Web Authn Reference Implementation

Julian Stampfli (julianjimmy.stampfli@students.bfh.ch)

February 4, 2019

Contents

1 Introduction

Passwords are the most commonly used way to authenticate a user. They are easy to implement and have been used since the beginning of the internet. People are used to passwords and use them everywhere, but most of them use weak ones. Remembering strong passwords is not easy; thus, most opt for easier passwords or password reuse them. The newest recommendations from the National Institute of Standards and Technology (NIST) are actually to use very long passwords that make sense for a human — for instance a sentence in a fictional language or with slang words. An appropriate password would then look something like 'awsumfurrycybercat', but not that. [1]

Even when the user employ strong passwords, it is hard to remember a new password for every site. Thus, they reuse passwords for multiple sites. Password reuse can lead to credential stuffing attacks. [2], [3] Those are two weaknesses that passwords always had. Firstly, used passwords are generally too weak; secondly, they are reused for multiple services.

FIDO2 is a specification that aims to increase security for end users by eliminating the weaknesses that come with passwords. The main advantages are strong privacy, privacy protection, multiple choices, cost-efficiency, and a layered approach. It tries to do that by using secure authenticators that are used instead of a password. Additionally to the already discussed attacks FIDO2 also protects against Man in the Middle (MitM) attacks. [4]

The main force behind FIDO2 is the Fast IDentity Online (FIDO) Alliance. Some notable members are Microsoft, Google, and Yubico. Microsoft, for instance, wants to use FIDO2 for the windows hello login. [5] With that force behind the standard has is implemented in all major web browser. [6] There are also already authenticators available from Yubico that can support it. [7]

This paper discusses the implementation for the replying party when the user already has a FIDO2 authenticator.

2 FIDO2

FIDO2 is an authentication standard which uses public key cryptography to authenticate the user. Compared to a password the user doesn't prove that he knows something but that he has something that he previously has registered with the application. It is also possible to secure this authenticator with a second factor like a pin or biometrics. The authenticator is an external device that interacts with the application through a protocol called Client-to-Authenticator Protocol (CTAP). CTAP is also part of the FIDO2 specification but this paper does not go into detail about how this protocol works. [8]

The Web Authentication (WebAuthn) specification forms the second part of FIDO2. This part deals with the Application that wants to authenticate the user. It contains two steps. Attestation which deals with registering a new user and assertion which handles the authentication of an existing user. Both functions are explained in the Section 3.

2.1 Main Ideas

Traditionally, when a user wants to register or authenticate, he supplies the application with a password. A shared secret that only the user is supposed to know. We already discussed how passwords usually lead to weak authentication. This issue is normally circumvented by adding a second factor to the authentication. With FIDO Universal 2nd Factor (U2F) this second factor would be robust, and the authentication would be resilient against scaled attacks. However, many sites offer weaker forms of second-factor authentication like codes via e-mail or SMS. Both of those methods are weak to message interception. [9]

By using certified authenticators, FIDO2 makes sure that a user needs to have access to his authenticator and be present when the authentication is triggered. With this, an attacker could still steal the physical authenticator and authenticate with it. However, this kind of physical attack can't be used to steal the authenticator for a large group of people. Additionally, once the user notices that his authenticator has been stolen, he can inform the service provider and disable this authenticator for future authentications. Moreover, to secure authenticators that are left in the USB slot of a machine, FIDO2 requires the user to be present during the authentication. With this, an attacker can't merely put some malware on the user's computer and trigger the login.

2.2 Authenticator

FIDO2 authenticators are certified devices that support the FIDO2 standard. They must at least fulfill level 1 of the certification levels. 3 levels can be achieved that all build on each other plus one for additional security. New authenticators need first to get a functional certificate to profe that they are interoperable with other clients and servers.

The authenticator used in this reference implementation is a Yubikey 5 which supports an interface for FIDO2 via Universal Serial Bus (USB) and via Near Field Communication (NFC). In this paper, only the USB interface is used.

2.2.1 Level 1

Level 1 is the basic level. It offers protection against basic, at-scale attacks. [10]

This level can range from an authenticator running in a High Level Operating System (HLOS) without effective protection against most other applications to an authenticator being implemented completely in an Allowed Restricted Operating System (AROE). Most authenticators that are used should belong to a security level above 1 to ensure the security of the authenticator and the authentication. [10]

2.2.2 Level 2

Level 2 protects against basic, scalable attacks. [11]

This level needs an authenticator to run in a FIDO AROE and needs to support allowed cryptography lists. Examples that would not achieve this level are: [11], [12]

1. Authenticator that doesn't support attestation
2. Authenticator that is an implementation without a restricted operating environment

2.2.3 Level 3

Level 3 protects against enhanced-basic effort software and hardware attacks. The fido alliance provides some examples that would pass level 3. [13]

Some simplified examples are:

1. Authenticator that can't be easily opened implemented on a CPU with RAM encryption and integrity check.

2.2.4 Level 3+

Level 3+ protects against moderate or high effort software or hardware attacks. Thus an attacker should be hindered from performing attacks on a chip level. [14]

2.3 FIDO2 vs FIDO U2F

FIDO U2F is the predecessor of FIDO2. Where FIDO2 covers more usecases than FIDO U2F. Where FIDO U2F is only for second factor authentication, FIDO2 can be used as a second factor but also for passwordless authentication and local multifactor authentication. [4]

It is possible to use a FIDO U2F token to authenticate with a FIDO2 replying party and vice verca. This mapping is mainly done in the CTAP version 1 for FIDO U2F and version 2 for FIDO2. In the FIDO2 version there is a mapping defined which can map a response of a FIDO U2F authenticator to a response of a FIDO2 authenticator. This mapping for the authentication can be seen in Figure 1. The full description is available in the CTAP specification. [4], [15]

3 Replying Party Reference Implementation

The replying party is one of the main components of the WebAuthn standard and the main component of this paper. It includes a sample implementation of the replying party meant as a reference. This part deals with the two phases attestation and authentication. First, there is a short section dealing with the setup of the reference code.

3.1 Environment

The reference code is written in Java and uses Maven for dependency management. To get the REpresentional State Transfer (REST) Application Programming Interface (API) set up Spring Boot is used such that most of the non-relevant code isn't needed.

The data from the registrations are stored in a set to decrease the non-relevant code even more. Additionally, there are some data classes which are used within the application.

The frontend part is from the FIDO Alliance demo [16]. Additionally, some inspiration was taken from the Yubico implementation of the Java-Webauthn-Server also available on Github [17].

For testing the YubiKey 5 was used [7] which is reflected in the code. There are multiple ways with which the standard can be implemented. Only the way to implement it with a Yubikey 5 is available in the reference code. The other ways are discussed in their paragraphs.

The most important classes are the LoginController, the RegistrationController, and the AuthenticatorDataParser. Those are supposed to be references for when one wants to implement the replying party oneself. Those classes are discussed in depth in the following sections.

For testing the replying party it is important to generate a valid certificate and to communicate with a valid domain. If the replying party listens to localhost the response is invalid. The same happens when the communication is not secured with a valid certificate and in a Transport Layer Security (TLS) connection.

3.2 Registration / Attestation

The registration which is reflected in the RegistrationController represents the process a user has to go through to register with the application using a FIDO2 authenticator. It consists of two main steps. First, the replying party has to create a JavaScript Object Notation (JSON) that is accepted by the token in the CTAP protocol as a valid challenge. After the response from the client, the response endpoint needs to validate the input from the authenticator. This can be seen in Figure 2.

3.2.1 Create registration

The create registration step consists of creating a JSON which represents a valid challenge. This JSON is sent to the browser where it is handed to the call 'navigator.create()'. The response is sent to the /response endpoint in the replying party.

A valid JSON representation of this response can be seen in Listing 1. A description to the important entries follows:

```
{
  "public-key": {
    "challenge": "UvwcjUM5UOASx2AK9MkVbXSmNoDc_AfhNMkDzhvFbXo",
    "fidoResponse": "direct",
    "rp": {
      "name": "BFH",
      "id": "dev.webauthn.demo"
    },
    "user": {
      "id": "lPVdjYV97NDuLLMgJMyg4PYWU9R9VvOZL1hrTN8_MlU",
```

```
      "name": "test",
      "displayName": "test"
    },
    "pubKeyCredParams": [
      {
        "type": "public-key",
        "alg": -7
      }
    ],
    "attestation": "direct",
    "timeout": 60000,
    "errorMessage": "",
    "status": "ok"
  }
}
```

Listing 1: Registration JSON

challenge
: The challenge to be signed by the Authenticator. Must be unique for each call and base 64 encoded.

fidoResponse Should be direct.

rp.id
: Must reflect the domain over which you communicate

user.id Must be unique and random.

pubKeyCredParams.type Should be public-key.

pubKeyCredParams.alg Must be a valid CBOR Object Signing and Encryption (COSE) (COSE) algorithm. [18]

attestation Should be direct.

timeout Should be defined.

3.2.2 Verify registration

To verify the response of an authenticator the specification provided 19 steps. The implementation closely followed those steps by creating a method for each. If something went wrong it will throw an exception and return with an error code. The following paragraphs will provide some further documentation and challenges that were faced while implementing the reference. The initial response can be seen in Figure 2.

```
{
  "rawId": "qKk7sFYyBRex6K6twW5_5UHy-2DwdDXX7lA...",
  "response": {
    "attestationObject": "o2NmbXRmcGFja2VkZ2F0dFN0bXSjY2FsZy
       ...",
    "clientDataJSON": "eyJjaGFsbGVuZ2UiOiJCbHpxbHpSNVVD..."
  },
  "getClientExtensionResults": {},
  "id": "qKk7sFYyBRex6K6twW5_5UHy-2DwdDXX7lA...",
  "type": "public-key"
}
```

Listing 2: Create Response JSON

Step1
In Java with Spring Boot, the first step is mostly not needed. But the check that the needed fields in the JSON are present can be done at this point.

Step2
In step 2 the clientDataJSON field has to be parsed. This is done by Base 64 URL decoding the clientDataJSON field and then representing the data as an Universal Transformation Format (UTF-8) string which can be parsed into JSON. Parsed it looks like in Listing 3

```
{
  "challenge": "D7WQryRg4-jfXvSNcw3a8YANinbL0yG4bdonqgoRigA",
  "new_keys_may_be_added_here": "do not compare clientDataJSON
     against a template. See https://goo.gl/yabPex",
  "origin": "https://dev.webauthn.demo:8888",
  "type": "webauthn.create"
}
```

Listing 3: Client Data JSON Parsed

Step3
Step 3 is trivial but important. The type of the parsed client data must be 'webauthn.create'

Step4
In step 4 we need to verify the challenge. In the code, this is done by searching for the user with this challenge.

Step5

Step 5 is trivial. The origin that is returned must match the origin of the replying party. For convenience in the code, just the presence of the origin is checked. It is important that in a production environment this doesn't suffice. This string needs to match the URL where it was called. This check offers protection against MitM attacks.

Step6

Step 6 is currently not covered by this documentation. It deals with token binding over the TLS connection.

Step7

In step 7 the SHA-256 hash for the signature is precomputed. It is important that the string value of the JSON property 'clientDataJSON' is compared, else the signature is incorrect.

Step8

Step 8 needs much work. First the 'attestationObject' from the 'response' has to be Base 64 URL decoded and then parsed into JSON using a Concise Binary Object Representation (CBOR) mapper. The result of this is shown in Listing 4. After that, the 'authData' field of the attestation data needs to be parsed. It is encoded in a specified format. The decoding of this is described in Section 3.4. [19]

Those two steps were split in the code such that the 'attestationData' is available for future use.

```
{
  "fmt": "packed",
  "attStmt": {
    "alg": -7,
    "sig": "MEQCIBRsm+gm5tY75S/uEkDb91bzq1LoDqSDyCw...",
    "x5c": [
      "MIICvDCCAaSgAwIBAgIEA63wEjANBgkqhkiG9w0BAQ..."
    ]
  },
  "authData": "kn8Lq9EV0MBhHa/k+ZEtK8Ne4ZJ+06YC/..."
}
```

Listing 4: Attestation Data Parsed

Step9

In step 9 the 'rpIdHash' from the decoded auth data is compared to the SHA-256 hash of the rp.id from the Listing 1. This process is used to verify again that the token has registered with the correct replying party. Even though this was tested already, it should be rechecked to protect against some forms of attack like MitM.

Step10
In step 10 the user present flag has to be verified. This process is one of the main features of FIDO2 as it ensures that a user was present when the attestation was created. However, this doesn't prove that the user was the user that he claims to be, but simply that he is a user. This step limits many kinds of remote attacks as they can't trigger the authenticator to set this flag to true. Physical attacks are still possible though.

Step11
In step 11 the user verified flag is checked. This verification is optional and doesn't necessarily need to be checked. Only if the authenticator needs to support it. If this is required, it is advisable to inform the user beforehand, and if the bit is not set, he should get informed that he should use a different token or that he needs to be verified on authentication.

Step12
Step 12 deals with extensions. Currently this implementation does not support extensions.

Step13
Step 13 verifies the 'fmt' statement. As the implementation only validates Yubikey 5 authenticators, it is set to the value that is given back by those which is packed.

Step14
In step 14 the signature is verified. For this, there are multiple paths defined by the format. With the packed format, there are multiple options of which Yubikey 5 uses the x509 cert. This certificate first needs to be parsed. After it has been parsed some attributes must be checked. Only after those steps, the signature is validated. This validation is done by first combining the authenticator data with the hash computed at step 7 and test this data against the signature value provided.

Step15
Step 15 validates the trust anchor, in other words, whether we should trust this authenticator or not. This process should be done by getting data from a FIDO metadata service. In the implementation, it is done by comparing the 'aaguid' from the authenticator to the one received from a Yubikey 5. This guid should be the same for each key of the same model.

Step16
In step 16 the trustworthiness of the certificate used above should be checked. Again as we only trust Yubico authenticators, the CA cert from Yubico is statically used.

Step17
Step 17 is testing for the same credential on multiple users. The behavior of this is not specific. Additionally while testing the same authenticator always returned a different credential id and public key. Still, the check is done by checking against the other users.

Step18
In step 18 the user is associated with the credential. It is added to that user and used later for an authentication. The state of the user is also set to register.

Step19
Step 19 only states that if step 16 failed, the ceremony should be failed. As this was already done at step 16, this step can be ignored.

3.2.3 Summary

Overall there are many steps needed for a successful ceremony. None of the steps should be skipped for convenience as all of them provide some protection against various forms of attack. Some checks are rather complex, and a real implementation should be wary of that. Maybe covering only one path is the simplest solution and fulfills all the requirements for a specific application. If that is the case only this way should be implemented. More complexity often only increases the risk of exposure.

3.3 Login / Authentication

The authentication is implemented in the LoginController. It deals with the process of authentication to a replying party by using an authenticator that has previously been registered. As with the registration, there are two main stages. The replying party again has first to create a JSON which contains a challenge. , and in a second step, the replying party has to verify that response by checking against the authenticator data that were registered before. The process is depicted in Figure 3.

3.3.1 Create authentication

Simmilar to the create registration section discussed above the create authentication step creates a JSON which is used to send to the authenticator. A valid JSON is shown in 5

```
{
  "challenge": "rfmx563tPscMUAKZdeyzttW4LQUU5JAqFJ8tP2diUzI",
  "rpId": "dev.webauthn.demo",
  "allowCredentials": [
    {
      "type": "public-key",
      "id": "NWIv8Cd3lXkE8r4_Ondf6fPDeYDEbuQSNWbR..."
    }
```

```
  ],
  "timeout": 60000,
  "errorMessage": "",
  "status": "ok"
}
```

Listing 5: Create Authentication JSON

challenge
: The challenge to be signed by the Authenticator. Must be unique for each call and Base 64 encoded.

fidoResponse Should be direct.

rpid
: Must reflect the domain over which you communicate

allowCredentials.type Should be public-key.

allowCredentials.id Should be the id the user used when registering.

timeout Should be defined.

3.3.2 Verify Authentication Assertion

The standard supplies a set of 18 steps to verify an authentication assertion. Should a step fail, a LoginFailedException is thrown which results in an error code in return. Before the first step, some serialization has to be done. The initial response is visible in Listing 6.

```
{
  "rawId": "NWIv8Cd3lXkE8r4_Ondf6fPDeYDEbuQSNWbRrDYmfweVCo...",
  "response": {
    "authenticatorData": "kn8Lq9EV0MBhHa_k-ZEtK8Ne4ZJ-06YC_t4
       YmMtA57gBAAAA7Q",
    "signature": "MEUCIAhdPPqzV69p2dFEXNHjkafVkELuCP1
       RgXdJbdKoYLz4AiEAm...",
    "userHandle": "",
    "clientDataJSON": "eyJjaGFsbGVuZ2UiOiJjYnVCTTVHVEZ0
       RFBjVzRCcVh5c0tkeWQ2..."
  },
  "getClientExtensionResults": {},
  "id": "NWIv8Cd3lXkE8r4_Ondf6fPDeYDEbuQSNWbRrDYmfweVCo...",
  "type": "public-key"
}
```

Listing 6: Attestation Response JSON

Step1
In the first step, the credential should be matched to the credentials requested. This match is done by first getting the user with the credential. Then the stored assertion request for this user is compared against the credential id given with the assertion.

Step2
The user has already been retrieved in step 1. In step 2 the user handle should be verified. The user handle is optional, but if it is present, it must reflect the user that was requested.

Step3
In step 3 the public key must be retrieved. It is stored with the user.

Step4
Step 4 deals with parsing of several fields.

Step5
Step 5 requires an UTF-8 decoding of the cData.

Step6
Step 6 requires a JSON parsing of the decoded 'cData'. Step 5 and 6 are to close together that the reference implementation combined them into one step which deals with both at the same time. The final clientDataJSON can is displayed in Listing 7.

```
{
  "challenge": "pT5Z7r07quoiDUbLxBGN1I9sy-W8I7Jcm0SfailZ3Lc",
  "origin": "https://dev.webauthn.demo:8888",
  "type": "webauthn.get"
}
```

Listing 7: Attestation Client Data JSON

Step7
In step 7 the type of the client data must be set to 'webauthn.get'.

Step8
In step 8 the challenge has to be verified to be sure that the correct challenge was responded.

Step9
In step 9 the origin has to be verified. Again as in the registration process, this is done by checking if the origin contains the domain. In a productive environment, this should be done by comparing the expected URI.

Step10
Step 10 deals with the token binding over TLS which is optional and currently not supported in the reference implementation.

Step11
In step 11 the rpIdHash from the 'authData' must equal to the SHA-256 hash of the RP ID. For this first, the 'authData' needs to be parsed.

Step12
In step 12 the user present flag is checked. This check is the same process as in the registration and must be verified to protect against remote attacks.

Step13
In step 13 the user verified flag is checked. This check is optional and requires an authenticator that supports it.

Step14
In step 14 the extensions are checked. This check is optional and is currently not supported by the reference implementation.

Step15
In step 15 the client data is hashed using SHA-256. The data provided needs to be the exact string that was received by accessing the clientDataJSON field.

Step16
For step 16 first the public key which is in the COSE format needs to be parsed. Then the binary representation of the auth data and the hash calculated in step 15 are concatenated which represents the signed data. Then the signature value parsed in step 4 can be checked against the signed data using the parsed public key.

Step17
Step 17 is needed to check for cloned authenticators. If the sign count is present, it should always increase. It is important to verify this behavior to avoid cloned authenticators.

Step18
Step 18 states that should any step above have been invalid; the authentication ceremony must be failed.

3.3.3 Summary

After already knowing what happens in the registration the authentication is very similar as many steps exist in both ceremonies. The steps of the assertion sometimes can't be represented correctly in code because with java it is sometimes easier to combine two

steps into one. However, it is important that all of the steps are followed as required. Missing some steps could lead to broken authentication.

3.4 Auth Data

In both the registration and authentication ceremony, there is one step where the authenticator data must be parsed. Thus this was done at a central place called the 'AuthenticatorDataParser'. This data comes in binary and needs to be parsed according to the specification. The first 32 bytes generate the rpIdHash, the next byte the flags sector. Then there are 4 bytes which represent the signature Counter. After that, there might exist some attestedCredentialData. A table with all the information can be seen in Figure 4.

The attestedCredentialData if present contains 16 bytes of the 'aaguid', with which the authenticator model can be determined, 2 bytes which set the length of the following field which is the credential id and the rest which makes the credential Public Key. This attestedCredentialData is present for registration to verify the assertion. A table with all the information can be seen in Figure 5.

4 Conclusion

The FIDO2 standard increases the security of the end user by eliminating passwords with a MitM resistant factor. The authenticator can be supplied by many parties which makes wide adoption easier. However, currently, the implementation of the standard for a webpage owner is probably to complex. There are not yet convenient libraries for all languages available. This needs to change if wide adoption is to be achieved. Additionally, there are not yet many FIDO2 authenticators available on the market. Which results in an even poorer adoption rate.

However, the standard is considerably new and could be adopted widely. It has strong support from Microsoft, where it is implemented in Windows Hello, and other big technology companies. The implementation in Windows Hello could result in many companies supplying authenticators to their employees which would lead to a higher incentive to provide web applications with FIDO2 support. It is crucial that other big players indeed implement the standard for core products so that it truly can replace passwords.

An exciting application of this would be within companies to replace the need for passwords. Also, it could be used by governments in combination with an electronical IDentitification (eID) to give access to specific resources by using a physical card as an authenticator. This way every citizen of a country already has access to a FIDO certified authenticator which would lead to country-specific sites implementing the standard as an authentication method.

4.1 Further Research

This implementation only covered the Yubikey 5 authenticator. It would be interesting to see how other authenticators react. Also, it would be interesting to see how the

Yubikey 5 interacts via the NFC interface as this implementation was only tested using the USB interface of the authenticator.

Additionally, it only dealt with authentication and registration. It is essential to give the user the option to add multiple authenticators to his account such that he can recover his account should he lose his key. Additionally, it is vital that the user can revoke their authenticators. This possibility should be done by giving the user the option to set a name that is relevant to him for each authenticator. He should then be able to manage those authenticators. Either adding new ones or removing existing ones.

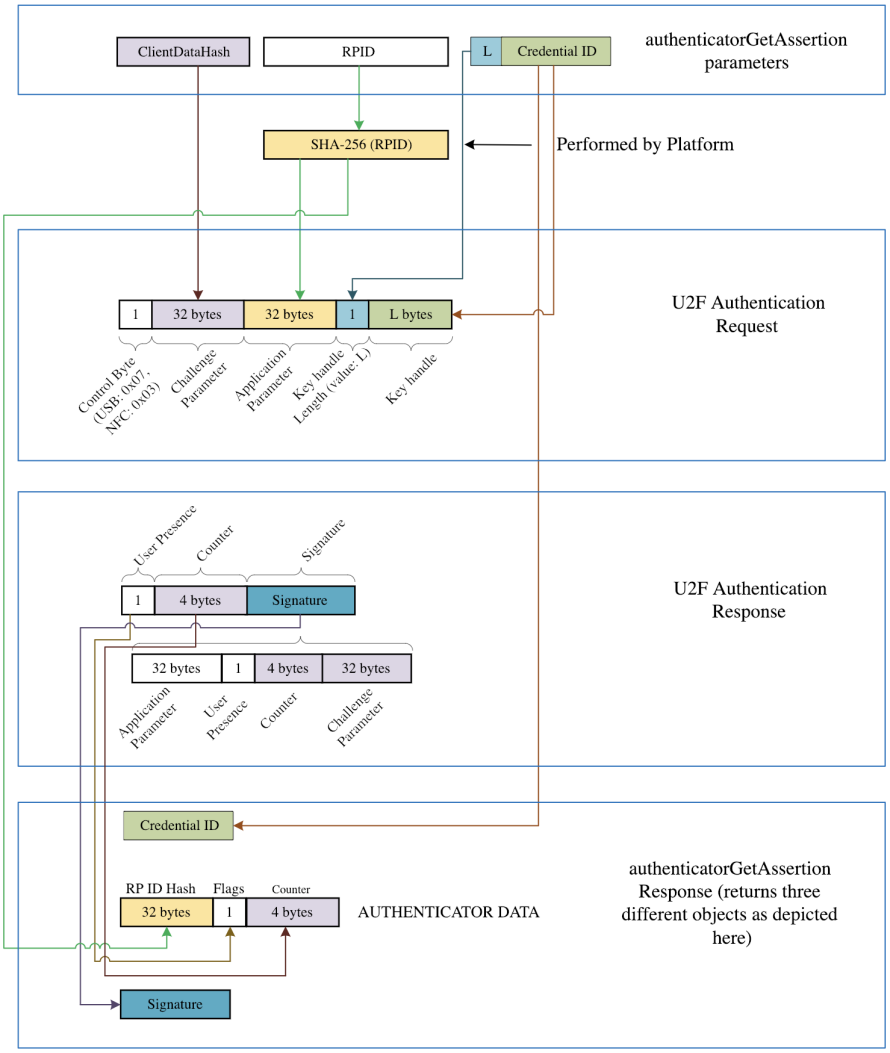

Figure 1: Authenticator Data

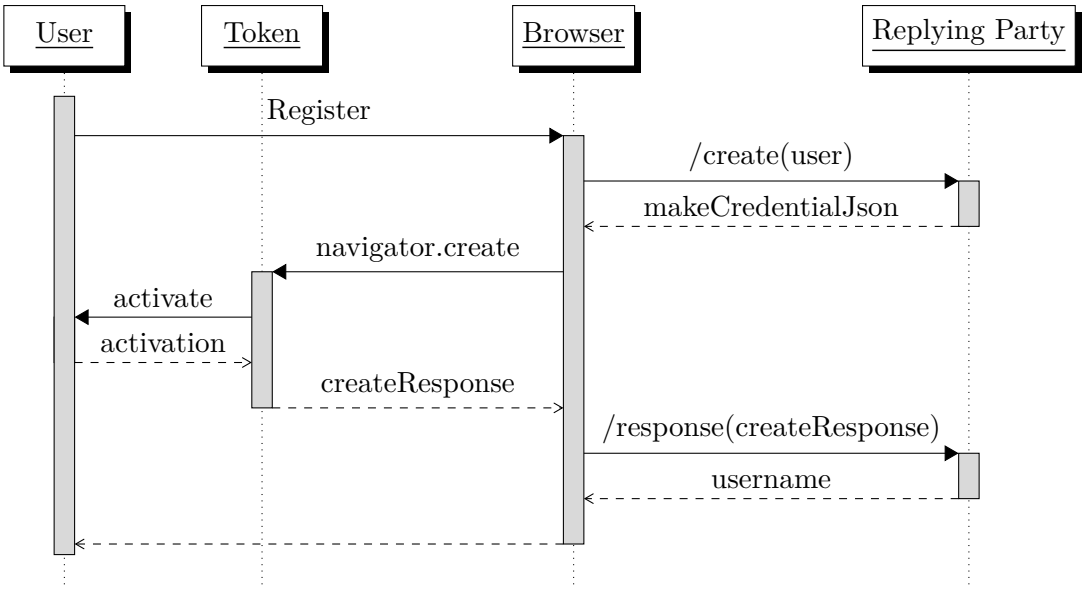

Figure 2: User Registration

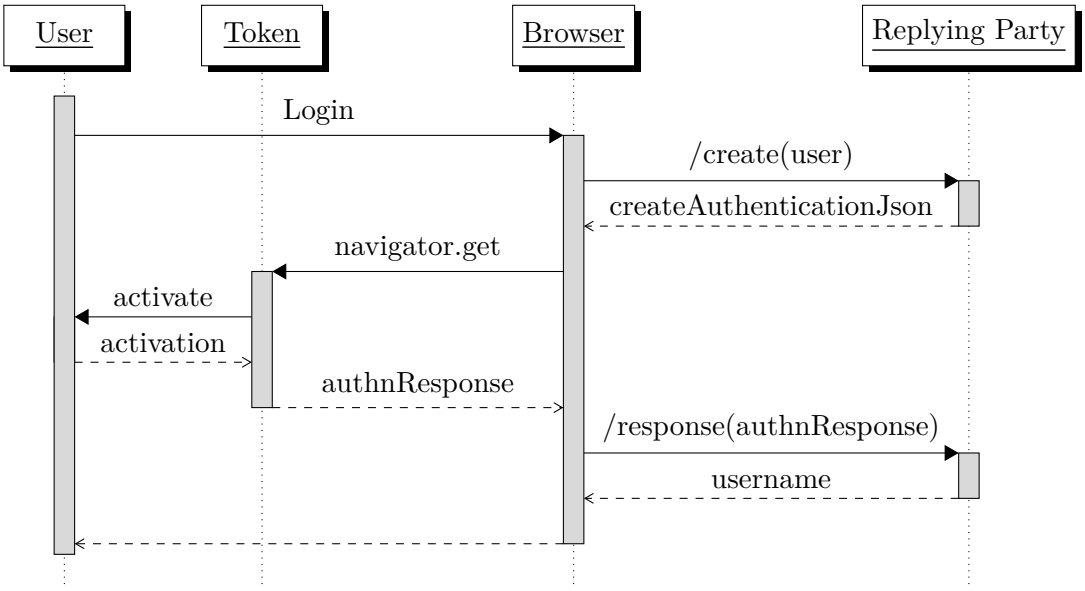

Figure 3: User Authentication

Name	Length (in bytes)	Description
rpIdHash	32	SHA-256 hash of the RP ID the credential is scoped to.

flags	1	Flags (bit 0 is the least significant bit):
		• Bit 0: User Present (UP) result.
		  ◦ 1 means the user is present.
		  ◦ 0 means the user is not present.
		• Bit 1: Reserved for future use (RFU1).
		• Bit 2: User Verified (UV) result.
		  ◦ 1 means the user is verified.
		  ◦ 0 means the user is not verified.
		• Bits 3-5: Reserved for future use (RFU2).
		• Bit 6: Attested credential data included (AT).
		  ◦ Indicates whether the authenticator added attested credential data.
		• Bit 7: Extension data included (ED).
		  ◦ Indicates if the authenticator data has extensions.

signCount	4	Signature counter, 32-bit unsigned big-endian integer.
attestedCredentialData	variable (if present)	attested credential data (if present). See §6.4.1 Attested Credential Data for details. Its length depends on the length of the credential ID and credential public key being attested.
extensions	variable (if present)	Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and authenticator extension outputs as values. See §9 WebAuthn Extensions for details.

Figure 4: Authenticator Data

Name	Length (in bytes)	Description
aaguid	16	The AAGUID of the authenticator.
credentialIdLength	2	Byte length **L** of Credential ID, 16-bit unsigned big-endian integer.
credentialId	L	Credential ID
credentialPublicKey	variable	The credential public key encoded in COSE_Key format, as defined in Section 7 of [RFC8152], using the CTAP2 canonical CBOR encoding form. The COSE_Key-encoded credential public key MUST contain the "alg" parameter and MUST NOT contain any other OPTIONAL parameters. The "alg" parameter MUST contain a `COSEAlgorithmIdentifier` value. The encoded credential public key MUST also contain any additional REQUIRED parameters stipulated by the relevant key type specification, i.e., REQUIRED for the key type "kty" and algorithm "alg" (see Section 8 of [RFC8152]).

Figure 5: Attested Credential Data

Glossary

API Application Programming Interface. 5

AROE Allowed Restricted Operating System. 3, 4

Base 64 Base 64 is a binary-to-text encoding that uses 64 ASCII characters to represent 6 bit. This results in an overhead of 2 bit per byte. There is also a Base 64 URL encoding which uses slightly different characters.. 7, 8, 11

CBOR Concise Binary Object Representation. 8

COSE CBOR Object Signing and Encryption (COSE). 6, 13

credential stuffing attacks Using credentials that were leaked on one site to access other sites. This is often an attack against passwords when passwords are reused. 2

CTAP Client-to-Authenticator Protocol. 2, 4, 5

eID electronical IDentitification. 14

FIDO Fast IDentity Online. 2–5, 9, 14

FIDO2 New standard from the FIDO Alliance which combines CTAP and WebAuthn. 2–5, 9, 14

HLOS High Level Operating System. 3

JSON JavaScript Object Notation. 5, 7, 8, 10, 12

MitM Man in the Middle. 2, 8, 14

NFC Near Field Communication. 3, 15

NIST National Institute of Standards and Technology. 2

REST REpresentional State Transfer. 5

SHA-256 Secure Hash Algorithm 2 with 256 Bit Output. 8, 13

TLS Transport Layer Security. 5, 13

U2F Universal 2nd Factor. 3, 4

USB Universal Serial Bus. 3, 15

UTF-8 Universal Transformation Format. 7, 12

WebAuthn Web Authentication. 2, 4

References

[1] M. Garcia, *Easy ways to build a better p@$5w0rd*, NIST, 2018. [Online]. Available: https://www.nist.gov/blogs/taking-measure/easy-ways-build-better-p5w0rd.

[2] *52% of users reuse their passwords*, Panda Security, 2018. [Online]. Available: https://www.pandasecurity.com/mediacenter/security/password-reuse/.

[3] R. Munroe, *Password reuse*, 2014. [Online]. Available: https://xkcd.com/792/.

[4] *What is fido2?* Yubico, 2019. [Online]. Available: https://developers.yubico.com/FIDO2/.

[5] R. Manning, *Passwordless login with the yubikey 5 comes to microsoft accounts*, Yubico, 2018. [Online]. Available: https://www.yubico.com/2018/11/passwordless-login-with-the-yubikey-5-comes-to-microsoft-accounts/.

[6] *Fido2 browser support, new certified products continue momentum towards passwordless future*, fidoalliance, 2018. [Online]. Available: https://fidoalliance.org/fido2-browser-support-new-certified-products-continue-momentum-towards-passwordless-future/.

[7] J. Chong, *Introducing the yubikey 5 series with new nfc and fido2 passwordless features*, Yubico, 2018. [Online]. Available: https://www.yubico.com/2018/09/introducing-the-yubikey-5-series-with-new-nfc-and-fido2-passwordless-features/.

[8] *7. interoperating with ctap1/u2f authenticators*, fidoalliance, 2018. [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-client-to-authenticator-protocol-v2.0-id-20180227.html.

[9] D. Price, *It's time to stop using sms and 2fa apps for two-factor authentication*, 2018. [Online]. Available: https://www.makeuseof.com/tag/two-factor-authentication-sms-apps/.

[10] *Authenticator level 1*, fidoalliance, 2019. [Online]. Available: https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-1/.

[11] *Authenticator level 2*, fidoalliance, 2019. [Online]. Available: https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-2/.

[12] *Fido authenticator allowed restricted operating environments list*, fidoalliance, 2019. [Online]. Available: https://fidoalliance.org/specs/fido-security-requirements-v1.0-fd-20170524/fido-authenticator-allowed-restricted-operating-environments-list_20170524.pdf.

[13] *Authenticator level 3*, fidoalliance, 2019. [Online]. Available: https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-3/.

[14] *Authenticator level 3+*, fidoalliance, 2019. [Online]. Available: https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-3-plus/.

[15] *Client to authenticator protocol (ctap)*, fidoalliance, 2018. [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-rd-20170927/fido-client-to-authenticator-protocol-v2.0-rd-20170927.html#interoperating-with-ctap1-u2f-authenticators.

[16] *Fido alliance webauthn demo*, fidoalliance, 2018. [Online]. Available: https://github.com/fido-alliance/webauthn-demo.

[17] *Java-webauthn-server*, Yubico, 2019. [Online]. Available: https://github.com/Yubico/java-webauthn-server.

[18] *Cbor object signing and encryption (cose)*, IANA, 2019. [Online]. Available: https://www.iana.org/assignments/cose/cose.xhtml.

[19] *7.1. registering a new credential*, W3C, 2019. [Online]. Available: https://www.w3.org/TR/webauthn/#authenticator-data.