ImperialViolet

Security Keys (27 Mar 2018)

Introduction

Predictions of, and calls for, the end of passwords have been ringing through the press for many years now. The first instance of this that Google can find is from Bill Gates in 2004, although I suspect it wasn't the first.

None the less, the experience of most people is that passwords remain a central, albeit frustrating, feature of their online lives.

Security Keys are another attempt address this problem—initially in the form of a second authentication factor but, in the future, potentially as a complete replacement. Security Keys have gotten more traction than many other attempts to solve this problem and this post exists to explain and, to some extent, advocate for them to a technical audience.

Very briefly, Security Keys are separate pieces of hardware capable of generating public/private key pairs and signing with them. By being separate, they can hopefully better protect those keys than a general purpose computer can, and they can be moved between devices to serve as a means of safely authorising multiple devices. Most current examples attach via USB, but NFC and Bluetooth devices also exist.

Contrasts with existing solutions

Security Keys are not the first attempt at solving the problems of passwords, but they do have different properties than many of the other solutions.

One common form of second factor authentication is TOTP/HOTP. This often takes the form of an app on the user's phone (e.g. Google Authenticator) which produces codes that change every minute or so. It can also take the form of a token with an LCD display to show such codes (e.g. RSA SecurID).

These codes largely solve the problem of password reuse between sites as different sites have different seed values. Thus stealing the password (and/or seed) database from one site no longer compromises accounts at other sites, as is the case with passwords.

However, these codes are still phishable: the user may be tricked into entering their password and code on a fake site, which can promptly forward them to the real site and impersonate the user. The codes may also be socially engineered as they can be read over the phone etc by a confused user.

Another common form of second factor authentication are SMS-delivered codes. These share all the flaws of HOTP/TOTP and add concerns around the social engineering of phone companies to redirect messages and, in extreme cases, manipulation of the SS7 network.

Lastly, many security guides advocate for the use of password managers. These, if used correctly, can also solve the password reuse problem and significantly help with phishing, since passwords will not be auto-filled on the wrong site. Thus this is sound advice and, uniquely amongst the solutions discussed here, can be deployed unilaterally by users.

Password managers, however, do not conveniently solve the problem of authenticating new devices, and their automated assistance is generally limited to a web context. They also change a password authentication from an (admittedly weak) verification of the *user*, to a verification of the *device*; an effect which has provoked hostility and counter-measures from relying parties.

In light of this, Security Keys should be seen as a way to improve upon, and exceed the abilities of, password managers in a context where the relying party is cooperating and willing to make changes. Security Keys are unphishable to a greater extent than password managers because credentials are bound to a given site, plus it's infeasible to socially engineer someone to read a binary signature over the phone. Also, like TOTP/HOTP, they use fresh credentials for each site so that no reuse is possible. Unlike password managers, they can work outside of a web context and they can serve to authenticate new devices.

They aren't magic, however. The unphishability of Security Keys depends on the extent to which the user may be mislead into compromising other aspects of their device. If the user can be tricked into installing malware, it could access the Security Key and request login signatures for arbitrary sites. Also, malware may compromise the user's login session in a browser *after* successfully authenticating with a Security Key. Still, that's a heck of a lot better than the common case of people using the same password across dozens of sites.

All the different terms

There is a lot of terminology specific to this topic. The first of which I've already used above: "relying parties". This term refers to any entity trying to authenticate a user. When logging into a website, for example, the website is the relying party.

The FIDO Alliance is a group of major relying parties, secure token manufacturers, and others which defines many of the standards around Security Keys. The term that FIDO uses for Security Keys is "Universal 2nd factor" (U2F) so you'll often see "U2F security key" used—it's talking about the same thing. The terms "authenticator" and "token" are also often used interchangeably to refer to these devices.

At the time of writing, all Security Keys are based version one of FIDO's "Client To Authenticator Protocol" (CTAP1). This protocol is split between documentation of the core protocol and separate documents that describe how the core protocol is transported over USB, NFC, and Bluetooth.

FIDO also defines a U2F Javascript API for websites to be able to interact with and use Security Keys. However, no browser ever implemented that API prior to a forthcoming (at the time of writing) version of Firefox.

But sites have been able to use Security Keys with Google Chrome for some years because Chrome ships with a hidden, internal extension through which the U2F API can be implemented with a Javascript polyfill, which Google also provides. (Extensions for Firefox were also available prior to native support in that browser.)

Thus all sites which supported Security Keys prior to 2018 used some polyfill in combination with Chrome's internal extension, or one of the Firefox extensions, to do so.

The FIDO Javascript API is not the future, however. Instead, the W3C is defining an official Web Authentication standard for Security Keys, which is commonly called by its short name “webauthn”. This standard is significantly more capable (and significantly more complex) than the U2F API but, by the end of 2018, it is likely that all of Edge, Chrome, and Firefox will support it by default.

The webauthn standard has been designed to work with existing (CTAP1-based) devices, but FIDO is working on an updated standard for tokens, CTAP2, which will allow them to take advantage of the new capabilities in webauthn. (The standards were co-developed so it’s equally reasonable to see it from the other direction and say that webauthn allows browsers to take advantage of the new capabilities in CTAP2.)

There are no CTAP2 devices on the market yet but their major distinguishing feature will be that they can be used as a 1st (and only) factor. I.e. they have enough internal storage that they can contain a username and so both provide an identity and authenticate it. This text will mostly skim over CTAP2 since the devices are not yet available. But developers should keep it in mind when dealing with webauthn as it explains many, otherwise superfluous, features in that standard.

CTAP1 Basics

Since all current Security Keys use CTAP1, and webauthn is backwards compatible with it, understanding CTAP1 is pretty helpful for understanding the space in general. Here I’ll include some Python snippets for communicating with USB CTAP1 devices to make things concrete, although I’ll skip over everything that deals with framing.

CTAP1 defines two operations: creating a new key, and signing with an existing key. I’ll focus on them in turn.

Creating a new key

This operation is called “registration“ in CTAP1 terminology and it takes two, 32-byte arguments: a “challenge” and an “application parameter”. From the point of view of the token these arguments are opaque byte strings, but they’re intended to be hashes and the hash function has to be SHA-256 if you want to interoperate.

The challenge argument, when used with a web browser, ends up being the hash of a JSON-encoded structure that includes a random nonce from the relying party as well as other information. This nonce is intended to prove freshness: if it was signed by the newly generated key then the relying party could know that the key really was fresh and that this was the only time it had been registered. Unfortunately, CTAP1 doesn't include any self-signature (and CTAP2 devices probably won't either). Instead the situation is a lot more complex, which we'll get to.

The application parameter identifies a relying party. In U2F it's a hash of the origin (e.g. SHA-256("`https://example.com`")) while in webauthn it's a hash of the domain (e.g. SHA-256("`example.com`")). As we'll see, the signing operation also takes an application parameter and the token checks that it's the same value that was given when the key was created. A phishing site will operate on a look-alike domain, but when the browser hashes that domain, the result will be different. Thus the application parameter sent to the token will be different and the token will refuse to allow the key to be used. Thus keys are bound to specific origins (or, with webauthn, domains) and cannot be used outside of that context.

Here's some sample code that'll shine some light on other aspects of the protocol, including the outputs from key creation:

```
while True:
    challenge_hash = hashlib.sha256('challenge').digest()
    app_param_hash = hashlib.sha256('https://example.com').digest()
    status, reply = transact(1, 3, 0, challenge_hash + appid_hash)
    if status == 0x6985:
        time.sleep(0.5)
        continue
    print 'Public key: ' + reply[1:66].encode('hex')
    key_handle_length = ord(reply[66])
    print 'Key handle: ' + reply[67:67 + key_handle_length].encode('hex')
    reply = reply[67 + key_handle_length:]
    # This is a fragile way of getting an ASN.1 length;
    # just to keep the code small.
    cert_len = struct.unpack('>H', reply[2:4])[0]
    print '-----BEGIN CERTIFICATE-----'
    print reply[:4+cert_len].encode('base64'),
    print '-----END CERTIFICATE-----'
    print 'Signature: ' + reply[4+cert_len:]
    break
```

(The full source is available if you want to play with it, although it'll only work on Linux.)

The first thing to note is that the operation runs in a loop. CTAP1 devices require a "user presence" test before performing operations. In practice this means that they'll have a button or capacitive sensor that you have to press. While the button/sensor isn't triggered, operations return a specific error code and the host is expected to retry the operation after a brief delay until it succeeds.

A user-presence requirement ensures that operations cannot happen without a human being physically present. This both stops silent authentication (which could be used to track people) and it stops malware from silently proxying requests to a connected token. (Although it doesn't stop malware from exploiting a touch that the user believes is authorising a legitimate action.)

Once the operation is successful, the response can be parsed. In the spirit of short example code everywhere, errors aren't checked, so don't use this code for real.

Key generation, of course, produces a public key. For CTAP1 tokens, that key will always be an uncompressed, X9.62-encoded, ECDSA P-256 public key. That encoding happens to always be 65 bytes long.

After that comes the key handle. This is an opaque value that the token uses to identify this key, and this evinces another important facet of CTAP1: the tokens are nearly stateless in practice.

In *theory*, the key handle could be a small identifier for a key which is stored on the device. In practice, however, the key handle is always an encrypted version of the private key itself (or a generating seed). This allows storage of the keys to be offloaded to the relying party, which is important for keeping token costs down.

This also means that CTAP1 tokens cannot be used without the user entering a username. The relying party has to maintain a database of key handles, and that database has to be indexed by something. This is changing with CTAP2, but I'll not mention that further until

CTAP2 tokens are being commercially produced.

Lastly, there's the attestation certificate (just one; no chain) and a signature. As I mentioned above, the signature is sadly not from the newly created key, but from a separate attestation private key contained in the token. This'll be covered in a later section

Signing with a key

Once a key has been created, we can ask the token to sign with it.

Again, a "challenge" and "application parameter" have to be given, and they have the same meaning and format as when creating a key. The application parameter has to be identical to the value presented when the key was created otherwise the token will return an error.

The code looks very similar:

```
while True:
    challenge_hash = hashlib.sha256('challenge').digest()
    appid_hash = hashlib.sha256('https://example.com').digest()
    status, reply = transact(2, 3, 0, challenge_hash + appid_hash +
                                  chr(len(key_handle)) + key_handle)
    if status == 0x6985:
        time.sleep(0.5)
        continue
    if status != 0x9000:
        print hex(status)
        os.exit(1)
    (flags, counter) = struct.unpack('>BI', reply[:5])
    if flags & 1:
        print 'User-presence tested'
    print 'Counter: %d' % counter
    print 'Signature: ' + reply[5:].encode('hex')
    break
```

The same pattern for waiting for a button press is used: the token is polled and returns an error until the button/sensor is pressed.

Three values are returned: a flag confirming that the button was pressed, a signature counter, and the signature itself—which signs over the challenge, application parameter, flags, and counter.

In the web context, the challenge parameter will be the hash of a JSON structure again, and that includes a nonce from the relying party. Therefore, the relying party can be convinced that the signature has been freshly generated.

The signature counter is a strictly monotonic counter and the intent is that a relying party can record the values and so notice if a private key has been duplicated, as the strictly-monotonic property will eventually be violated if multiple, independent copies of the key are used.

There are numerous problems with this, however. Firstly, recall that CTAP1 tokens have very little state in order to keep costs down. Because of that, all tokens that I'm aware of have a single, global counter shared by all keys created by the device. (The only exception I've seen is the Bluink key because it keeps state on a phone.) This means that the value and growth rate of the counter is a trackable signal that's transmitted to all sites that the token is used to login with. For example, the token that I'm using right now has a counter of 431 and I probably use it far more often than most because I'm doing things like writing example Python code to trigger signature generation. I'm probably pretty identifiable because of that.

A signature counter is also not a very effective defense, especially if it's per-token, not per-key. Security Keys are generally used to bless long-term logins, and an attacker is likely to be able to login once with a cloned key. In fact, the login that violates the monotonicity of the counter will probably be a *legitimate* login so relying parties that strictly enforce the requirement are likely to lock the real user out after a compromise.

Since the counter is almost universally per-token, that means that it'll commonly jump several values between logins to the same site because the token will have been used to login elsewhere in-between. That makes the counter less effective at detecting cloning. If login sessions are long-lived, then the attacker may only need to sign once and quite possibly never be detected. If an attacker is able to observe the evolution of the counter, say by having the user login to an attacker-controlled site periodically, they can avoid ever triggering a counter regression on a victim site.

Finally, signature counters move the threat model from one that deals with phishing and password reuse, to one where attackers are capable of extracting key material from hardware tokens. That's quite a change and the signature counter is not optional in the protocol.

Attestation

When creating a key we observed that the token returned a certificate, and a signature over the newly created key by that certificate. Here's the certificate from the token that I happen to be using as I write this:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 95815033 (0x5b60579)
    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: CN = Yubico U2F EE Serial 95815033
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:fd:b8:de:b3:a1:ed:70:eb:63:6c:06:6e:b6:00:
                    69:96:a5:f9:70:fc:b5:db:88:fc:3b:30:5d:41:e5:
                    96:6f:0c:1b:54:b8:52:fe:f0:a0:90:7e:d1:7f:3b:
                    ff:c2:9d:4d:32:1b:9c:f8:a8:4a:2c:ea:a0:38:ca:
                    bd:35:d5:98:de
                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
    Signature Algorithm: sha256WithRSAEncryption
```

Clearly, it identifies the manufacturer and the unknown extension in there identifies the exact model of the key.

The certificate, despite the 32-bit serial number, *doesn't* uniquely identify the device. The FIDO rules say that at least 100 000 other devices should share the same certificate. (Some devices mistakenly shipped with uniquely identifying attestation certifications. These will be recognised and suppressed in Chrome 67.) This device happens to be a special, GitHub-branded one. I don't know whether this attestation certificate is specific to that run of devices, but the same model of device purchased on Amazon around the same time (but unbranded) had a different certificate serial number, so maybe.

But the practical upshot is that a relying party can determine, with some confidence, that a newly created key is stored in a Yubico 4th-gen U2F device by checking the attestation certificate and signature. That fact should cause people to pause; it has weighty ramifications.

Traditionally, anyone who implemented the various specifications that make up a web browser or server has been an equal participant in the web. There's never been any permission needed to create a new browser or server and, although the specifications on the client side are now so complex that implementing them is a major effort, it's possible to leverage existing efforts and use WebKit, Firefox, or Chromium as a base.

(The DRM in Encrypted Media Extensions might be an exception, and there was an appropriately large fight over that. I'm not making any judgment about the result of that here though.)

But with attestation, it's quite possible for a site to require that you have particular hardware if you want to use webauthn. The concerns about that vary depending on the context: for an internal, corporate site to demand employees use particular hardware seems unobjectionable; a large public site raises many more concerns. There appear to be two, main ways in which this could develop in an unhealthy direction:

Firstly, as we've experienced with User-Agent headers, sites are not always as responsive as we would like. User-Agent headers have layers upon layers of browsers spoofing other browsers—the history of browser development is written in there—all because of this.

Spoofing was necessary because sites would implement workarounds or degraded experiences for some browsers but fail to update things as the browser landscape changed. In order to be viable, each new browser entrant had to spoof the identity of the currently dominant one.

But attestation is not spoofable. Therefore, if sites launch webauthn support and accept attestations from the current set of token vendors, future vendors may be locked out of the market: Their devices won't work because their attestations aren't trusted, and they won't be able to get sites to update because they won't have enough market presence to matter.

If things get really bad, we may see a market develop where attestation certificates are traded between companies to overcome this. The costs of that are ultimately borne by increased token prices, which users have to pay.

The second concern is that, if different sites adopt different policies, users can have no confidence that a given token will work on a given site. They may be forced to have multiple tokens to span the set of sites that they use, and to remember in each case which token goes with which site. This would substantially degrade the user experience.

FIDO does not dismiss these worries and their answer, for the moment, is the metadata service (MDS). Essentially this is a unified root store that all sites checking attestation are supposed to use and update from. That would solve the problem of site stagnation by creating a single place for new vendors to submit their roots which would, in theory, then auto-update everywhere else. It would also help with the problem of divergent policies because it includes a FIDO-specified security-level for each type of token, which would at least reduce the variety of possible policies and make them more understandable—if used.

The challenge at the moment is that major vendors are not even included in the MDS, and using the MDS properly is harder for sites than throwing together a quick hack that'll be "good enough for now".

Thus my advice is for sites to ignore attestation if you're serving the public. As we'll see when we cover the webauthn API, attestation information is not even provided by default. Sites newly supporting webauthn are probably just using passwords, or maybe SMS OTP, and thus Security Keys offer a clear improvement. Worrying about whether the Security Key is a physical piece of hardware, and what certifications it has, is a distraction.

webauthn

Now that we understand the underlying protocol, here's how to actually use it. As mentioned above, there's an older, "U2F" API but that'll disappear in time and so it's not covered here. Rather, the future is the W3C Web Authentication API, which builds on the Credential Management API.

Right now, if you want to experiment, you can use Firefox Nightly, Chrome Canary, or Edge 14291+. In Firefox Nightly, things should be enabled by default. For Chrome Canary, run with `—enable-features=WebAuthentication —enable-experimental-web-platform-features`. For Edge, I believe that you have to enable it in about:flags, but I'm just going off documentation.

Testing for support

Before trying to use webauthn, you should use feature detection to ensure that it's supported by a given browser:

```
if (!window.PublicKeyCredential) {
    // webauthn is not supported.
}
```

Creating a new key

The following are partial snippets of Javascript for creating a key. In-between the snippets is discussion so you would need to concatenate all the snippets in this section to get a complete example.

```
var makePublicKey = {
    // rp specifies things about the "relying party", i.e. the website.
    rp: {
        // A friendly display name for the site.
        name: 'ACME Widgets',
        // Optional: RP ID for the key.
        id: 'www.example.com',
    },
```

The `name` field is required, but currently discarded. In the future it's intended that CTAP2 tokens will be able to store this and so it could be used in an account chooser interface.

The `id` field is optional and it sets the relying party ID (RP ID) of the credential. This is the domain name associated with the key and it defaults to the domain that's creating the key. It can only be overridden to set it within the eTLD+1 of the current domain—like a cookie.

```
user: {
    // id is a opaque user-id for this account.
    id: new Uint8Array([0, 1, 2, 3, 4, 5, 6, 7]),
    // name is an account identifier. Doesn't have to be an
    // email-like string, just whatever your site uses.
    name: 'jsmith@example.com',
    // displayName is a more friendly version of |name|.
    displayName: 'Joe Smith',
},
```

All these fields are required. However, like the previous chunk, they're currently discarded and intended for forthcoming CTAP2 tokens.

The `id` identifies an account. A given CTAP2 token should not store two keys for the same RP ID and user ID. However, it's possible that CTAP2 keys will allow blind enumeration of user IDs given physical possession of the token. (We have to see how that part of the CTAP2 spec is implemented in practice.) Therefore, you don't want to store a username in the `id` field. Instead you could do something like HMAC-SHA256(key = server-side-secret, input = username)[:16]. (Although you'll need a reverse index in the future if you want to use CTAP2 tokens in 1st-factor mode.)

The `name` and `displayName` are again strings intended for a future account chooser UI that doesn't currently exist.

```
// challenge is an ArrayBuffer containing the nonce.
challenge,
```

The `challenge` field for a new key is a little complex. The webauthn spec is very clear that this must be a strong, server-generated nonce and, for the assertion request which we'll get to next, that's correct. Also, if you're checking attestation then this challenge is your only assurance of freshness, so you'll want it in that case too.

However, as I mentioned above, it's far from clear how well attestation will work outside of a controlled environment and I recommend that you ignore it in the general case. Given that, the utility of the challenge when creating a key is questionable. Generated U2F keys don't sign over it and, while CTAP2 keys have the option of covering it with a self-signature, it remains to be seen whether any will actually do that.

One advantage that it does bring is that it stops CSRF attacks from registering a new key. But, if you don't have a solid, general solution to CSRF already then you have far larger security issues.

Thus, since it's easy and you'll need it for assertions anyway, I still recommend that you generate a 16- or 32-byte nonce on the server as the spec suggests. I just thought that you should be aware that the benefit is a little fuzzier than you might imagine.

```
    pubKeyCredParams: [
        { type: 'public-key', alg: -7 },
    ],
```

This enumerates the types of public keys that the server can process. The `type` field is always `public-key` and the `alg` comes from the IANA COSE list. The value -7 means ECDSA with P-256, which is effectively mandatory since it's what all U2F tokens implement. At the moment, that's the only value that makes sense although there might be some TPM-based implementations in the future that use RSA.

```
    timeout: 10000,
    excludeCredentials: [],
};

let credPromise = navigator.credentials.create(
    {publicKey: makePublicKey}
);
```

The `timeout` specifies how many milliseconds to wait for the user to select a token before returning an error.

The `excludeCredentials` is something that can be ignored while you get something working, but which you'll have to circle back and read the spec on before deploying anything real. It allows you to exclude tokens that the user has already created a key on when adding new keys.

The promise will, if everything goes well, resolve to a PublicKeyCredential, the `response` member of which is an AuthenticatorAttestationResponse. I'm not going to pointlessly re-write all the server-side processing steps from the spec here but I will note a couple of things:

Firstly, if you don't know what token-binding is, that's fine: ignore it. Same goes for extensions unless you previously supported the U2F API, in which case see the section below. I also suggest that you ignore the parts dealing with attestation (see above), which eliminates 95% of the complexity. Lastly, while I recommend following the remaining steps, see the bit above about the `challenge` parameter and don't erroneously believe that checking, say, the `origin` member of the JSON is more meaningful than it is.

I'll also mention something about all the different formats in play here: In order to process webauthn, you'll end up dealing with JSON, ASN.1, bespoke binary formats, and CBOR. You may not have even heard of the last of those but CBOR is yet another serialisation format, joining Protocol Buffers, Msgpack, Thrift, Cap'n Proto, JSON, ASN.1, Avro, BSON, …. Whatever you might think of the IETF defining another format rather than using any of the numerous existing ones, you will have to deal with it to support webauthn, and you'll have to deal with COSE, the CBOR translation of JOSE/JWT. You don't have to support tags or indefinite lengths in CBOR though because the CTAP2 canonicalisation format forbids them.

The problem is that there's going to be a lot of people having to implement converters from COSE key format to something that their crypto library accepts. Maybe I should start a GitHub repo of sample code for that in various libraries. But, short of that, here's a quick reference to help you navigate all the different formats in play:

COSE Key

Used in: the public key in the attested credential data when creating a key. You have to parse this because you need the public key in order to check assertions.
Format: a CBOR map, defined here. However, you'll struggle to figure out, concretely, what the keys in that map are from the RFC so, for reference, if you're decoding a P-256 key you should expect these entries in the map: 1 (key type) = 2 (elliptic curve, x&y), 3 (algorithm) = -7 (ECDSA with SHA-256), -1 (curve) = 1 (P-256), and then x and y coordinates are 32-byte values with keys -2 and -3, respectively.
How to recognise: the first nibble is 0xa, for a small CBOR map.

X9.62 key

Used in: lots of things, this is the standard format for EC public keys. Contained within the SPKI format, which is used by Web Crypto and X.509.
Format: a type byte (0x04 for standard, uncompressed keys), followed by x and y values.
How to recognise: for P-256, it's 65 bytes long and start with 0x04. (There's also a compressed version of X9.62 but it's exceedingly rare.)

SPKI

Used in: X.509 and Web Crypto.
Format: an ASN.1 structure called SubjectPublicKeyInfo: an algorithm identifier followed by a lump of bytes in an algorithm-specific format, which is X9.62 for elliptic-curve keys.
How to recognise: starts with 0x30.

ASN.1 signatures

Used in: nearly everything that deals with ECDSA signatures.
Format: an ASN.1 SEQUENCE containing two INTEGER values for ECDSA's *r* and *s* values.
How to recognise: starts with 0x30 and you're expecting a signature, not a key.

"Raw" signatures

Used in: Web Crypto.
Format: a pair of 32-byte values (for P-256).
How to recognise: 64 bytes long.

Getting assertions

Once you have one (or more) keys registered for a user you can challenge them for a signature. Typically this is done at login time although CTAP2 envisions *user verifying* tokens that take a fingerprint or PIN number, so signatures from those devices could replace reauthentication requests. (I.e. those times when a site asks you to re-enter your password before showing particularly sensitive information.)

When getting an assertion, the *challenge* (i.e. nonce) really must be securely generated at the server—there's none of the equivocation as with generation.

```
var authPromise = navigator.credentials.get({publicKey: {
  // A 16- or 32-byte, server-generated, nonce.
  challenge,
  // The number of milliseconds before timing out.
  timeout: 10000,
  // Optional: the relying party ID. Defaults to the current domain.
  rpId: document.domain,
  // A list of credentialIDs of registered keys.
  allowCredentials: [{
    type: "public-key",
    id: credentialID,
  }],
}});
```

The `credentialID` values are those extracted from the attested credential data from the registration.

Again, there's no need for me to reiterate the processing steps already enumerated in the spec except to repeat that you can ignore token-binding if you don't know what it is, and to reference the comments above about signature counters. If you implement the signature counter check, you should think through how the user will experience various scenarios and ensure that you're monitoring metrics for the number of counter failures observed. (I don't think we know, in practice, how often counter writes will be lost, or counters will be corrupted.)

Supporting keys created with the U2F API

This is just for the handful of sites that supported Security Keys using the older, U2F API. If you have keys that were registered with that API then they aren't immediately going to work with webauthn because the AppID from U2F is a URL, but the relying party ID in webauthn is just a domain. Thus the same origin is considered to be a different relying party when using U2F and webauthn and the keys won't be accepted. (Just the same as if you tried to use a credential ID from a different site.)

However, there is a solution to this: in the `publicKey` argument of the `get` call add an extensions field containing a dict with a key `appid` whose value is the AppID of your U2F keys. This can be done uniformly for all keys if you wish since both the relying party ID and AppID will be tried when this is asserted. With this in place, keys registered with U2F should work with webauthn.

There is no way to create a U2F key with webauthn however. So if you rollout webauthn support, have users create keys with webauthn, and have to roll it back for some reason, those new keys will not work with the U2F API. So complete the transition to webauthn of your login process first, then transition registration.

Concluding remarks

It's exciting to see webauthn support coming to most browsers this year. I get to use Security Keys with about the same number of sites as I use SMS OTP with, and I use Security Keys anywhere I can. While I, like everyone technical, *assumes* that I'm less likely to get phished than most, I'm still human. So I hope that wide-spread support for webauthn encourages more sites to support Security Keys.

Challenges remain, though. Probably the most obvious is that NFC is the ideal interface for mobile devices, but it doesn't work with iOS. Bluetooth works for both Android and iOS, but requires a battery and isn't as frictionless.

Security keys will probably remain the domain of the more security conscious in the short-term since, with CTAP1, they can only be an additional authentication step. But I hope to see CTAP2 tokens generally available this year with fingerprint readers or PIN support. It might be that, in a few years time, a significant number of people have a passwordless experience with at least one site that they use regularly. That’ll be exciting.