following keys:

Key	Value
`x5c`	The X.509 certificate as a byte string
`sig`	A P-256 ECDSA signature in ASN.1 DER format

The data signed by the attestation private key for this format is the following, concatenated:

- A zero byte.
- The SHA-256 hash of the RP ID.
- The SHA-256 hash of the `clientDataJSON` value.

- The credential ID.
- The credential public key in uncompressed X9.62 format. (See chapter 13 for details of different formats.)

The X.509 certificate contains the attestation public key, but how do you trust it? One answer is that, if your company is purchasing a large number of security keys to distribute to its employees, it can simply ask the vendor of those security keys for their root certificate. The attestation certificate in `x5c` should be signed by that root certificate to show that it is authentic. The FIDO Alliance also provides a repository of information about certified security keys, indexed by AAGUID, which is covered in the next section.

The U2F protocol, however, doesn't include any AAGUIDs, so the AAGUID for a `fido-u2f` attestation will always be zero and some bespoke configuration will be needed to know the attestation roots for validation. That's fixed with CTAP2 authenticators which use the `packed` attestation format.

Packed attestation

The significant changes with packed attestation are the following:

- The `fmt` is `packed`.
- The `x5c` key in `attStmt` now contains a CBOR array of one or more byte strings. The first is the attestation certificate and the remainder (if any) are additional X.509 certificates that form a certificate chain to the root.
- There is also an `alg` key in the `attStmt` that contains the COSE algorithm ID of the signature scheme used to produce the signature.
- The signed data is now the concatenation of the authenticator data and the SHA-256 hash of the `clientDataJSON`—matching the form used for regular credential signatures.

CTAP2 security keys will produce attestations in this form. Let's do a worked example with one.

After creating a credential with `attestation` set to `direct`, we inspect `response.attestationObject` in the resulting `PublicKeyCredential` and decode it as the CTAP2 subset of CBOR (see chapter 6) to get:

```
{
  "fmt": "packed",
  "attStmt": {
    "alg": -7,  # ECDSA with P-256 and SHA-256
    "sig": h'30460221...',
    "x5c": [h'308202be30...']
  },
  "authData": h'f95bc73828...'
}
```

The authenticator data from `authData` (see page 39) breaks down as:

```
# SHA-256 hash of the RP ID
f95bc73828ee210f9fd3bbe72d97908013b0a3759e9aea3d0ae318766cd2e1ad
# Flags: AT, UV, and UP set.
45
# Signature counter
0000001f
# Attested credential data
#  AAGUID
f8a011f38c0a4d15800617111f9edc7d
#  Credential ID length
0040
#  Credential ID
3429904107e65bf06f19fd8fa55b4bda
04ede99c1a6994c6bc315252cc6940bf
aeb0c7c62dc88214fc52cb7105aa33da
7b480da9012c36853d4179f159c9348c
#  Public key in COSE format
a501020326200121582 0bc767fb6069f
fd51dbd04916030ec23399e72eefab22
352f29906621351dc83122582066c21d
877c48527407f891ba9611ba85eed1b9
b00164daf2f0a67c39038d771f
```

While packed attestation can contain multiple X.509 certificates to form a chain to the root, this attestation only includes a single certificate, which contains the following:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1955003842 (0x7486fdc2)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=Yubico U2F Root CA Serial 457200631
        Validity
            Not Before: Aug  1 00:00:00 2014 GMT
            Not After : Sep  4 00:00:00 2050 GMT
        Subject: C=SE, O=Yubico AB, OU=Authenticator Attestation,
                 CN=Yubico U2F EE Serial 1955003842
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:95:5d:f3:ad:f7:24:7d:31:75:ef:fd:9c:c4:f3:
                    1a:4e:87:8e:ba:e1:81:09:56:61:50:fb:38:8b:2e:
                    5f:65:27:bf:57:40:9a:a5:81:a5:0d:0a:c5:2f:18:
                    44:5c:0a:13:54:8a:13:53:c8:a4:e5:9a:70:4e:52:
                    3b:c0:4d:eb:ed
                ASN1 OID: prime256v1
```

```
            NIST CURVE: P-256
        X509v3 extensions:
            1.3.6.1.4.1.41482.2:
                1.3.6.1.4.1.41482.1.1
            1.3.6.1.4.1.45724.2.1.1:
                ...
            1.3.6.1.4.1.45724.1.1.4:
                .......M........}
            X509v3 Basic Constraints: critical
                CA:FALSE
    Signature Algorithm: sha256WithRSAEncryption
         31:5c:48:80:e6:9a:52:7e:38:66:89:bd:69:fd:0a:a8:6f:49:
         ...
```

The AAGUID (`f8a011f3-8c0a-4d15-8006-17111f9edc7d`) from the authenticator data can be used to look the security key up in the FIDO Alliance's metadata. The set of metadata can be downloaded by fetching `https://mds.fidoalliance.org/`, which results in a signed JWT. (Information about the metadata service can be found at `https://fidoalliance.org/metadata/`[1].)

We can find the record for this security key by running:

```
curl https://mds.fidoalliance.org/ | \
cut -d. -f2 | \
python3 -c "import sys, base64;
sys.stdout.buffer.write(
  base64.urlsafe_b64decode(sys.stdin.read().strip()))" | \
jq '.entries[] |
    select(.aaguid == "f8a011f3-8c0a-4d15-8006-17111f9edc7d")'
```

> While the data is being downloaded using HTTPS, this one-liner doesn't validate the JWT signature. You may wish to do that.

Within the JSON record is an `attestationRootCertificates` field that contains the attestation root certificate for this device. Now we can check that the attestation certificate is valid using standard X.509 libraries. For illustration we'll use the OpenSSL command-line tool:

```
openssl verify -CAfile root-certificate.pem attestation-
certificate.pem
attestation-certificate.pem: OK
```

The public key from the attestation certificate can be used to verify the attestation signature by appending the SHA-256 hash of `clientDataJSON` to the authenticator data. This provides evidence that the COSE-format public key in the authenticator data really was generated inside a security key of the attested make & model. A real

implementation wouldn't use command line tools, of course, and should perform each step of the validation procedures[2] from the specification.

> CTAP2 security keys often also still support the U2F protocol, and will produce `fido-u2f` attestations if used via that protocol. Since early CTAP2 security keys could not create credentials without doing user verification once a PIN had been configured, attempts to create non-discoverable credentials without user verification may still use the U2F protocol even though a security key supports CTAP2.

You might also come across a packed attestation that is missing the `x5c` key completely. That is a variant of packed attestation called a self-attestation.

Self-attestation

Regular credential creation in WebAuthn does not involve the newly created private key signing anything. Thus, when a credential is created, there's no proof that the entity submitting the new credential actually holds the private key, and thus it's possible to submit someone else's public key as your own.

This does not obviously cause any problems in WebAuthn but the standard solution is a "self-signature" and self-attestation provides this. It is a packed attestation, as described above, where there is no `x5c` key in the attestation statement, and the public key used to validate the attestation signature is the newly-created credential public key. The signature shows that the private key was involved in the creation.

However, self-attestations are not really attestations! They claim nothing about how the private key was generated nor where it is stored. They simply reuse some of the mechanisms. Because of this, self-attestations are rarely encountered.

Other attestation formats

There are several other, less common attestation formats. If you are building a WebAuthn deployment and come across an unexpected attestation `fmt`, you may be able to find details in this section[3] of the WebAuthn spec. Hopefully now that you understand the broad shape of attestation, the specification will be easier to understand.

Enterprise attestation

None of the attestations above identify an individual security key, otherwise it would be possible to track a specific security key as it was used on different websites and in apps. However, if you are a company purchasing security keys for your employees, you might legitimately want to be able to individually track these security keys.

There is one straightforward way to do that, which is to pre-create credentials on the security keys before distribution. This is a viable solution in many cases but it does not always work so, for the exceptions, CTAP2 and WebAuthn define a concept of *enterprise attestation*, where the attestation certificate individually identifies a specific security key for inventory tracking purposes.

If this simply replaced the standard attestation certificate, that would cause all of the privacy problems outlined above. So an enterprise attestation certificate is *in addition* to a regular attestation certificate and requests have to be authorized to use it. Security keys with enterprise attestation must also be specially purchased from the vendor.

There are two ways that enterprise attestation can be authorized for a specific request:

1. The platform communicating with the security key can be configured to authorize the use of enterprise attestation for certain relying party IDs.
2. The security key itself can recognize specific relying party IDs and use enterprise attestation when credentials are created for them.

In both cases the WebAuthn `create` request has to request enterprise attestation by setting the `attestation` parameter to `enterprise`. If the security key has an enterprise attestation certificate, and if the request meets at least one of the two requirements above, then the enterprise attestation certificate will be returned (and the corresponding attestation private key will be used to sign the attestation).

Implementation in CTAP2

When a request specifies enterprise attestation, the platform will consider whether its configuration specifies that the relying party ID from the request is authorized to receive it. The mechanism of this configuration is specific to the platform but, for Chrome/Edge, the policy is called `SecurityKeyPermitAttestation`[4].

If, by whatever mechanism the platform uses, the relying party ID is authorized, then the platform will send an extra `ep` parameter with the CTAP2 `authenticatorMakeCredential` command, and will give it the value `2`. This informs the security key that the platform believes that the request is authorized to use any enterprise attestation certificate that the security key may have configured.

Otherwise, the platform sends the value `1`, which informs the security key that enterprise attestation has been requested, but the platform policy doesn't authorize it. Still, the security key itself may recognize the relying party ID and decide to use enterprise attestation.

The security key is always free to decide that it doesn't want to return enterprise attestation for any request, and the feature is disabled by default and must be ex-

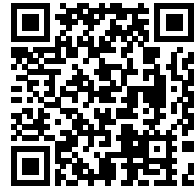
plicitly enabled with an `authenticatorConfig` command after purchase, and after each reset (see page 57). The presence of the `ep` field in the `authenticatorGetInfo` response indicates that enterprise attestation is supported by a security key, and its value indicates whether it is currently enabled.

The enterprise attestation signature signs the authenticator data. When considering the output of extensions (see chapter 9), a server may need to extract extension outputs directly from the authenticator data where possible, rather than use the browser's reflection of them in the `PublicKeyCredential` object, in order to ensure that it's getting authentic extension results. This is most applicable to the `minPinLength` extension (see page 83).

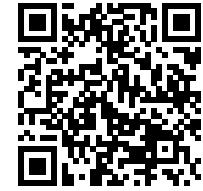
---

1 fidoalliance.org

2 www.w3.org



3 w3c.github.io



4 chromeenterprise.google

CHAPTER 8
WebAuthn on the web

Chapter 4 covered the core of WebAuthn, which is reflected in the platform-specific APIs detailed in chapter 11. However there are several aspects of WebAuthn that are specific to operating on the web. This chapter covers those parts.

Feature detection

Nothing works unless WebAuthn is available in the browser. While WebAuthn is very widely supported now, there are still contexts (such as WebViews) where support might be lacking. To check whether WebAuthn is available, test for the existence of `window.PublicKeyCredential`:

```
if (!window.PublicKeyCredential) {
  // WebAuthn is not available in this context.
}
```

Platform authenticator detection

If WebAuthn is available then sites can silently detect whether a platform authenticator that supports user verification exists:

```
const promise = window.PublicKeyCredential.
  isUserVerifyingPlatformAuthenticatorAvailable();
promise.then((hasUVPlatformAuthenticator) => ..., (error) => ...);
```

Sites would typically check this before proactively prompting users to create a credential. Users may wish to use security keys, so the option to create a credential should still appear in the account settings, but it is unwise to try and upsell the user unless this promise resolves with `true`.

Conditional UI

Hopefully there will come a day when WebAuthn dominates the authentication landscape and sites just have a sign-in button that starts a WebAuthn flow. But today that is not the case.

Today, most users recognize a pair of text boxes as the way that they start a sign-in process, and they may well be used to their password manager auto-filling their username and password into them. Conditional UI is a way for that autofill to also include WebAuthn credentials.

The way it works is that a page can make a `navigator.credentials.get` call and pass `mediation: "conditional"` in the top-level dictionary. (So at the same level

as `publicKey`, not inside the assertion options.) That will cause the request not to show a modal UI, but instead the returned promise will hang around, unresolved.

```
const abortController = new AbortController();
const promise = navigator.credentials.get({
  // The `requestOptions` are the same as for a regular request.
  publicKey: requestOptions,
  mediation: 'conditional',
  signal: abortController.signal,
});
// `promise` may not ever resolve but, if it does, it'll return a
// `PublicKeyCredential`, just like a regular `get()` call.
```

The contract with the browser is that it may present WebAuthn as an option to the user in whatever unobtrusive manner that it wishes. Currently, that means that credentials can be offered in autofill for fields that have `webauthn` as the *final* autocomplete token:

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

If the user selects a WebAuthn credential from the browser's autofill menu then any needed user verification will be completed and the promise from the conditional request will be resolved. The Javascript for the page is then responsible for sending the assertion to the server and getting the user signed in.

Before starting a conditional request, check that the browser supports them: (Otherwise the `mediation` parameter will be ignored and the call will trigger a modal UI.)

```
if (PublicKeyCredential.isConditionalMediationAvailable) {
  const promise = PublicKeyCredential.
      isConditionalMediationAvailable();
  promise.then((isSupported) => ..., (error) => ...);
}
```

A conditional request should be made as soon as possible after page load because credentials won't appear in autofill unless a conditional request is pending to receive the resulting credential.

Mixing modal and conditional requests

A page may well use conditional UI and also have a "Sign in with passkey" button that triggers a modal request. However, only one WebAuthn request can be outstanding at any given time. So if the conditional request is hanging, waiting for a possible credential from autofill, trying to make a regular request in response to the user clicking on the button will fail immediately.

Instead, conditional requests need to be aborted before the modal request can be started. In the example above, the conditional request also took a `signal` parameter. This allows an `AbortController`[1] to abort a conditional request by calling

`abortController.abort()` and waiting for the promise from the conditional request to fail. Then another WebAuthn request can be started.

If the modal request fails, then you need to restart the conditional request; otherwise credentials won't appear in autofill after the user clicks the button.

The challenge of challenges

Conditional UI makes generating the `challenge` parameter in the request more, well, challenging. This is also the case with the `preferImmediatelyAvailableCredentials` option to the platform APIs that we'll cover in chapter 11.

Recall that the properties that we want of a challenge are that:

1. It has never been used before, so no previous signature can be reused for the current request.
2. It is unpredictable, so that an attacker with temporary access cannot generate signatures that will be valid in the future.

The ideal way to meet these requirements is to generate a random challenge at the server, record it, and check against it when the signature is received. But conditional requests want to be started as soon as possible after page load. Any delay to fetch a challenge is a problem. This is also the case with mobile apps that want to show the option to sign in as soon as the app is opened.

With conditional UI, the challenge can be dynamically embedded in the page contents to avoid making a separate request for it. But this isn't applicable when opening an app because that is a purely local operation.

Conditional UI challenges also have a second tension (which the app case doesn't) because the signature from a conditional request could come days later. How long does the server have to store session information for the possibility that a page load is still in a tab somewhere, waiting for the user to sign in?

The best answer for these issues is probably a feature that doesn't exist at the time of writing but which has been proposed: a `challengeUrl` parameter as an alternative to the `challenge` parameter. This would specify a URL from which a challenge can be downloaded at the point that one is needed. For conditional UI, the challenge wouldn't be downloaded until the user has selected a credential and the signing operation is ready to happen. For apps, fetching the challenge could happen concurrently with showing UI to the user, thus minimizing latency.

An alternative that is often suggested is encrypted timestamps; i.e. distribute a stateless service around the world (to minimize latency) that encrypts and returns a timestamp on demand to serve as a challenge. When validating signatures, the timestamp can be decrypted and checked to be reasonably recent.

On the plus side, this avoids needing storage and can make latency acceptable. But there are several drawbacks. Firstly, it limits replay but doesn't prevent it. The extent that replays are possible is bounded by how old a received challenge is allowed to be. If it's just a few minutes, perhaps you deem that acceptable, but larger windows are progressively more concerning. Thus any conditional UI requests need to be restarted frequently.

Second, it would also obviously be bad if the encryption key leaked. Public-key cryptography doesn't help here because, in order for future valid challenges to be unpredictable, it mustn't be possible for an attacker to generate a valid encrypted timestamp. Thus public-key cryptography *shouldn't* be used because public keys are assumed to be public, and making non-standard assumptions of cryptosystems is usually disastrous.

Conditional create

If a user is already using a password manager to store a password for a site it would be great if that password manager would just start storing a passkey instead. That's why browsers increasingly support setting `mediation: "conditional"` on `create` calls. This requests that a passkey be automatically created, i.e. without the user having to confirm or present any biometrics.

Before attempting this, check whether the browser supports conditional create calls:

```
if (window.PublicKeyCredential &&
    window.PublicKeyCredential.getClientCapabilities) {
  window.PublicKeyCredential.getClientCapabilities().then(
    (capabilities) => {
      if (capabilities.conditionalCreate ?? false) {
        attemptConditionalCreate();
      }});
}
```

Since conditional creation doesn't involve any confirmation UI, the user presence and user verification bits in the resulting authenticator data will both be false. Thus, the request should set `authenticatorSelection.userVerification` to `discouraged`.

A site can technically attempt to conditionally create a passkey at any time. However, passkey providers will impose requirements before doing so. They will likely require that they already have an account with the same username to confirm that the user is happy storing that credential. They may also require that the password has been recently filled to try and confirm that any saved password is valid. Because of these requirements, the correct time to attempt a conditional create is immediately after a user has signed in using a password.

If the conditional creation attempt is rejected, the promise will fail immediately with a `NotAllowedError`.

(Conditional creation may also be available in platform APIs. See chapter 11.)

iframes

WebAuthn works without fuss in iframes that are same-origin with the main frame. But when people ask about iframes, they usually mean cross-origin iframes. WebAuthn `get` requests do work in cross-origin iframes, but the parent frame has to grant permission for the iframe to make that call. To do so, use the permissions policy framework:

```
<iframe src="..." allow="publickey-credentials-get">
```

Chromium-based browsers also allow `create` calls within cross-origin iframes. Again, the parent frame has to grant permission, this time with the `publickey-credentials-create` permission. At the time of writing, Safari does not allow this, however.

JSON conversion

Ideally, WebAuthn requests would be created in your backend and sent to the frontend to be performed by the browser. But because WebAuthn requests and responses contain ArrayBuffers, which can't be expressed in JSON, this is not as easy as it should be.

Thus dedicated JSON conversion functions were added to WebAuthn. These functions are aware of the WebAuthn structures and, wherever an ArrayBuffer is needed, accept a base64url-encoded string instead.

Here's an example WebAuthn registration converted to JSON by taking all ArrayBuffers and base64url encoding them:

```
createJSON = `
{
  "challenge": "cmFuZG9tIGNoYWxsZW5nZQ",
  "rp": { "id": "example.com", "name": "example.com" },
  "user": {
    "id": "dXNlcmlk",
    "name": "name",
    "displayName": "displayName"
  },
  "pubKeyCredParams": [ {"type": "public-key", "alg": -7} ],
  "authenticatorSelection": {
    "requireResidentKey": true,
    "authenticatorAttachment": "platform"
  }
}`;
```

A browser with support for these JSON functions will accept the following to trigger a creation request:

```
navigator.credentials.create({
  publicKey: window.PublicKeyCredential.parseCreationOptionsFromJSON(
      JSON.parse(createJSON)),
}).then(console.log, console.log);
```

The static `parseCreationOptionsFromJSON` method on `window.PublicKeyCredential` takes a parsed JSON object and converts it to a WebAuthn creation request. Similarly, `PublicKeyCredential.parseRequestOptionsFromJSON` also exists for assertion requests.

Once a promise has resolved with a `PublicKeyCredential` object, there's also a function to convert it to JSON for sending back to the server:

```
JSON.stringify(pubKeyCred.toJSON());
```

That results in JSON string where, again, all ArrayBuffers have been encoded as base64url strings.

To test whether a browser supports these functions, do:

```
if (window.PublicKeyCredential.parseCreationOptionsFromJSON) {
  // JSON functions supported.
}
```

These JSON formats are also used by the Android platform APIs (see page 96) and so it's possible for a backend to generate WebAuthn JSON objects that will transparently work for both web and Android clients.

Testing

WebAuthn involves interacting with security keys or local biometric sensors. These are all things that make automated testing very challenging. However, browsers can implement a virtual authenticator—a fake WebAuthn authenticator that skips showing any UI, but which creates credentials and generates signatures like any other.

Chromium-based browsers can add a virtual authenticator to a tab for when different types of security keys aren't available for manual testing, or for when a platform authenticator isn't available (although that's rare these days). To do so, find "WebAuthn" under "More tools" in the developer tools, check the option to enable the environment, select the configuration of the virtual authenticator, and click "Add". WebAuthn requests will now use the virtual authenticator and the state of the authenticator appears in the developer tools pane.

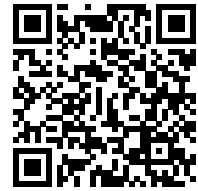
It's also possible to do completely automated tests with a virtual authenticator by configuring it using WebDriver[2]. See the WebDriver section[3] of the WebAuthn specification for more details.

---

1 developer.mozilla.org

2 developer.mozilla.org

3 www.w3.org



CHAPTER 9
Extensions

Recall from chapter 4 that WebAuthn accepts an `extensions` parameter when creating or asserting a credential. These parameters are dictionaries mapping extension names to extension-specific inputs. When an operation completes successfully, the resulting `PublicKeyCredential` has a `getClientExtensionResults` method that returns a dictionary mapping those same names to the extension's outputs.

This extension mechanism allows a number of non-core features to be cleanly added to WebAuthn and several of them are covered in this chapter, along with descriptions of how they are implemented for security keys.

To save reading this whole chapter just to discover whether any of these extensions are useful to you, here's a quick summary of what each does:

Extension	Use
credProps	Learning whether a newly created credential is discoverable or not
PRF	Getting secret keys for encrypting data
credProtect	Setting a minimum security bar for credentials kept on security keys
credBlob	Storing 32 bytes of data with a credential
minPinLength	Checking that company PIN-length requirements are being enforced
largeBlob	Storing certificates for offline operation of a security key
appId / appIdExclude	Backwards compatibility with credentials created via the U2F web API

Extensions are processed by the platform or browser and some of them, like `credProps` don't involve any explicit work by the authenticator or security key. For others, the extension is forwarded entirely to the authenticator for processing. There is also a third class of extensions where the authenticator is involved in the processing of the extension, but the platform also has to do work.

In all cases, the results are returned via `getClientExtensionResults`. Even for extensions that are entirely processed by the authenticator, the platform will generally take the authenticator's output and rewrite it into a JavaScript object to include there.

If you are using attestation (see chapter 7) then you need to consider that the platform's output is not signed by the attestation private key. So for each extension used, you need to decide whether your implementation will extract the extension output from the returned authenticator data instead—which is signed. This is not always possible, for example with the `prf` and `largeBlob` extensions, but sometimes it's nonsensical not to, for example with the `minPinLength` extension.

> Attestation is not commonly used and most implementations do not need to worry about this.

credProps

Recall from chapter 4 that the `authenticatorSelection` field of the creation parameters lets you specify `residentKey` as `preferred`. (And recall that a "resident key" is a historical name for a discoverable credential.)

If you configure that, `credProps` is the way that you learn whether a discoverable credential was actually created or not. To request this information, set this extension in the creation options:

```
options.extensions = {credProps: {}};
```

Then there are three possible outcomes in a successful response:

- The extension isn't supported: the credential probably isn't discoverable if it was created on a security key, but we don't know.
- The extension is supported, but the discoverability isn't reported. (With some security keys the platform can't know whether a credential is discoverable or not. Rather than guess, it'll say nothing at all.)
- The credential explicitly is (or isn't) discoverable.

These possibilities can be extracted from the resulting `PublicKeyCredential` object like this:

```
const extensionOutputs = credential.getClientExtensionResults();
const hasCredProps = 'credProps' in extensionOutputs;
if (!hasCredProps) {
  // Platform doesn't support the extension.
  return "probablynot";
}
const propertyReported = 'rk' in extensionOutputs.credProps;
```

```
if (!propertyReported) {
  // The platform doesn't know whether the authenticator used creates
  // discoverable credentials or not.
  return "probablynot";
}
return extensionOutputs.credProps.rk ? "yes" : "no";
```

PRF

A pseudo-random function (PRF) is a cryptographic abstraction that approximates a random oracle. A random oracle is a function that takes an arbitrary byte-string input and produces a fixed-sized output. It works like this:

Conceptually, the oracle contains a table mapping inputs to outputs that starts off empty. Each time the oracle is evaluated, it looks in its table to see whether the input has been seen before. If so, it returns the corresponding output from its table. Otherwise, it generates an output uniformly at random, records it in its table, and returns it.

As an example, we start off with an oracle with an empty table, and we evaluate it on the input “apples”. The table is empty, so “apples” isn’t in it, and so the oracle generates a random output, records it, and returns it.

Next, we evaluate it on the input “bananas” and the same thing happens. (Although the output, being randomly generated, will be different with very high probability.)

Finally, we evaluate it on “apples” again. This time “apples” is in its table, and it so returns the same output as it did the first time we evaluated it.

If you built such a function, that would be an ideal random oracle. But storing the table is problematic so, instead, PRFs approximate a random oracle by using functions like HMAC-SHA256. They are computationally indistinguishable from a random oracle but, by using a hash function, do not require storing all the input and output pairs.

The result of all this theory is that attaching a PRF to a credential allows you to derive an unlimited number of secret keys from it. These secret keys can be used for whatever you want, but encrypting data is the most common use.

The `prf` extension to WebAuthn lets you attach a PRF to each credential. The PRF is credential-specific and can only be evaluated when the credential is created or asserted. During each operation the PRF can be evaluated on up to two inputs in order to support key rotation.

To use the `prf` extension with a credential, it should be requested at creation time. (It is possible for authenticators to support PRF evaluation at assertion time even

when it wasn't configured at creation time, but this is authenticator-specific behavior and can't be depended upon.)

To request PRF support during credential creation, just create an empty extension:

```
options.extensions = {prf: {}};
```

In the resulting `PublicKeyCredential`, see whether PRF was supported:

```
const extensionOutputs = credential.getClientExtensionResults();
const hasPrf = 'prf' in extensionOutputs &&
               extensionOutputs.prf.enabled === true;
```

Unlike capabilities such as user verification, there is no way to express that the PRF extension is required, so the possibility that an authenticator doesn't support it must always be handled.

It's also possible to opportunistically evaluate the newly-created PRF during the creation operation. (Because security keys can't support this, this is not guaranteed to succeed.)

```
options.extensions = {prf: {
  eval: {
    first: new Uint8Array([1,2,3,4]),
    second: new Uint8Array([5,6,7,8]),
  },
}};
```

The PRF can be evaluated at two inputs, as shown above, but `second` is optional if you don't need that ability.

The outputs will always be 32 bytes long and are in the `results` member of the extension outputs if supported:

```
const extensionOutputs = credential.getClientExtensionResults();
const hasPrf = 'prf' in extensionOutputs &&
               extensionOutputs.prf.enabled === true;
const hasOutputs = hasPrf && 'results' in extensionOutputs.prf;
if (hasOutputs) {
  const output1 = extensionOutputs.prf.results.first;
  const output2 = extensionOutputs.prf.results.second;
}
```

The same code works at assertion time except that there will always be PRF results if `hasPrf` was true at creation time.

But at assertion time, there might be multiple credential IDs listed in `allowedCredentials` and the PRF of each may need to be evaluated at different inputs depending on which credential was used. If that's the case, `evalByCredential` can be set instead of `eval`.

```
options.extensions = {prf: {
  evalByCredential: {
    "Y3JlZGVudGlhbElEMQ": {
      first: new Uint8Array([1,2,3,4]),
      second: new Uint8Array([5,6,7,8]),
    },
    "b3RoZXJJRA": {
        first: new Uint8Array([4,3,2,1]),
        second: new Uint8Array([8,7,6,5]),
    },
}};
```

The keys in `evalByCredential` are base64url-encoded credential IDs. It can only be used if `allowCredentials` is non-empty and every credential ID listed in `evalByCredential` must be present in `allowCredentials`. If both `evalByCredential` and `eval` are both set then the former will be used for any credential ID listed in it, with `eval` used as the fallback for any other credentials.

Choosing the inputs

As a general rule, one shouldn't use the same secret key for multiple purposes. So, if you're using a PRF output to encrypt data with AES-GCM, and then you later switch to a different algorithm, you should use a different key obtained from a different PRF input.

However, that doesn't exclude the possibility that a service uses the same PRF input globally for all users. For example, you could always use `new TextEncoder().encode("user data encryption key").buffer` as the sole PRF input.

A worry with that design would be that, if an attacker were ever to be able to request an assertion with your RP ID, then they could get that secret key for a specific credential. Thus a step up from that design is to have random, per-user PRF inputs that an attacker would struggle to obtain: even if the attacker were able to request an assertion somehow, they would not know what input to use to get the secret key.

A further step up is to be continually rotating the secret key, which is why it's possible to evaluate the PRF at two different inputs each time.

In this design, in addition to a per-user PRF input, each account also has a second, random PRF input that is "pending". The PRF is then evaluated for both inputs and the data can be decrypted with the secret key from the first evaluation and then re-encrypted with the secret key from the second. Then the second PRF input becomes the primary one and the server generates a new "pending" input for the next rotation.

In this case, even if the server's database of PRF inputs leaked, it would rapidly and automatically become out of date.

The decision of which style of PRF evaluation makes sense for a given service has to be made in light of the security needs of that service. Each additional step adds a meaningful amount of complexity and so there's no uniformly applicable advice about which is appropriate.

Input hashing

PRF inputs from WebAuthn, and WebAuthn-like APIs, are prefixed with the string `"WebAuthn PRF"`, followed by a zero byte, and hashed with SHA-256 before being used. This means that there are two layers of "access" to the PRF. An API that allows PRF inputs to be specified without hashing has more authority than WebAuthn, because it can evaluate the PRF at all the inputs that WebAuthn can, *plus* it can evaluate the PRF at inputs that WebAuthn cannot express.

PRF inputs sent over CTAP2 and hybrid (the protocol used between computers and phones) are already hashed. So applications that have direct CTAP2 access can choose to use PRF inputs that are inexpressible in WebAuthn. But this also causes problems when CTAP2 requests are sent over hybrid and need to be translated back into WebAuthn-like requests, because the hashing of PRF inputs cannot be undone. Thus credential providers on Android might see a `prfAlreadyHashed` extension. See page 90.

Implementation in CTAP2

The `prf` extension is implemented for security keys by using a different extension: `hmac-secret`; security keys don't implement `prf` directly.

When creating a credential, the `hmac-secret` extension only takes a boolean parameter to specify whether HMAC support is requested or not. Because of this it's not possible to evaluate any PRFs at creation time when using a security key, and that's why creation time evaluation is optional in WebAuthn.

At assertion time, in order to protect data when it is on the USB bus, or transmitted to an NFC security key, the PRF inputs and results are encrypted using the same Elliptic Curve Diffie–Hellman derived key as described in the section on user verification. (See page 55.)

Those `hmac-secret` inputs simply specify the input(s) for evaluation, i.e. there is no equivalent of WebAuthn's `evalByCredential` field. So to implement `evalByCredential` with security keys, platforms have to probe the possible credentials from `allowCredentials` to find a match, and only then select the PRF inputs for the final `authenticatorGetAssertion` command.

The `hmac-secret` extension also defines that *two* PRFs are created per credential. Which PRF is used is determined automatically based on whether user verification was done for a request. These dual PRFs are not exposed through WebAuthn, however. Rather, WebAuthn defines that when two PRFs are present in an authenticator, the user verification PRF must always be used. Thus, if a WebAuthn request includes PRF evaluation, user verification will be done with security keys if they support it, irrespective of the `userVerification` parameter in the WebAuthn request itself.

Because the `prf` extension is implemented this way on security keys, you're actually able to see `hmac-secret` appearing in the authenticator data when getting an assertion. We can take a look at the authenticator data from a security key operation after requesting a PRF evaluation:

```
# Hash of RP ID
26bd7278be463761f1faa1b10ab4c4f82670269c410c726a1fd6e05855e19b46
# Flags: ED + UV + UP
85
# Signature counter
00000023
# Because the ED flag is set, the remainder is extension data.
# This is CBOR-encoded.
a16b686d61632d7365637265745840a2
0f1e5cd69d341c5e98fe1f2e90834a10
d1df55f835f45e69f2e53650bc3c579a
08d3919308582656a4658c876d1521f6
d703a63a55db81ad9c64b94808a454
```

The CBOR-encoded extension data decodes as `{"hmac-secret": h'A20F1E5CD69D341C5E9...'}`. The payload is 64 bytes long because two PRF inputs were sent, and two 32-byte outputs thus require 64 bytes. The result is encrypted with an ephemeral key that isn't accessible outside of the platform/browser, so neither Javascript nor the server can learn anything about the PRF results from this. It's mentioned here to connect different concepts rather than for any practical use.

credProtect

Security keys can get lost. If someone picks up your lost security key in the car park, how much can they learn about you?

They have physical access to the security key, so they can send `authenticatorGetAssertion` commands to it and query for discoverable credentials associated with any RP ID. Should security keys better protect this information?

Perhaps, but maybe you also want to use a security key like an access card, where you tap it on an NFC reader and a door unlocks for you. That inherently requires that the security key disclose the existence of credentials to any NFC reader that gets close to it.

The `credProtect` extension exists to let the privacy level of a credential be specified at creation time. It only applies to security keys, since phones and laptops have screen locks to protect the information on them already.

> Security keys may disclose the existence of a discoverable credential, its credential ID, and its user ID without user verification being performed. But they *never* disclose the user name or user display name without it. That's why WebAuthn says that the user ID should not contain identifiable information.

There are three levels of credential protection:

- "userVerificationOptional": a credential's existence may be disclosed.
- "userVerificationOptionalWithCredentialIDList": a credential's existence may be disclosed only if user verification is done, or if a request specifies its credential ID.
- "userVerificationRequired": a credential's existence may not be disclosed without user verification.

If you want to use a credential like an access card then it must use "userVerificationOptional". Otherwise, you might want a more restrictive value. Security keys can enforce a higher level of privacy for all credentials, but most don't.

Tucking this away into an extension is a little obscure and so some platforms will set a higher default. Chromium (and thus Chrome and Edge) will default to at least "userVerificationOptionalWithCredentialIDList" whenever a WebAuthn creation request requires or prefers a discoverable credential.

Additionally, if a discoverable credential is required, and `userVerification` is set to `preferred` (which is the default in WebAuthn), then Chromium will set the credProtect level to "userVerificationRequired". This protects users when a site accepts both user-verified and non-verified assertions. That's a reasonable thing to do in the context of a platform authenticator where access is protected by a screen lock, even if user verification isn't done for a specific assertion. But with a security key, a discoverable credential with optional user verification may mean that finding a lost security key grants immediate access to the owner's accounts.

We can observe this happening because the security key will echo the `credProtect` setting in the authenticator data. When inspecting the authenticator data from a creation request made in Edge with `requireResidentKey` set, we see the ED (Extension Data) flag set. After skipping over the attested credential data, the extensions decode as the following CBOR map: `{"credProtect": 3}`. So Edge has set this extension for us, based on the parameters. The value "3" corresponds to "userVerificationRequired" as, over CTAP2, the three protection levels are simply numbered 1, 2, and 3.

A WebAuthn request can overwrite these defaults if it wishes:

```
options.extensions = {
  credentialProtectionPolicy: "userVerificationOptional",
  enforceCredentialProtectionPolicy: false,
}
```

The `enforceCredentialProtectionPolicy` can be set to `true` to require that a credential *only* be created if the specified protection policy can be implemented. So it excludes security keys that don't implement the `credProtect` extension. (For this purpose, platform authenticators are always considered to be sufficiently privacy-preserving.)

After setting the extensions shown above, and doing a creation in Edge, the resulting extensions in the authenticator data decode as `{"credProtect": 1}`. So indeed, the explicit extension overrode the defaults.

credBlob

The `credBlob` extension allows at least 32 bytes of arbitrary data to be stored with a credential. The data is set once, when the credential is created, and can be read when it's asserted. At the time of writing no platform authenticators implement this, only some security keys.

To attempt to set a blob, just set the extension to an `ArrayBuffer` containing the blob's data:

```
options.extensions = {
  credBlob: new TextEncoder().encode("credBlob contents").buffer,
}
```

> If the contents of the blob are sensitive, combine this extension with `credProtect` to ensure that user verification is required for the credential to be asserted.

Just because this extension was set doesn't mean that the security key supports storing a blob, so the extension results have to be checked after a successful creation to see whether the blob was stored:

```
const extensionOutputs = credential.getClientExtensionResults();
const credBlobStored =
      'credBlob' in extensionOutputs &&
      extensionOutputs.credBlob === true;
```

Later, when asserting the credential, the credBlob data can be requested:

```
options.extensions = { getCredBlob: true };
```

And the contents, if any, will be found in the extension outputs:

```
const extensionOutputs = credential.getClientExtensionResults();
let credBlob = 'getCredBlob' in extensionOutputs ?
    (new TextDecoder().decode(extensionOutputs.getCredBlob)) :
    undefined;
```

(`TextEncoder` and `TextDecoder` are used here only for illustration. The contents of a credBlob can be arbitrary binary data.)

However, this extension is not suitable for storing secret keys because the contents end up in the authenticator data, which has to be sent to the server for the assertion signature to be validated. If we look at the authenticator data that resulted from the assertion request above, the extensions within decode as `{"credBlob": h'63726564426C6F6220636F6E74656E7473'}`, and that hex string is the UTF-8 encoding of "credBlob contents". Not very secret! To store secret keys, use the PRF extension (see page 76).

minPinLength

This extension requests that a security key report its configured minimum PIN length. In an enterprise environment, this can be used to enforce that a minimum length policy is in effect for all created credentials.

The minimum PIN length can be configured on some security keys using the `authenticatorConfig` CTAP2 command (see page 57). The minimum can be raised but cannot be reduced without resetting the security key, which erases all credentials. So the minimum length reported during credential creation will be in effect for the lifetime of that credential.

To request that the minimum PIN length be reported during a credential creation, just set the `minPinLength` extension:

```
options.extensions = { minPinLength: true };
```

The minimum will only be reported if the RP ID has been previously configured via the `authenticatorConfig` command. The result can be found in the extensions block of the authenticator data under the key `minPinLength`. The result must be taken from the authenticator data because this extension typically has to be combined with attestation (see chapter 7) to meet compliance requirements, and only the authenticator data is signed by the attestation private key.

largeBlob

Sometimes the entity that creates a credential is not the entity that verifies assertions from it, and communication between those two entities may be difficult or impossible. The classic case for this are air-gapped systems, where authority to ac-

cess them may be issued centrally, but has to be checked by a system that cannot communicate with that central authority.

Typically the central authority will sign a public key with a certificate. The certificate specifies what the holder of the corresponding private key is permitted to access. Such a private key can be stored in a security key, but where are we going to put the certificate? That's what the largeBlob extension is for.

Keep in mind that while this extension does have the word "large" in the name, security keys are embedded devices and the term is relative. The amount of storage available on a security key that supports this extension is only guaranteed to be 1024 bytes.

A largeBlob can only be read and written during assertions, meaning that it cannot be written during creation. But you don't get the public key to put in the certificate until the creation is complete, so that shouldn't be a constraint.

You won't see `largeBlob` appearing in the authenticator data extensions. Given the size, the implementation in security keys is a bit more complex than that.

Security keys that support largeBlob expose a single storage extent to the platform. That storage can only be read or written completely, albeit in a streaming fashion to satisfy message buffer limits. The largeBlob values for every credential on the security key have to fit within that storage and the platforms perform a read/update/write pattern in order to update any part of it.

The storage extent is formatted as a CBOR array by the platforms. Each element of the array is a CBOR map that contains a compressed and encrypted largeBlob value. The encryption key is returned by the authenticator when a credential is asserted, thus, while platforms can read the security key's storage at will, they cannot learn the contents of a largeBlob without its corresponding encryption key, which requires them to have successfully asserted the credential. If `credProtect` is set on the credential (see page 80) then the platform can be required to know the credential ID, or to complete user verification, before that occurs.

The largeBlob storage does not include any credential IDs because doing so would render credProtect's "userVerificationOptionalWithCredentialIDList" policy moot. Instead platforms trial-decrypt each element of the CBOR array in order to find the correct largeBlob entry for a credential once they have the key.

Platforms can, however, always see the number of largeBlobs stored by a security key, and their compressed and uncompressed sizes.

appId / appIdExclude

Before WebAuthn existed there was a U2F web API. Firefox supported it and Chrome shipped a hidden, internal extension by which it could be polyfilled. It has

long been deprecated but had meaningful usage while it existed. It had an equivalent of relying party IDs called AppIDs, but AppIDs were origins, rather than domain names.

When WebAuthn was introduced, sites that had used this U2F API faced a problem. All the credentials that they had registered were associated with AppIDs, like `https://example.com`. But WebAuthn used RP IDs, like `example.com`. Those two strings are different and so security keys would consider them distinct, forever making all AppID-based credentials inaccessible to WebAuthn. That would block a transition to WebAuthn.

The `appId` extension exists so that sites can express that they may also have credentials registered to an AppID. The value of the extension is the AppID that they used, and platforms have to validate that they can request that AppID, similar to how they have to validate RP IDs.

When doing an assertion that contains an `appId` extension, platforms will check for any credentials that match an ID from the `allowCredentials` list, and then check all those IDs again using the given AppID.

This gets complex when doing user verification with a PUAT (see chapter 6) because using a PUAT will bind it to the RP ID or AppID requested. Naively, platforms would have to request two PUATs, and get the user to complete user verification twice. However, any credential created with an AppID must have been created with the old U2F web API. That API didn't support user verification and didn't support `credProtect`, therefore any credential with an AppID can be probed without user verification. So platforms actually probe for each credential ID with the AppID first, and then probe for each ID again with an RP ID and with the PUAT.

The `appIdExclude` extension has the same shape but applies at creation time. It specifies that one or more of the credential IDs in `excludeCredentials` may have been created with an AppID rather than an RP ID. The processing complexity is similar.

At this time, these extensions are only of historical interest because they are only useful if a site needs compatibility with credentials created with the old U2F web API. The set of sites that ever used that API is small, and no new site will ever have had an AppID. But these extensions are mentioned here because they appear in modern API references, and so they should be explained.

CHAPTER 10
Hybrid transport

Security keys are great, but realistically regular people are not going to carry one around with them. But people do carry around their phones. Wouldn't it be great if a phone could work like a security key?

That's what the *hybrid transport* is for. It allows phones or tablets to communicate with laptops and desktops in a way that shows that they are physically proximal. The evidence of proximity is the key factor here: there are plenty of "sign in by scanning this QR code" schemes in use today that simply work across the internet, and they're phishable. An attacker can request a sign-in QR code from the legitimate site, display it on their own site and, if they can persuade a user to scan it with their phone, that user will be authorizing *the attacker's computer*.

In contrast, a security key requires that a computer be physically connected to its USB connector, or within NFC range, thus no remote attack is possible. (Unless the computer itself is compromised.)

Sadly, NFC is rare on laptops, but Bluetooth support is common. Could we have the phone and laptop communicate via Bluetooth to ensure proximity? Unfortunately this didn't turn out to be practical. Measurements of this scheme in the real world showed an unacceptably high rate of Bluetooth communication failures. But broadcasting a single Bluetooth message from the phone to establish proximity, and running the rest of the communication over the internet, did work acceptably well. This scheme was called "hybrid[1]" (Bluetooth + internet).

> During development, this transport was called caBLE, for Cloud Assisted Bluetooth Low Energy. That was a cute name, but calling a wireless protocol "caBLE" caused some confusion and thus it was changed. But you might still see the old name hanging around as caBLE reached version 2.1 before being renamed.

How it works

The device making a WebAuthn request, usually a laptop, will display a QR code if it wants to use the hybrid transport. That QR code contains a CBOR map that contains the following values:

- A public key for the laptop.
- A secret key, so that the phone can prove that it has seen the QR code.

- Whether the request is to make a credential, or to get a signature, so that the phone can show a specific message when scanning the QR code.

If the user scans the QR code and triggers the operation on the phone, then the phone connects to its *tunnel service*. This is some service on the internet that is willing to relay messages between laptops and phones as part of this protocol. It's run by whomever is implementing the phone side of the protocol, so iPhones use a tunnel service operated by Apple and Android phones (at least if they're using Google's Play Services) use one operated by Google. The phone asks the tunnel service to wait for a connection from the laptop which will be identified by a long, random ID.

Then the phone starts broadcasting a Bluetooth Low Energy (BLE) advert to tell the laptop that it's ready. Adverts are small messages used by BLE devices to advertise their supported services. Importantly, they can be sent and received without needing to do a Bluetooth pairing between the devices. The BLE advert sent by the phone advertises a service number assigned to the FIDO Alliance (0xfff9) and the advert can include a small (20-byte), service-specific payload.

Twenty bytes is not a lot! For this protocol it's split into 16 bytes of encrypted message and four bytes of authentication tag. Both are keyed based on the secret key that was in the QR code and thus is shared by the two devices. The authentication tag means that two different hybrid transactions happening within BLE range of each other are very unlikely to interfere because the tag will only be valid for one of them.

As soon as the laptop displays a QR code, it starts listening for a matching BLE advert. When it receives one with a correct authentication tag, it decrypts it. The resulting 16 bytes specify the domain of the phone's tunnel server and the laptop connects to it. By knowing the contents of the QR code and the BLE advert, the laptop can calculate the random ID that the phone told the tunnel server to expect and now the tunnel server can relay messages between the two devices.

> A domain name often won't fit in a 16-byte message and so the tunnel server's domain isn't included directly. Rather a 16-bit field is used. The first 256 values specify pre-defined tunnel server domains (of which only two values have been defined so far). The remaining values are hashed to generate a random-looking domain name. So, to set up your own tunnel service, either get its name into the FIDO Alliance specification as a pre-defined name, or else register one of the random-looking domains generated by the hash function.

We don't want the tunnel server to be able to see the contents of any of the messages, thus the two devices run a cryptographic handshake. During this protocol

the laptop proves to the phone that it holds the private key corresponding to the public key in the QR code, and that it received the BLE advert and thus is in Bluetooth range. Also, the phone proves to the laptop that it knows the secret key from the QR code.

The two devices can now exchange encrypted messages. The phone is convinced that the device on the other end of the connection received a BLE message that it broadcast, and thus is physically close. CTAP2 is used between the two devices and, in order to save a round-trip, the phone preemptively sends the result of the `authenticatorGetInfo` command (see page 52) so that the laptop can immediately send an `authenticatorMakeCredential` or `authenticatorGetAssertion` command. The messages are padded up to the next multiple of 32 bytes to reduce the amount that a tunnel server could learn from seeing the lengths of the messages exchanged.

Threats

This protocol is a practical way for a phone to act as an authenticator, but it's worth knowing about the limits of its threat model.

The protocol ensures that the device connected to the phone was able to learn the contents of a BLE advert that the phone broadcast. So one option for an attacker is to have a BLE receiver with internet access physically close to the victim, for example hidden in a busy coffee shop. The attacker can then email a QR code to the victim and, if the victim scans it, the BLE receiver can play the part of the laptop, proxy a challenge from an important site, and send a signature request to the phone. Hopefully the victim notices the UI on their phone and declines to authorize the operation, but some small fraction of people may fall for it.

Simpler still, an attacker could hide a BLE receiver somewhere near a poster with a QR code claiming "Free WiFi!" Again, the BLE receiver would need internet access, but could proxy a challenge and hope that the user ignores all the messages and signs in.

In both cases, from the phone's perspective the transaction is indistinguishable from a legitimate sign-in request. If these attacks start happening in the real world, there are a couple of defenses. Firstly, the messaging on the phone can be sharpened to make people less likely to misunderstand what they're authorizing. Secondly, the QR code already includes a timestamp and phones could start enforcing that it's current. This will have false negatives because clocks are not always accurate, but it would force attackers to produce a fresh QR code—a "Free WiFi" poster would no longer suffice.

While the hybrid transport isn't perfect, these attacks are far more difficult and far less scalable than phishing attacks.

Skipping the QR code

There is another part of this protocol that allows scanning the QR code to be skipped if the phone and laptop have interacted previously.

During a QR-initiated connection, the phone can optionally send information to the laptop that will allow the laptop to contact the phone again in the future. This includes a public key for the phone and an identifier that the laptop can send to the phone's tunnel server in order to request a connection to that phone.

> Since the phone picks which tunnel server will be used, we can assume that the tunnel server knows some private way to contact the phone. So these connections are always triggered by the laptop connecting to the phone's tunnel server. They are not triggered by the laptop broadcasting any kind of Bluetooth message.

Later, when the laptop wants to contact the phone again, it connects to the tunnel server and sends the phone's identifier. Using that, the tunnel server figures out which phone to contact and establishes a connection with it, forwarding a small message from the laptop as it does so. But, even though there is a pre-established relationship between the phone and laptop, we always want to establish proximity for security reasons. So the phone starts broadcasting a BLE advert.

Both parties to the transaction already have a connection to the tunnel server, so the BLE advert doesn't need to include the tunnel server's domain again, it just needs to include random data so that it's unpredictable. Once the laptop receives the BLE advert, it starts the cryptographic handshake over the already established tunnel and proves receipt to the phone. Now the devices once again have an encrypted tunnel established between them and no QR scanning was necessary.

CTAP2 changes

While CTAP2 is used over the encrypted tunnel, it is a slight variant of the protocol. As discussed in chapter 6, a single WebAuthn request may generate many CTAP2 commands: in order to respect the memory limits of security keys, lists of credential IDs might need to be batched, and extensions such as largeBlob are implemented using a sequence of commands to manage the storage on the security key.

But smartphones do not have the tight resource limits of an embedded device. They also don't want to disclose any information without the user authorizing it. So, when working over a hybrid connection, the phone really wants to receive a single CTAP2 command that contains everything.

A phone also doesn't want to produce a PUAT to represent user verification (see page 55) because it wants to display the request to the user before asking the user

for biometrics. At best, the phone would return a dummy PUAT without actually collecting user verification from the user and then do the actual user verification after processing the main command. But all that would consume round trips which, given that a hybrid connection runs over the internet, could cause a significant delay.

Thus the flavor of CTAP2 used over hybrid does not do any PUAT exchange, nor does it do any batching of credential IDs. Also, there is a special flavor of the largeBlob extension used over hybrid which looks just like the WebAuthn extension (but expressed in CBOR), and which avoids the many round-trips used to implement largeBlob on a security key. There is also a special version of the PRF extension, again looking like the WebAuthn version, where the evaluation points for all possible credentials are sent.

The PRF extension has another problem when going over hybrid. If you recall, the evaluation points are hashed before being sent to security keys, so the evaluation points sent over hybrid are already hashed. But when the request is received at the phone, the platform APIs typically expect unhashed inputs and would normally hash them again, which would result in the wrong value. But it is not possible for the phone to unhash the inputs as hash functions are, by their very nature, irreversible.

So, on Android, a synthetic extension called `prfAlreadyHashed` is synthesized for requests received over hybrid, which has the same shape as the regular PRF extension, but where the evaluation points are already hashed.

These tweaks to CTAP2 suggest that it was probably not the correct protocol to use over hybrid. Instead, JSON-encoded WebAuthn requests and responses should have been used. A future revision of the protocol may thus change this.

---

1 fidoalliance.org

CHAPTER 11
Platform APIs

Much of this book focuses on WebAuthn as implemented in browsers—its original context. But Android, iOS, macOS, and Windows all implement WebAuthn-inspired APIs that produce compatible signed messages. These are the *platform APIs*.

Apple platforms

In order to use the WebAuthn-like API on Apple platforms[1], start by importing `AuthenticationServices`. (This book assumes that you're using Swift. If you're using Objective C then everything is the same, modulo syntax, and you're probably very used to translating from Swift at this point.)

Creating credentials

Apple's API is divided between one set of classes for handling security keys and another for handling credentials on the local device or over the hybrid transport. They have overlapping, but distinct, sets of parameters. If you want to handle both in a given request (which will often be the case) then you can pass one request of each type to the ultimate `ASAuthorizationController`.

To create credentials on the local device (or on another phone via scanning a QR code) then start with:

```
let provider = ASAuthorizationPlatformPublicKeyCredentialProvider(
      relyingPartyIdentifier: "example.com")
let request = provider.createCredentialRegistrationRequest(
      challenge: Data([0]),  // fine unless attestation is used
      name: "user.name",
      userID: Data("user.id".utf8))

let controller = ASAuthorizationController.init(
    authorizationRequests: [request])
controller.delegate = self
controller.presentationContextProvider = self
controller.performRequests()
```

(Make sure that you've read chapter 5, have configured the associated domains for your RP ID, and have set the `webcredentials` entitlement for your project.)

The names in the API mirror the WebAuthn parameters and so should be immediately clear. If not, see chapter 4.

There's no `displayName` parameter. Apple platforms only support display names for security key requests and never show them in their UI.

In iOS 18, the `request` object has a `requestStyle` property that can be set to `.conditional` to request that the creation happen silently. This will only succeed soon after the user has filled a password from a password manager into your app, but it lets you easily upgrade users from passwords to passkeys. (See page 70.)

There are also `largeBlob` and `prf` properties to enable support for those extensions. See chapter 9.

You might notice that there's no `excludeList` mentioned so far. The Apple API doesn't support setting an exclude list, except for security key requests. Thus it's not possible to avoid overwriting existing credentials! Perhaps Apple will address this in a future revision but currently you'll need to think carefully about the implications of this.

As a common pattern across all the calls documented here, `controller` is passed two delegates (which can be the same object). The `presentationContextProvider` answers a `presentationAnchor` message which provides the `UIWindow` for the app:

```
class ExamplePresentationDelegateClass: NSObject,
      ASAuthorizationControllerPresentationContextProviding {
  var anchor: ASPresentationAnchor?

  ...

  func presentationAnchor(for controller: ASAuthorizationController)
         -> ASPresentationAnchor {
    return anchor!
  }
}
```

The `delegate` object handles success and failure callbacks from the controller:

```
class ExampleDelegateClass: NSObject,
      ASAuthorizationControllerDelegate {
  ...

  func authorizationController(
    controller: ASAuthorizationController,
    didCompleteWithAuthorization authorization: ASAuthorization
  ) {
      switch authorization.credential {
      case let registration as
           ASAuthorizationPublicKeyCredentialRegistration:
          let credID = registration.credentialID
```

```
            let clientDataJSON = registration.rawClientDataJSON
            let attestationObject = registration.rawAttestationObject!
            Self.logger.log(
                "success: \(credID.base64EncodedString()) \
                (clientDataJSON.base64EncodedString())    \
                (attestationObject.base64EncodedString())")
        default:
            Self.logger.error("unknown ASAuthorization type received")
        }
    }

    func authorizationController(
      controller: ASAuthorizationController,
      didCompleteWithError error: Error
    ) {
      Self.logger.error("failed: \(error)")
    }
}
```

Unfortunately, unlike with WebAuthn and the Android APIs, the authenticator data and public key aren't exposed directly. Instead you must parse the CBOR in the attestation structure and extract the public key and authenticator data from the raw contents. See chapter 7 for details on how to do this, or seek a wrapper library that makes iOS more friendly in this respect.

Let's have a look at the authenticator data returned after creating a credential in iCloud Keychain, the default passkey provider on Apple platforms. We'll break down the hex-encoded data with comments:

```
# The hash of the RP ID
26bd7278be463761f1faa1b10ab4c4f82670269c410c726a1fd6e05855e19b46
# Flags: AT + BE + BS + UV + UP
5d
# Signature counter; always zero
00000000

# Attested credential data
#  The AAGUID of iCloud Keychain
fbfc3007154e4ecc8c0b6e020557d7bd
#  Credential ID length: 20 bytes
0014
#  Credential ID
df46b51df21331fb23bbfa3e9622ae9fc92fc9ea
#  Public key in COSE format.
a501020326200121582071f5ce7ba3e4
4960ddc7f7026e708fc98a835039aa58
97f0d3c80373f3759d542258209e5f26
ec5c054c5841ac4b331bd79b196f006b
7a75e8e5ad585947b4edd3a5ac
```

Since iCloud Keychain syncs credentials, the BE (Backup Eligible) and BS (Backup State) flags are both set. The signature counter is always zero because it would be implausible to synchronize a signature counter between devices. iCloud Keychain sets a distinct AAGUID so that it can be identified in account management UIs. (See page 107.)

To support creating a credential on a security key, add a security key credential provider too:

```
let provider =
ASAuthorizationSecurityKeyPublicKeyCredentialProvider(
  relyingPartyIdentifier: "example.com")
let skRequest = provider.createCredentialRegistrationRequest(
  challenge: Data([0]),
  displayName: "user.displayName",
  name: "user.name",
  userID: Data("user.id".utf8))
skRequest.credentialParameters = [
  ASAuthorizationPublicKeyCredentialParameters.init(
      algorithm: ASCOSEAlgorithmIdentifier.ES256)
]

let controller = ASAuthorizationController.init(
    authorizationRequests: [request, skRequest])
controller.delegate = self
controller.presentationContextProvider = self
controller.performRequests()
```

This time `displayName` and `credentialParameters` are required to be set. Other properties that can be set are `excludedCredentials`, `residentKeyPreference`, `attestationPreference`, and `userVerificationPreference`. Although the names are slightly different, the meaning of all of these is the same as in WebAuthn.

Most of the time you'll want to support creating a credential both locally and on a security key and thus will create requests from *both* types of provider and pass both requests when initializing the `ASAuthorizationController`.

Getting signatures

When requesting a signature, the classes are again split between using security keys or using a local credential / showing a QR code.

```
func assertCredentialOnPlatform(anchor: ASPresentationAnchor) {
    self.authenticationAnchor = anchor
    let provider = ASAuthorizationPlatformPublicKeyCredentialProvider(
      relyingPartyIdentifier: "example.com")
    let request = provider.createCredentialAssertionRequest(
      challenge: Data("SHOULDBERANDOMVALUEFROMSERVER!".utf8))
```

```
    let controller = ASAuthorizationController.init(
        authorizationRequests: [request])
    controller.delegate = self
    controller.presentationContextProvider = self
    controller.performRequests()
  }
```

The properties `allowedCredentials`, `prf`, and `largeBlob` are available on the `request` object, with the same meanings as in WebAuthn. Obviously the challenge value should be random and come from the server—the fixed value used in this code example here is only for illustration and a fixed value should never be used in real code.

The same pair of delegates is used and results are returned via a different subclass of `ASAuthorization`:

```
func authorizationController(
    controller: ASAuthorizationController,
    didCompleteWithAuthorization authorization: ASAuthorization
  ) {
    switch authorization.credential {
    case let assertion as ASAuthorizationPublicKeyCredentialAssertion:
        let credID = assertion.credentialID
        let userID = assertion.userID;
        let clientDataJSON = assertion.rawClientDataJSON
        let signature = assertion.signature
        let authenticatorData = assertion.rawAuthenticatorData
        Self.logger.log(
            "success: \(credID.base64EncodedString()) \
            \(clientDataJSON.base64EncodedString())   \
            \(signature!.base64EncodedString())       \
            \(authenticatorData!.base64EncodedString())"
        )
    default:
      Self.logger.error("unknown ASAuthorization type received")
    }
  }
```

Recall from chapter 5 that, for Apple platforms, the client data's `origin` is always a web origin (taken from the RP ID) even when called from an app. For example, here's the client data JSON from the sample request above:

```
{
  "type": "webauthn.get",
  "challenge": "Y2hhbGxlbmdl",
  "origin": "https://example.com"
}
```

You would have expected the origin to be something like `ios:T7AYYU7S6A.com.-YourApp`, but Apple platforms don't let you distinguish between apps and web origins!

The `performRequests` call also has a useful option to control whether UI is shown or not:

```
controller.performRequests(
    options: .preferImmediatelyAvailableCredentials)
```

When `preferImmediatelyAvailableCredentials` is given, requests that don't have any matching local credentials will fail immediately and no UI will be shown. So this option lets you prompt for a passkey only if one exists. When a request fails for this reason, the error passed to `authorizationController(controller:didCompleteWithError:)` has code `ASAuthorizationErrorCanceled` and domain `ASAuthorizationErrorDomain`. That is the same error that you get if the user declines to use a passkey that does exist, although you can tell the two apart based on how fast the error occurs. (And based on the error message, although you can't assume that will be stable.)

To accept a signature from a security key, as you generally should, a second request object is needed:

```
  self.authenticationAnchor = anchor
  let provider =
  ASAuthorizationSecurityKeyPublicKeyCredentialProvider(
    relyingPartyIdentifier: "example.com")
  let skRequest = provider.createCredentialAssertionRequest(
    challenge: Data("SHOULDBERANDOMVALUEFROMSERVER".utf8))
  // Omit setting this property to request a discoverable credential.
  skRequest.allowedCredentials = [
    ASAuthorizationSecurityKeyPublicKeyCredentialDescriptor(
        credentialID: Data(base64Encoded: "AM==")!,
        transports: ["usb"])
  ]

  let controller = ASAuthorizationController.init(
      authorizationRequests: [request, skRequest])
  controller.delegate = self
  controller.presentationContextProvider = self
  controller.performRequests()
```

Android

To use WebAuthn on Android[2], start by importing these libraries:

```
implementation("androidx.credentials:credentials:1.2.2")
implementation(
  "androidx.credentials:credentials-play-services-auth:1.2.2")
```

(Although keep in mind that there's likely a newer version[3] of them that you should use by the time you're reading this.)

Don't forget to configure the `assetlinks.json` for your RP ID. See chapter 5.

Rather than having an API that mirrors WebAuthn, Android just uses JSON-encoded WebAuthn structures with any `ArrayBuffers` encoded using base64url. This is the same format that `window.PublicKeyCredential.parseCreationOptionsFromJSON` takes on the web, allowing a backend to generate this form of request and have it be easily consumed by web and Android frontends. (See page 71.)

```
val request = CreatePublicKeyCredentialRequest(
        requestJson = """
{
  "challenge": "cmFuZG9tIGNoYWxsZW5nZQ",
  "rp": { "id": "example.com", "name": "example.com" },
  "user": {
    "id": "dXNlcmlk",
    "name": "name",
    "displayName": "displayName"
  },
  "pubKeyCredParams": [ {"type": "public-key", "alg": -7} ],
  "authenticatorSelection": {
    "requireResidentKey": true,
    "authenticatorAttachment": "platform"
  }
}""",
    )

    val credentialManager = CredentialManager.create(requireContext())
    coroutineScope.launch {
        try {
            val result = credentialManager.createCredential(
                context = requireActivity(),
                request = request,
            )
            when (result) {
                is CreatePublicKeyCredentialResponse -> {
                    // This is a JSON-encoded response. See below for
                    // an example.
                    println(result.registrationResponseJson)
                }
                else -> {
                    // Unknown response type.
                }
            }
        } catch (e: CreateCredentialException) {
            when (e) {
                is CreatePublicKeyCredentialDomException -> {
```

```
                    when (e.domError) {
                        is InvalidStateError -> {
                            // Credential already exists
                        }
                        // Other error
                    }
                }
                else -> {
                  // Other error
                }
            }
        }
    }
```

The only non-boilerplate above is the request JSON, and it should be familiar to you from chapter 4. The responses are also just JSON-encoded WebAuthn structures:

```
{
  "rawId": "uGZDSrbiPsPDJ1gv1ebluA",
  "id": "uGZDSrbiPsPDJ1gv1ebluA",
  "authenticatorAttachment": "platform",
  "type": "public-key",
  "response": {
    "clientDataJSON": "eyJ0eXBlIjoi...",
    "attestationObject": "o2NmbXRkbm...",
    "transports": ["internal", "hybrid"],
    "authenticatorData": "Jr1yeL5GN2Hx-qGxCrTE-
CZwJpxBDHJqH9bgWFXhm...",
    "publicKeyAlgorithm": -7,
    "publicKey": "MFkwEwYHKoZ..."
  },
  "clientExtensionResults": {
    "credProps": {"rk": true}
  }
}
```

The first two lines of the JSON might look odd but, if you recall from the WebAuthn chapter, the `id` field of WebAuthn's `PublicKeyCredential` dictionary is the base64url-encoded credential ID, while the `rawId` field is the credential ID as an `ArrayBuffer`. Since, in the JSON form, `ArrayBuffers` are base64url-encoded, the JSON does indeed end up with two copies of the same value!

But otherwise there's nothing new that you need to learn here due to the adherence to regular WebAuthn. Also note that the `getPublicKey`, `getPublicKeyAlgorithm`, and `getAuthenticatorData` helpers in WebAuthn have been turned into fields in the above JSON. So the public key is directly available in the more useful SPKI format.

the JSON result is the same as `PublicKeyCredential.toJSON` will produce on the web.

Long fields have been elided in the example above, but here's the decoded `clientDataJSON`:

```
{
  "type": "webauthn.create",
  "challenge": "cmFuZG9tIGNoYWxsZW5nZQ",
  "origin": "android:apk-key-hash:wGsazqR2MsDW-
DBK0TJQqBlYUK2MD59aPxzt5rl5Bsc",
  "androidPackageName": "com.example.webauthn"
}
```

Note that the origin reflects that the caller was an app, not a website. The app is identified by the hash of the signing certificate but the package name is also available. Your server will need to be updated to recognize the app as legitimate.

(This value after `apk-key-hash` is the same SHA-256 signing-certificate hash as you put in your `assetlinks.json` file, but base64url-encoded, rather than hex-encoded.)

Getting an assertion

Getting an assertion looks very similar, just with some different classes. Again, the request and response are just JSON-encoded WebAuthn structures:

```
val request = GetPublicKeyCredentialOption(
    requestJson = """
{
  "challenge": "cmFuZG9tIGNoYWxsZW5nZQ",
  "rpId": "example.com"
}
""")
val credentialManager = CredentialManager.create(requireContext())
coroutineScope.launch {
    try {
        val result = credentialManager.getCredential(
            context = requireActivity(),
            request = GetCredentialRequest(listOf(request)),
        )
        when (val cred = result.credential) {
            is PublicKeyCredential -> {
                println(cred.authenticationResponseJson)
            }
            else -> {
                // Unknown response type.
            }
        }
    } catch (e: GetCredentialException) {
```

```
            println(e)
        }
    }
```

Just for reference, here's the resulting JSON. Again, none of the fields should be surprising. (The `clientDataJSON` has a similar form to the one shown above.)

```
{
  "rawId":"uGZDSrbiPsPDJ1gv1ebluA",
  "id":"uGZDSrbiPsPDJ1gv1ebluA",
  "authenticatorAttachment":"platform",
  "type":"public-key",
  "response":{
    "clientDataJSON":"eyJ0eXBlIjoid2ViYXV...",
    "authenticatorData":"Jr1yeL5GN2Hx-qGxCrTE-CZw...",
    "signature":"MEUCIQDi0o8OOtUJQDKtFLBMU_Cnuycd...",
    "userHandle":"dXNlcmlk"
  },
  "clientExtensionResults":{}
}
```

Like the Apple platform API, `GetPublicKeyCredentialOption` supports a `preferImmediatelyAvailableCredentials` argument, which will cause the operation to return immediately if there are no local credentials available.

Windows

Although non-browser applications use WebAuthn much less frequently on Windows than on mobile platforms, Windows does provide an API for it. We won't dive into it in detail, but it is well explained in the header file[4] provided by Microsoft.

Unlike the mobile platforms, there is no need to configure any files on your server to authorize the use of an RP ID, as the Windows API trusts applications to assert any RP ID.

Windows also enjoys the most complete security key support of any of the platform APIs, although the platform authenticator, Windows Hello, does not sync credentials at the time of writing.

---

1 developer.apple.com

2 developer.android.com

3 developer.android.com

4 github.com

CHAPTER 12
The server side

This chapter will cover some of the details of a server-side implementation of WebAuthn. This helps you understand what's going on, but doing everything yourself is not necessarily the best choice: when implementing the server side you may well be best served by using an existing library for WebAuthn support. However, I do claim that it's perfectly practical to build support yourself if you wish.

This chapter will assume that you're building a flow based on discoverable credentials because that's the most common option. Some changes are needed if you want to build a purely 2nd-factor flow but, if you've read chapter 4, you should be well positioned to make those tweaks.

First, a brief checklist of basics to take care of:

1. If your website doesn't already use HTTPS, fix that first. WebAuthn only works on secure origins.
2. Pick your RP ID. (See chapter 5.) We'll assume here that the RP ID will be `example.com`.
3. Set up assetlinks and associated domains files to support mobile platforms as needed. (See page 45.)

Presumably you already have a database table of users. Perhaps it contains salted and hashed passwords for authentication, and maybe phone numbers for SMS OTP. Passkeys aren't just another column because, while an account can only have one password, *an account can have multiple passkeys*. Don't make the mistake of limiting accounts to a single passkey!

So you need to have a separate table for passkeys with a foreign-key relation to the primary key used to identify accounts:

```
CREATE TABLE passkeys (
  cred_id BLOB PRIMARY KEY,
  username STRING NOT NULL,
  public_key_spki BLOB,
  backed_up BOOLEAN,
  /* You may also want creation_time, last_used_time, and perhaps
     aaguid columns. */
  FOREIGN KEY(username) REFERENCES users(username));
```

Recall from page 31 that the `user.id` in a credential creation request should be an opaque identifier for an account, and from page 81 that security keys don't con-

sider this value to be sensitive thus it shouldn't be possible to identify the user from this value. You may already have a user ID in your system, but are you sure that it doesn't leak out anywhere else? If so, perhaps it's a reasonable value to use as the user ID. But otherwise it might be safer to generate a fresh identifier for just this purpose:

```
ALTER TABLE users ADD COLUMN passkey_id blob DEFAULT(randomblob(16));

/* The CASE expression causes the function to be non-constant. */
UPDATE users SET passkey_id=hex(randomblob(CASE rowid WHEN 0
                                                      THEN 16
                                                      ELSE 16 END));
```

(This SQL is just an example. You'll need to adjust it for your specific environment.)

These SQL snippets assume that your table of passkeys is keyed by the credential ID, and that your user IDs are random. These are the correct choices for the majority of sites.

However, there are some cases where your users are split up. Perhaps as the result of an acquisition or the merging of disparate systems, you might need to know in which universe an account lives in order to be able to efficiently look it up. But when you get a WebAuthn assertion that contains a credential ID, the ID is random: you don't know from which universe it came.

In this case, you can use structured user ID values. Because an assertion from a discoverable credential also returns the user ID, and that's a value chosen by the server, you can encode whatever universe information you need in it.

We won't develop this possibility any further as it's mentioned only for the handful of people who will find this hint useful.

Enrolling existing users

When a user signs-in with a password, you might want them to create a passkey on the local device for easier sign-in next time. First, check that a local platform authenticator exists and that the browser or mobile platform supports passkeys / conditional UI. (See page 67.)

Next, try conditionally creating a passkey; see page 70. (Conditional creation may also be available via platform APIs.) If this doesn't work, you may want to prompt the user to create a passkey and use the traditional modal UI flow.

In both the conditional and modal cases, the server will need to send creation parameters to the client. Below the parameters are represented as a Javascript object for the purposes of exposition but note that both Android and, increasingly, the web support accepting JSON-encoded requests. (See page 71.) Unfortunately, the iOS API does not accept JSON, and so either your backend will need to produce a

different style of output for any iOS apps, or else you'll need to implement a converter in the app from JSON to the native API on that platform.

```
var createOptions : CredentialCreationOptions = {
  publicKey: {
    rp: {
      // The RP ID.
      id: "example.com",
      // This field is required to be set to something but is not
      // currently used by any implementations.
      name: "",
    },

    user: {
      // `userIdBase64` is the passkey_id field from the users table,
      // base64-encoded.
      id: Uint8Array.from(atob(userIdBase64), c => c.charCodeAt(0)),
      // `username` is the username field from the users table.
      name: username,
      // `displayName` can be a more human name for the user, or
      // just leave it blank.
      displayName: "",
    },

    // This lists the ids of the user's existing credentials. I.e.
    //   SELECT cred_id FROM passkeys WHERE username = ?
    // and supply the resulting list of values, base64-encoded, as
    // existingCredentialIdsBase64 here.
    excludeCredentials: existingCredentialIdsBase64.map(id => {
      return {
        type: "public-key",
        id: Uint8Array.from(atob(id), c => c.charCodeAt(0)),
      };
    }),

    // Boilerplate that advertises support for P-256 ECDSA and RSA
    // PKCS#1v1.5. Supporting these key types results in universal
    // coverage so far.
    pubKeyCredParams: [{
      type: "public-key",
      alg: -7
    }, {
      type: "public-key",
      alg: -257
    }],

    // Unused during registrations, except when doing attestation.
    // (But don't do this during sign-in!)
    challenge: new Uint8Array([0]),
```

```
    authenticatorSelection: {
      authenticatorAttachment: "platform",
      requireResidentKey: true,
    },

    // Five minutes.
    timeout: 300000,
  }
};

navigator.credentials.create(createOptions).then(
  handleCreation, handleCreationError);
```

Recording a passkey

When the promise from `navigator.credentials.create` resolves successfully, you have a newly created passkey! Now you have to ensure that it gets recorded by the server.

The promise will result in a `PublicKeyCredential`[1] object, the `response` field of which is an `AuthenticatorAttestationResponse`[2].

Call `getAuthenticatorData()` and `getPublicKey()` on `response` and send those ArrayBuffers to the server. (These fields also exist in the JSON output from the Android platform APIs but, sadly, not on iOS where they would have to be polyfilled.)

At the server, we want to insert a row into the `passkeys` table for this user. The authenticator data[3] is a fairly simple, binary format (see page 39). Offset 32 contains the flags byte. Sanity check that bit 6 is set and then extract:

1. Bit 4 as the value of `backed_up`. (I.e. `(authData[32] >> 4) & 1`.)
2. The big-endian, uint16 at offset 53 as the length of the credential ID.
3. That many bytes from offset 55 as the value of `id`.

The ArrayBuffer that came from `getPublicKey()` is the value for `public_key_spki`. That should be all the values needed to insert the row.

> Neither the user presence nor user verification bits are checked above. This works for conditional creation (see page 70) but might not be right for every deployment.

Handling a registration exception

The promise from `create()` might also result in an exception. `InvalidStateError` is special and means that a passkey already exists for the local device. This is not an error, and no error will have been shown to the user. They'll have seen a UI just like they were registering a passkey but the server doesn't need to update anything.

`NotAllowedError` means that the user canceled the operation. Other exceptions mean that something more unexpected happened.

The WebAuthn-family APIs on mobile platforms will have similarly structured errors. See chapter 11.

Signing in

See chapter 8 for details on using conditional UI for signing in on the web, and chapter 11 for details of invoking the APIs on mobile platforms to do something similar.

One thing that all of these APIs will need is a challenge value. See page 69 about picking challenge values. Otherwise, assuming that you're using discoverable credentials, there aren't any other inputs. (There couldn't be because, at this point, you don't know who the user is!)

A successful response from all these APIs will include the:

- Credential ID (called the `rawId` in WebAuthn's result).
- Client data JSON (see page 40).
- Authenticator data (see page 39).
- Signature.

Those values should be sent to the server for validation. At the server, first look up the passkey: `SELECT username, public_key_spki, backed_up FROM passkeys WHERE cred_id = ?` and give the credential ID value for matching. The `cred_id` column is a primary key, so there can either be zero or one matching row(s). If there are zero rows then the user is signing in with a passkey that the server doesn't know about—perhaps they deleted it. This is an error, reject the sign-in.

Otherwise, the server now knows the claimed username and public key. To validate the signature you'll need to construct the signed data and parse the public key. The `public_key_spki` values from the database are stored in SubjectPublicKeyInfo format and most languages will have some way to ingest them. See chapter 13.

Your language's crypto library should provide a function that takes a signature and some signed data and tells you whether that signature is valid for a given public key. For the signed data, calculate the SHA-256 hash of the client data JSON and append it to the contents of the authenticator data. If the signature isn't valid, reject the sign-in.

But there are still a number of things that you need to check!

Parse the client data as UTF-8 JSON and check that:

1. It's valid JSON.
2. The `type` member is `webauthn.get`.

3. The `challenge` member is equal to the base64url encoding of the challenge that the server gave for this sign-in.
4. The `origin` member is equal to your site's sign-in origin (e.g. a string like `"https://www.example.com"`), or is a recognised Android app.
5. The `crossOrigin` member, if present, is false.

There's more! Take the `authenticatorData` and check that:

1. The first 32 bytes are equal to the SHA-256 hash of the RP ID that you're using.
2. That bit zero of the byte at offset 32 is one. I.e. `(authData[32] & 1) == 1`. This is the user presence bit[4] that indicates that a user approved the signature.

If all those checks work out, then sign in the user whose passkey it was. E.g. set a cookie and respond to the running Javascript so that it can update the page.

If the stored value of `backed_up` is not equal to `(authData[32] >> 4) & 1` then update that in the database.

> The user verification bit isn't checked above, but some sites might want to require user verification.

Removing passwords

Once a user is using passkeys to sign in, great! But if they were upgraded from a password then that password is hanging around on the account, doing nothing useful yet creating risk. It would be good to ask the user about removing the password.

Doing this is reasonable if the account has a backed-up passkey. I.e. if `SELECT 1 FROM passkeys WHERE username = ? AND backed_up = TRUE` returns results. A site might consider prompting the user to remove the password on an account when they sign in with a passkey and have a backed-up one registered.

Settings

If you've used passkeys with any sites, then you'll have noticed that they tend to list registered passkeys in their account settings, let users name each one, show the last used time, and let them be individually removed. If you want to do this then you'll need to add more columns to the `passkeys` table that we sketched above to support this. You might also want to record the AAGUID in order to automatically show where a passkey is stored.

Recall from page 60 that the AAGUID is the first 16 bytes of the attested credential data that is inside the authenticator data when creating a credential. It will generally reveal which passkey provider created a given credential. (If it was a passkey provider and not a security key, that is.) If the AAGUID is not the all-zero value, then it might be one of the following values:

ID	Name
08987058-cadc-4b81-b6e1-30de50dcbe96	Windows Hello
0ea242b4-43c4-4a1b-8b17-dd6d0b6baec6	Keeper
17290f1e-c212-34d0-1423-365d729f09d9	Thales PIN iOS SDK
39a5647e-1853-446c-a1f6-a79bae9f5bc7	IDmelon
50726f74-6f6e-5061-7373-50726f746f6e	Proton Pass
531126d6-e717-415c-9320-3d9aa6981239	Dashlane
53414d53-554e-4700-0000-000000000000	Samsung Pass
6028b017-b1d4-4c02-b4b3-afcdafc96bb2	Windows Hello
66a0ccb3-bd6a-191f-ee06-e375c50b9846	Thales Bio iOS SDK
771b48fd-d3d4-4f74-9232-fc157ab0507a	Edge on Mac
8836336a-f590-0921-301d-46427531eee6	Thales Bio Android SDK
891494da-2c90-4d31-a9cd-4eab0aed1309	Sésame
9ddd1817-af5a-4672-a2b9-3e3dd95000a9	Windows Hello
adce0002-35bc-c60a-648b-0b25f1f05503	Chrome on Mac
b5397666-4885-aa6b-cebf-e52262a439a2	Chromium Browser
b84e4048-15dc-4dd0-8640-f4f60813c8af	NordPass
bada5566-a7aa-401f-bd96-45619a55120d	1Password
cc45f64e-52a2-451b-831a-4edd8022a202	ToothPic Passkey Provider
cd69adb5-3c7a-deb9-3177-6800ea6cb72a	Thales PIN Android SDK
d548826e-79b4-db40-a3d8-11116f7e8349	Bitwarden
dd4ec289-e01d-41c9-bb89-70fa845d4bf2	iCloud Keychain (Managed)
ea9b8d66-4d01-1d21-3ce4-b6b48cb575d4	Google Password Manager
f3809540-7f14-49c1-a8b3-8f813b225541	Enpass
fbfc3007-154e-4ecc-8c0b-6e020557d7bd	iCloud Keychain
fdb141b2-5d84-443e-8a35-4698c205a502	KeePassXC

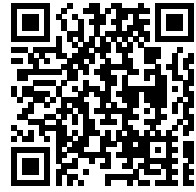
(This comes from https://github.com/passkeydeveloper/passkey-authenticator-aaguids[4], which may be more up to date by the time that you read this, and also contains icons for many of the providers.)

For simpler sites it's also perfectly valid to avoid all that complexity and have a "reset passkeys" button (like a "reset password" button). It would prompt for a new passkey registration (with no `excludeCredentials` listed), delete all other passkeys, and invalidate all other active sessions for the user. Unfortunately the deleted passkeys would still exist on the client side. Work is underway to add an API (called the *signal API*) that would let sites inform platforms during subsequent sign-ins that other passkeys have been invalidated, but it's not ready at the time of writing.

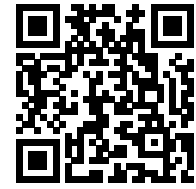
---

1 www.w3.org

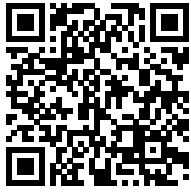
2 www.w3.org



3 w3c.github.io



4 www.w3.org



5 github.com

CHAPTER 13
Public key formats

There are several different public key formats that you'll encounter around WebAuthn. This chapter will help you to recognize and handle them. Overwhelmingly, WebAuthn uses a signature scheme called ECDSA P-256, so we'll discuss the different public key formats for this scheme.

ECDSA

X9.62 format

ECDSA public keys are (x, y) coordinate pairs. The coordinates in P-256 are 256-bit numbers. The simplest public key format zero-pads the coordinates so that they're each 32-byte, big-endian values, sticks them together, and prepends an 0x04 byte. This is *X9.62* format.

You can recognize it because these public keys are always 65 bytes long and they start with an 0x04 byte. Although not commonly encountered directly in WebAuthn, raw X9.62 keys are found within the next format discussed.

Note that, technically, this is the *uncompressed* X9.62 format. Since the coordinate values are related by an equation, you can derive the y value given the x value, although the square root operation means you obtain a value that is either equal to y or $-y$.

Because of this, there's also *compressed* X9.62 format where only the x coordinate is given and the leading byte is 0x02 or 0x03, depending on which of the two y values from the square root operation is the correct one. However, this is *extremely* rare in general. In WebAuthn, it only appears inside the QR code of a hybrid connection (see chapter 10) and hybrid is a feature implemented only by the platform.

SubjectPublicKeyInfo format

X9.62 works great, but it's nice for public key formats to be able to describe which signature scheme they apply to. There are other elliptic curves with 256-bit coordinates and it would be nice if the public keys for all these schemes weren't mutually ambiguous.

So a SubjectPublicKeyInfo[1] (SPKI) wraps a public key in ASN.1 that identifies a signature scheme. This is the format used inside of X.509 certificates, and so is quite widely supported. It's also the format that WebAuthn's `getPublicKey()` returns.

You can recognize it because it starts with an 0x30 byte. (Although that's common to all ASN.1-based formats.) You can convert from X9.62 format to SPKI by prepending the following bytes:

```
3059301306072a8648ce3d020106082a8648ce3d030107034200
```

Those bytes include the algorithm identifier for ECDSA P-256 and the needed prefix such that the X9.62 bytes can follow, without any suffix. However, the format is sufficiently flexible that it would be unwise to try matching and removing that prefix to convert in the other direction. Since SPKI is widely supported, that shouldn't be necessary anyway.

If you're looking for functions in your favorite language to parse an SPKI, see `java.security.spec.X509EncodedKeySpec` in Java, `System.Security.Cryptography.ECDsa.ImportSubjectPublicKeyInfo` in .NET, or `crypto/x509.ParsePKIXPublicKey` in Go.

COSE format

COSE[2] is the "CBOR Object Signing and Encryption" framework and it's the format used by public keys in the authentication data. If you're doing attestation (see chapter 7) or dealing with the Apple API (see chapter 11) then you'll have to process this format. Unfortunately, it's rare and is not commonly supported by general cryptographic libraries.

As the name suggests, it's encoded with CBOR and, in the case of WebAuthn, that'll be the CTAP2 subset of CBOR (see chapter 6). Technically speaking you can read RFC 8152 and figure out how to process a COSE public key, but the RFC is not straightforward. You're better off looking at an example:

```
{
  1: 2,   # key type = elliptic curve
  3: -7,  # alg = ECDSA P-256
  -1: 1,  # curve = P-256
  # x and y coordinates
  -2: h'950F7AF17D9E...',
  -3: h'7A6B0654742C...',
}
```

The x and y coordinates in the CBOR are zero-padded and so, for P-256, must always be 32-bytes long.

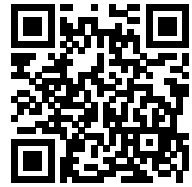
You can recognize these keys because they'll start with 0xa5 (for a CBOR map with five entries) and, as mentioned, they appear in the authenticator data. In order to convert to X9.62 format you can parse the CBOR, check that keys 1, 3, and −1 are present with the expected values, check that keys −2 and −3 are present with 32-

byte values, then concatenate an 0x04 byte and the x and y values. Once you have X9.62 format, see above for how to convert to SPKI format if you need.

---

1 datatracker.ietf.org

2 datatracker.ietf.org



CHAPTER 14
Index