# FIDO2 and WebAuthn

Authentication has been an essential part of applications for some time now because applications need to know some information about the user who’s using the application. For the longest time, the solution to this has been username and passwords. Username passwords are popular because they’re convenient to implement. But they aren’t secure. There are many issues with passwords.

First, there’s the problem of transmitting this password securely. If you send the password over the wire, a man-in-the-middle could sniff it. That pretty much necessitated SSL over such communication or the equivalent of creating a hash of the password that’s sent over the wire instead of the actual password. But even those techniques didn’t solve the problem of the server securing the password, or a secure hash of the password. Or, for that matter, keeping you safe from replay attacks. Increasingly complex versions of this protocol were created, to the point where you could, with some degree of confidence, say that you were safe from man-in-the-middle attacks or replay attacks.

Users created a simple, easy to remember password, and brute force techniques guessed those passwords. So we came up with complex requirements for passwords, such as your password must contain an upper case, lower case, special character, and minimum length—and yet people still picked poor passwords. When they didn’t pick poor passwords that were easy to remember, they would reuse passwords across different systems. Or they would use password managers to store their passwords, until the password manager itself got compromised.

But even then, you’re not safe from passwords being leaked. Worse, leaked passwords are not detected—you don’t know if your password has been leaked until the leak is discovered. And these leaks could occur on a poorly implemented service. This means, no matter what you do, you’re still insecure.

## Don’t Despair

There are solutions. There are concepts like MFA or one-time passwords that can be used in addition to your usual password. This is what you’ve experienced when you enter a credential, but in addition, you have to enter a code sent to you via SMS or from an authenticator app on your phone.

MFA and one-time passwords are great. In fact, I’d go to the extent of saying that if there’s a service you‘re using that uses only username password, just assume it’s insecure, and don’t use it for anything critical. Additionally, pair it with common-sense practices like own your domain name, and a separate email address from your normal use email address for account recovery. Secret questions and answers that aren’t easy to guess, and answers that don’t make sense to anyone.

As great as MFA and one-time passwords are, they’re still not a perfect picture. There are a few big issues with this approach.

First, they are cumbersome to manage for the end user. I work with this stuff on a daily basis, and I find it frustrating to manage 100s of accounts, multiple authenticator apps, and I worry that if I ever broke my phone accidentally, I’d be transported to neanderthal times immediately. I can’t imagine how a common non-technology-friendly person deals with all this.

Second, MFAs and one-time passwords are both cumbersome and expensive for the service provider. All those SMS messages and push notifications cost money. This creates a barrier to entry for someone trying to get a service off the ground. Then there’s the question of which authenticator app to trust and whether that app be trusted. Is SMS good enough?

Third, there’s the issue of phishing. As great as MFA is, someone can set up a service that looks identical to a legit service, and unless you have very keen eyes watching every step, you may fall for it. Unfortunately, even the best of us is tired and stressed at times, and that’s when you fall for this. In fact, the unscrupulous service that pretends to be a legit service could simply forward your requests to the legit service after authentication while stealing your session. So you may think everything is hunky dory but your session has effectively been stolen.

Finally, there is authentication fatigue. Hey, I just want to login and use a system. Zero trust dictates that you assume a breach, so it’s common for services to over-authenticate. This creates authentication fatigue, and an already fatigued user could blindly approve an MFA request, especially if it’s cleverly disguised. It only takes one mistake for a hacker to get in the house, then they can do plenty of damage, potentially remaining undetected for a long time.

## What am I Trying to Solve?

I’m not trying to secure passwords or make a better MFA solution here. The fundamental problem I wish to solve here is how an application can securely trust a user’s identity, such that the identity is not cumbersome to manage, is secure, convenient, and not stealable.

Let’s refine this problem further. The problem I’m really trying to solve is that a user goes through a registration process, typically the first time you encounter the user. The next time the user shows up on the application, you want to make sure it’s the same person behind the user ID. You want to do so with 100% confidence, and you want to do so with relative ease for users and the application.

If you had a clean slate to architect this with the technology available today, how would you do it?

Imagine if, during registration, the user generates a key pair. This is a typical certificate. There’s a private key, and there’s a public key. With the private key, you can sign stuff, you can encrypt stuff, but you never share that private key. You can keep the private key in your private possession forever. But the public key is public information. With the public key, you can only decrypt or verify the signature.

During registration, you generate a key pair that’s unique for the service, and the public portion of that key pair is shared with the service. The server then stores it securely and connects it with this particular user (you).

Next time you wish to authenticate, the server generates a challenge that’s just a random string. This challenge is communicated over to the user securely over HTTPS. This challenge is encrypted or signed by you using your private key that’s unique to this service. This encrypted or signed string is now sent back to the service. The service can now validate the signature via the public key associated with the user.

This sounds like something that could work, but a few interesting things happened here. At no point was the private key communicated anywhere except on your device. As you’ll see later in this article, there are plenty of hardware devices that allow for the storage of this private key securely.

There’s also no need for a cumbersome MFA prompt or the maintenance of it. There’s also no risk of SMS spoofing or your phone number rolling over to the next subscriber.

There’s no additional cost for the service to send MFA prompts. The service just needs to remember a mapping of the public key with the user, so it’s no worse than remembering a password. This can be paired with existing MFA techniques, if you choose to do so.

Sounds like we’re on to something interesting here. Let’s dig further.

## Hardware Support

I’ve boiled the problem down to the much simpler problem of keeping your private keys secure. The good news is that there exists a lot of interesting hardware to help you do so. These look like USB keys, or even NFC keys, that securely store your private keys. An example of this key is a YubiKey, as shown in **Figure 1**.



Figure 1: YubiKeys

These private keys require some interaction from the user to extract the private key briefly when it is needed. Additionally, you now see things like trusted platform module (TPM) requirements baked into Windows 11 and MacOS/iOS, supporting things like TouchID on Macs and FaceID on phones, which, paired with secure enclaves, give you a pretty neat solution around storing these keys securely in iCloud.

The Apple ecosystem moves faster because they have full control on the end-to-end story, and it also helps that they make a phone. I’m particularly excited about the possibility of roaming these keys using iCloud. What this means for the end user is that they just use their devices as they normally do. They don’t need to carry a separate dongle or device or risk losing it. And yet they gain the convenience of never having to bother with a password while remaining secure. This is the holy grail of security—security and convenience, so users won’t try to work around inconveniences. All this is pretty new at the time of writing this article; the support for this technology was introduced in iOS 15 and future versions of OSs will improve this and make it more accessible. I’m quite excited about where this is headed.

Hardware aside, you probably need a common understanding of protocols for this standard to be implemented, right? Let’s talk about that next.

## Protocols

The overall concept sounds great, but if various services don’t speak a common language, this concept will never gain foothold. This is why this concept has been solidified as protocols. Like anything else in identity, protocols around this concept have been evolving.

The word “FIDO” comes from the FIDO alliance, which is the organization pushing for this standard. You can check them out at [https://fidoalliance.org](https://fidoalliance.org/). If you check out their website, you’ll see them describe specs on UX guidelines around strong authentication, but more interestingly, they talk of specific specs such as FIDO universal second factor (FIDO U2F), FIDO universal authentication framework (UAF), and FIDO2, which includes W3C’s Web authentication (WebAuthn) spec, and FIDO client-to-authenticator protocol (CTAP).

All right, that was a lot of acronyms I just threw at you. Let’s break it down in **Figure 2**. FIDO2 is the umbrella term of what I’m concerned with here. When the user needs to register or authenticate, they interact with an external authenticator or a platform authenticator. An external authenticator could be a USB key, such as a YubiKey. It looks just like USB flash storage but may have additional biometric protection on it. Or the user could use a platform authenticator, such as FaceID, TouchID, Windows Hello, etc. When the user interacts with a relying party (the service you are trying to access), it uses a protocol called WebAuthn.

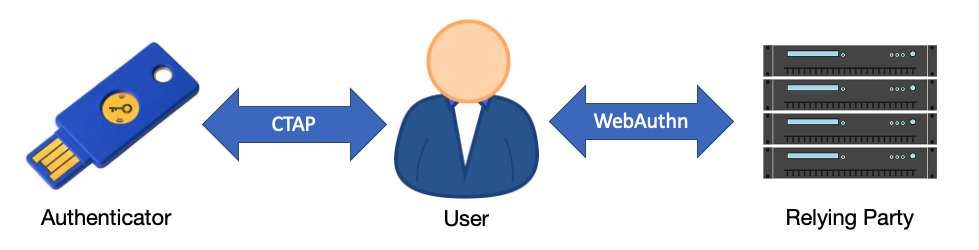


Figure 2: FIDO2 and its moving parts

## Registration Process

When a user first lands on a site, they create an account. This is called the registration process. Here’s how it would work if you were to do this under the FIDO2 protocol.

The user lands on the site, and says, “hey, I want to register a key.” The server then generates a challenge, a random string, and passes it over TLS to the user along with a bunch of other information. A critical part of this information is the relying party ID. The relying party ID must match the TLD or top-level domain of the site the user is on. This is verified with the SSL cert being used by the server. Once the client has verified the identity of the server, the client then generates a public-private key pair. The private key is never sent over the wire. But the public key, along with the signed challenge, is sent back to the server. Along with this, it also sends a credential ID generated by the security key.

The server then verifies the signed challenge with the public key. If it passes signature verification, the server then stores the credential ID and the public key, and sets the counter to zero. Every time an authentication is performed, this counter increments, to prevent the cloning of keys. There should be only one instance of the key in the wild, and if the counter isn’t sequential, authentication is denied. This entire process can be seen in **Figure 3**.

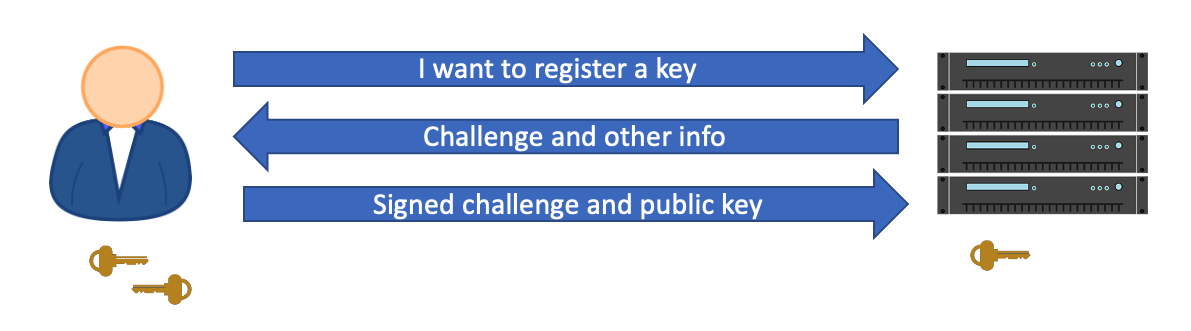


Figure 3: The registration process

## Authentication Process

At a later time, the user lands on the site and wishes to authenticate themselves. The server communicates back to the user a randomly generated challenge, which is just a string, and a list of credential IDs for the user.

Why are there multiple credential IDs? It’s because you want to support more than one key per user, just in case one key gets lost. Or perhaps one key lends you a greater level of access than the other.

The user now receives the challenge. At this point, the user’s computer verifies the server identity, and uses the credential ID to find the appropriate key. It then increments the counter so it stays in synch with the server, and it signs the challenge using the private key.

This challenge is then communicated back to the server, which then verifies the signed challenge with the public key, and increments the counter. This entire process can be seen in **Figure 4**.

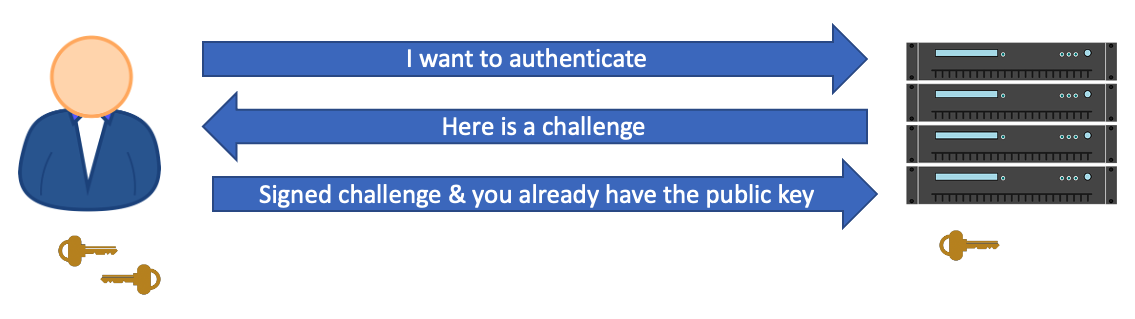


Figure 4: The authentication process

## Set Up FIDO2 Auth in Azure AD

Many websites support FIDO2. Microsoft has been a prominent participant in this ecosystem as well, along with Google, AWS and many others. Azure AD fully supports FIDO2 authentication for its users. This means that if your application uses Azure AD for authentication, you can make use of FIDO2 easily today. Additionally, you can also lock down access to critical resources such as the Azure Portal or Office 365 using FIDO2. Strong Auth that’s convenient for users is a win for everyone. If users use FIDO2, they are less susceptible to MFA fatigue and accidentally completing the MFA challenge. They’re also more secure as a result.

Let’s see how you can go about setting up FIDO2 authentication in Azure AD. To follow through these steps, you’ll need a physical FIDO2 key. I’m using a YubiKey.

Start by logging into the Azure Portal at portal.azure.com as a tenant admin, and then navigate to the Azure Active Directory blade. In that blade, look for Security, and navigate to Authentication Methods. Here, under policies, choose **FIDO2 security key**, and choose to enable it for select users. As you can see in **Figure 5**, I’ve chosen to enable it for testuser10.

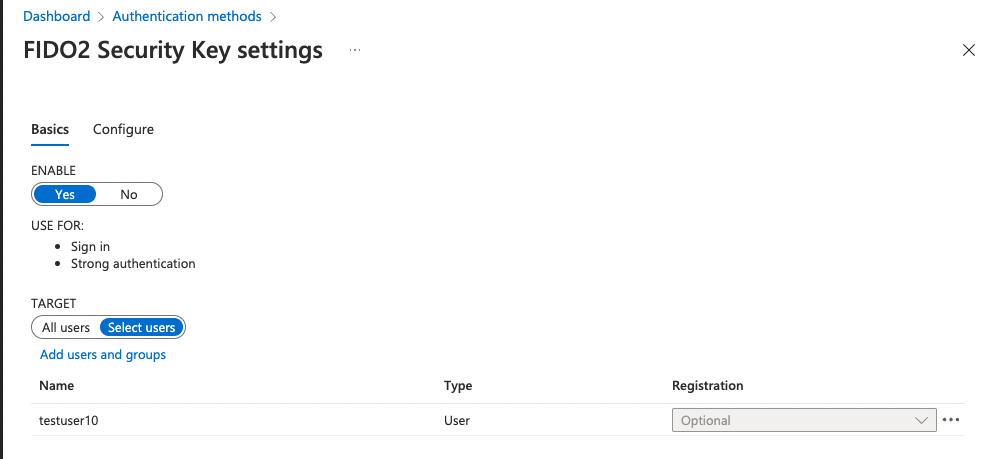


Figure 5: Enabling testuser10 to use FIDO2 as an authentication method

Note that, optionally, you can configure FIDO2 at the tenant level under the Configure tab in the same area. This can be seen in **Figure 6.** There are a number of settings here, and you can allow users to set up FIDO keys themselves. The other option is for an admin to set these keys for the user. You can choose to enforce attestation or not. Usually, attestation is useful in enterprise scenarios where you want to disallow certain keys from being used. However, using any key is better than a username password, so it’s okay to leave this set to “no”. You can restrict keys to certain well-known keys, so users don’t just buy their own keys and start registering them. You’d do this by adding AAGUIDs of the keys. This creates a huge management overhead, but it’s incredibly secure.

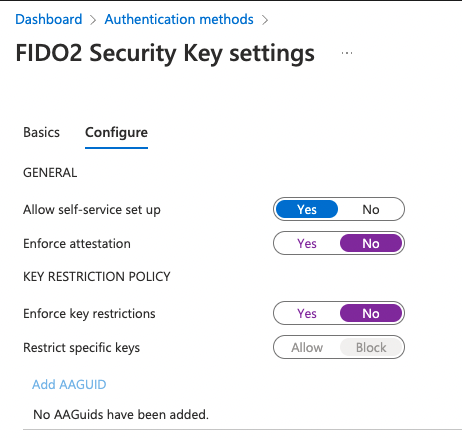


Figure 6: Configure FIDO2 settings at the tenant level.

As you can see from **Figure 6**, I’ve allowed self-service set up. Also, as you can see from **Figure 5**, I’ve enabled testuser10 to use FIDO2 as an authentication method. In a relatively modern browser (I’m using Chrome), in a non-private window, visit [https://myprofile.microsoft.com](https://myprofile.microsoft.com/) and sign in as the user you’ve enabled FIDO2 authentication for. In my case, that’s testuser10@sahilmalikgmail.onmicrosoft.com. Note that you may already have MFA enabled on this user, and that’s okay—one user can have numerous authentication methods.

Once signed in, visit the “Security” section and click on **Add method**, as shown in **Figure 7**.

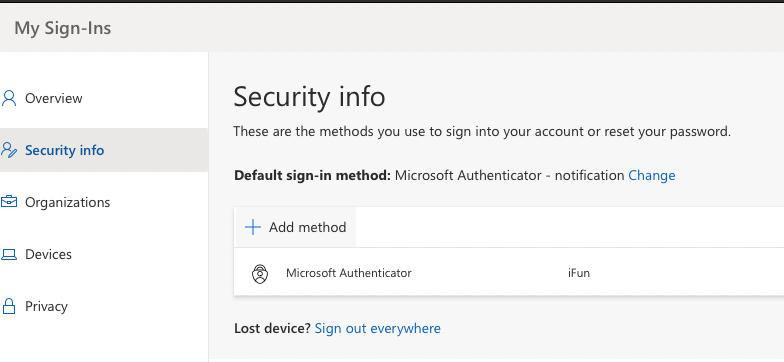


Figure 7: Add an authentication method for the user.

When prompted, choose **Security key** as the authentication method you’d like to use, and click **Add**. You’ll then be prompted to pick what kind of device you wish to use. This can be seen in **Figure 8**.

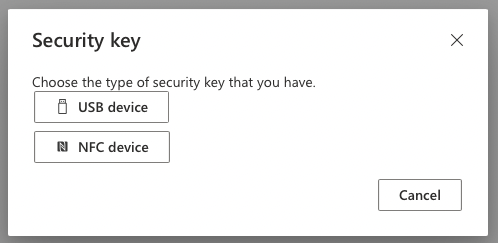


Figure 8: Security key type

I have a USB-C YubiKey, so I’ll pick “USB device”. Next, I’m shown a message saying that I should have my key ready, and when prompted, plug in the key and touch the key’s sensor or button to finish setting it up. As soon as I click Next, I’m redirected to a new window to finish set up.

Here, Azure AD shows you a message, but the real authentication dance is built into the browser using the WebAuthn protocol. The browser now prompts you to plug your security key in. This can be seen in **Figure 9**.

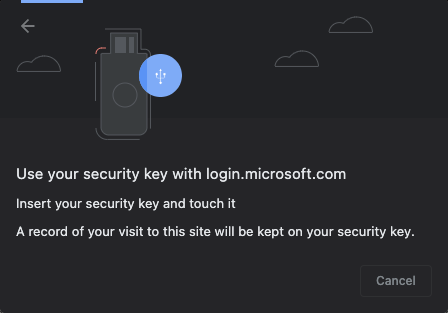


Figure 9: Chrome prompts you to plug your key in.

In my case, I use this key for numerous purposes, so it’s locked with a PIN. As soon as I plug it in, I’m asked to enter the PIN. Once I do that, the browser then asks me if I wish to use this key with the given website, which is, in this case, login.microsoftonline.com. Although it’s not very difficult to build FIDO2 authentication right on your website, most identity providers already support this, and delegating this responsibility to them is usually what we do these days anyway. This can be seen in **Figure 10.**

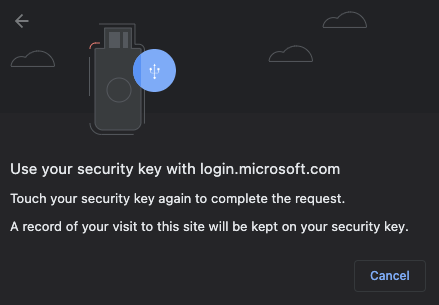


Figure 10: Chrome prompting you if you wish to use your FIDO2 key with AAD

To allow this key to be used with login.microsoftonline.com, you now have to touch the key. This proves physical possession of the key. Remember: The key, if cloned, can easily be detected using an ever-increasing counter.

As soon as I touch the key, I’m shown a third prompt, asking me if I allow the site (in this case AAD), to see the details of my key. Say **Allow**. This can be seen in **Figure 11**.

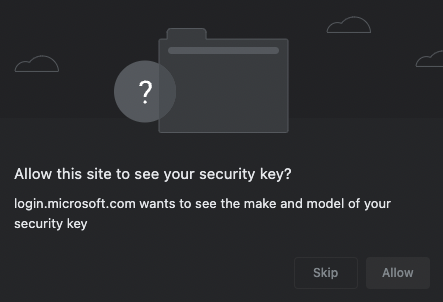


Figure 11: Sending information about the key

What’s interesting is that all this was built right into the browser. AAD has been patiently waiting for your key to be registered before moving further. At this point, your key is registered, and AAD asks you to name it. Giving it a meaningful name, I called mine **SahilKey**, and I soon see a message confirming that the key is ready for use. This can be seen in **Figure 12**.

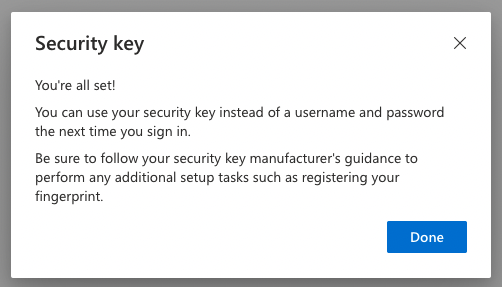


Figure 12: FIDO2 key is ready for use with AAD.

Now let’s see the sign-in experience.

Go ahead and sign out from myprofile.microsoft.com. You can do so by clicking the person-like icon on the top right-hand corner and choosing **Signout**. Now relaunch the browser, and visit any site protected by the same AAD. I’ll just use myprofile.microsoft.com again. Enter your username, (testuser10 in my case), and pick “Sign in with Windows Hello or a security key.”

Here’s a pet peeve. I’m on a Mac, and this system should be smart enough to not confuse the user with “Windows hello” on a Mac. But I digress.

I do have a security key, so I’ll click on that link. Chrome now shows me a bunch of options to sign in using. This can be seen in **Figure 13.**

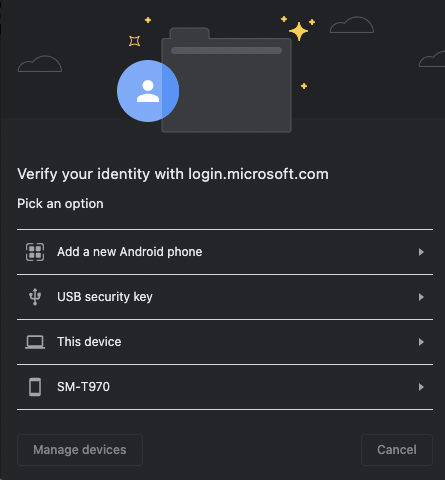


Figure 13: Many ways to sign in.

The exact list you see may be different. You may also be prompted for Bluetooth permissions at this point. I intend to use a USB YubiKey, so I pick **USB security key**. Now Chrome takes you through a simple sign-in process that involves touching the key, entering a PIN, and boom, you’re signed in.

See how easy that was? Not only that, when I signed in using FIDO2, I didn’t have to enter a password or remember a password, and the server’s workload is also greatly reduced. Plus, I never sent anything sensitive over the wire. It’s a win-win for all.

What if you lose the key? Well, you can always fall back to a back-up authentication method, such as an authenticator app in Azure AD. However, it’s also not atypical to register more than one security key.

## Summary

Passwords suck and I hate dealing with them. I like MFA, but it’s so inconvenient to deal with MFA sometimes. FIDO2 is supported by a number of organizations. And it really simplifies the log-in process while keeping my credentials secure. Some of the places I use FIDO2 already are Facebook, Twitter, GitHub, my Azure and Google accounts, plus a few others. If you wish to see who supports FIDO2, visit www.dongleauth.info. You’d be pleasantly surprised to see how many sites already support FIDO2.

No doubt this identity and security space will continue to evolve, but everyone has, at this point, unanimously agreed to kill passwords. If username password is your line of defense, I have bad news for you.

FIDO2 keys aren’t perfect. There’s a physical key that you must carry. But with platform authenticators, and technologies such as FaceID that are better at identifying individuals than fingerprints, the keys really are very compelling argument.

The best part is that almost every major identity provider already supports it, and it’s not hard to set up. If I use a username password as the only protection on a website, I just assume it to be insecure. I won’t use anything useful if it at least doesn’t support MFA. But MFA is inconvenient. So if a site gives me the option to use FIDO2 keys, that’s my solution.

How about you? Do you have any critical sites that you use just username password on?