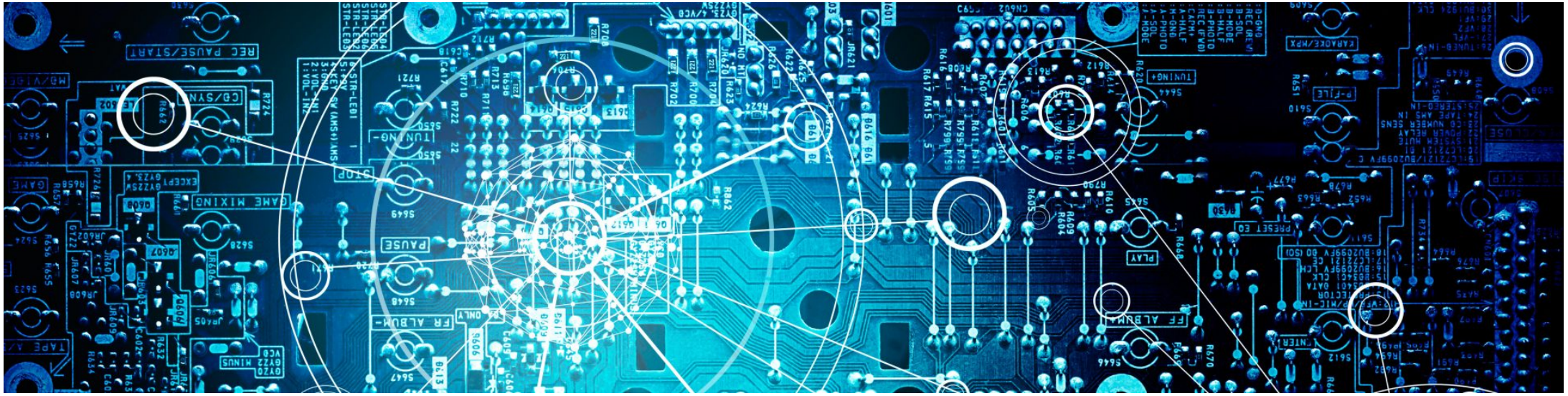
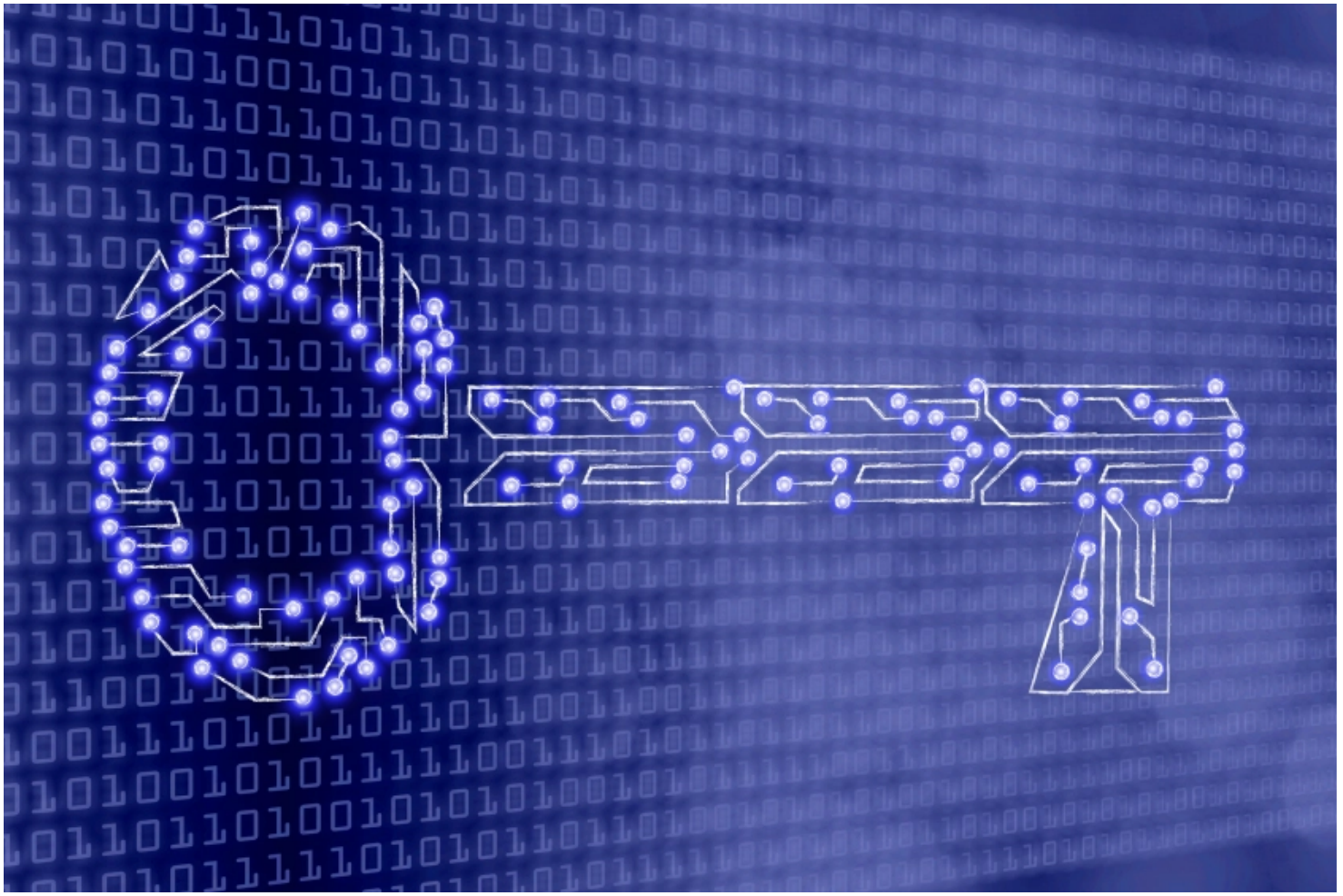
KUDELSKI SECURITY | RESEARCH

The Latest News from Research at Kudelski Security

HOME CATEGORIES





PASSKEYS – UNDER THE HOOD

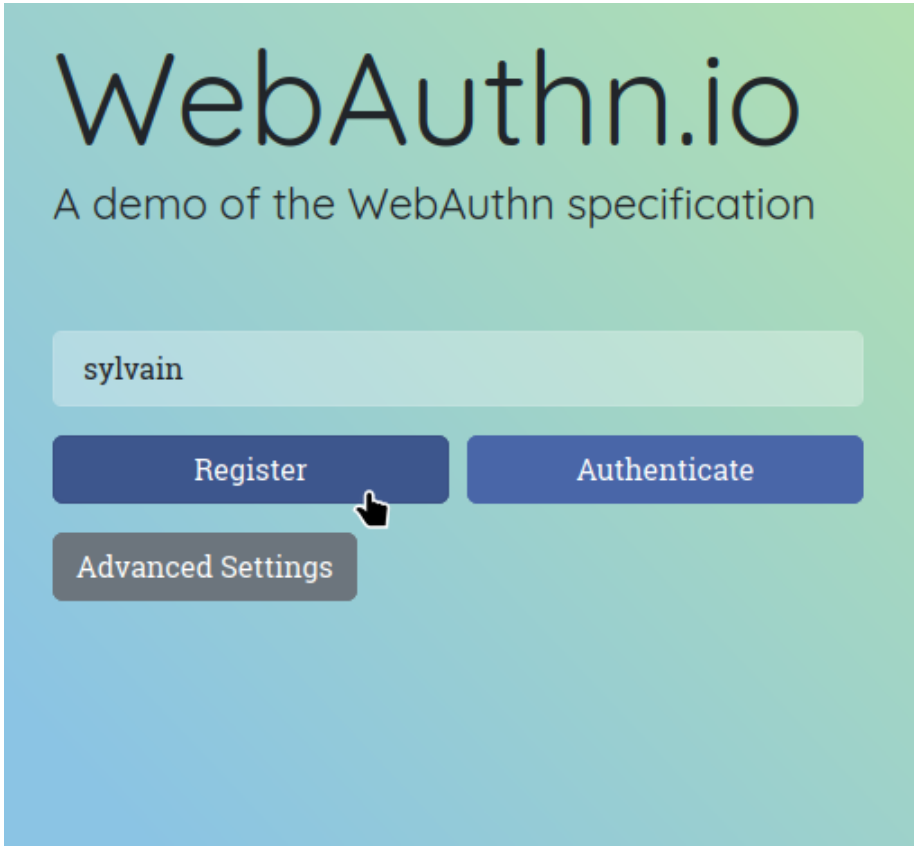
March 14, 2024 Sylvain Pelissier Authentication, cryptography Leave a comment

There was considerable attention around Passkeys last year. It was sometimes presented as the password killer technology. This came from the announcements of Apple and Google to support this technology and they were followed by many other services. The main advantages of passkeys compared to traditional passwords is their ability to be phishing resistant and server breach resistant.

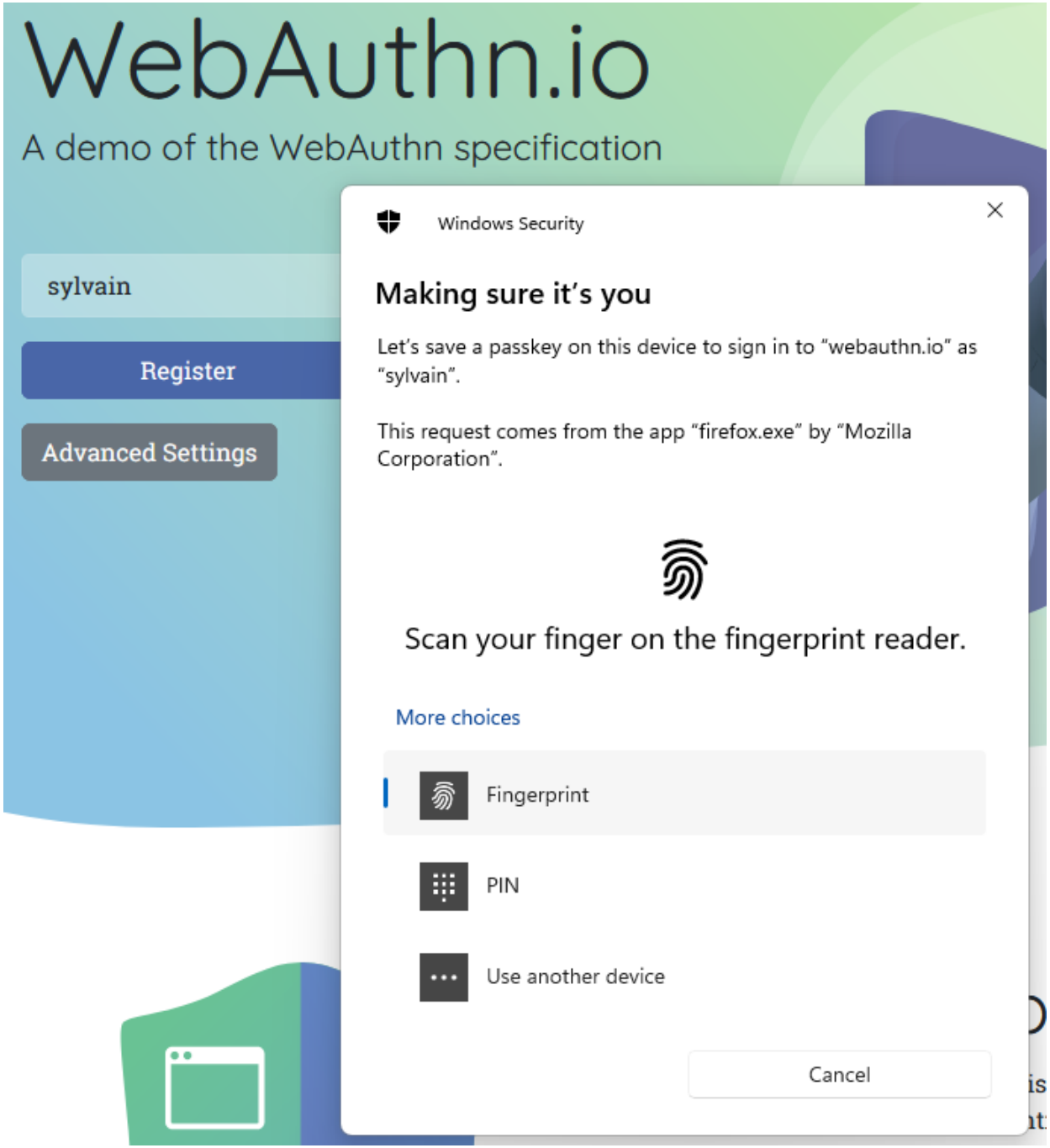
Another feature pushed by some actors is the ability to synchronize passkeys to multiple devices, even though this is not yet implemented everywhere. This would solve a big drawback of hardware security keys: the user credentials back-up. However the term passkey is confusing, many articles have explained how passkeys work conceptually but few explain how things work in practice and how they are implemented. In this blog we want to dig deeper and see how some of the existing solutions work in practice and to compare them to hardware security keys.

First, a passkey is a FIDO credential and it is created by a browser according to the WebAuthn specification. As detailed in a previous blog post, Webauthn specifies an API allowing a website to authenticate users using their browser. As a big picture, a service or a website (called relying party in Webauthn) authenticates a client by asking to sign a randomly generated challenge and other information with a client private key matching the public key known by the service. By design, the service will only store a public key and thus, if at some point it is breached, it cannot leak any information about the user private key. This feature is a big advantage when compared to traditional password authentication. In addition, the service address is included in the signature by the browser, therefore it thwarts phishing attacks.

As an example, the website webauthn.io allows for testing passkey creation.



If we click on “Register” button, the browser will start the credential creation. Practically speaking, the navigator.credentials.create() function is called to generate an asymmetric key pair for the service. Under compatible Microsoft Windows operating systems and browsers like Firefox, the following pop-up appears:



It asks us to enter our fingerprint or our PIN code to validate that we want to create a credential for this service. To have a glimpse of what is actually happening, we can open the browser console (F12

key) and check messages:

```
REGISTRATION OPTIONS
{
  "rp": {
    "name": "webauthn.io",
    "id": "webauthn.io"
  },
  "user": {
    "id": "c3lsdmFpbg",
    "name": "sylvain",
    "displayName": "sylvain"
  },
  "challenge": "5MvoufqYlltIT9JaQFMGG83ej7yeHqxOYmzE0vFkzVs2bIJEesg7zGoYi(
  "pubKeyCredParams": [
    {
      "type": "public-key",
      "alg": -7
    },
    {
      "type": "public-key",
      "alg": -257
    }
  ],
  "timeout": 60000,
  "excludeCredentials": [],
  "authenticatorSelection": {
    "residentKey": "preferred",
    "requireResidentKey": false,
    "userVerification": "preferred"
  },
  "attestation": "none",
  "hints": [],
  "extensions": {
    "credProps": true
  }
}
```

The service (or relying party) **webauthn.io** requires the generation of a public key with algorithms -7 and -257, meaning ECDSA with SHA-256 or RSASSA-PKCS1-v1_5 with SHA-256. As soon as we have

scanned our fingerprint or entered our PIN code, we have the freshly generated public key in the console:

```
REGISTRATION RESPONSE
{
  "id": "j5MX4uBITwi0zQBMyu5CaQ",
  "rawId": "j5MX4uBITwi0zQBMyu5CaQ",
  "response": {
    "attestationObject": "o2NmbXRkbm9uZWdhdHRTdG10oGhhdXRoRGF0YViUdKbqkhP
    "clientDataJSON": "eyJ0eXBlIjoid2ViYXV0aG4uY3JlYXRlIiwiY2hhbGxlbmdlIj
    "transports": [
      "internal"
    ],
    "publicKeyAlgorithm": -7,
    "publicKey": "MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEc9C6bLjbr1myHSzFFrU
    "authenticatorData": "dKbqkhPJnC90siSSsyDPQCYqlMGpUKA5fyklC2CEHvBFAAA
  },
  "type": "public-key",
  "clientExtensionResults": {},
  "authenticatorAttachment": "cross-platform"
}
```

The browser also answers with a random id which allows for registering several passkeys for the same login. The private key is stored on the user side exclusively and therefore cannot be leaked by the server. Later, when the user authenticates on the same service, the function navigator.credentials.get is called. Again the following pop-up from Windows appears:

In the console, the following message appears at the same time:

```
AUTHENTICATION OPTIONS
```

```
(index):639 {
  "challenge": "tGrdV4e5c2Ysb2ESzSOoje9nZk0ExA-RkG7j-rejmryRdPM02Mtr-f_gE
  "timeout": 60000,
  "rpId": "webauthn.io",
  "allowCredentials": [
    {
      "id": "j5MX4uBITwi0zQBMyu5CaQ",
      "type": "public-key",
      "transports": [
        "internal"
      ]
    }
  ],
  "userVerification": "preferred"
}
```

Essentially, the service is asking us to sign a challenge together with the service address and other information. The service also displays the credential ids allowed to login and their types. For a passkey it is labeled **“internal”**, while for a hardware security key it would be **“usb”**. Again we enter our PIN code and the service authenticates us. In the console, we have the following message:

```
AUTHENTICATION RESPONSE
{
  "id": "j5MX4uBITwi0zQBMyu5CaQ",
  "rawId": "j5MX4uBITwi0zQBMyu5CaQ",
  "response": {
    "authenticatorData": "dKbqkhPJnC90siSSsyDPQCYqlMGpUKA5fyklC2CEHvAFAAA
    "clientDataJSON": "eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY2hhbGxlbmdlIjoidE
    "signature": "MEUCIHbydreK68UUV7fEcFPDn3vEbmHL4AIyA6xYIWClv5GdAiEAidt
    "userHandle": "c3lsdmFpbg"
  },
  "type": "public-key",
  "clientExtensionResults": {},
  "authenticatorAttachment": "cross-platform"
```

We notice that the field “clientDaraJSON” which is part of the message signed has the “origin” field in its content:

```
>>> from base64 import urlsafe_b64decode
```

```
>>> urlsafe_b64decode("eyJjaGFsbGVuZ2UiOiJyUkYtSUxGNGN6dklObGpnbnhfUXVFd1dI
b'{"challenge":"rRF-ILF4czvINljgnx_QuEwWQsbTnkycdqq2cUVTR2SOsZfkliOYguq2Bj
```

This field is read by the browser directly and not given by the service. It allows to detect any phishing tentative, since the signature is not valid for another service.

Finally in Windows, the passkeys are secured by Microsoft Hello using the system TPM (if available). We can manage the saved passkeys in the Passkey settings menu:

The only problem so far, is that we do not have much information on how the passkeys are generated, stored and secured. We can neither export them to another device, for example, a Linux machine. We can get some additional information with certutil tool in command line:

```
> certutil -csp "Microsoft Passport Key Storage Provider" -key -v
Microsoft Passport Key Storage Provider:
S-1-12-1-3939627729-1327541301-18900911-3508007247/a946056c-151d-469b-8fb

ECDSA_P256
RSA
Key Id Hash(rfc-sha1): 32384a43c96daa0f4a46652d479a9b075227b0f8
Key Id Hash(sha1): ae7d6c5050694a46dbfcb94d8104d07e59252c11
Key Id Hash(bcrypt-sha1): d62022dbd4eef93cb7268d6abd59da6cf931aace
```

```
Key Id Hash(bcrypt-sha256): 0b0d814e0bcba1bbd31f1d4b9f362b621790694f3220c
Container Public Key:
0000  04 be a2 6b 2f 32 96 ab  75 b8 b7 c6 7e 5d 1b 93
0010  29 f8 79 4b 48 e4 85 22  06 2d 99 58 bc 1e d1 f3
0020  65 dc 11 98 85 17 5b 4a  6b c0 83 dc 3d 24 b3 3b
0030  0c dc ec fe 47 62 3c 53  75 7d 6f b4 31 82 54 a3
0040  ad
```

It displays the public key value and some other information but not much about how it is stored or encrypted. In addition, Microsoft does not allow synchronization of passkeys with other devices. On Apple or Android devices, this feature is enabled. It solves one of the main problems of previous security keys which was the user back-up. However, this may lock the user to a specific vendor because passkeys are not synchronized between devices of different ecosystem like Apple and Google. For example, users with an Apple laptop would not be able to retrieve their passkeys on an Android phone. With hardware security key, since the private key is not accessible anytime, a second hardware security key needs to be enrolled for each services in case the first one is broken or lost. This creates a big drawback for such devices.

On another hand, the security model has changed. With a security key, the private key is stored inside a secure element and an attacker with physical access to a security key would not be able to recover the private key value. With passkeys, the private key is decrypted and stored in memory at some point and thus maybe accessible by an attacker with access to the machine. This change of threat model needs to be known and chosen accordingly to the requirements of the user.

Bitwarden

To dig a bit deeper we can inspect the Bitwarden password manager which recently implemented the passkey support. The main advantage is that Bitwarden is open-source, therefore we may inspect the implementation. The browser extension can be downloaded from their website, but to be able to debug the extension we used the source code from the GitHub repository.

Lets see how the Bitwarden browser extension works. As soon as the extension is installed in the browser, when we browse to a service using passkeys we see the extension intercepting the Webauthn calls and displaying its own pop-up allowing to save the passkey in Bitwarden.

Indeed, in the code we noticed that the Webauthn calls are overridden:

```
const browserCredentials = {
  create: navigator.credentials.create.bind(
    navigator.credentials,
```

```
  ) as typeof navigator.credentials.create,
  get: navigator.credentials.get.bind(navigator.credentials) as typeof nav
};

const messenger = ((window as any).messenger = Messenger.forDOMCommunicat

navigator.credentials.create = createWebAuthnCredential;
navigator.credentials.get = getWebAuthnCredential;
```

Now each time the browser calls navigator.credentials.create it ends calling the function createWebAuthnCredential which itself calls the function makeCredential. The previous browser function pointer is kept in browserCredentials in case the user chooses the option “hardware key”. In this case, the previous operating system passkey mechanism (like Microsoft Hello) or a hardware security key will be used.

If we set-up a breakpoint at the end of the makeCredential function we may inspect the FIDO2 credential created:

It is interesting to see how everything is generated in the case of Bitwarden compared to a hardware security key where information like the private key is never accessible. Finally, when the passkey is created, it is stored encrypted in the same way as the Bitwarden passwords. The passkeys are also synchronized to the Bitwarden server with a end-to-end encryption and may be accessible to other devices of any brand with Bitwarden installed. This slightly mitigates the problem of vendor lock-in as described previously. An additional interesting feature is that the private key can be exported from the vault in JSON format. This may allow using passkeys in another password manager:

We recover the same information as before with the breakpoint. We can verify that the private key stored in "keyValue" is indeed a valid ECDSA key:

```
>>> from base64 import urlsafe_b64decode
>>> from Crypto.PublicKey import ECC
>>> key = urlsafe_b64decode("MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBA(
>>> mykey = ECC.import_key(key)
>>> mykey
EccKey(curve='NIST P-256', point_x=9859877056943199504836753142060708547357
```

Similarly, when the browser calls navigator.credentials.get function in Bitwarden code, then the function getAssertion is called. When the signature is returned, we can verify its validity with the public key in

Python as well:

```
>>> from Crypto.PublicKey import ECC
>>> from Crypto.Hash import SHA256
>>> from Crypto.Signature import DSS
>>> signature = urlsafe_b64decode("MEYCIQCwDTCys2jgUyfnArlYrVeByRuasP8sjM
>>> authData = urlsafe_b64decode("dKbqkhPJnC90siSSsyDPQCYqlMGpUKA5fyklC2C
>>> clientDataJSON = urlsafe_b64decode("eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY
>>> clientDataHash = SHA256.new(clientDataJSON)
>>> h = SHA256.new(authData+clientDataHash.digest())
>>> verifier = DSS.new(mykey, 'deterministic-rfc6979', 'der')
>>> try:
...     verifier.verify(h, signature)
...     print("The message is authentic.")
... except ValueError:
...     print("The message is not authentic.")
...
False
The message is authentic.
```

To sum-up, we have seen how passkeys are used in practice and how they are implemented in the Bitwarden password manager. We noticed that the threat model has changed between hardware security keys and passkeys since at some point the user private key is present in the user's system for passkeys. Even if passkeys solved the user credential back-up problem, the threat model needs to be assessed according to the use cases.

Positive Hack Days V
June 26, 2015
In "Conferences and events"

Taking the (quantum) leap with go
May 27, 2021
In "quantum computing"

Wire Cryptography Audit (with X41 D-Sec)
February 9, 2017
In "Crypto"

CRYPTOGRAPHY | PASSKEYS | SECURITY | WEBAUTHN

« WCCA+CTF Info Page

A Look Into LockBit – PART 1 »

LEAVE A REPLY