Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

DebAuthn: a Relying Party Implementation as a WebAuthn Authenticator Debugging Tool

Estudante: Martiño Rivera Dourado
Dirección: José M. Vázquez Naya
Dirección: Marcos Gestal Pose

A Coruña, setembro de 2020.

A meu pai e a miña nai

Acknowledgements

Firstly, I would like to show gratitude to my tutors José and Marcos for their time in mentoring and guiding this project. I also want to acknowledge the RNASA-IMEDIR group and the CITIC Research for supporting the project with their resources. More personally, I would like to show my gratitude to my close friends for their help and care during my studies and primarily during these last months. Many thanks also to my classmates who were always willing to give me a hand. Finally, I could not finish these acknowledgements without expressing my gratitude to my parents and my sister for the great support through all these years.

Abstract

Passwords as an authentication method have become vulnerable to numerous attacks. During the last few years, the FIDO Alliance and the W3C have been working on a new authentication method based on public key cryptography and hardware authenticators, which avoids attacks like phishing or password stealing. This degree thesis focuses on the development of a web application as a flexible testing and debugging environment for developers and researchers of the protocol, still under development. Moreover, the developed tool is used for testing the most relevant hardware authenticators, showcasing their main characteristics.

Resumo

Os contrasinais como método de autentificación volvéronse vulnerables a numerosos ataques. Durante os últimos anos, a FIDO Alliance e a W3C estiveron traballando nun novo sistema de autentificación baseado en criptografía de chave pública e autentificadores hardware, o que evita ataques como phishing ou roubo de contrasinais. Este traballo de fin de grao céntrase no desenvolvemento dunha aplicación web como un entorno flexible de probas e depuración para desenvolvedores e investigadores do protocolo, aínda en desenvolvemento. Ademais, a ferramenta desenvolvida é usada para probar os autentificadores hardware máis relevantes, mostrando as súas características principais.

Keywords:

- WebAuthn
- Authentication
- Debugging
- Testing
- Authenticator

Palabras chave:

- WebAuthn
- Autentificación
- Depuración
- Evaluación
- Autentificador

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Authentication is one of the most critical parts of an application. It is a security service that aims to guarantee the authenticity of an online identity. This can be done by using several security mechanisms but currently, without a doubt, the most common is the username and password method. Although authentication based on username and password is easy for a user to conceptually understand, it presents many security problems.

One of the problems is password stealing through phishing. This attack consists in sending emails that include a hyperlink that, in general, directs the victim to a clone of the official web page, misleading the user to introduce their credentials directly to the attacker's hands. Furthermore, user passwords can also be compromised at the client side by the use of malware or hardware devices. Most of the techniques that capture the password on user input involve the keyloggers, which can be software, hardware, acoustic or even wireless tools [1].

It is a fact that information leaks often appear in the news. They usually happen once servers hosting web applications have been compromised. This leaked information usually includes password hashes, susceptible to be cracked using dictionary attacks [2]. All phishing, password stealing and cracking attacks entail a security risk for password-based authentication methods that leave the user unprotected and without control over their credentials.

This is the reason why there is the need for a solution that facilitates the credential management and protects users from phishing and password capture attacks. One of the approaches that big technology companies such as Facebook or Google started to implement is based on the use of physical authenticators [3], such as the Yubikey [4] or the Titan Security Key [5]. This idea evolved into a Web Authentication protocol, called WebAuthn, which was made public in March 2019 as a W3C Recommendation, result of some collaborations with the FIDO Alliance under the FIDO2 project.

The FIDO2 project aims to develop a new standard that constitutes an uncomplicated and phishing resistant authentication method, which returns the user the power over their credentials. The WebAuthn standard and, more often, its predecessor FIDO U2F, are already

implemented in several web services. Some of them are the Google account, the most outstanding cloud services such as Azure, and even the Github platform.

For configuring an authentication factor with the new standard, it is necessary to be in possession of an authenticator compatible with WebAuthn [6], usually in the form of an USB hardware device. Both for registering the device and logging in with it, the user can directly plug the device into the computer and declare user presence by tapping on a physical sensor. Therefore, this hardware authenticator can be treated as a physical token independent of a personal computer or any other device that needs user interaction to approve registration and authentication operations.

Although these standards have already been implemented in some platforms, it is notable the evolution within the authenticator model implementations. From the FIDO CTAP1/U2F standard in 2014 until the current CTAP2 [7], new functionalities have been added during these last few years. Moreover, despite the fact that the WebAuthn standard has become a W3C Recommendation last year, the consortium is already building a second version. This version is currently as a Working Draft and corrects and improves its first version. In fact, developers implementing WebAuthn in their systems are giving feedback to the W3C working group.

Because of this current standard adoption and improvement, an accessible and manageable test environment can be of use for developers. Mainly, a debugging tool would evade the need of an *ad hoc* implementation for reproducing an use case and, eventually, help the developer to guide their implementations. This aids projects planning to implement WebAuthn to test support of browsers and authenticators beforehand. For instance, a researcher would be able to test if both some concrete browser and an authenticator support resident credentials, which are defined in the new CTAP2 standard.

As a result of the current interest on the recent standards, there is also a need to audit browser implementations, new authenticator devices and the standard itself. During these years, some projects have started the development of many libraries supporting the standard and also some testing systems. Most of them are meant for the final user to test the functioning of a device and understand the new authentication flow, like the WebAuthn.io [8] tool. There also exist some applications like WebAuthn.me [9] which are more intended to test the protocol. However, the most relevant are linked with a commercial solution or it provides limited flexibility regarding the configuration of some configurations like the cryptographic algorithms.

This project will focus on WebAuthn and its main features, while incrementally implementing a useful tool for developers and researchers of the new protocol, named DebAuthn. Once implemented, it will be used to test the most relevant authenticators available on the market.

1.1 Objectives

The main objective of this work is the study of the WebAuthn protocol and the implementation of a Relying Party [1], in the form of a web application that can be used as a debugging tool for WebAuthn Authenticator Model compatible authenticators, meant to serve as a testing environment for researchers.

The specific objectives are the following:

- Design and implement a REST service for the access to Relying Party operations.
- Design and implement a user web interface, using client scripts executable in a web browser compatible with the WebAuthn standard.
- Design the tool such that it is easily extensible, enabling the possibility of adding functionaties that support new WebAuthn extensions or new attestation formats and types, as it is considered in the standard.
- Perform tests with different physical authenticators by using the implemented tool.

1.2 Structure of the degree thesis

According to the objectives, this thesis structures the content in a specific order, namely:

- **State of the Art**(chapter 2): covers the analysis of the WebAuthn standard while drawing an image of the projects and companies around WebAuthn, as well as analyzing tools similar to the developed one.
- **Planning and methodology** (chapter 3): explains the engeneering methodology and states the project planning.
- **Analysis** (chapter 4) and **Development** (chapter 5): they cover the full iterative and incremental development of the tool.
- **Testing and studying authenticators** (chapter 6): covers the testing of five hardware authenticators by using options of the developed tool. Also, it includes a study of other extra characteristics.
- **Results** (chapter 7): summarizes the main results according to the objectives.

[1] "The entity whose web application utilizes the Web Authentication API to register and authenticate users." [6]

- **Conclusion and future research** (chapter 8): includes the conclusion of the author and it enumerates some of the future research lines.

Chapter 2

State of the Art

The WebAuthn API [6] is the result of the collaboration of the FIDO Alliance and the W3C, although a new version is already being developed [10]. Both the FIDO Alliance and the W3C are joined efforts of projects and companies that have designed and developed their protocols and software solutions. The State of the Art explores the WebAuthn standard itself and some of the most relevant software and hardware implementations around WebAuthn, including similar tools to the developed one.

2.1 Existing standards

The FIDO2 project (see figure 2.1) is a joint effort between the W3C and the FIDO Alliance to develop standards for the public-key cryptography authentication [11]. It was born at the FIDO U2F standard in 2014, a standard that designed this type of authentication as a second-factor authentication method.

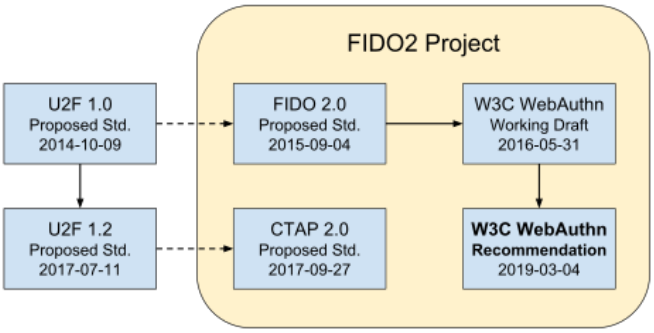


Figure 2.1: Evolution of the standards. The FIDO2 project.

Now, U2F has been renamed to FIDO CTAP1 and its successor, the FIDO CTAP2, was designed together with the W3C WebAuthn API. In 2019, the W3C reached a consensus publishing the W3C WebAuthn Recommendation. Nowadays, the consortium is developing a

second version of the standard, currently as a Working Draft.

- FIDO
  - FIDO U2F [12], renamed as CTAP1.
  - CTAP2 [7]
- WebAuthn W3C: API built into browsers to enable support for FIDO Authentication
  - W3C Recommendation: Level 1 [6].
  - Working Draft: Level 2 [10].

2.2 WebAuthn: a W3C recommendation

This section covers the concepts of the WebAuthn standard [6]. However, before getting into details, it is advisable for the reader to review some basic concepts on cryptography. This degree thesis does not cover these concepts as WebAuthn is a recent standard, subject to change, that involves a deep level of detail on cryptography. Some of them are digital signatures, public key cryptography and Public Key Infrastructure (PKI) certificates. However, Appendix C covers a brief insight on public key cryptography, together with external documentation.

2.2.1 General concepts

The WebAuthn API [6] was developed by the W3C aiming to create a common interface built on web browsers to access the public key credentials managed by an authenticator. The communication between the web browser and the authenticator is led to the CTAP protocol [7] that OS and/or web browsers will implement on top of transports like USB, BLE or NFC.

There are two ceremonies, one for registration called *Attestation* and other for authentication, named *Assertion*. Both of them are based on public key cryptography, where the *private key* is held by the authenticator and the *public key* is stored at the server for validation. All cryptographic operations are performed at the authenticator after the user's interaction, preventing a malware to trigger any operations without the user's awareness (see figure 2.2).

Relying Party (hereinafter, R.P.) is the name given to the service web application, including the browser scripts and the server application. On the other hand, the authenticator is a software or hardware implementation of the WebAuthn Authenticator Model, having to implement several operations compatible with the FIDO CTAP protocol.

The WebAuthn API is built on top of the W3C Credential Management API [13], used to request password credentials stored in a web browser. WebAuthn, instead of working with stored passwords of a browser, it uses the new API to request the browser for public-key credentials managed by an authenticator.

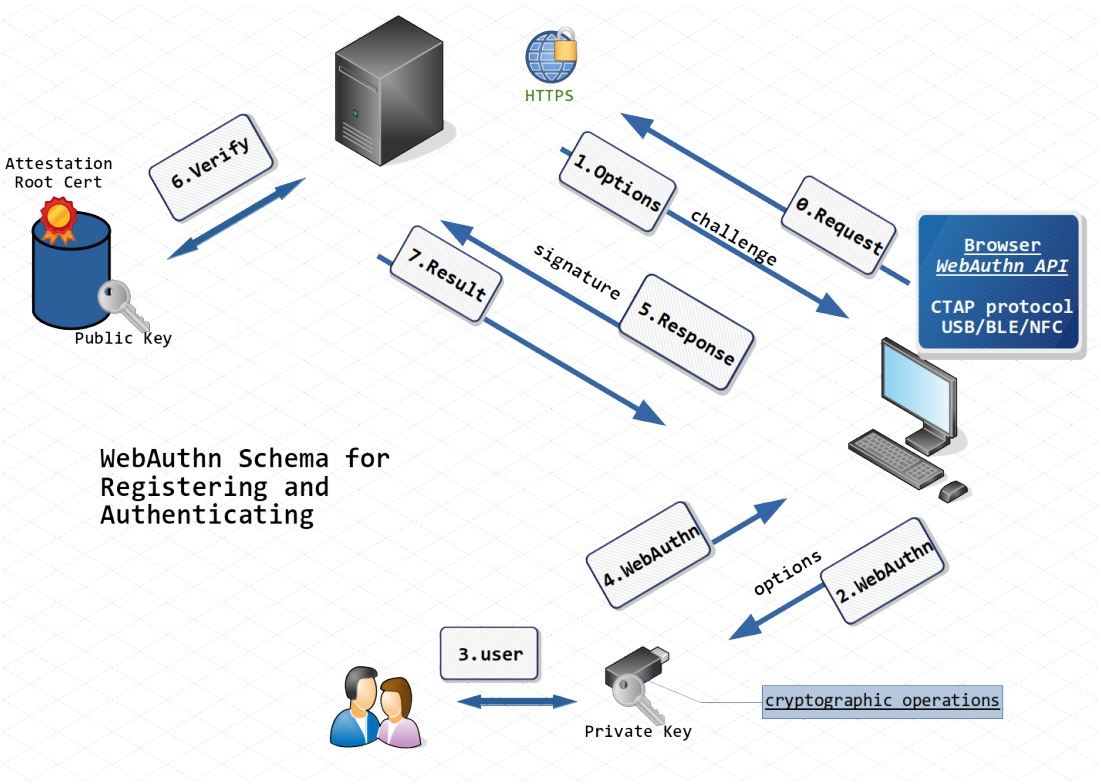


Figure 2.2: WebAuthn Schema for Registering (Attestation) and Authenticating (Assertion)

2.2.2 Attestation

Attestation is the process of registering a credential in the R.P. by using an authenticator. This credential will always be created by the authenticator and, in hardware authenticators, all cryptographic operations occur inside the key's hardware.

The WebAuthn API call for performing the Attestation operation is `navigator.credentials.create()`. This call is parametrized with the Attestation options, which include, among many other configurations, the challenge buffer, the user info and the R.P ID. This call obviously will be triggered by the user, requesting the R.P. JavaScript to initiate the process, requesting the options to their server.

Apart from the options, the call to the authenticator will also include some client data, being the most important the domain name of the web page, named *origin*. This is used afterwards by the R.P. to identify a possible phishing attack (see figure 2.3).

Once the authenticator receives the request by the `authenticatorMakeCredential()` CTAP protocol call, it performs the user presence or user verification and it creates a new public key pair. After creating this key pair, it will concatenate the client data with the authenticator data (the public key and other specific data). This results in the attestation ob-

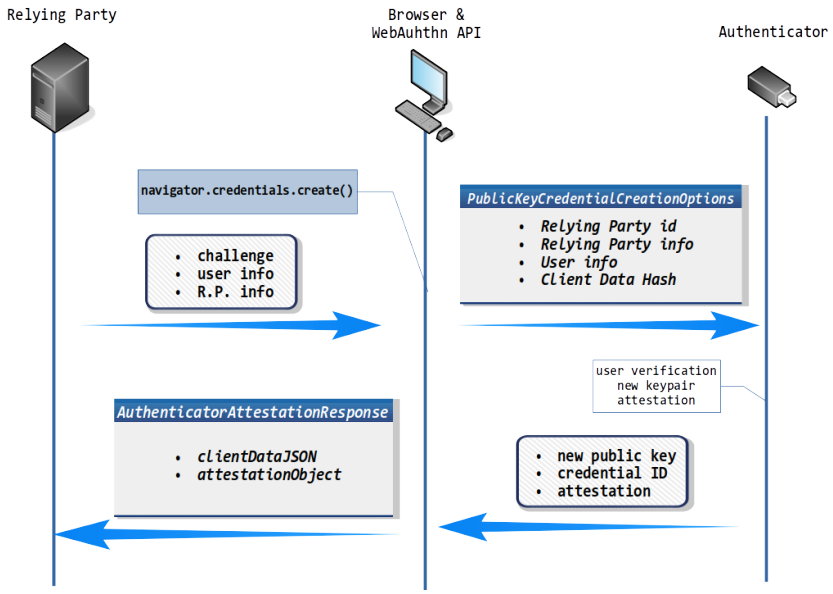


Figure 2.3: Attestation network diagram

ject that will be signed resulting in the attestation signature, that follows a specific attestation format (see section 2.2.4).

However, this attestation signature is done by a specific private key and will be linked with an attestation certificate. At the R.P. this certificate will serve to verify the certificate chain and ensure the authenticator model is actually the model it pretends to be and that the key pair was actually generated by the authenticator. This certificate chain is validated up to a common root of trust obtained through a FIDO Metadata Service (MDS) [14].

Finally, it is important to distinguish between the two credential storage modalities: resident and non-resident credentials, explained later in section 2.2.5.

2.2.3 Assertion

Assertion is the operation that authenticates a user with a credential that was previously generated and registered in the R.P. Like during registration, all cryptographic operations will occur at the authenticator.

The WebAuthn API call is `navigator.credentials.get()`, parametrized with many options. The most important is the challenge, fetched from the server. The challenge will serve as the payload that the authenticator must sign with the credential private key, so it can be verified with the correspondent public key stored at the R.P. server.

In this operation, the browser prepares the client data and does the request via `authenticatorGetAssertion()` CTAP protocol operation. After the user presence or user verification test, the authenticator will increment the signature counter (if present) and then

it generates the assertion signature using the private key of the selected credential. The payload being signed will contain the challenge, the authenticator data and the hash of the client data, ensuring the integrity of the information.

The credential private key is fetched from the authenticator memory in the case of resident credentials, or deciphered from the allowed credentials ids list sent with the options in the case of non-resident credentials (see section 2.2.5).

Finally, the authenticator response will include the signature together with the authenticator data (see figure 2.4). Also, it will include the credential id that was used during Assertion. Once at the R.P., the signature will be verified with the correspondent credential public key.

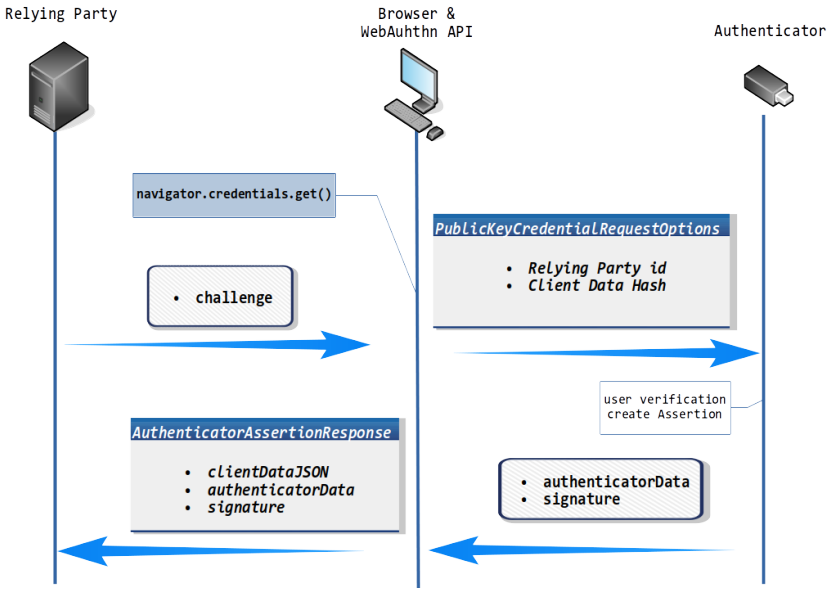


Figure 2.4: Assertion network diagram

2.2.4 Attestation types and formats

Attestation types and formats are one of the key points during the implementation of WebAuthn. Not only the server should be compatible with them, but also the authentiators and browsers.

As Amiet explains on [15], attestation has several purposes. The first of them is to "attest to the provenance of an authenticator and the data it creates, [...] (and) it allows RPs to verify that the authenticator that generated this public key is really the specific model of authenticator it claims to be".

The result of the attestation operation is called *attestation object* and contains an *attestation statement*, that contains an attestation signature. Usually it also includes a certificate chain, used at the R.P. to verify the signature up to a common root of trust. The attestation statement

is generated differently depending on the attestation type used. Besides, each attestation type provides a different trust model [15]. There are five main attestation types:

- **Basic attestation:** Uses an attestation private key securely contained in the authenticator burned in at the factory, common for all key pairs generated at the authenticator. It also contains an attestation certificate.
- **Self attestation:** Uses the generated private key to sign the correspondent public key. This does not prove the authenticator authenticity but the ownership of the public key.
- **Attestation CA:** Used by authenticators that have a Trusted Platform Module (TPM). This guarantees the authenticator uses a TPM and preserves the user's privacy as it generates a new trusted signing key each time.
- **Elliptic Curve based Direct Anonymous Attestation (ECDAA):** Uses the FIDO ECDAA algorithm [16]. It ensures that the signing key is not common to other authenticators of the same model, ensuring user's privacy and reduces the impact of compromising one attestation private key.
- **None.**

On the other hand, for a given attestation type, the same information can be expressed in multiple attestation formats. The most used is the "packed" format, as it supports all the attestation types except None, which has its own "none" format. The WebAuthn standard defines the following attestation formats:

- **"packed":** Optimized for WebAuthn, supports the four main attestation types.
- **"tpm":** Generally used by authenticators that use a Trusted Platform Module as their cryptographic engine.
- **"android-key":** Used by the platform-provided authenticator on Android "N" or later, based on the Android key attestation [17].
- **"android-safetynet":** Used by the platform-provided authenticator on certain Android platforms based on the SafetyNet API [18].
- **"fido-u2f":** The format used with FIDO U2F authenticators.
- **"none"**

2.2.5 Resident and non-resident credentials

From the authenticator point of view, the credentials can be resident or non-resident. Resident credentials refer to the ones that are stored on the authenticator's memory: the private key is indexed by the web page origin (domain name).

In contrast, when the private key is not stored in the authenticator memory, the credential is said to be *non-resident*. Although the private key is not stored inside the authenticator, it will be needed to sign the authentication (Assertion) challenge.

For this, when the device creates the key pair during registration (Attestation), it creates a ciphered version of the private key, which is sent to the server as the credential id. Later, during authentication, this credential id will be included by the server in the allowCredentials list, so the authenticator will receive it and decipher the credential ID to obtain the correspondent private key. It is also important to mention that the ciphered private keys sent to the server can only be deciphered with an unike key embedded in each authenticator.

FIDO-U2F uses non-resident credentials as the protocol was only used for second-factor authentication. Because the authenticator is not storing the credentials, there is no constraint on the amount of credentials an authenticator can manage.

Later, FIDO2 introduced the concept of *resident credentials*, as an idea for providing a way to use authenticators as a first-factor or "passwordless" authentication method. Storing the credential private key in the device memory, as the standard [6] explains in its section 6.2.2, allows the authenticator to use a credential given only a Relying Party ID, with no need of the a credential ID.

This design is meant to avoid the server sending a list of credential IDs during authentication, without firstly identify the user behind the request. It is during authentication (Assertion), when the authenticator provides the user ID (userHandle) so the R.P. can identify the user and correctly verify the authentication.

User verification Usually when using resident credentials, the authenticator uses user verification before authenticating. User verification (UV), according to the WebAuthn standard, has the intention "to be able to distinguish individual users". That means the authenticator may use, for example, a device PIN or a fingerprint recognition, protecting it from other users using the resident credentials held by the key.

It is important to differentiate it from user presence, when only the button should be pressed to perform a user presence test with an authorization gesture.

2.2.6 WebAuthn charecteristics and use cases

The WebAuthn architecture is based on public-key cryptography. This means that the server will only store the public key so, should the server be compromised, the information leak will

not jeopardise the account security. Besides, WebAuthn makes phising attacks are more difficult thanks to digital signatures, which provide authenticity and integrity to the information.

Signed information during authentication includes not only the challenge but also browser data, such as the ”origin”, which is the domain name of the web page. For instance, if an attacker clones a login web page and tricks the user to use the authenticator on their clone, they will not be able to perform a replay attack by sending the authenticator response to the original page. This is because of the domain name, which would be different to the original page, resulting in a signature validation failure.

WebAuthn can be configured as a first or second authentication factor. As a second factor authentication method, the user would need to use the authenticator to prove their identity after a first authentication factor, usually a password. If the password is stolen, the attacker would also need to compromise the second factor (the authenticator) for accessing the account.

On the other hand, WebAuthn can be also used as a first factor authentication method. WebAuthn would then replace the password method so the user would directly login by using the authenticator. This schema is called the “passwordless” authentication.

With this schema, the authenticator is required to use resident credentials (see section 2.2.5). When using resident credentials, the authenticator resturns a user handle or user id. This user handle will serve the server to identify the user behind it for validating their authentication.

2.2.7 Backwards compatiblity with FIDO U2F

The FIDO CTAP protocol has two versions, as seen before. The first version, or FIDO CTAP1 [12], is the renamed version of the first approach: FIDO U2F (Universal Second Factor). The second version, the FIDO CTAP2 [7], is part of the FIDO2 project [11] that involves the W3C WebAuthn API and seeks for a first-factor authentication mechanism, introducing the concept of resident credentials, explained in the previous section.

However, WebAuthn is backwards compatible with FIDO U2F, based on the concept of non-resident credentials. Also, as explained in the standard in its section 2.2 [6], authenticators compatible only with FIDO U2F use a specific Attestation Statement Format and do not have any mechanism to store a user handle, or user identifier. With resident credentials, this handler is stored along with the private key, allowing the R.P. to identify a user during authentication. However, with FIDO U2F, this user handle will always be null (see section 2.2.6).

2.3 Projects and companies around WebAuthn

FIDO and WebAuthn have raised interest among some companies and organisations, starting some projects around this type of authentication.

- **Yubico.** The Yubikeys are the most well known security keys in the market. Yubico [19] also contributed actively on the development of FIDO standards and has many projects around WebAuthn and their hardware devices.

- **FIDO Alliance.** Fast IDentity Online Alliance [20] joins big companies like Google, Microsoft, Facebook or Amazon towards the same goal: solve the password's problem. It has been the most important organisation around the standards and that have developed the FIDO U2F/CTAP1 standard, precursor of the FIDO CTAP2. It is the main actor of the FIDO2 project [11], involving the W3C and WebAuthn. The FIDO alliance also has a public discussion community to help FIDO2 developers [21].

- **World Wide Web Consortium (W3C).** The W3C is an international community that develops open standards [22]. Working with the FIDO Alliance, they developed the current W3C Recommendation: the WebAuthn API [6]. The working group that is developing this standard second version has a public mailing list, used to stay updated on the Github changes of the standard [23].

- **Duo Labs.** Duo [24] is a CISCO security company that developed some WebAuthn projects and demos, being the most important WebAuthn.io.

- **Auth0.** As an authentication service provider, Auth0 is one of the highlighted companies around authentication research [25]. The company built WebAuthn.me, a demo and a debugger tool for the WebAuthn.

- **SoloKeys.** Solokeys is a company that developed the first open-source FIDO2 security key, creating a community around its firmware project. They aim to provide a verified and trustworthy hardware solution as a WebAuthn authenticator [26].

2.4 Implementations of the WebAuthn Relying Party

During the last few years, many developers have demonstrated interest in WebAuthn. The most common concern is the implementation of a web application that can make use of this new standard. According to WebAuthn, the entity that authenticates a user by using WebAuthn is called a Relying Party. This includes both the server logic and the front-end scripts that make use of the API.

The following projects are some examples, although there are more:

- **py_webauthn.** Built as a Python module, it can be used to implement the server operations of a Relying Party. It also provides a class for the user, storing a public key and a counter. This user is then used for authentication. The project also provides a

Flask complete demo with a simple frontend. Only the three main attestation formats are supported [27].

- **Fido2-lib.** A NodeJS library that implements the server functionality for WebAuthn. It implements the registration and authentication operations and provides extensibility for MDS and some extensions. The library parses the attestation certificates and supports several attestation formats. Its main developer is one of the engineers at the FIDO Alliance [28].
- **Fido2-net-lib.** A WebAuthn library for .NET, implementing Attestation and Assertion operations. It also provides some demos, even some examples for integration with Active Directory [29].
- **webauthn-ruby.** A Ruby library for implementing a Ruby/Rails server conformant with a WebAuthn Relying Party. The project provides a demo and extense API documentation [30].
- **webauthn.** A library for Go developed by Duo-Labs. It is the server implementation of the WebAuthn.io demo site. It provides extensive documentation [31].

2.5 Implementations of the WebAuthn Authenticator Model

The section 6 of [6] describes the WebAuthn Authenticator Model. This is an abstract model implemented, for example, by the FIDO CTAP standards, so they are compatible with WebAuthn. FIDO standards are implemented by hardware devices, but the model is not restrictive and also allows for software implementations that offer the same abstract interface. The following are some examples of devices and software implementations available on the market:

- **Google Titan security keys:** Hardware devices used by Google to provide a second-factor solution to their users. It also works with any FIDO U2F compatible service [32].
- **Yubikeys:** Yubico is one of the main actors of the new standards. They have implemented several series of hardware devices: the yubikeys [4].
- **Solokeys:** An open-source project that began as a FIDO U2F firmware evolved into a FIDO2 firmware implementation, also with open-source hardware [33].
- **OnlyKey:** This device is conceived as a hardware password manager, being able to store static passwords as well as OTP codes. Besides, it implements the FIDO U2F standard [34].
- **Thetis:** A FIDO U2F certified hardware authenticator [35].

- **WearAuthn.** An open-source software implementation for Android wearable devices like smartwatches [36].
- **android-webauthn-authenticator:** An open-source software implementation as a library for Android [37].
- **OpenSK:** An open-source project from google that offers firmware that can be loaded on some Norton boards. It is intended for research purposes to inspire new authenticator developments [38].

2.6 WebAuthn testing tools

In order to serve as implementation references, there exist an amount of demo websites that implement an example of authentication with WebAuthn. Besides, some of them allow some configuration and more information that can help developers to test the protocol. Some of the most relevant tools are the following:

- **WebAuthn.io**. A WebAuthn demo page sponsored by Duo. It works with an open-sourced backend and frontend that showcases a real implementation with usernames and a "passwordless" authentication with WebAuthn. It also allows some configurations like the attestation type, authenticator selection and user verification (see figure 2.5). [8].

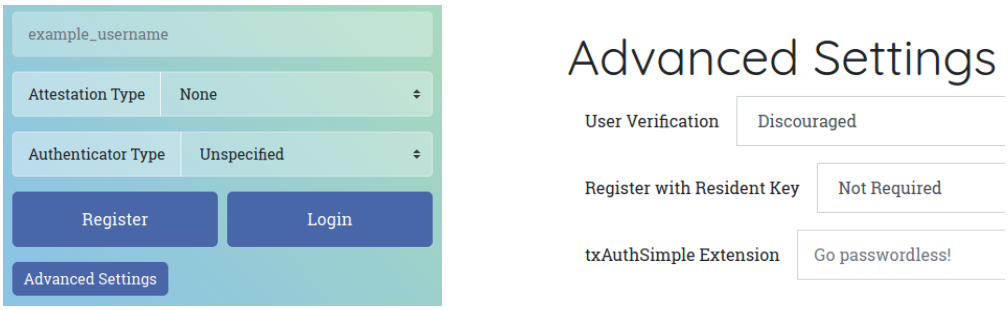


Figure 2.5: WebAuthn.io demo page.

- **WebAuthn.org.** This site provides an open-source demo with advanced information about the WebAuthn registration and login operations. It does not provide any configuration options (see figure 2.6) [39].
- **Yubico demo**. This tool is proprietary software and is intended to showcase all Yubikey capabilities [40]. It provides an elaborated playground mimics that allows a user to configure the account with a realistic multi-factor authentication use case, with recovery

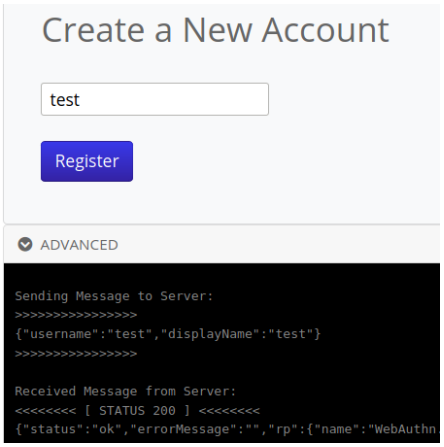
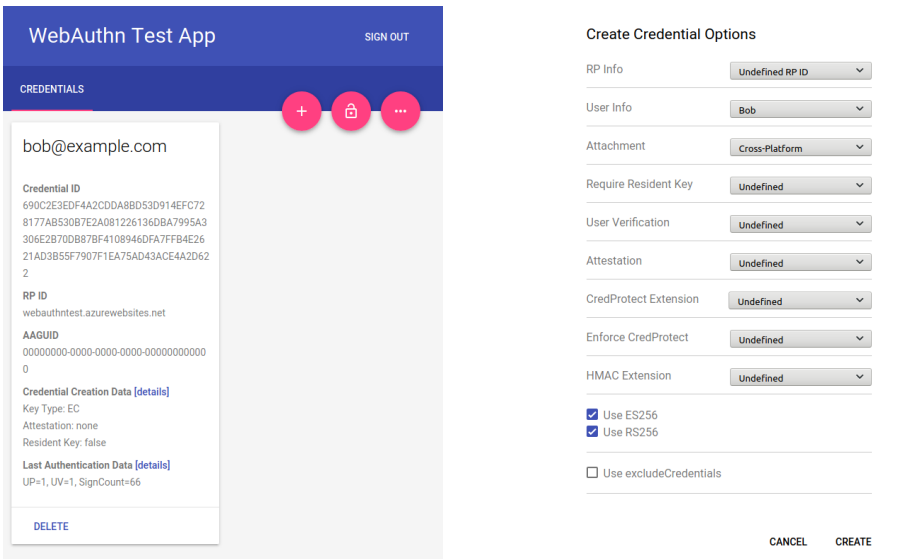


Figure 2.6: WebAuthn.org demo page.

codes, OTP codes and security keys. It also provides a registration and authentication page for WebAuthn, with no configuration options (see figure 2.7). However, it does provide all technical information of the data exchanged during the whole process.

Figure 2.7: WebAuthn.io demo page.

- **WebAuthn Test App**. This web application provides an extensive amount of information and configuration options (see figure 2.8b) as it is intended to serve as a testing tool. It lets registering and authenticating several independent credentials along with all technical information retrieved during the registration and authentication (see figure 2.8a) [41].
- **WebAuthn checker**. A web application that provides some information and verifies that the operation was performed correctly (see figure 2.9). It shows the validation details on both registration and authentication operations. Finally it also demonstrates some WebAuthn features like a concrete extension and some configurations [42].
- **WebAuthn.me**. This site provides an interactive WebAuthn tutorial and a debugger,



(a) Main dashboard.

(b) Registration configuration.

Figure 2.8: WebAuthn Test App.

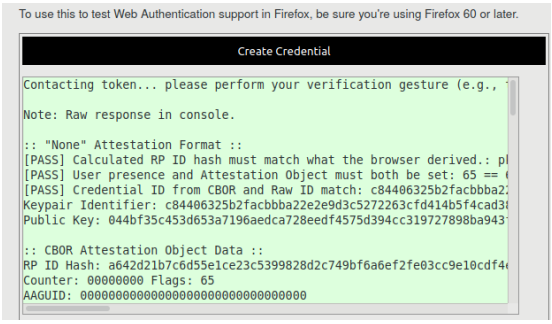
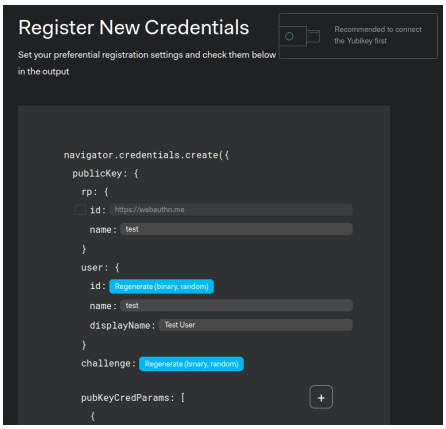


Figure 2.9: WebAuthn Checker tool.

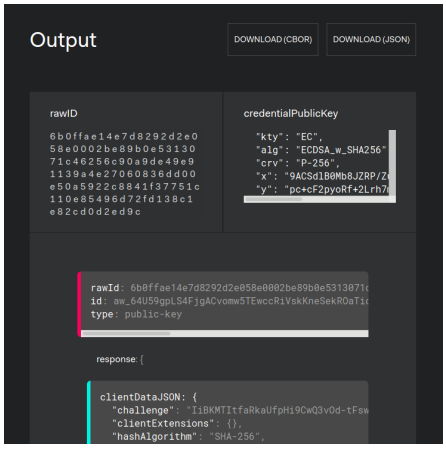
crafted by Auth0 company. The debugger provides all the configuration options (see figure 2.10a), intended for developers to understand and test the registration and authentication processes. After the process has finished, it shows all related data and allows downloading it, both in CBOR encoding and in JSON (see figure 2.10b) [9].

2.7 Conclusion

As soon as the W3C got involved in the FIDO2 project, many companies and developers started to build demonstrations of the new API the consortium was developing: the WebAuthn API. This new standard was based on previous works of the FIDO Alliance, a group of big companies such as Google or Microsoft that started the path towards this new public-



(a) Registration configuration.



(b) Output information.

Figure 2.10: WebAuthn.me App.

key cryptography authentication. In the first place, the new standards were designed as a second-factor authentication mechanism, protecting users from phishing attacks. However, the idea of a "passwordless" authentication flow soon appeared, making these new standards a potential first-factor authentication mechanism.

At this point, several developers and companies worked to create reference implementation examples and public demonstrations, to introduce and boost the adoption of the new protocol. The first working examples and demo web pages are linked with the main companies involved, like Auth0, Duo Security or Yubico. In parallel, while the main browsers were implementing the WebAuthn API, some developers created web tools that allowed anyone to test the protocol. Furthermore, new companies like Solokeys have also shown interest in building alternative authenticators, as in the first place Yubico was the main actor.

Nowadays, the WebAuthn API has been released as a W3C recommendation and the consortium is working on a Level 2 version to improve it. This situation demonstrates that there still is a need for flexible and independent testing environments that allow protocol researchers and browser and authenticator developers to test different use cases. Currently, there are some tools that offer configuration options, although they do not allow manipulating every field within the API calls. Also, most of them allow no control over the registered credentials and they are usually tight with a user account, as their main purpose is not testing, but to represent a realistic use case. Finally, some of the tools show the details at the end of the operation and do not allow the developer to stop, modify parameters and replay the API call during the process.

Chapter 3

Planning and methodology

Before getting into details about the tool development and the authenticators testing, this section will explain the election of the engineering methodology used in this project. Besides, the project defines a main time and cost estimate, represented in a Gantt diagram.

3.1 Engineering methodology

The methodology used in this project is based on Adaptive Software Development (ASD) [43], an agile methodology that replaces determinism with emergence. ASD's main idea is to focus on collaboration and learning as a technique to build complex systems, introducing the concept of Speculation in contrast to Planning. During Speculation, the project initiation information is used to define a set of release cycles or software increments that will be required for the project.

This degree thesis fits in ASD methodology in two ways. First, there are no initial set of features but a mission statement. This is due to the fact that the development team does not have a prior exposure to the used technology, as WebAuthn is a recent standard. After an initial Speculation phase, there will be defined a set of software increments which are adaptive to changes on the protocol specification or support. Secondly, the learning cycles are based on short iterations that involve design, build and review. These short iterations allow basic functionality to be delivered fast. Besides, during each iteration, the knowledge gathered by correcting small mistakes lead to greater experience and eventually mastery in the problem domain: strong authentication.

3.1.1 Tool development

ASD lifecycle is based on an incremental and agile lifecycle. For the development of the tool, the Systems Development Life Cycle (SDLC) Iterative Incremental model is used [44]. In each iteration or cycle, a set of requirements are defined, which will be analysed for designing and

coding the deliverable solution, which will constitute a software increment (see figure 3.1). Next iterations will include increased functionality, as well as correcting defects, if any, from the prior delivery.

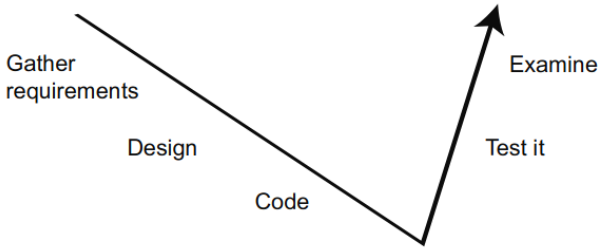


Figure 3.1: Iterative cycle in SDLC Iterative Incremental model.

Each iteration of the SDLC Iterative Incremental model will be tracked by using a Kanban board [45], composed of several columns, which aim to track the amount of Work In Progress (WIP). The "To Do" column holds the pending features for the iteration, followed by a "In progress" and a "Test" columns, used for tracking the progress of development of each feature (see figure 3.2).

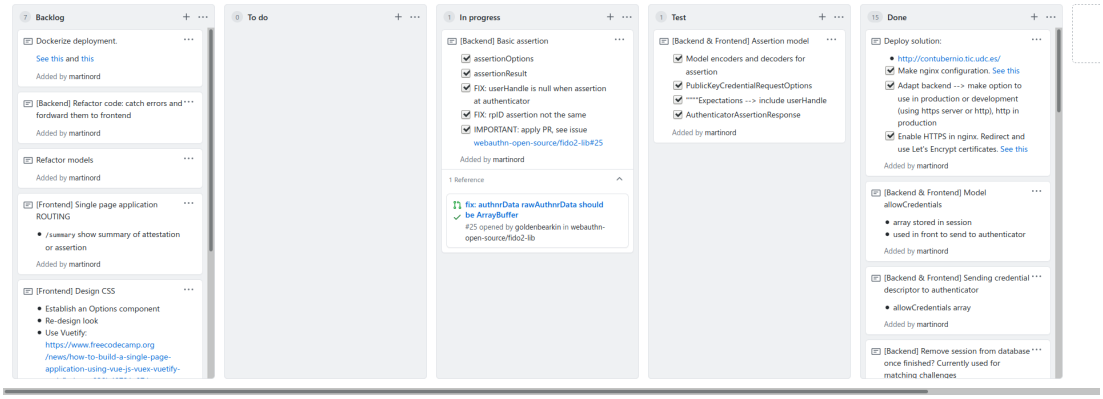


Figure 3.2: Kanban board used in the first iteration. Hosted at Github.

At the end of each iteration, integration testing is performed, which is based on End to End (E2E) tests. E2E testing allows performing, mostly, manual tests to the final software product. This is specially interesting in this project as most of the features are related with the integration of the hardware authenticators with the developed system. As a result of the testing and ASD evaluation, the requirements for the next iteration are defined, correcting errors and including new features. These new features are included in the Kanban board in a "Backlog" column, which will be transferred to the next iteration.

Finally, the defined iterations are:

1. **Basic operations implementation.** Implement REST backend and the minimal frontend to trigger registration and authentication.
2. **Displaying information to the user.** Show the information involved in the process at the frontend.
3. **Modifiable options.** Implement forms for the user to be able to modify the configuration options of registration and authentication.
4. **Structuring the displayed data.** Parse and structure the involved data and better display it in the frontend, allowing to copy the data.
5. **Support for several registered credentials.** Allow registering more than one credential or authenticator for multi-credential testing environments.
6. **Android Safetynet: a new Attestation format.** Implement the Android Safetynet attestation format for using the tool with the Android platform.
7. **Improving user experience.** Frontend improvements regarding the displayed data, browser feature detection and instructing the user on how to use the tool.
8. **Extending possible tests.** Allow deletion of registered credentials and implement resident credentials support.

Regarding the used tools, the project uses a local Git for the version control, synchronized with a Github remote repository. At the end of each iteration, the git tags are used for establishing numbered versions, following the format v<main release>.<iteration>.<minor release>. For instance, the iteration 2 will be tagged as v0.2.0.

During the development phase, some debugging tools are used for white box E2E testing. The main used tool was Visual Studio IDE in debugging mode for NodeJS.

Once an iteration is finished, it will be delivered by deploying on the production server as explained in the Appendix A. Moreover, for automated deployments, refer to Appendix B, where it is explained the development with Docker containers.

3.1.2 Authenticators testing

This project also includes the testing of physical authenticators, aiming its feature exploration and validation. After performing a market study of the most relevant available WebAuthn authenticators, a set of the most relevant devices is acquired (see chapter 6). In order to conclude the compatibility of each device, some compatibility tests are defined, namely: supported attestation mechanisms, communication transports, cryptographic algorithms and resident credentials.

Therefore, compatibility tests will be applied to the most representative authenticators. These tests will use features of the WebAuthn standard by using the available options of the developed tool (DebAuthn), configuring it accordingly and documenting the device response under equivalent testing environments:

- Chromium browser v83 on Ubuntu 18.04.5 LTS.
- Chrome browser v83 on Android 7.

3.2 Project planning and monitoring

This project is structured taking into account the initial phase of research on WebAuthn for defining the tool basic functionality.

In coherence to the ASD methodology, the tool implementation is divided in several software increments that represent the basic functionality of the tool. As it can be seen in the project planning, future iterations are expected but not yet defined. The project management in ASD is based on the fact that many resource management activities are experiments, used as learning opportunities for the improvements in the future.

The monitoring of the project is performed by doing periodic reviews at the end of each development iteration. During them, the project manager meets the development team to evaluate the iteration result and define future functionality increments. In this project, the role of project manager was shared among the thesis directors and the student. Due to external circumstances, most of the review meetings were performed by videoconference.

In terms of time cost and project planning, the following project phases were initially defined:

1. Researching on WebAuthn.
    (a) Study of Relying Party configuration options.
    (b) Study of the Authenticator Model implementations.
    (c) Study of the client implementations in a form of web browsers.
2. Tool implementation.
    (a) Design and analysis: research, technology and compatibility.
    (b) Implementation of Attestation and Assertion operations in the server.
    (c) Implementation of a web interface for the tool as a client.
    (d) Software increments: add functionality.
3. Testing the tool with different web browsers: Chrome and Firefox.

4. Testing different physical authenticators.

   (a) Testing with different configurations.

5. Documentation.

   (a) Writing the dissertation final report.

   (b) Writing installation documentation for the use of the open source project.

In terms of project monitoring, the figure 3.3 shows a Gantt diagram with the final time cost of each project phase at the end of the project. Red week numbers correspond with delivery times for the project draft and the forecasted degree thesis submission.

At the end of the project, some of these phases changed due to the project needs. For instance, testing the tool with browsers (phase 3), according to the methodology, was included in each iteration. Moreover, the implementation of the web interface (phase 2.c) was done in parallel to the basic operations implementation. Therefore, these phases do not appear in the Gantt diagram.

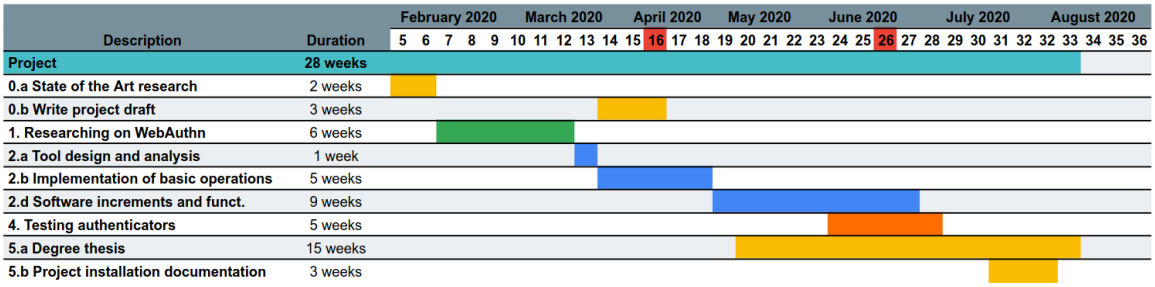


Figure 3.3: Gantt diagram.

3.2.1 Materials and cost estimate

The project has materials and human resources needs for its completion. This section includes the allocation of these resources and its associated cost.

Materials and infrastructure The following are the minimal main infrastructure and materials for the tool development, deployment and test:

- Server for the backend installation. Preferably Linux based, but in any case NodeJS and NPM manager is required.
- Personal computer serving as a testing environment, with USB port and Chromium >=70 (or Chrome) [46] and Firefox >=60 [47].

- Physical authenticators compatible with FIDO CTAP1/U2F or CTAP2 (FIDO2) [7].

The cost estimate for the hosting of a Virtual Private Server (VPS) based on a quick market study would be 38€ per 30 days, 4GB of RAM and 1 assigned CPU. Therefore, 1.26€ a day. The personal computer used for the development and testing has a rounded cost of 1200€. Assuming the lifetime of a laptop is four years, it would result in 0.82€ a day. Finally, the costs of physical authenticators are determined as a fixed cost. The used authenticators are enumerated in the section 6, resulting in a total cost of 140€.

Human resources For this project, the tem consists of three people. A full stack developer that also assumes a role of project manager, guided and supervised by two additional project managers that will assist and guide the research tasks. The associated costs can be based on the official bulletin release of minimum salaries on the sector [48]. The cost estimate per 8h of daily work would result in: for the project managers or analysts 85€, while for the development team it would be 65€.

3.2.2 Estimated and monitored cost

The cost estimate would be calculated in terms of the minimal amount of days the resource was used. For human resources, it is measured in terms of daily work. Every week of the project it is estimated to take 5 days of work for the development team and 2 for the project managers. Taking into account the previous sections, the cost estimate is summarized in the following table:

Description	Cost per day	Forecasted days	Monitored days	Forecasted cost	Real cost
2 Project managers	170	40	56	6800	9520
Development team	65	100	140	6500	9100
Hosting	1.26	140	196	176.40	246.96
Hardware	0.82	60	84	49.20	68.88
Devices	-	-	-	140	140
TOTAL				**13665.60**	**19075.84**

Table 3.1: Cost estimate and monitored cost in euros (€).

In summary, the project was planned to be delivered by June but it was delayed until August, as shown in the Gantt diagram (see figure 3.3). Delaying the project 8 weeks increased the associated costs by 39.61%.

Chapter 4
Analysis

This section analyzes main purpose of the tool and argues the technological and architecture initial decisions, before explaining the development of the tool.

4.1 Mission statement

First of all, it is greatly important to identify the tool's main purpose. The designed web application was designed to serve developers, testers and researchers of the WebAuthn protocol to design and audit a specific use case, by manipulating the protocol options testing the behaviour. This testing environment should be open for anyone and does not attempt to serve, as other tools, as a proof of concept or demonstration of a concrete production system that uses WebAuthn as a protocol to authenticate (log in) users in a system.

This approach helps researchers not to be tight to a persistent user account. As seen in other tools, the user accounts lasted a limited amount of time. Despite this, there were no restrictions on registering the same user several times for providing support for multiple registered credentials. In these tools, some usernames could be used by two different researchers during the same time period so the registered credentials will belong to either of them. That means that a real user of the system, when requesting authentication with a username also used by another researcher, will find credentials that they have not registered themselves, but some others have.

This restriction could have been solved by registering real usernames that are not conflicting. However, this solution obligates potential users to create an account for performing their tests, even if they wish to test a concrete case once. Also, this means persistent storage is needed at the server side, containing user's information.

With no doubt, in a web application that uses WebAuthn as authentication mechanism for user authorization, this persistence is required. In contrast, a web application that does not use information persistence offers users to perform tests freely with no constraints. For

example, by not linking the user to the credentials, a tester could modify the user field when registering a different credential, and check how credentials that were registered with different usernames are allowed to authenticate in the same operation. In fact, the protocol does not restrict how users are linked with credentials, so by not implementing this link, the tool becomes more flexible.

Another feature of the tool is to provide the insights of the information treated at the server side, regardless of the operation result. That is, the tool user may intentionally provoke a failed validation with the purpose of debugging a specific use case.

In brief, the tool designed here was not intended to provide a demonstration of a real authentication for users but to serve as a testing environment with no constraints. Therefore, it was not built to support user accounts in the first place, nor native roaming credentials among devices.

The application, however, provides an interface to request the creation of new credentials and authentication of registered credentials during a limited amount of time. The server handles the minimal amount of information that is required by the standard to validate the authentication of the user and, also, provides the server validation data during the process.

By doing this, the user is able to debug both registration and authentication use cases as they are informed of successful and failed operations. Then, the tool can serve as a reference for developers to debug the outcome of authenticators and browsers with different configurations before starting coding their own implementation.

4.2 Actors and use cases

The debugging tool is intended to serve users with visual feedback of the operations performed in two main operations: Attestation, or registering ceremony and Assertion, or authentication ceremony. Therefore, two main use cases are defined:

- The user uses an authenticator to perform Attestation, registering a credential.
- The user uses an authenticator to perform Assertion, authenticating with a credential.

This tool is conceived as a web application, showcasing the main squema of the WebAuthn protocol [6]. The web application is a server-client architecture, consisting of a server, alias backend, and a client running on a web browser, alias frontend. The user accesses the application through the user interface, provided by the Relying Party. Therefore, two actors within the protocol are defined, namely:

- A Relying Party, that supplies information such as randomly generated challenges through REST endpoints as well as serving the static user interface with client-side logic to a compatible web browser.

- The user, that uses an authenticator compatible with the WebAuthn Authenticator Model.

4.3 Architecture and technology election

Regarding the server-client communication, the application involves the use of a HTTPS/TLS web browser and a REST server. The server exposes endpoints for requesting and posting registration and authentication information in form of attestation and assertion WebAuthn objects.

The server exposing the REST service can be implemented in many languages. One well known library that enables the use for both serving static web content as well as the exposure of a REST service is the ExpressJS library [49], for NodeJS JavaScript framework [50]. This is a well maintained and widely used server-side JS framework with many pluggable modules or libraries.

On the other hand, for the frontend, there exists an obvious need for web logic, namely JavaScript code. This logic is used to interact with the WebAuthn API of the browser. However, JS can also be used for controlling an asynchronous communication between the server and the client. This communication involves the registration and authentication information, which can be represented in JSON files. For this reason, the JS code can make use of AJAX (Asynchronous JavaScript and XML) requests to the REST endpoints through HTTP.

The user interface aims to dynamically show the information exchanged with both the Relying Party server and the user's authenticator. Although WebAuthn does not require an application to perform asynchronous requests to be compatible, it can be a great improvement in two ways:

- It would better manage data consistency and user experience.
- It will reduce network traffic as it does not load static web content more often than required.

Besides, the JS script should be able to modify HTML DOM objects for displaying the information needed dynamically resulting from the asynchronous requests. This DOM manipulation can directly be done with JS scripts. However, for this purpose there also exist two main JS frameworks that provide an extensive tool for reactive elements and routing, implementing a Single Page Application (SPA): ReactJS and VueJS. Both of them allow building minified static files that will compose the SPA with the desired asynchronous functionality.

Additionally, having in mind the original idea of creating an easily extensible tool, it is a better idea to use a well known JS framework that enables the capability of creating loose coupled components. In VueJS [51], each reactive component can be defined in a single file,

binding CSS, HTML and JS logic together. This approach makes the development easily pluggable, as well as compatible with the already exposed features needed on the client-side.

Last but not least, the server needs to implement all attestation and assertion operations defined by WebAuthn. The development of these cryptographic operations is approached by several libraries in different programming languages, as analyzed in chapter 2.

In this case, the fido2-lib NodeJS library was used, as it provides an open source solution published as a npm package with online documentation. Nevertheless, as it will be explained later, it was necessary to adapt and extend the functionality of the library.

Chapter 5
Development

According to the Incremental Iterative Model, the tool, named DebAuthn, is developed in software increments. This chapter includes a section per software development iteration, including the analysis of requirements, technical details and the iteration evaluation or testing.

5.1 Basic operations implementation

This first iteration aimed to cover as a first step the implementation of basic functionality of the two main operations: Attestation and Assertion, corresponding to registration and authentication respectively. Besides, it served as a first experiment on the project domain: WebAuthn. The first subsection argues the general project setup and architecture while the following subsections will individually approach registration and authentication operations.

5.1.1 General aspects

Both Attestation (registration) and Assertion (authentication) require an options object that will serve as input configuration for the WebAuthn API requests to the authenticator. These requests will answer with the correspondent response from the authenticator, sent back to the server for validation, as shown in the table 5.1.

Operation	WebAuthn API Request	Input	Output
Attestation	navigator.credentials.create()	{ publicKey: PublicKeyCredentialCreationOptions }	PublicKeyCredential
Assertion	navigator.credentials.get()	{ publicKey: PublicKeyCredentialRequestOptions }	PublicKeyCredential

Table 5.1: WebAuthn API Requests: Attestation and Assertion.

As said, both `create` and `get` WebAuthn API requests require an object as input. This object contains, as it will be seen later, some configurations or `options`. Among other parameters, they include the challenge, which is generated at the server side. Regarding the

rest of the parameters contained in the object, most of them are dependent on the server configuration. Hence, the whole object will be created at the server side and sent to the client when requested via a HTTP GET method request.

Secondly, the WebAuthn API requests will return a response from the authenticator, which needs to be sent to the server in order to obtain the operation result, after validation. For this purpose, the HTTP POST method is used to send the response object and to obtain this result from the server. Having in mind the explained design, four endpoints were defined (see table 7.1).

REST Endpoint	Method	Request payload	Response payload
/attestation/options	GET	*None*	PublicKeyCredentialCreationOptions
/attestation/result	POST	AuthenticatorAttestationResponse	AttestationResult
/assertion/options	GET	*None*	PublicKeyCredentialRequestOptions
/assertion/result	POST	AuthenticatorAssertionResponse	AssertionResult

Table 5.2: Initial REST endpoints exposed by the backend for Attestation and Assertion.

The asynchronous implementation of the REST service in ExpressJS is based on callback functions, called middleware functions, which serve as handlers for the incoming requests. These are used for declaring the already mentioned endpoint routing, registering in each router a controller that will call the specific webauthn validators that will be explained in the following subsections.

With respect to server information management, session information was used. This is key for the web server to keep the state of the requests. As explained, when the options are requested to the server, it creates a challenge, among other parameters. This challenge, however, is needed for validating when the authenticator response is posted. On the other hand, when registering a credential in the system, the public key needs to be stored in the server so it can later be authenticated.

As explained in the section 4, at first, there are no user accounts and the tool does not manage persistent data, so the registered credential public key should be attached to the user temporal session.

For the implementation, the selected approach is to store all this data on the server side, while a client-side cookie holds a session identifier. A technology that implements this is the ExpressJS session middleware. It can be configured with many stores, like in-memory implementations. In this case, in order to allow easy management and storage of data like Buffer objects in an object-like structure, `connect-mongo` storage was used. For storing the session data, it uses an external MongoDB [52], a database based on NoSQL documents. Connect-mongo library also can be configured with a `time-to-live` parameter, deleting the stored data after a defined time. In this case, the session data will be deleted after 1 hour

so the application becomes more scalable.

The server, apart from exposing the REST service, it should serve as a web server for accessing static public files. This was achieved in ExpressJS by the use of the static middleware provided by the framework. This middleware allows to set a folder within the working directory to serve the static files, like the HTML, CSS and JavaScript files of a web page. Then, this middleware was assigned to the '`/`' route, so both the REST service and the static content are served by the same server.

Finally, regarding the WebAuthn Relying Party operations, as it will be explained later, the server needs some login. The server-side attestation and assertion validation is well implemented in the asynchronous `fido2-lib` NodeJS library [28], so all library configuration has been defined in a separate module, importing the WebAuthn operations into the project. These functions will be used by the REST controllers when receiving the requests from the client.

On the other hand, the user interface was implemented in VueJS, as discussed in the Analysis section. This iteration involved the implementation of the basic operation of registration and authentication so, for this reason, only two buttons were required at the user interface. The logic behind the button triggers the AJAX requests to the defined endpoints by using the axios library [53]. The scripts will also utilise the WebAuthn API that interfaces with the authenticator through the browser, as explained later.

VueJS composes a Single Page Application, enabling a powerful kit for the development of complex functionalities for later iterations. Accordingly, two main components are defined: `Attestation` and `Assertion`. At this point, each VueJS component only included a button, and the related JavaScript functions (see figure 5.2a). Figure 5.1 shows the common steps for both operations, involving the asynchronous REST requests, namely:

1. The user clicks the button.
2. The options are requested to the server.
3. The server generates the options and sends them.
4. The call to the authenticator is done using the requested options.
5. The user performs interaction with the authenticator.
6. The authenticator response is posted to the server.
7. The server replies with the operation result.
8. The result is parsed. If successful, an alert is shown to the user (see figure 5.2b).

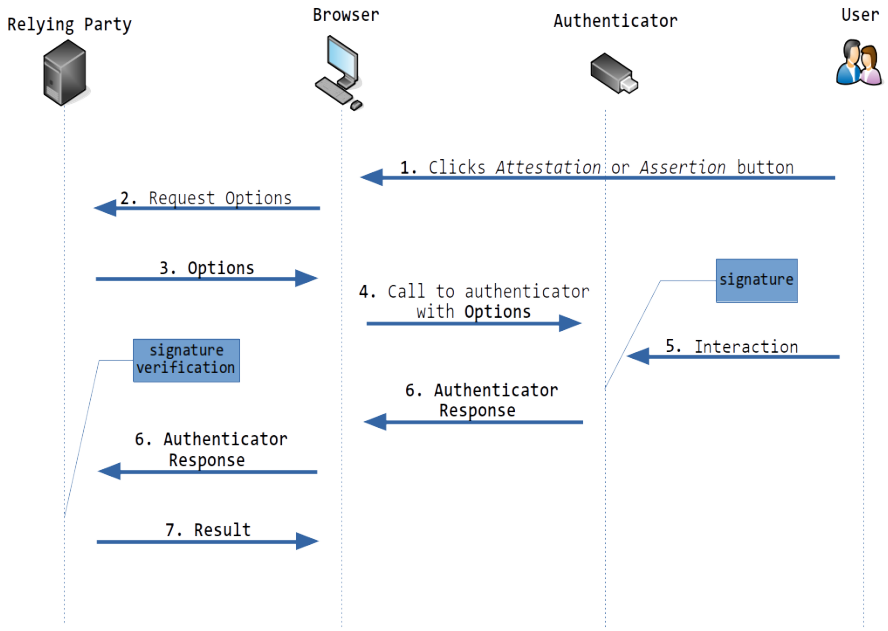


Figure 5.1: WebAuthn frontend complete flow.

5.1.2 Attestation

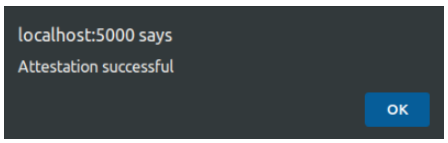
Attestation is the process of registering a credential generated by the authenticator in the system, for later authentication. This process involves signing a challenge together with other client data as the origin and the operation type, concatenated with the authenticator data (the generated public key and the authenticator properties). Once the authenticator response is received, it is validated by the library at the server-side and the result is returned to the client.

In order to send the options and the authenticator result object using a JSON, Buffers (binary data) such as the challenge should be sent in an encoded form, decoding them at the other side. For instance, binary buffers can be encoded in base64 format. In this first iteration it was defined one class per object, with two static methods: `encode` and `decode`, that return the corresponding encoded and decoded object respectively.

On the other hand, two request handlers were defined in the attestation controller, corresponding to each endpoint. The beginAttestation handler will send a `PublicKeyCredentialCreationOptions` object once encoded. Then, this object is decoded at the client-side component's logic and passed to the authenticator via the WebAuthn API `navigator.credentials.create()`. Figure 5.3 shows the prompts displayed by the browser after the call.

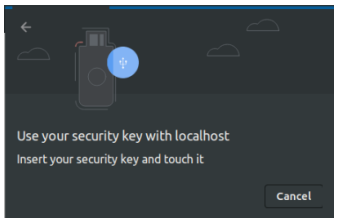
Once the authenticator returns the `AuthenticatorAttestationResponse` object, it is encoded and sent to the server. This object is then handled by the `finishAttestation` function. It will then decode the object and use the fido2-lib correspondent function

(a) Attestation and Assertion VueJS components.



(b) Alert

Figure 5.2: Basic operations implementation: frontend.



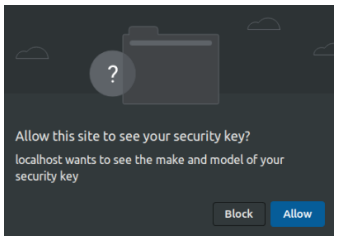


Figure 5.3: Browser prompts in Chrome when performing Attestation.

to validate the Attestation.

However, in order for this last function to validate the response from the authenticator, it needs to correlate both separate requests. For this purpose a new object was designed: `AttestationExpectations`. Once the first request is performed, this object containing the challenge, a factor and an origin configuration is saved by using the session data. From now on, this data will be referred to as '`expectations`', as they are the parameters expected at the authenticator response.

Finally, when the second request arrives, the session information is retrieved and both the authenticator response and the expectations objects are used for validation by the fido2-lib library (see figure 5.4). The validation consists in several checks related to the protocol, like parsing the attestation certificate and checking the signature. By using the expectations, the library checks that the same challenge was signed in the authenticator response and that the origin has not changed, detecting a potential MitM attack.

5.1.3 Assertion

Assertion is the process of authenticating using an authenticator holding the private part of the credential. The process, again, involves signing a challenge provided by the server to demonstrate the credential authenticity. Therefore, in the same way as it was done during At-

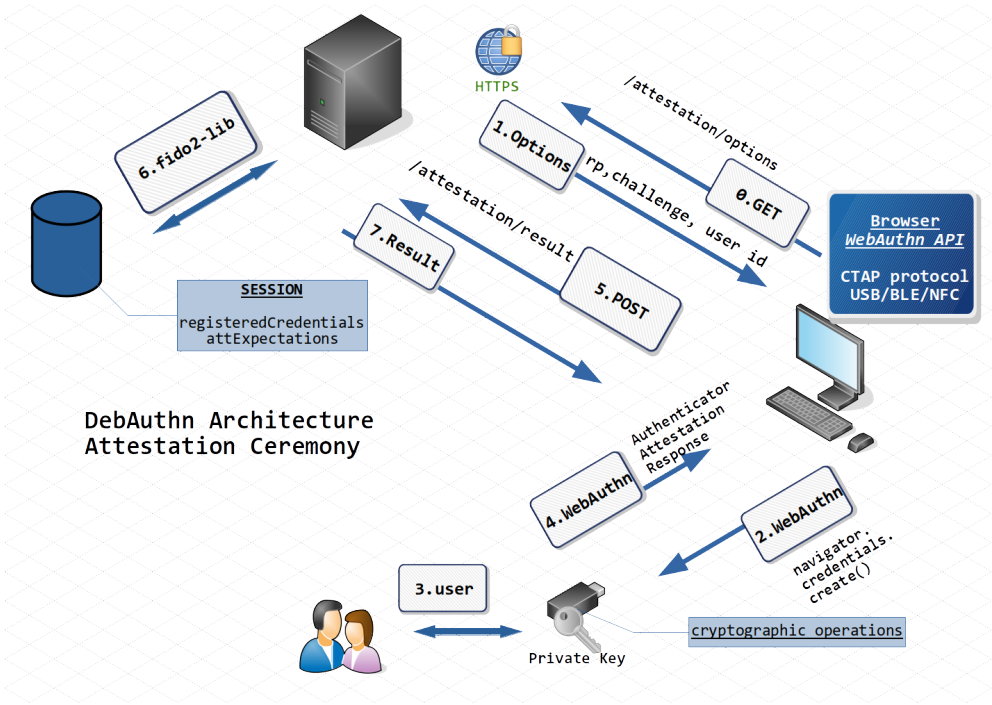


Figure 5.4: DebAuthn architecture Attestation cermony.

testation, two classes were defined with the correspondent encode and decode static functions, in order to send the options and the response Buffers. Also, two controllers were implemented for the correspondent REST endpoints : `beginAssertion` and `finishAssertion`.

In this case, `PublicKeyCredentialRequestOptions` object is sent to the client upon a first request and, when the authenticator responds with the `AuthenticatorAssertionResponse` object, it is sent from the client to the server, in order to be validated.

At this point, the registered credentials are non-resident (see section 2.2.5) due to the default registration options, as it is done in FIDO U2F (FIDO CTAP1), ensuring that the tool is working with any authenticator. That means the private keys are not actually stored in the authenticator but sent encrypted to the server as the credential id. It is only the authenticator who can asymmetrically decrypt the credential id. Therefore, the original id generated at attestation operation needs to be returned at assertion so the authenticator can decrypt it to obtain the private key.

In order to do that, this original id of the registered credential (named `rawId`) needs to be stored in the backend so it can be sent back when authenticating. The WebAuthn standard supports this by `allowCredentials` Assertion `options` field. This is a list of all credential ids of the credentials that can be used to authenticate the given user, by using the WebAuthn API `navigator.credentials.get()` (see step 2 of figure 5.5). Once this is given to the authenticator, it will extract the needed credential id, decrypt it and obtain the

credential private key, used for completing the operation.

In this iteration, a bug was found in the library. When calling the fido2-lib library function that validates the assertion authenticator response, it was giving an error related to the type of the authenticator response. After some research and testing, this issue [54] was found in the github repository of the fido2-lib, and a pull request fixing the issue, so that it typecasts the authenticator data to an ArrayBuffer [55]. However, this pull request has not been yet merged, so it was necessary to manually patch the NodeJS dependency inside the `node_modules/` directory.

Finally, just like during attestation, the library function that validates the authenticator response also required the design of the `AssertionExpectations` object. Like the attestation procedure, these expectations are saved upon the first request, saving the challenge, the origin and the factor parameters, which are later used during the validation (see figure 5.5).

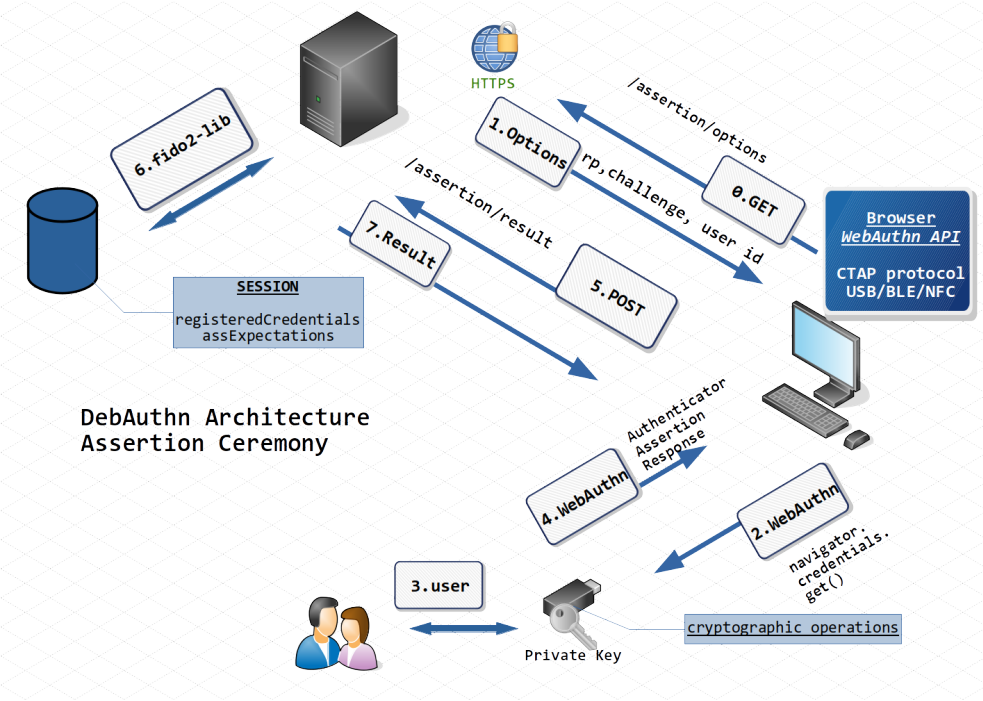


Figure 5.5: DebAuthn architecture Assertion cermony.

5.2 Displaying information to the user

At this point, the user was able to perform both Attestation and Assertion successfully by using an authenticator. However, two buttons were not enough for the user to be aware of the actions performed by the scripts and, should an error occur, no alert is triggered and

displayed to the user. In this second iteration, the main idea was to give some information to the user, including both alerts and the basic data involved.

To this effect, as there exists a need for further display functionality, it was introduced the use of Vuetify [56], a VueJS plugin for reactive UI components for showing dialogs or managing routing through the Single Page Application. Also, this module served as a CSS library with design helpers as its grid system.

5.2.1 Error handlers

Error handling is key in any application and this web application is not an exception, and more especially in this tool, whose purpose is debugging. Also, as a debugging tool, it is interesting for the user to visualize all error occurring both at the server during validation and at the browser during the communication with the authenticator. By having this information, the user can better debug the complete functioning of the protocol.

For this purpose, error catchers and handlers were included both at the server and client (see Appendix F). At the server, all validation functions of the library were wrapped with error catchers, triggering a general callback function acting as handler. This function returns a HTTP 500 error code to the client with the correspondent message from the event.

At the client, the asynchronous requests need to handle these errors. On error, an alert including the corresponding error message (see figure 5.6) is rendered. Apart from the catched backend errors, the WebAuthn API calls were also wrapped around error catchers so, upon failure, it also prompts an error message using the same alert component.

5.2.2 Frontend routing

The previous user interface was composed of two simple components with a single button in each. However, next iterations will include more information and controls for users to configure and debug the operations. Also, from the user experience point of view, it is interesting to divide registration and authentication in two different sections of the tool. For this purpose, the two operations were separated in two different screens or pages: one for registration of credentials (Attestation) and one for authenticating with them (Assertion).

For rendering different pages of a SPA, a Vue Router [57] is configured. This module intercepts the links and renders the correspondent component within the app template. That means that static components that are common to both pages, namely, the header and the footer, do not need to be reloaded (see figure 5.10).

Finally, Vuetify tabs were added for allowing the user to switch between the two pages, triggering the renderization of the correspondent component (see figure 5.10).

5.2.3 Alerts and dialogs

Apart from the dynamic alerts for errors, a success dialog is shown upon success on the Attestation or Assertion operation (see figure 5.7). The dialog component from the Vuetify library is displayed after the operation result from the backend is checked. This check is performed by using the `audit` data embedded into the backend webauthn library response, forwarded to the frontend.

Figure 5.6: Alert component of the frontend.

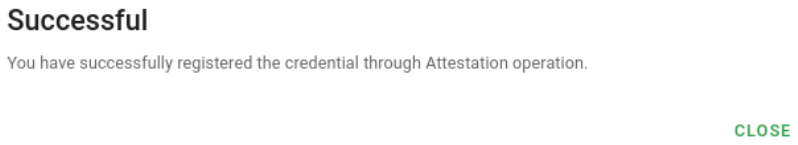


Figure 5.7: Dialog component of the frontend.

5.2.4 Splitting functionality and displaying information

Finally, both Attestation and Assertion operations are composed of three calls or requests (see figure 5.8), as mentioned in the previous iteration, namely:

- Options request to the backend.
- Request to the authenticator through WebAuthn API.
- Authenticator response POST and validation request.

As the tool itself is intended to help with debugging tasks, these three steps were separated, instead of having a single button. Having three different sections allows the user to visualize the protocol flow and to stop at each step to verify all the data involved. For this first approach, the Vuetify stepper UI component was used. Also, the function with all functionality was splitted in three, each one containing the logic and data of a request.

Once all the procedures were divided, each step can display the data involved in the correspondent request (see figures 5.9 and 5.10). For instance, when Attestation options are fetched from the server, they can be displayed to the user before triggering the request to the authenticator. This was configured with VueJS data binding.

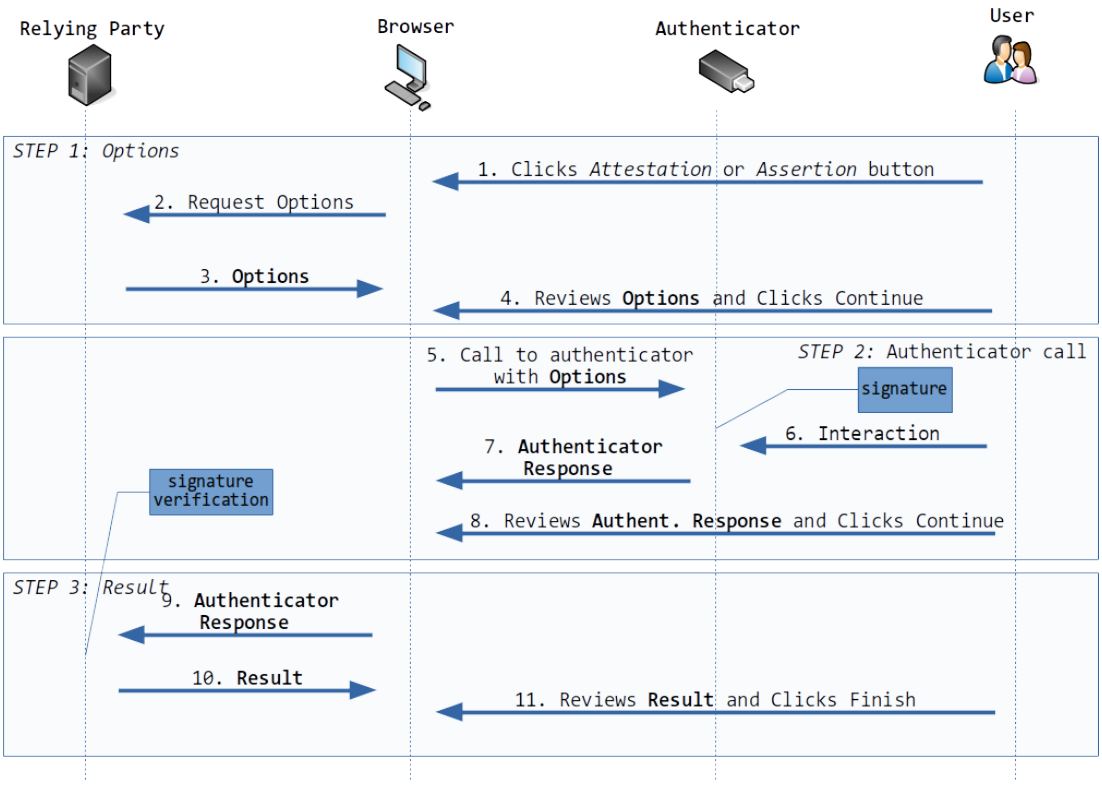
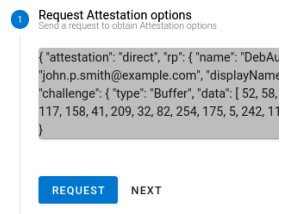
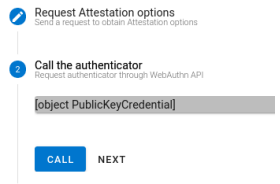


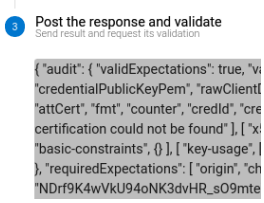
Figure 5.8: Frontend flow with the three steps.



(a) Step 1: Options request to the backend.



(b) Step 2: Request to the authenticator.



(c) Step 3: Send authenticator response for validation

Figure 5.9: Frontend steps with VueJS components.

5.3 Modifiable options

Dividing functionality in steps made possible to display the data involved within each operation, including the configuration options of the operation, the authenticator response and the validation data. In this iteration, the options requested to the backend in the step 1 (see figure 5.9) for both attestation and assertion options will become editable.

That means that the user can manipulate the configurations once received from the server. The result of editing the options with research and/or debugging purposes will then be used

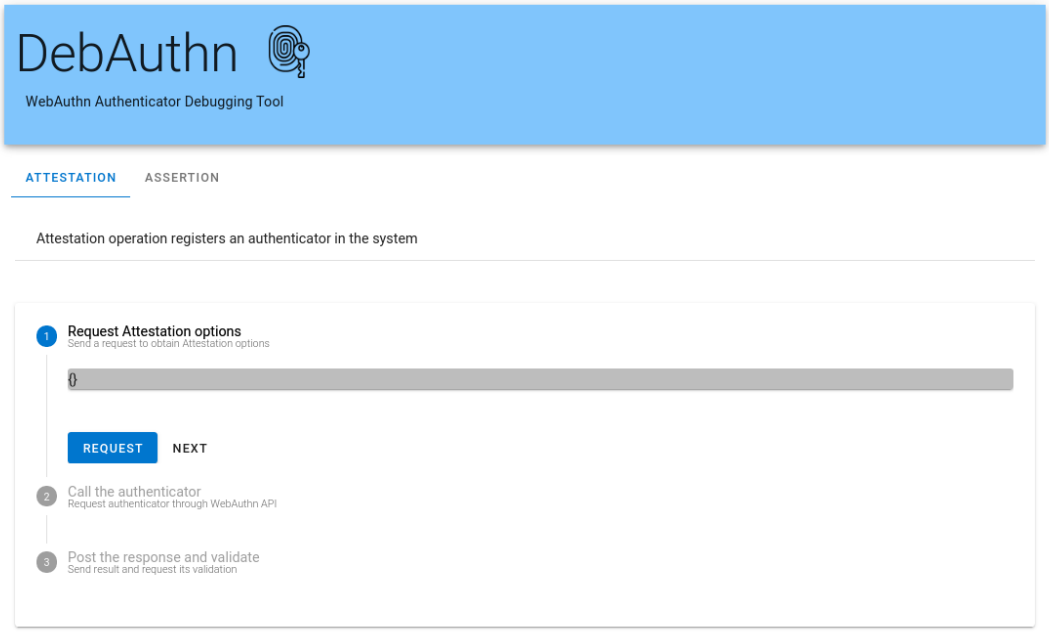


Figure 5.10: Complete frontend with routing and steps.

in the operation, instead of the requested default configurations. This feature had become one of the main characteristics of the tool.

It is important to mention that the options are modified after having fetched them from the server, for both attestation and assertion operations. These default options are the ones that the Relying Party server is using as default and are dependent on the server configuration, so they should be used for a successful operation.

Despite this, the user may cause the server to fail the validation once they manipulate the default options. For instance, if the challenge is modified, the server will fail the operation as the expected and the received signed challenge will not match. Another example can be the supported signing algorithms. If another algorithm is included and used by the authenticator, the R.P. server will fail the authenticator response validation as it does not support it.

5.3.1 Enabling the reuse of options at the backend

As discussed in previous sections, the *expectations* of both Attestation and Assertion are kept in the session information to perform the validation of the authenticator response. When this validation occurs, it was found that the fido2-lib was typecasting the `factor` parameter within the expectations to the correspondent flags as string codes. This implies that the validation fails, when reusing the same options for repeating the operation and, therefore, using the same expectations.

Strictly, the three steps are sequential: request options to the server, call the authenticator and finish the operation by validating. However, when reusing the same options for repeating the operation after a successful data validation, the library could not parse the `factor` inside the expectations, as it was typecasted to string codes. In order to avoid this, and allow reusing the same options after a validation, the library code was manually modified to skip the deletion of factor after translating it.

By doing this, the DebAuthn tool keeps the interesting feature of reusing the very same options with different authenticators. For instance, when testing several hardware keys, the tester may use the same registration options for all the keys. Of course, this characteristic only applies for testing environments so this change in the library does not correct a bug but adds an interesting feature for implemented tool.

5.3.2 Improving models and encoding in base64url

For the user to edit all the registration and authentication options, this iteration also redesigned the models to include all the optional fields defined by the WebAuthn standard [6].

Furthermore, this iteration changed the models to better support serialization to JSON by encoding all binary data like the credential rawId or the operation challenge with the same encoding and decoding functions, which are also improved.

For DebAuthn to be flexible in its REST design, it is interesting to be compatible with REST endpoints with encoded data in query parameters[1]. For this, encoding and decoding functions were changed to also support another format: base64url. This format is often used in different implementations around WebAuthn, like the fido2-lib library and webauthn-json [58]. base64url is defined by the RFC 4648 [59] in its section 5. The code of both encoding and decoding functions can be found in Appendix F.

5.3.3 Attestation options form

According to the section 5.4 of the WebAuthn standard [6], the `PublicKeyCredentialCreationOptions` object has the attributes shown in the table 5.3.

At the first step (see figure 5.12), the required parameters can be loaded into a form allowing the user to directly change them, before passing them to the authenticator (see figure 5.11). Some fields such as `pubKeyCredParams` will not be editable at this point, as they cannot be treated as a string input. The `user` parameter is used for specifying the username and a display name and the `rp` holds the id (usually the domain name of the site) and a name. The most important is the `challenge`, which is the binary buffer to be signed by the authenticator.

[1] For example, `/attestation?challenge=c29tZSBleGFtcGxl`

Required	**rp**	{id,name}
	user	{id: Buffer, name, displayName}
	challenge	Buffer
	pubKeyCredParams	Sequence of {type, alg}
Optional	**timeout**	Number
	excludeCredentials	Sequence of {type, id, transports}
	authenticatorSelection	authenticatorAttachment: “platform” | “cross-platform”
		requireResidentKey: Boolean
		userVerification: “required” | “preferred” | “discouraged”
	attestation	“none” | “indirect” | “direct”
	extensions	-

Table 5.3: PublicKeyCredentialCreationOptions.

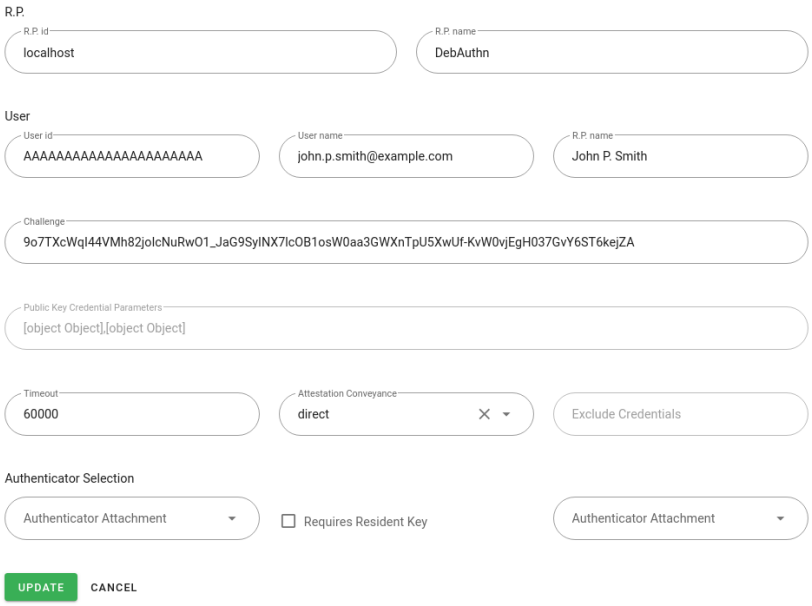


Figure 5.11: Attestation form initial implementation.

Finally, all the optional parameters are loaded if present in the options requested from the server. If they are not present, the user can eventually add them in an empty field.

5.3.4 Assertion options form

The Assertion options defined by the standard [6], are the ones included in the table 5.4.

Similarly, the required `challenge` field is loaded and becomes editable as a first step (see figure 5.13). Once this is implemented, the optional fields were loaded if they are present and, if not, the user will be allowed to add them (see figure 5.12).

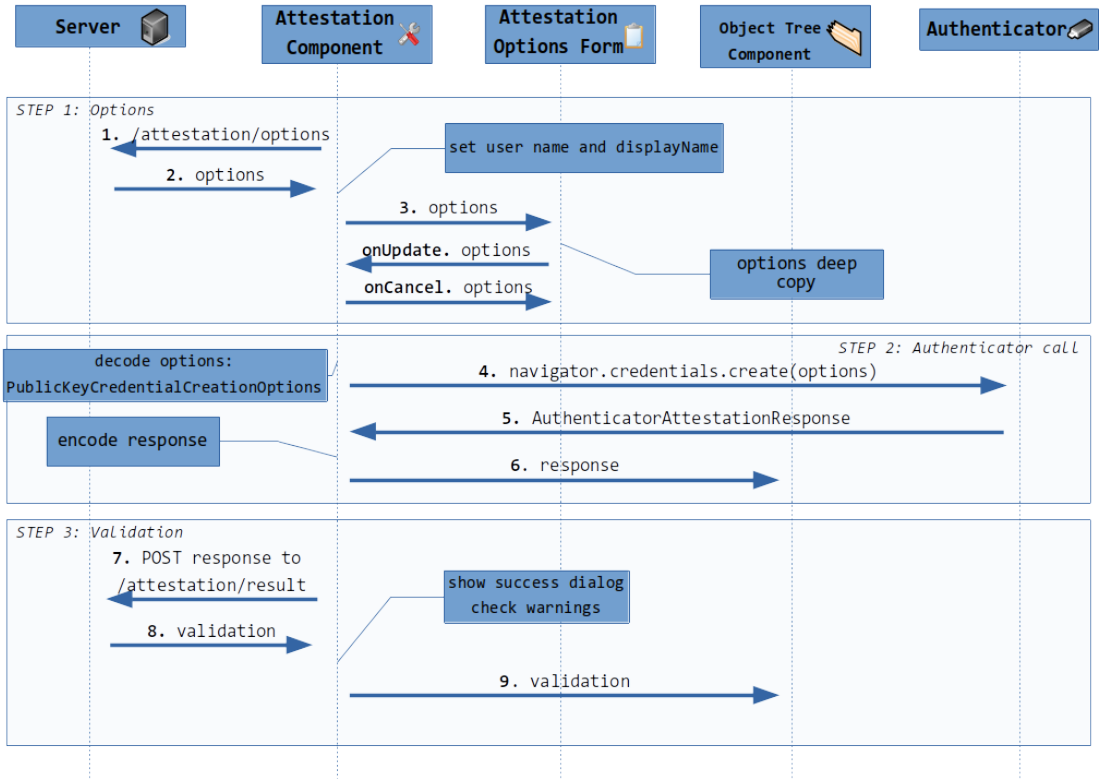


Figure 5.12: DebAuthn frontend flow for Attestation.

Required	**challenge**	
Optional	**timeout**	
	rpId	USVString
	allowCredentials	Sequence of {type, id, transports}
	userVerification	"required" | "preferred" | "discouraged"
	extensions	-

Table 5.4: PublicKeyCredentialRequestOptions.

5.3.5 User input validation

As explained in the previous subsections, some of the fields were optional, as defined by WebAuthn. Nonetheless, some of them are mandatory for the authenticator call to be sucessful. For this reason, the user input should be validated before updating the opreration configuration options. For example, mandatory fields of the object should not be empty.

Implementing this feature improved user experience and usability by using reactive forms. These forms can highlight an input field if it is not valid, while it deactivates the "Update" option of the form until all fields are valid (see figure 5.15).

Also, for providing usability a "cancel" button is useful for being able to roll-back the

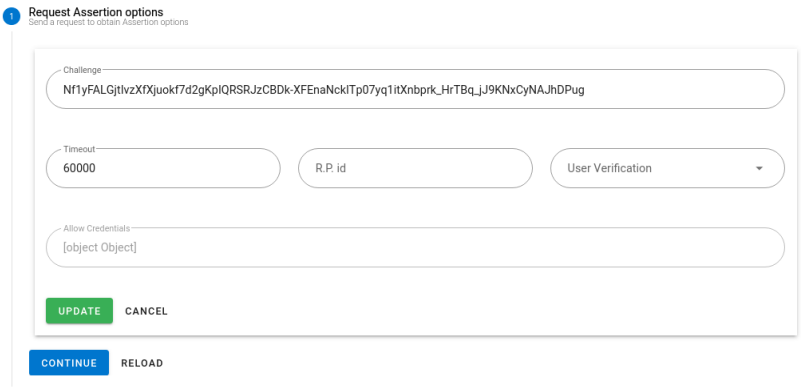


Figure 5.13: Assertion form initial implementation.

changes in the form. This functionality was achieved by doing a *deep copy* of the options to the form when loading them and restoring these options when cancelling the edition. At all times, the object copy is the updated state of the reactive form and will only be used for copying this state to the original options object when the user triggers the `updated` event via the "update" button.

By using Vuetify form functionality, the forms implemented have become reactive. That means that, upon the user input on a field, it gets dynamically validated. This validation can be programmed in validation functions. At this point, all additional validation functions for all fields are extensible, for future implementations such as the base64 correct encoding.

5.4 Structuring the displayed data

The version of the tool so far had no easy way to check all fields of the object returned by the Authenticator as a response to the operation, before it is sent to the Relying Party server. On the other hand, the validation the server was returning had many information that was not being used and has no meaning for the user. This iteration covered the improvement on how this data is parsed and shown to the user.

5.4.1 Relying Party validation data processing

Currently, the validation data that the server returns included innecessary data and follows a structure not coherent with the client needs, as it is directly forwarded from the fido2-lib library response. For this reason, the information like the operation warnings, the output were structured in a new data model that the frontend can use for finding the information needed.

This `Validation` object contains a boolean stating the validation of the operation,

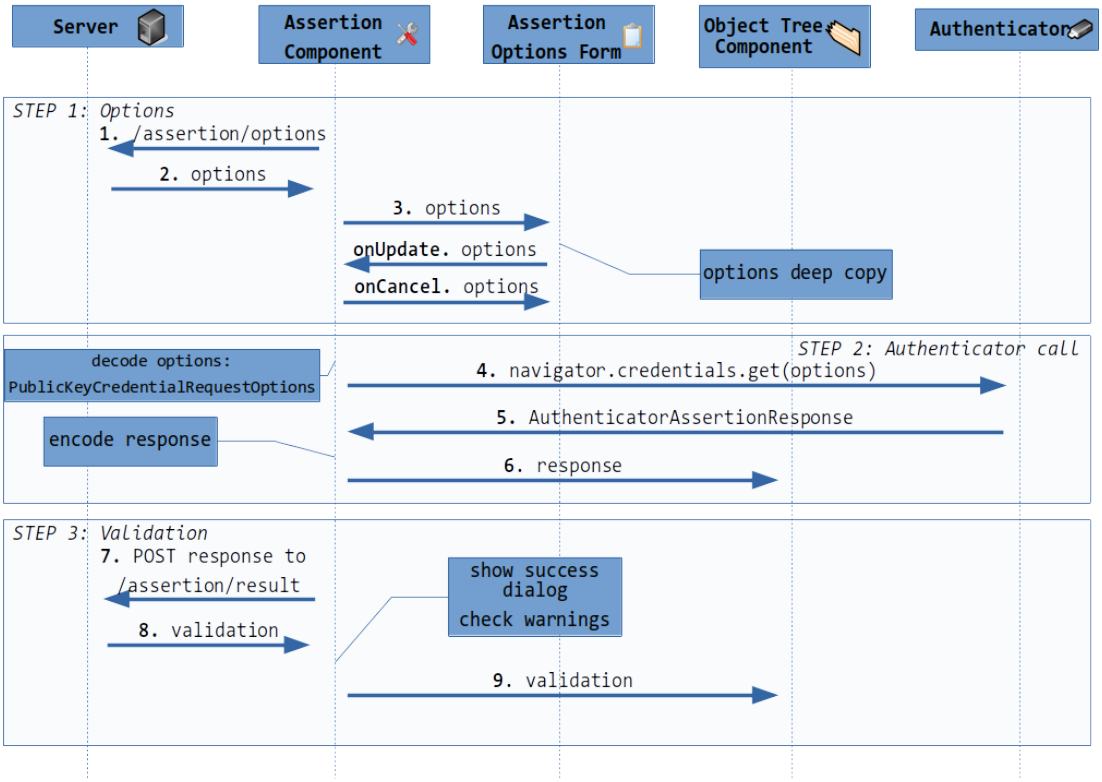


Figure 5.14: DebAuthn frontend flow for Assertion.

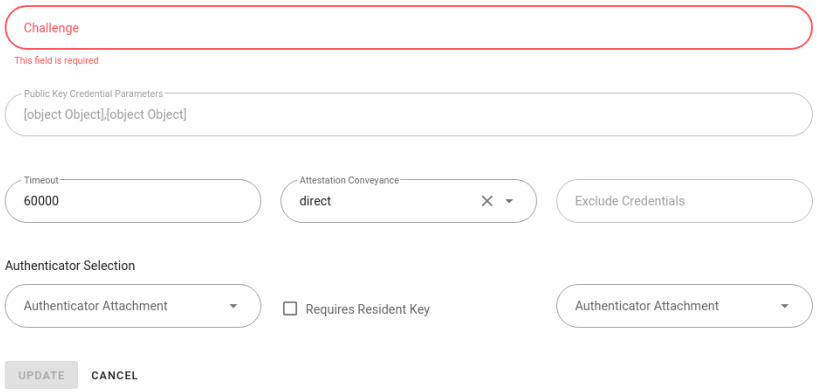


Figure 5.15: Reactive invalid form.

obtained from the audit data contained in the Result object the fido2-lib. Also, this audit data contains some warnings containing information about the validation process, so they are also included in this model. Finally, the data object includes the information contained in the audit data and both authenticator and client data (see table 5.5).

This model is used at the server to send the validation result in both Attestation and

complete	Boolean	
warnings	Array of Strings	
data	**info**	Object
	authnrData	Object
	clientData	Object

Table 5.5: Designed Validation object as the server response.

Assertion operation. At the client, the complete and the warnings section are used as control while the data section is directly displayed to the user.

At this point, some data loss was found during serialization. As it can be seen in the table 5.5, the data fields are Objects. However, the fido2-lib library provides Map objects. It was found that the `JSON.stringify()` function, used in serialization of this data as JSON, did not support this object structure. Therefore, the Map objects were converted to standard objects before their serialization. This was implemented as a recursive function that creates plain objects from Map and Set objects, checking that all fields can be correctly stringified, avoiding any data loss due to serialization.

5.4.2 Operation warnings after validation

According to the `validation` model and the fido2-lib, the server's final response may include some warning messages, containing relevant information for the user to understand any issues not constituting an error on the operation. For instance, one case considered by the library not finding the attestation root certificate of an authenticator maker on the server side so the attestation could not be verified.

The information can be shown to the user by using a warning alert component inside the dialog showing the operation success. In order to do this, the VueJS for directive structure was used, rendering an alert component for each error message (see figure 5.16).

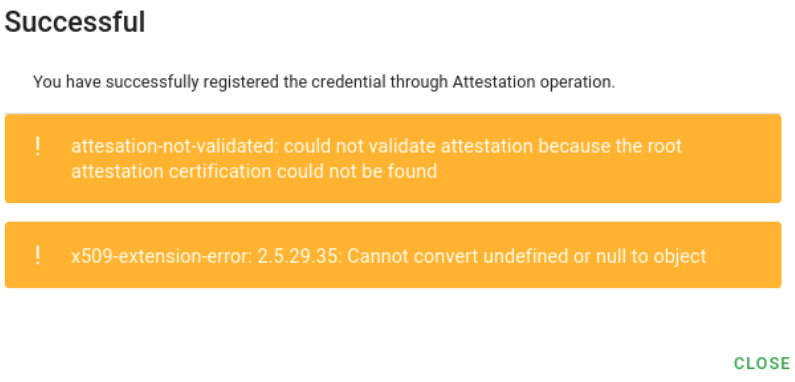


Figure 5.16: Warnings included in the VueJS dialog.

5.4.3 Structuring Authenticator response and validation data

Once the output data is available, it should be shown in an ordered way and also allow the user to copy and navigate through any data that would be useful for the debugging process. For instance, copying the attestation certificate, the signature or the public key. This applies both for the Authenticator Response object and the Validation response fetched from the server. The Authenticator Response would serve the user with the credential id and the response data, namely the authenticator data and the attestation or assertion object and the client data JSON. The Validation response would provide the information explained in the previous section: the validation.

One option to show this data is adding a text field for each piece of data. With this solution, should the object containing the data change its structure, the frontend code would have to be updated.

The other option was to build a common solution for all these cases where the user needs to navigate through some data stored in an object. This information has been represented with a component that is able to show all object attributes keeping the object structure in a parent-children tree similar to the JSON structure.

Therefore, the designed component goes through all the attributes of the concrete object, creating the tree structure component. The attributes serve as the tree leafs, stringifying their contents. In the case an attribute is itself an object containing more attributes, it is recursively processed, generating the parent-children aforementioned structure. Also, if the attribute does not contain information, it is shown with a different icon next to the attribute name (see left part of figure 5.17).

Figure 5.17: Object tree component for showing objects at the frontend.

Additionally, for allowing the user to copy the string contained in each of the tree leafs, the component was implemented to listen to the onClick events. These events trigger a Vuetify card component that shows the selected data (see right part of figure 5.17).

5.5 Support for several registered credentials

Until this iteration, the system allowed registering a credential with an authenticator and then authenticating with the credential in the system through the Assertion operation. This registered credential (the public key) was stored in the session information so it can later be used for authentication, according to the design explained in the first implementation, where all data is stored in the session information. However, an authentication system would eventually allow a user to register more than one credential in a system. This is unusual in password-based authentication, but it is particularly interesting in systems based on authenticators for supporting a backup authenticator.

For instance, a user could have a second factor bluetooth authenticator and a backup key as an usb authenticator, as proposed in the Titan Security Keys from Google [5], in the case the first registered key is lost, following the FIDO recommendations [2].

In order for the system to support having several registered credentials[3], it should store all of them. From a debugging or testing point of view it is interesting, so, when authenticating in WebAuthn, the credential ids could be included in the allowCredentials optional Assertion option. This option allows the authenticator to select from several registered available credentials. Until now, DebAuthn was including a single credential id, as there was only one registered credential.

Like when registering a credential, during authentication, the server creates some expectations after generating the challenge when the Assertion options are requested. The information includes the challenge, the publicKey and the counter, so when validating the operation that is taken into account. However, due to the implementation of the fido2-lib library, when generating the expectations needed for validating an authenticator response in the Assertion operation, only one public key is taken into account. This means the server needs to know which credential is going to be used to authenticate before the authenticator response is validated through the Assertion Result function.

For supporting several credentials, first, the user interface was redesigned to include all allowed credentials during authentication in a list with a form, so the user can modify it and add new credential ids. Moreover, the server now stores all registered credentials and the verification process takes now all of them into account.

5.5.1 Allowed credentials in Assertion

Manipulating a list of allowed credentials with variable length is more complex than modifying a simple text field. This required the design of a new component. Once the allowed

[2] Although the problem of recovering from a device lost is also treated by the Preemptively Synced Key protocol [60], the most common solution is the registration of a backup authenticator.

[3] There can be more than one registered credential in the same authenticator.

credentials are requested to the server, they are displayed to the user. Also, it can be interesting for the user to remove one or more credential ids from the list. Besides, allowing the user to append a new credential id can be used to check if a credential is still present at the authenticator, although removed from the server (see figure 5.18).

Figure 5.18: AllowCredentials form for the user to add a new credential id.

As there are many elements inside the list fetched from the server, the form needs to render a card component per credential id. This was implemented by using a VueJS for directive structure, looping on the list elements and rendering each of them.

On the other hand, in order to implement the insertion of a new credential id, there should be a text input for the user to copy and insert the correspondent id. Once validated, the credential id is pushed into the allowCredentials list of the form. It is worth mentioning that the credentials included in allowCredentials have the following structure:

```
{
    type: “public-”key
    id: pubCredId
}
```

Additionally, in order to delete an id from the list (see figure 5.19), each card includes a delete button, represented by a key with an 'x'.

Finally, the model representing the `PublicKeyCredentialRequestOptions` has been modified to take into account the existence of multiple credentials in allowCredentials, looping through each of them in order to encode and decode the credential id. Like the challenge, the credential id is a Buffer, and it is encoded into a Base64Url string.

5.5.2 Registered credentials at the server

The implementation until this point was based on storing a single credential. When a new credential was registered, it would replace the last registered credential. However, in order to keep track of the registered credentials for being able to authenticate with any of them, all of their respective public keys, ids and counters should be stored in the session.

As seen in the figure 5.20, for supporting several credentials, a new component was designed: `registeredCredentials`. This is a list containing all registered credentials,

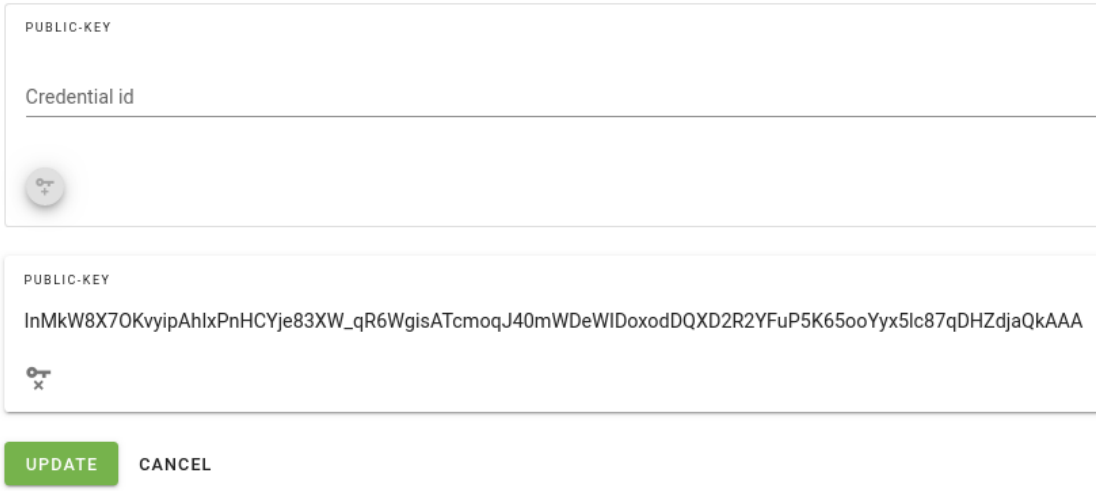


Figure 5.19: AllowCredentials list and the new form for adding credential ids.

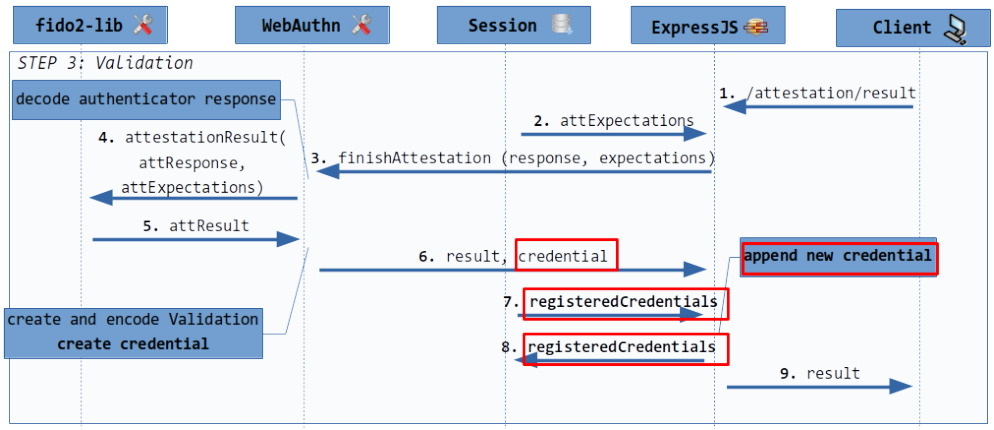
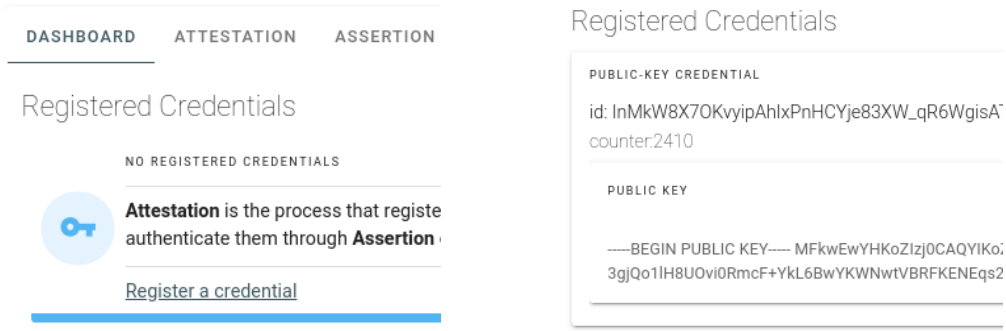


Figure 5.20: Validation flow at the backend for registering several credentials.

and it is stored in the session information. Then, when registering a new credential through Attestation, the server needs to add the credential, composed the public key, id and the counter, to this array, creating a list of all registered credentials.

Despite the fact that the server was keeping the public key and counter of several credentials, the user had no way of fetching this information. This is interesting so the user increases their awareness on which information the relying party is storing in the server. The most relevant may be the counter, as it changes with each authentication.

This feature required adding a new page in the user interface, in this case called Dashboard, that can be used as the root of the web, enrouted by default (see figure 5.21). The Dashboard renders a card for each registered credential, like `allowCredentials` list shown in the Assertion form, but also showing the counter.



(a) No registered credentials

(b) One registered credential

Figure 5.21: New Dashboard component: registered credentials.

For fetching these registered credentials, a new GET endpoint is designed (see table 5.6), which will serve the list of registered credentials to be rendered at the frontend.

Endpoint	Method	Request payload	Response payload
/registered	GET	*None*	registeredCredentials

Table 5.6: Designed REST endpoint for requesting the registered credentials.

5.5.3 Verifying Assertion

Supporting several credentials involves not only the registration process but also authentication. As said, the Assertion operation requires a list of the allowed credential ids, corresponding to the registered credentials at the server, which are sent to the authenticator. With this list, the authenticator will select one of the registered credentials it can use for authentication. However, in order to validate the authentication, the validation needs the public key and the last counter of the credential that the authenticator used in the operation [4].

This new feature implied a redesign of the authentication flow. As it can be seen in figure 5.22, now the allowCredentials list is built based on the *registered credentials*. It is important to notice that during step 1 (see figure 5.22), the server does not know which credential will be selected by the authenticator, so the backend does not create the expectations with a concrete credential public key and counter. Instead, at this step, the session information was changed to store only the challenge (together with origin and factor).

Then, during step 3, once the authenticator performs the Assertion, the Authenticator Response will include the credential id that was used in the operation. Therefore, this id is used to find the correspontent public key and counter in the aforementioned `registeredCre-`

[4] As an authenticator can have more than one credential registered in the system.

dentials component. Then, these used together with the challenge, origin and factor to create the Assertion expectations for validation (see figure 5.22).

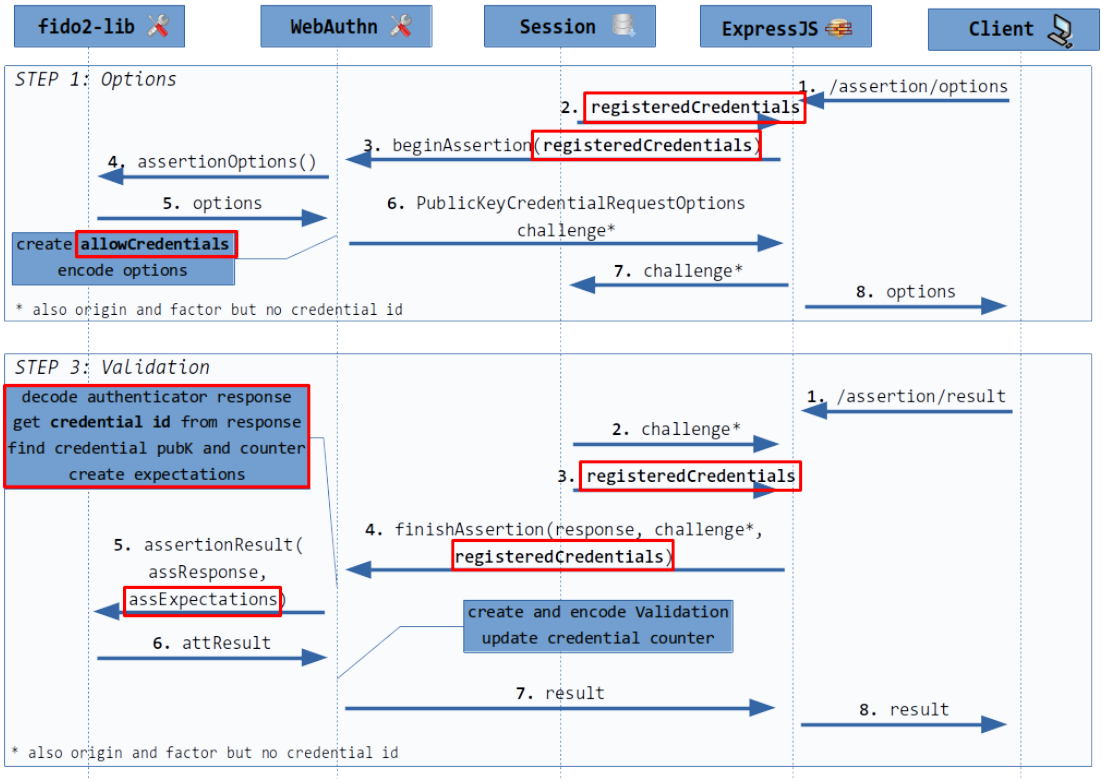


Figure 5.22: Backend flow for authenticating when having several registered credentials.

5.6 Android SafetyNet: a new Attestation format

Attestation formats are used by authenticators to pack the attestation data that is sent to the Relying Party (see section 2.2.4). In order to read this packed data, the server must be able to parse and validate the concrete format. So far, the main formats were supported by the fido2-lib library. However, Android Safetynet is not yet supported.

Adding support for this new Attestation format allows users to use a compatible Android device (>= v.7) with no need of a physical authenticator. Android SafetyNet attestation format is mainly used by the Android platform when the device screen lock option is used.

5.6.1 Implementing validation for Android Safetynet attestation format

As the fido2-lib library does not include support for Android Safetynet, for addding support of the format, it was necessary to implement a custom validation function that parses and validates the attestation. This attestation format has a different cryptographic schema compared

to physical authenticators, as seen in Appendix D. The implementation of the attestation validation was based in this schema.

Once the server receives the response from the client authenticator, it needs to validate the response. First, according to the format, it is parsed and then verified. In this case, the verification steps are both included in the Android documentation and the WebAuthn standard, as mentioned above. Generally, it includes the following steps:

1. "Verify that `attStmt` is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields."
2. "Verify that response is a valid SafetyNet response of version `ver`."
3. "Verify that the `nonce` in the response is identical to the Base64 encoding of the SHA-256 hash of the concatenation of `authenticatorData` and `clientDataHash`."
4. "Let attestationCert be the attestation certificate."
5. "Verify that `attestationCert` is issued to the hostname "attest.android.com" ([30])."
6. "Verify that the `ctsProfileMatch` attribute in the payload of response is true."
7. "If successful, return implementation-specific values representing attestation type Basic and attestation trust path `attestationCert`."

```
safetynetStmtFormat = {
                              ver: text,
                              response: bytes
                        }
```

Most of the steps are common to other attestation validations, like the checks on the attestation certificate or the CBOR decoding, although specific to this format. However, the response includes two important fields that need to be verified: the nonce and the integrity verdict ctsProfileMatch.

The nonce is the data sent to the Android SafetyNet API by the browser. This is the information that is attested by the Google servers. It is generated, as the protocol explains, with the concatenation of the authenticator data and the client data hash. Then, this concatenation is hashed and encoded in Base64. Therefore, in order to check the integrity at the server side it should perform the same operation and check both the generated nonce and the one provided in the response match.

$$nonce = sha265(concat(authenticatorData, sha265(clientData))) \tag{5.1}$$

Furthermore, Android SafetyNet also provides two integrity verdicts. WebAuthn specifies that ctsProfileMatch should be checked. According to the API documentation, this verdict states a strict device integrity, as opposed to basicIntegrity verdict.

In order to add an attestation format to the fido2-lib library, it is required a format name, a parsing function and a validation function. Parsing involves taking into account the format stated above, with a version and a response field. When validating, the response needs to be parsed as a JWS (JSON Web Signature) to extract all the fields to perform the validations explained before.

For this purpose, the `jose` library [61] is used, as it provides functionalities for JWT, JWS, etc. Once parsed and verified, the payload field should contain the Android SafetyNet data. After checking the device integrity with the verdict, the function generates the nonce from scratch by using the `crypto` library, together with the raw authenticator and client data.

Finally, according to the API documentation, the payload includes an error and an advice field so, should an error occur, an exception is thrown with the advice message, so the R.P. can better handle it.

5.6.2 Integration of the implementation in DebAuthn

This implementation drived into a contribution to the library used by DebAuthn. As explained before, the library did not have support for the format. Appendix E explains the contribution, also describing the unit tests written for this implementation.

It is worth mentioning that, as the main library (`fido2-lib`) had no recent maintenance, from now on, the project uses a maintained fork (`fido2-library`).

5.7 Improving user experience

This iteration was intended to improve the user interface. Previous iterations were focused on adding functionality with no instructions to the user on how to use the tool. Also, this iteration included a new feature for detecting browser compatibility, warning the user before using the developed tool. Finally, two of the fields inside the registration options were improved to better display the information and making them editable.

5.7.1 Giving instructions to the user

The current version of the tool at this point did not provide the user any information related to the tool design and purpose. Showing more information makes the tool more approachable for non-expert users. This was fixed by making the new Dashboard page a welcome page, as well as instructing the user on the tool usage in some sections (see figure 5.23).

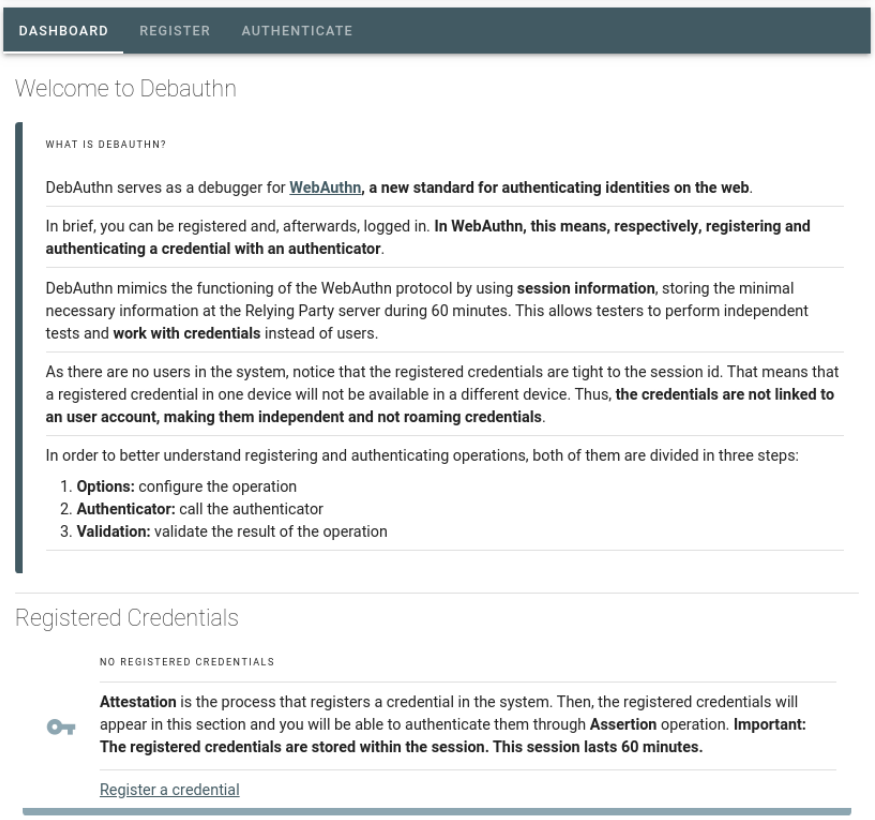


Figure 5.23: New Dashboard page.

5.7.2 Feature detection

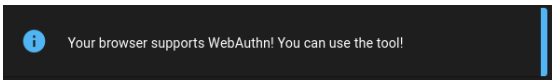
A web application running a script should ensure that the code is compatible with the browser, as to avoid running time errors. In the case of a debugging tool for WebAuthn, which is a brand-new protocol implemented only in the main browsers, this is particularly useful. By using feature detection, the web application can warn the user beforehand, so they try with another browser (see figure 5.24).

The development of this feature can be approached in two main ways: using the user agent string or using feature detection.

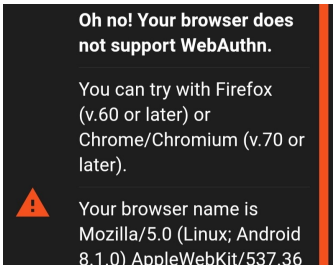
Parsing a user agent gives the name and version of the browser the code is running on. However, this string is easily modifiable and does not guarantee a correct identification of the browser. Also, in order to match the browser name and version with the compatibility information, a database should be often updated to be accurate.

On the other hand, feature detection is totally accurate when checking if a specific feature is available in a browser, by running some tests before executing the code. In this tool this is very useful, as it detects a compatible browser independently of the browser name and

version. This means that, if a new browser implements this protocol, the tool does not need to know this in advance, as it can identify the feature in running time.



(a) Supported browser.



(b) Unsupported browser.

Figure 5.24: Feature detection fontend information boxes.

For this purpose, the developed piece of code checks the availability of the Credential Management API [13], and also for the `PublicKeyCredential` class, that is used by the browser for managing public-key credentials, used in WebAuthn.

5.7.3 Improving two key fields of the attestation options

Previous iterations have included all registration options in the attestation options form. However, the user could only edit or configure some of them. This iteration involved the implementation of two registration options fields: **pubKeyCredParams** and **excludeCredentials**. After this, the user can select the algorithms requested to the authenticator during registration and the list of already registered credentials. For example, by allowing the user to manipulate the list of already registered credentials, the authenticator could be registered again with a different credential (see figure 5.25).

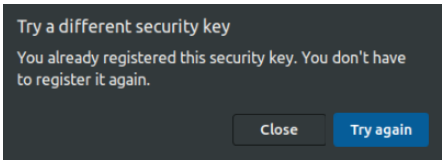


Figure 5.25: Failure to register an authenticator that was already registered.

Regarding the user interface, both of them were based on the approach designed for allowCredentials when authenticating (see figure 5.19). That is, an input field with an “add” button and a component rendered for each of the items contained in the list, along with a delete button (see figure 5.26).

For the public key credential parameters, the cards contain the algorithm identifier according to the IANA registries [62]. It is the user responsibility to include a valid identifier. All the identifiers included here are then attached to the Attestation options so the authen-

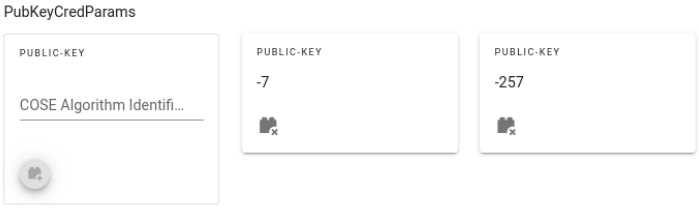


Figure 5.26: PubKeyCredParams new component in the Attestation form.

ticator will know which cryptographic algorithms the Relying Party supports. It is worth mentioning that the algorithms included in the options fetched by the server are the only ones supported by the server. Any change on this list may cause an invalid validation due to unsupported formats when parsing the authenticator response.

5.8 Extending possible tests

This last iteration included the development of a two extra functionalty that extended the the available testing options of DebAuthn. The most important of them is the support of resident credentials, which includes this type of credentials available for FIDO CTAP2 [7] authenticators. On the other hand, a new option was implemented to allow users to delete all their registered credentials. This is interesting for testers of the protocol that need to register credentials several times.

Finally, the tool has been published at `https://debauthn.tic.udc.es`.

5.8.1 Adding support for resident credentials

Resident credentials are compatible with the FIDO CTAP2 authenticators, which store the private key in the device memory, as explained in section 2.2.5. Currently, not all authenticators, browsers, operating systems and platforms support resident credentials. For DebAuthn, it is interesting to support registrating an authenticator with a resident credential, demonstrating its compatibility. As explained in section 2.2.5, this type of credentials are used in ”passwordless” authentication flow, the main idea of the FIDO2 project [11].

This implementation required redesigning the server Attestation and Assertion flow. Also, it was necessary to patch the fido2-lib library, as explained at the end of this section.

It is worth mentioning that, when authenticating with a resident credential, the authenticator will return a userHandle, set during registration as the user id (see figure 5.28). In the previous design, the assertion expectations do not include the userHandle, as when authenticating with non-resident credentials the authenticator, this field was null.

Therefore, a redesign was needed for saving the user id, set during registration, together

with the other credential information in the server (see figure 5.27). Moreover, during authentication, the assertion expectations need to include the userHandle as the user id, for its correct validation of the authentication operation with resident keys.

Usually, in an ordinary WebAuthn flow, the `userHandle` would serve the system to retrieve the list of registered credentials for a specific user from the database. In this case, as the registered credentials are not linked with a user account, this `userHandle` is saved together with the id, public key and counter of the credential, inside the `registeredCredentials` component.

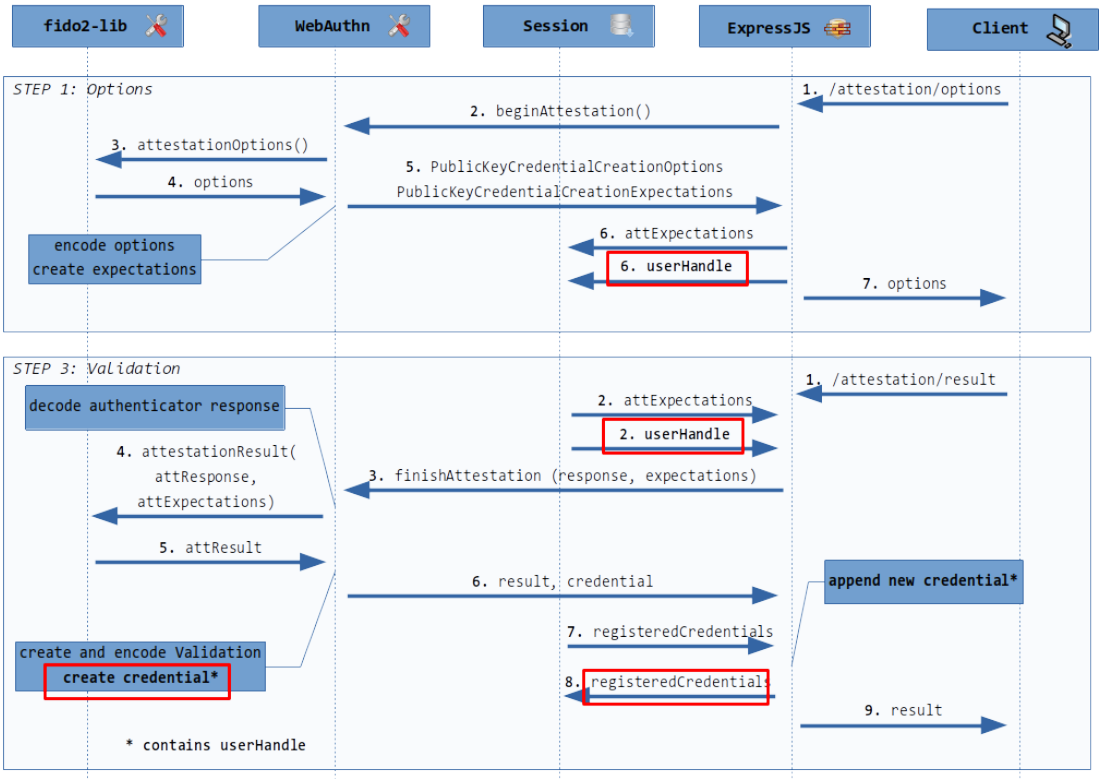


Figure 5.27: Backend Attestation flow with the user handle to support resident credentials.

Finally, when using resident credentials, the authenticator may return some data in the extensions. However, as the fido2-library did not support WebAuthn extensions, it was throwing an error during validation of when registering resident credentials.

For this implementation, the Relying Party can ignore the extensions during validation, as they are optional data. According to the standard, ignoring extensions is not considered a failure. In order to ignore such extensions, the library code that throws this error was also patched.

This patch could be done in two ways. First, by contributing to the library code, like in some previous implementations. Secondly, NodeJS dependencies can be patched by using the

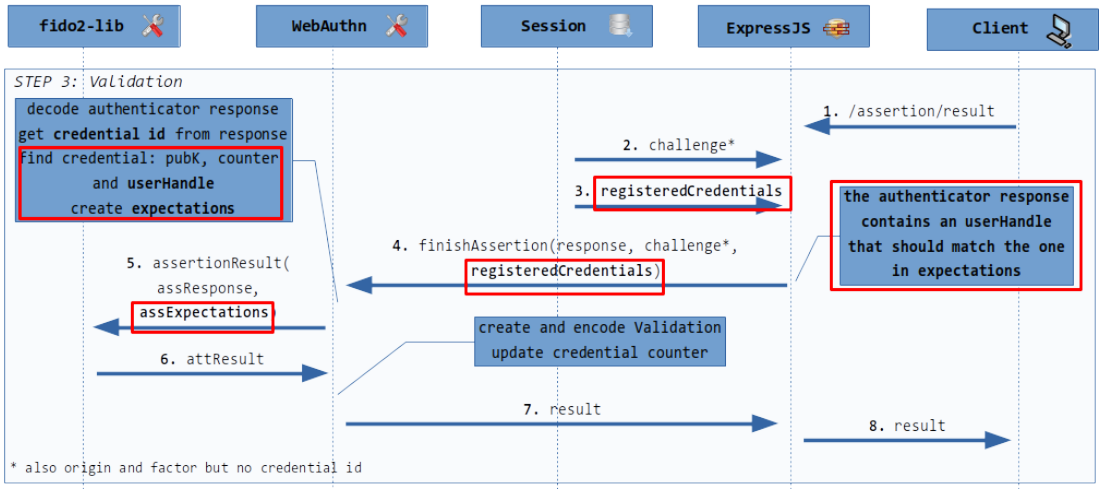


Figure 5.28: Assertion validation flow with the user handle to support resident credentials.

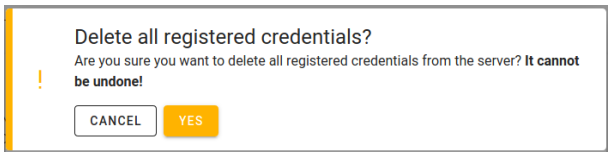


Figure 5.29: Confirmation dialog before deleting all credentials.

`patch-package` utility [63]. It allows generating and applying patches to the dependencies' code, once installed in the project. This second option is independent of the library code and adaptive to changes. Also, previous patches of the library like when allowing the reuse of the attestation options, explained in the section 5.3, can be included.

5.8.2 Delete all registered credentials

Until now, if a user was registering many times for testing purposes, they would find the amount of registered credentials are kept and, therefore, returned during Attestation and Assertion within the options, as described in previous sections.

However, a user may want to delete all registered credentials at some point, cleaning the workspace. For supporting the deletion of the credentials, a new option was implemented and added to the developed tool. This triggers a confirmation dialog (see figure 5.29) and, upon confirmation, an asynchronous DELETE REST request (see table 5.7). At the backend, it will delete all elements of the registeredCredentials in any case.

Endpoint	Method	Request payload	Response payload
/registered	DELETE	*None*	*None*

Table 5.7: New DELETE REST endpoint for registered credentials.

Chapter 6

Testing and studying authenticators

The developed tool (DebAuthn) serves to researchers to test both authenticators and browsers. This chapter seeks to test interesting characteristics of five physical authenticators (see figure 6.1) under the same situations by using the available options of DebAuthn, whose development was explained in chapter 5. Table 6.1 shows the devices' details. Finally, in order to provide more information for helping organisations to potentially select one of the tested keys, the chapter also studies other additional features. These features are key to understand complete functionality or capabilities of the studied hardware devices.

Figure 6.1: The five hardware authenticators picture.

The testing environment is important. In this case, Ubuntu 18 Linux operating system has been used, together with the Chromium v83 browser, which is the best browser regarding the support of WebAuthn. In order to test the bluetooth capabilities of one of the keys, it was also used an Android 7 with Chrome v83. Moreover, before each test, the keys were reset to defaults.

The tests were run on a Linux OS because the browser is the one in charge of implementing

Maker	Model	Firmware
Yubico	Security Key [64]	5.2.4 [65]
Yubico	Yubikey 5 NFC [66]	5.2.4 [65]
Google	USB-A/NFC Titan Security Key (K9) [32]	(version T3)
Google	Bluetooth/NFC/USB Titan Security Key (K13T) [32]	(version T3)
Solokeys	Solo Hacker [67]	4.0.0

Table 6.1: Hardware authenticators tested in this degree thesis.

the CTAP communication. This does not happen in other OSs. Windows and Android, for example, would limit the tests as browsers interact with authenticators through a platform implemented by the OS.

6.1 Attestation mechanisms in hardware authenticators

The first of the testing studies covered the attestation mechanisms in hardware authenticators. As explained in section 2.2.4, the Attestation operation returns an attestation object in one of the available formats. This formatted object was generated by one of the attestation types: Basic, Self, CA, ECDAA or None.

This section checks the Attestation formats and types the authenticators use under specific configurations. Finally, some tests were repeated with a different browser to verify the differences.

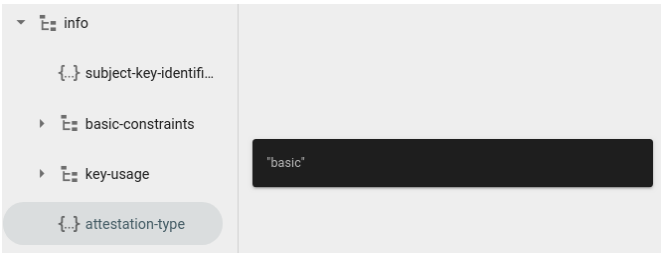
6.1.1 Authenticator compatibility

DebAuthn allows configuring and validating attestation for testing authenticators. Taking into account the available hardware authenticators, the attestation type and format can be checked upon validating the registration process. In order to see the differences among them, they should be tested under the same circumstances. Therefore, the keys should be reset to the default state.

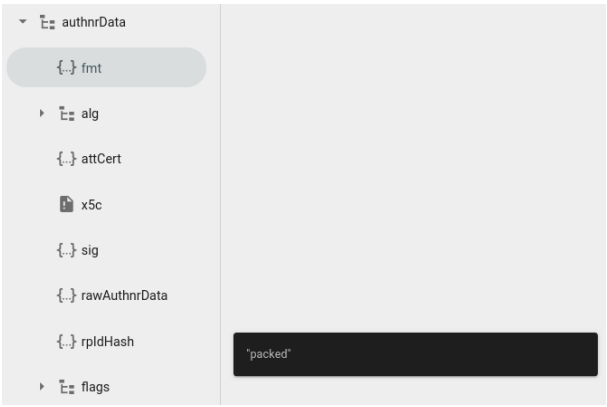
As expected, when setting the attestation conveyance to “none”, the attestation format found in the responses from all the devices is “none”. This is used by Relying Parties when they do not wish to receive attestation information. In order to check the attestation type and format, the conveyance should be set to “direct” or “indirect”. In this environment and with the study case, both “direct” and “indirect” answers have given the same result.

The main difference is that both Google Titan Security Keys use “fido-u2f” attestation format, while both Yubikeys and the Solokey use the “packed” format (see figure 6.2). All keys are performing, however, the same type of attestation: “basic”. According to the WebAuthn

standard, the only valid algorithm for “fido-u2f” attestation format signature is “ES256” (“-7” code).



(a) Attestation type.



(b) Attestation format

Figure 6.2: Attestation type and format with the SoloKey tested in DebAuthn.

For this reason, the experiment was repeated with the Google keys by excluding this algorithm and maintaining the “RS256” and the “EdDSA” algorithms, trying to force the keys to select another attestation format. As a result, both keys failed to complete the operation. While the test on Linux resulted as the key not responding to the request (timed out), the test on Android 7 with Chrome resulted in an browser error, as none of the algorithms provided were supported on the device.

On the other hand, Yubikey allows users to activate or deactivate the device’s interfaces by using their software “Yubikey Manager” [68]. Once deactivated FIDO2 in both Yubikeys, leaving only FIDO U2F, the test is repeated. In this case, like Google keys, responds with “fido-u2f” attestation format. Also, when excluding the “ES256” algorithm, both Yubikey devices will timeout the request.

As conclusion, the Google Titan Security Keys appear to be using FIDO U2F, backward compatible with FIDO2 (see section 2.2.7), while the Solokey and both Yubikeys are using the optimized “packed” attestation format used in WebAuthn. In fact, the Google Titan Security Keys are meant for being used as a second factor authentication method in the Google services and “many other services that support FIDO standards” [32].

6.1.2 Browser compatibility

The aforementioned tests were performed on Chromium on a Linux machine. However, when the tests are performed on Firefox v77.0.1 on the same Linux machine, all attestation signatures are formatted with "fido-u2f", instead of "packed".

One of the possible reasons is that this Firefox version does not support FIDO CTAP2, but it does support FIDO CTAP1 (a.k.a. FIDO U2F) [69].

6.2 Communicating with authenticators: the transports

The CTAP protocol [7] defines how the client or the platform communicates with the authenticators. Also, the standard defines transport-specific bindings for USB, NFC and Bluetooth.

One option for checking the available transports is testing the hardware devices with the specific protocols they are meant to support. However, there exists an attestation certificate extension defined by the FIDO U2F protocol, called fido-u2f-transports, which is a X.509v3 certificate extension [70]. Therefore, when the certificate is validated at the server, the library will parse this extension and its details will be included in the validation, sent and shown to the user.

Taking that into consideration, this experiment has used the displayed information in DebAuthn to check which transports were included in the attestation certificate extension and check whether they are the expected available transports.

After the experiment, both Yubikeys and both Google Titan keys successfully included all their respective available transports (see figure 6.3). For instance, the K13T key from Google, has included NFC, USB and BLE, while the Yubico Security Key only includes USB. However, the Solokey device seems not to be using this certificate extension. Currently, the Solo Hacker is only available for USB transport.

6.3 The cryptography: supported algorithms

This section explains the study that was designed to verify the available algorithms of the authenticators and identifying the most common ones. The result is interesting for guiding the Relying Party implementations in supporting these algorithms to be able to allow their users to use specific hardware authenticators.

An authenticator may implement several cryptographic signing algorithms. These algorithms are registered in the IANA CBOR Object Signing and Encryption registries [62], which includes the codes for all algorithms. A R.P. would include the supported algorithms in the Attestation options in a specific order. The authenticator will then use the most preferred available algorithm and use it to encode the credential public key.

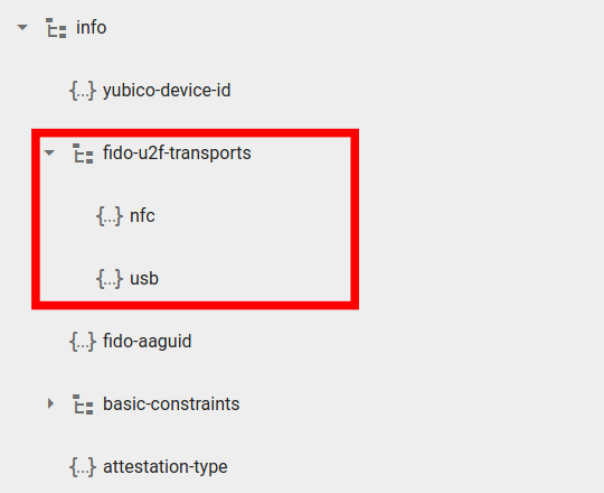


Figure 6.3: Supported communication transports of the Yubikey5.

The most common algorithms, according to the ones mentioned in WebAuthn and CTAP2 standard, are the ones contained in table 6.2. It is worth mentioning that according to [71], the RSA and the current ECDAA algorithms are not recommended, being ECDSA and EdDSA the best options.

Algorithm identifier	Algorithm name	Description
-7	ES256	ECDSA w/ SHA-256
-8	EdDSA	EdDSA
-257	RS256	RSASSA-PKCS1-v1_5 w/ SHA-256

Table 6.2: Main cryptographic algorithms used by authenticators in WebAuthn.

For verifying which are the available algorithms by each hardware key, an experiment was designed. By using a DebAuthn option during registration, the pubKeyCredParams list was edited so the tested algorithm will be included while excluding any other algorithm identifiers. Then, if the authenticator can successfully return a response, then the algorithm will be available. If the algorithm is not supported, the request will timeout as the authenticator will never respond to the request.

Regarding the Yubico hardware keys, there is some important information about the algorithms. From the firmware version 5.2.3, Yubico has added support for the Ed25519 algorithms as well removed support for RSA keys [65]. In this case, the firmware version of the tested keys is 5.2.4, so these changes are already in place.

Once tested on both Yubikeys, with the same firmware version, it was found that they both support EdDSA and ES256 algorithms, while the RSASSA algorithm is not supported. Note that, as EdDSA is not provided in the default registration (Attestation) options by the

R.P., so DebAuthn will fail validation as it does not support the algorithm. This confirms the tested algorithm is using another format, and that the server would need to support it in order to validate the authenticator response.

Regarding Google keys, the version T3 of the Google Titan security keys were conclued with no support EdDSA nor RSASSA algorithms, being ES256 the only supported algorithm, necessary for the fido-u2f attestation format, as explained in section 6.1. In contrast to the Yubikeys, Google has no documentation about their authenticator devices. In fact, the author has not found any information about how to check the firmware version or which characteristics it has.

The Solokey, although there is no documentation on the supported algorithms, its firmware is open source. After running the same tests, it only responds when the ES256 algorithm is in the parameters. Therefore, like the Google devices, it only supports this algorithm in its 4.0.0 firmware version.

Finally it is worth mentioning that, although these tests were performed in Chromium in a Linux environment, other browser and OS combinations responded in a different way, possibly because of the fact that the interaction with authenticators is performed by the OS platforms mentioned before. For example, when performing the same tests in Google Chrome on Android and in Firefox on Linux, they both rejected any algorithm other than ES256, that is supported by all tested keys.

In fact, when running the test with Firefox on Android, the OS platform will convert the request to support only ES256. This was found out when the EdDSA algorithm was requested to the Google BLE security key and, when checking the validation result, the algorithm used was "ECDSA_w_SHA256", also known as ES256. The same test using Chrome on Android did not succeed as the browser displayed an error message stating that none of the algorithms requested is supported by the device. This occurs both with EdDSA and RSASSA algorithms.

6.4 Testing support for resident credentials

This section studies the support of resident credentials which, as explained in section 2.2.5, are the ones saved in the authenticator memory. Testing how physical authenticators support resident keys is key for determining whether the key can be used in a "passwordless" authentication as a first factor. Apart from the web authentication, this type of credentials are starting to be used in other systems. For example, resident credentials are used for SSH authentication with physical security keys since its version 8.2 [72].

As resident credentials are designed for a first-factor authentication flow, the R.P does not provide the allowed credentials in the Assertion options during authentication. This forces the authenticator to find a credential in its memory only by the R.P. id, without a credential

id. Then, once Assertion is performed, it will return a user handle (user id) in its authenticator response.

However, for this type of authentication, the authenticator needs to be protected, as it will serve as a first-factor authentication method and may serve as the unique factor for authentication in a system. For this reason, when using resident credentials, a R.P. should force the authenticator to perform *user verification* during the registration process, explained in section 2.2.5.

6.4.1 Authenticator compatibility

As explained before, resident credentials is a feature of new authenticators that open new possibilities. The market differenciates compatible keys by tagging them with ”FIDO2 key” versus ”FIDO U2F key”. However, for some authenticators like the Google Titan Security Keys, do not specify its compatibility with resident credentials. This study has demonstrated its incompatibility.

This experiment considered keys reseted to defaults. Regarding the required Attestation options, requireResidentKey is set to true and the userVerification is set to “required”. This way we ensure the authenticator selection requests a resident key and performs user verification. Also, the browser should support user verification. As said, Chromium on Linux is used to ensure this capability.

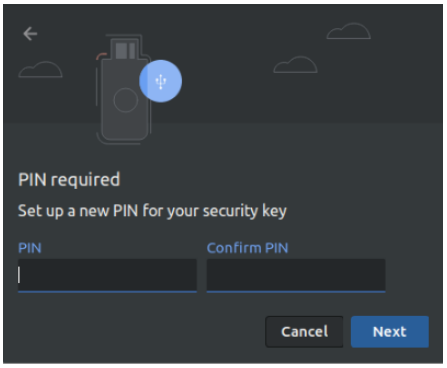
Both Yubikeys and the Solokey were proven to be compatible with resident keys, both having the same behaviour. Firstly, as they were reset and do not have a PIN by default, when the registration was requested, the browser prompted to set a new PIN, as shown in the figure 6.4). Once the PIN is set, the authenticator replies and the server validates the registration successfully.

Moreover, to ensure the key is resident at the authenticator, the authentication operation was tested. The assertion options should include an empty allowCredentials, so the only way the authenticator can sign is with a resident credential containing the private key. When the request is triggered, the dialog prompted by the browser also shows an additional message: “a record of your visit to this site will be kept on your security key”, referring to the resident information stored in the key (see figure 6.4).

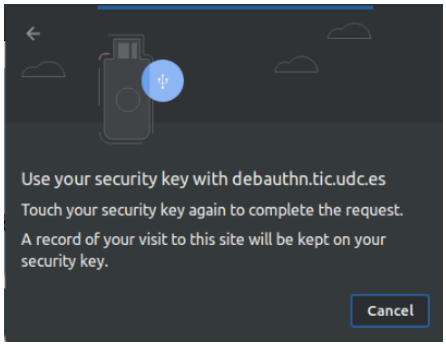
Also, when checking the authenticator response during authentication, a `userHandle` was found. It actually corresponds with the user id set during registration. This user handle or id is only returned when a resident credential is used, as it is stored along with the private key in the authenticator memory. As it will be shown in section 6.5, the resident credentials can be listed by using the configuration manager embedded in Chromium. After performing the operation, the key can be correctly listed, showing the user information as well as the web application origin or domain name.

On the contrary, when the same registration options were used with the Google keys, after performing the user presence, the browser shows a failure message: "your security key can't be used with this site". Also it suggests a newer or different key may be required to perform this type of registration (see figure 6.4).

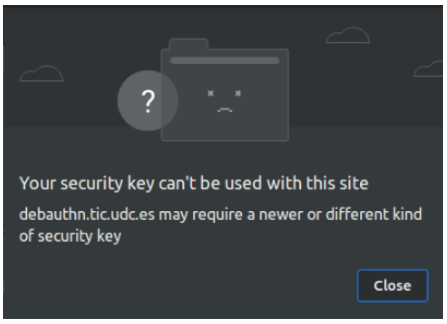
That concludes that the Google Titan security keys version T3 do not support resident keys. This result, joint with the study presented in the attestation format section, concludes that the Google Titan Security Keys do not work with FIDO2 and are FIDO U2F authenticators.



(a) Required PIN.



(b) Registering a resident credential.



(c) Authenticator not compatible.

Figure 6.4: Chromium dialogs when registering and authenticating with resident keys.

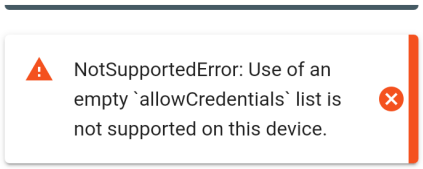
6.4.2 Browser compatibility

As mentioned before, browsers need to support user verification to be compatible with resident credentials. The browser used for testing the authenticators was Chromium, which was compatible. However, this is not the case for other browsers and platforms.

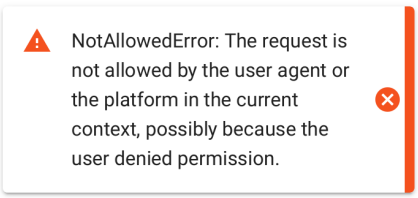
When registering resident credentials in Firefox on Linux the request will timeout in any case. This is possibly caused by the fact that this browser does not support the CTAP2 operations for requesting a PIN for the devices. According to the attestation formats section, Firefox may not support FIDO CTAP2 so far, only supporting FIDO U2F, which does not include user verification.

On Android 7.1.1, when using Firefox or Chrome with resident keys, the registration process did not fail, regardless of the authenticator used. However, when authenticating with an empty allowCredentials so that the resident key is used, both browsers fail with similar error messages. They inform the user that the device is not compatible with an empty allowCredentials. Also, when used with a compatible authenticator like the Solokey, the registration did not request any user verification.

Therefore, regardless of the browser that is used, the tested Android is not supporting registering or authenticating resident credentials (see figure 6.5), although the registration operation was not failing.



(a) Google Chrome



(b) Firefox.

Figure 6.5: Resident keys fail during Assertion on Android.

6.5 Configuration and management available tools

When using authenticators with a Relying Party like DebAuthn for registration and authentication there is no need for an additional tool for configuration. The R.P. is responsible for it by using the WebAuthn API [6]. However, there exist some external tools that allow managing some aspects of hardware authenticators. This section will explore the most relevant ones for the tested keys in this work.

One of the most relevant and basic tools may be the Chromium (or Chrome) browser, as it has a menu for managing security keys (see figure 6.6). This uses common operations implemented in the CTAP2 protocol [7], which allow to create a PIN, reset a security key and even retrieve the sign-in data (the resident credentials). As they implement operations from the CTAP2 protocol, any device that is compatible with it can be configured with these operations.

All tested keys but both Google Titan keys support these CTAP2 operations. This possibly means that the Google devices implement CTAP1 instead of the last version of the protocol, a result coherent with the previous tests.

On the other hand, the hardware authenticator makers can personalise their firmware to include other configuration operations that are specific to the device make and model. For example, SoloKey offers a tool called solo-python [73]. This serves both as a programming

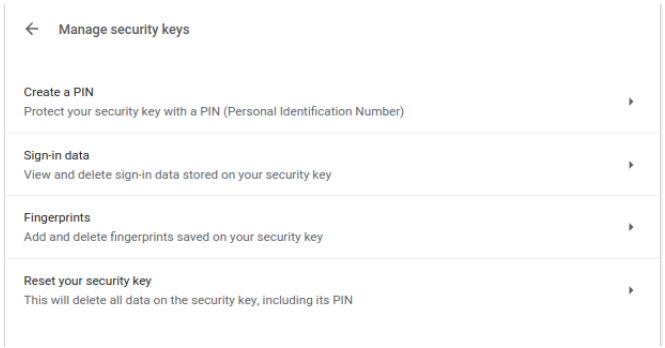


Figure 6.6: Chrome security keys embedded manager.

library and a CLI tool, with many options (see figure 6.7). For example, changing the attestation key, changing the PIN or using the generation of random hexadecimal bytes in the key.

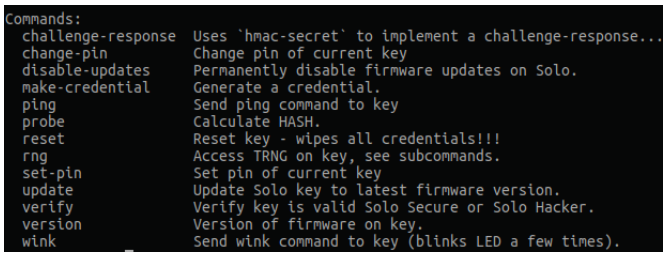


Figure 6.7: solo-python SoloKey CLI manager.

Another example is the Yubikey. Both tested models can use the Yubikey Manager, available as a CLI and a GUI. This tool can be used to verify the device firmware version, change the PIN, restore the key or set up other additional features available on the device. Besides, it includes a firmware specific configuration, being able to enable and disable the interfaces for all supported applications and transports (see figure 6.8). For instance, the Yubikey 5 can have FIDO U2F via USB and have it enabled via NFC.

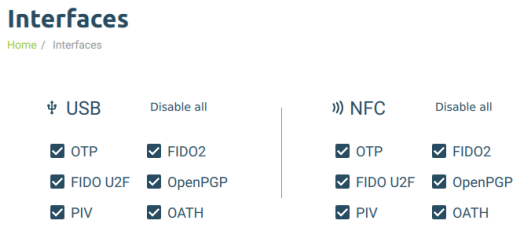


Figure 6.8: Yubikey Manager: enabling and disabling interfaces.

In addition to the Yubikey Manager, the maker also provides a more advanced tool that

only works with specific models like the Yubikey 5: the Yubikey Personalization Tool (see figure 6.9). This allows to check all programmable slots and modes, studied in the next section. For example, the tool allows programming a challenge-response mode in one of the slots. It is worth mentioning that this tool can also be used for batch programming of many keys, automatically loading the configuration when inserted.

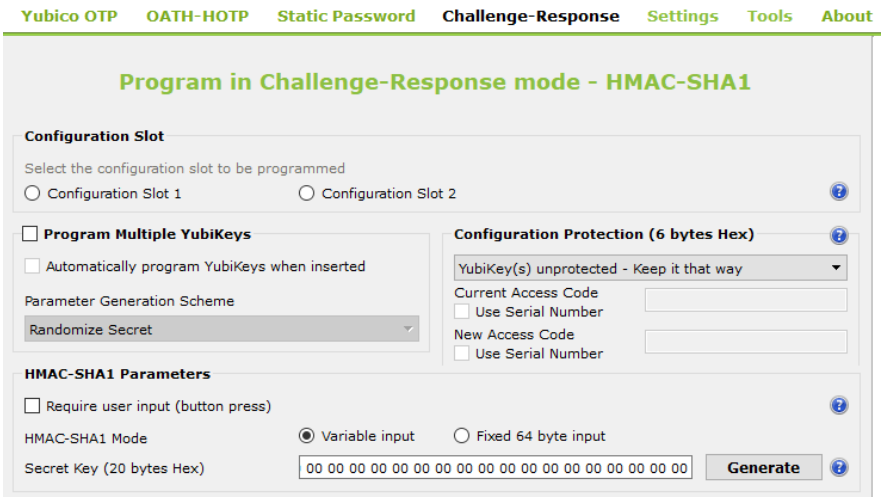


Figure 6.9: Yubikey Personalization Tool GUI: program HMAC-SHA1.

Finally, both Google Security Keys do not provide any specific configuration tool.

6.6 Additional features of the hardware devices

Some of the hardware devices compatible with WebAuthn implement other technologies in their firmwares, all related to keys and security, like OTP codes or PGP key storage. This section explores these features included in the tested hardware keys. Extra features may be of interest when selecting a WebAuthn compatible key for an organisation incorporating this new authentication method, as it can provide extra useful functionality.

After the research, the Yubikey 5 is the only key from the studied hardware devices that includes additional features not related with WebAuthn. There exist, however, other hardware devices that include more features, like the OnlyKey from CryptoTrust [74] [34].

One of the capabilities some keys include, like the Yubikey5, is the challenge-response authentication. As [75] explains in its chapter 10, challenge-response identification is based on demonstrating the knowledge of a shared secret without revealing the secret itself by using a time-variant challenge.

The most common algorithm used for challenge-response authentication is the Hash-based Message Authentication Code, or HMAC [76], which is a specific type of the MAC

algorithms. It is used for verifying both data integrity and the authenticity of the message, having many applications. As an example, there is a plugin for KeePass password manager, called KeeChallenge [77]. It allows using the Yubikey to add extra protection for the password database by using the HMAC-SHA1 algorithm with a shared secret, and can be programmed by using the Yubikey Personalization Tool, mentioned in the previous section (see figure 6.9).

Figure 6.10 shows the unlock screen of KeePassXC, a fork of KeePass that includes a similar approach and also lets using the Yubikey in this mode.

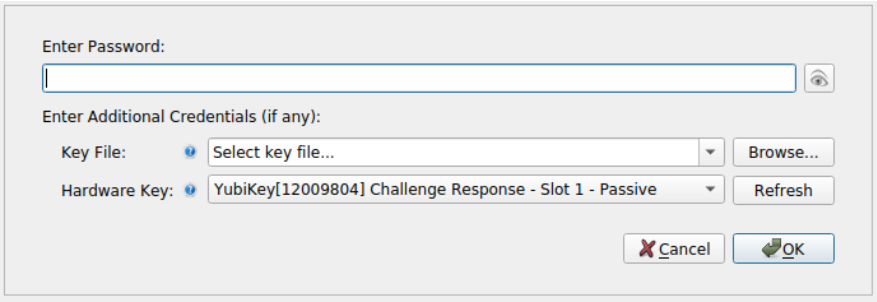


Figure 6.10: Add a hardware key in KeePassXC password manager with HMAC-SHA1.

Another example of the use of HMAC-SHA1 algorithm built into the Yubikey5 is the Yubico Login for Windows [78], which uses this programmable mode for storing and using the secret for Windows authentication.

The Yubikey 5 provides functionality to authenticate as a PIV, or Personal Identity Verification (FIPS 201), compliant smart card [79]. This feature allows storing certificates to work with smart card mini drivers in Windows or use public-key authentication through PKCS11 (see figure 6.11). For instance, the device can be listed by `pcsc_scan` tool on Linux, that lists all smart cards attached to the system (see figure 6.12). The Yubikey Manager software allows importing or generating certificates stored on the device and setting a PIN for protecting them.

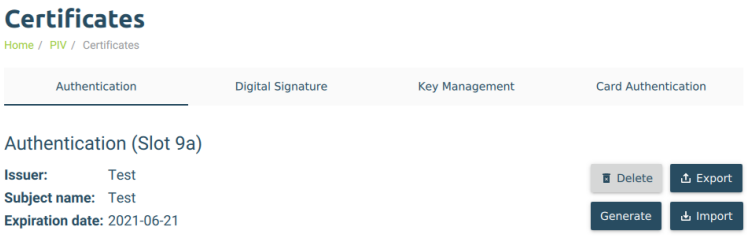


Figure 6.11: PIV setup with Yubikey Manager GUI.

Apart from the challenge-response, the Yubikey 5 can be configured with some popular One Time Password algorithms in both programmable slots. These are used for second-factor

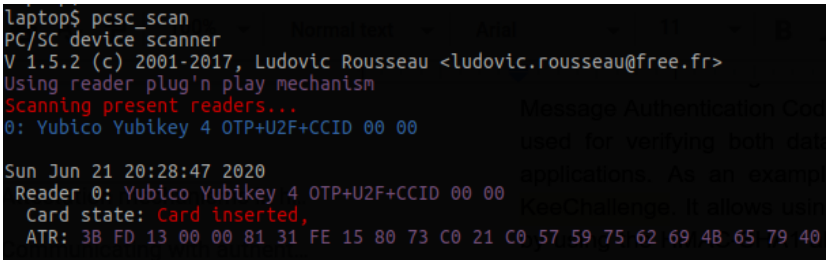


Figure 6.12: pcsc_scan detecting the Yubikey5 as a smart card.

authentication in many applications nowadays. Also, the slots can be programmed with a static password, used for storing a single static password that is typed as keystrokes when needed.

Finally, the Yubikey 5 also supports OpenPGP as a PGP smart card. For example, it allows both importing and generating keys on the device. With the gpg –edit-card command it shows all data related to the card, as well as the PGP keys stored: authentication, encryption and signature keys. Also, the yubikey manager allows configuring touch-protected OpenPGP, so when using this feature it requires the user to physically touch the device.

6.7 Device firmware

In addition to the already mentioned features of the tested devices firmware, it is interesting to highlight the difference between the closed source and open source firmware. From the set of tested devices, only the SoloKey has open source firmware [33]. Actually, this device also is well documented on its hardware.

Open source firmware allows cybersecurity experts to audit, test and deploy the code in order to search for vulnerabilities. This also helps to make sure the firmware inside the hardware device is doing what it is meant to do, and nothing more. Besides, the project will also create a community that can build user support.

On the other hand, closed source makes it more difficult to find vulnerabilities. Attackers would need performing reverse engineering techniques. This is the case of both Yubikeys and both Google Titan Security Keys.

However, Yubico has built an extensive amount of knowledge base by providing documentation for users and developers [80], as well as many open-source projects for configuring and interacting with their devices. The Yubikeys documentation contrasts with the one provided by Google, that is very limited and only provides support for their services. On the other hand, Solokey has some documentation, but it still lacks firmware details and a better explained documentation for its configuration tool.

Finally, most of the devices do not allow for firmware upgrades. The Yubikeys, the Google

Titan keys and the standard Solokeys have no way to upgrade the firmware. However, the SoloKey hacker is built to have an unlocked device for developing purposes. By using its configuration tool and the official code, a developer can build and load it on the device.

This feature, however, constitutes a security risk. An attacker could manipulate the firmware and load it on the victim's device, causing it to run the manipulated malicious code without the victim's awareness. This is the reason why most makers lock their devices' firmware, even from official firmware releases.

Chapter 7

Results

Once all iterations have been finished, the work resulted in a complete functional web application tool for debugging and testing purposes on the WebAuthn protocol that fits the objectives stated at the start of the project.

This web application result of this degree thesis project has been deployed and published for its open access and usege at `https://debauthn.tic.udc.es`. This working instance of the tool corresponds to its last version and uses the correspondent public Docker image (see Appendix B).

It is particularly noteworthy that the tool development, together with the authenticators testing, made possible a deep study of the protocol and its current implementation by different companies. This involves browsers like Firefox or Google Chrome, authenticators as the Yubikey and server implementations like the fido2-lib.

Moreover, for the development of some of the tool features like the operations validation, it required to use the code of an existing library (fido2-lib). The fido2-lib library main developer is one of the WebAuthn protocol engineers at the FIDO Alliance[28]. Notice that, during this project, the usage of this library was not based in interfacing with documented operations but adapting, patching and extending it with code contributions (see Appendix E).

In fact, due to the recent publishment of WebAuthn, the related articles and resources are still scarce. For this reason, along this project it was key the contact with some expert forums and mailing lists around WebAuthn, like the W3C mailing list [23] or the FIDO Alliance forum [21].

Finally, during the development of the degree thesis itself, the developed tool (DebAuthn) has proven to be useful for quickly obtaining information of the authenticators and their capabilities (see example in figure 7.1). This part involved obtaining knowledge that was not yet documented as well as verifying the already available one.

7.1 Tool implementation

Regarding the implementation of the tool, the three concrete objectives have been met. Also, extra results have been achieved, such as the automatic deployment with Docker.

7.1.1 REST service for the access to Relying Party operations

The first specific objective was to ”design and implement a REST service for the access to Relying Party operations”. The final design of the debugging tool is composed of a back-end server and a front-end user interface, communicating via an HTTP REST interface.

This REST model includes the endpoints shown in the table 7.1, which make possible fetching the default configuration options for registration and authentication, sending the authenticator response in each of the operations and managing the registered credentials.

Endpoint	Method	Request payload	Response payload
/attestation/options	GET	*None*	PublicKeyCredentialCreationOptions
/attestation/result	POST	AuthenticatorAttestationResponse	AttestationResult
/assertion/options	GET	*None*	PublicKeyCredentialRequestOptions
/assertion/result	POST	AuthenticatorAssertionResponse	AssertionResult
/registered	GET	*None*	registeredCredentials
/registered	DELETE	*None*	*None*

Table 7.1: All REST endpoints defined in the backend.

7.1.2 User web interface

In order to allow a researcher to check the raw data and debug the configurations, the tool provides two main features:

1. Configuring registration and authentication operations.

2. Displaying the information involved in an structured manner.

Firstly, the tool requests default registration and authentication options to the server and allows users to modify all of them, including the arrays of credential ids. Once this is done, the tool can then trigger the call to the authentiactor through the WebAuthn API. On the other hand, the tool displays the authenticator response before posting it to the server for validation, as well as all errors that occur at the browser during any of the operations. Finally, the tool registers a credential or authenticates with a registered one, while returning all parsed validation details to the user.

7.1.3 Extensible tool

Another of the objectives was to develop an extensible tool. Consequently, the software was designed taking into account the loose coupling principle.

On the server side, REST endpoints were defined by using one main router that uses callback functions controllers. All these loosely coupled controllers use a combined WebAuthn library, configured in a single module loaded in all controllers. For this design, ExpressJS library was used at the backend in a NodeJS environment. It uses callback functions named ”middleware” functions for these REST controllers, managed by a main router that provides the HTTP basic operations. Finally, data models are defined as external components, shared with the client side.

Moreover, on the client side, the design was also built to use loose coupling component definitions that group together the HTML, CSS and JavaScript logic. For instance, all logic and design related to the registration form is grouped in a single file, making it easy to change. For this purpose, VueJS Single File Components are used.

By using this loose coupling principle on both on the server side and the front-end side, make the tool flexible and adaptive to future changes, while it easies its maintenance as an open source project.

7.1.4 Extra results

Executing this web application provides researchers and developers to test both authenticators and browsers, with no need for an *ad hoc* implementation for a use case. The flexibility is extended by providing user forms for all WebAuthn registration and authentication options, apart from the multi credential support which allows selecting the identifiers included in the request. Also, all registered credentials that reside at the server side can be accessed and deleted at the client, showing the credential counter, the id and the public key.

Regarding the WebAuthn protocol, the tool is now supporting all the main attestation formats, including Android Safetynet. This allows to extend the range of authenticators and allow using compatible Android devices as authenticators.

Moreover, DebAuthn also supports resident credentials, allowing to use and debug the behaviour of authenticators compatible with this new technology defined by the CTAP2 standard. This is interesting as in-memory or resident credentials allow authenticators to be used for many different use cases.

It is worth mentioning that, during the development of the tool, the fido2-lib library was corrected and extended. Some of the corrections were in form of patches, while others drived to the contribution of the library (see Appendix E).

Finally, it is also worth mentioning that the tool was published as open-source (GPL-3.0) in

a public repository in Github, as explained in the Appendix A. Additionally, the deployment of the project has been automated using a bash script and Docker, so the server can be installed and deployed in a Docker container. As explained in Appendix B, the project includes a Docker Compose configuration file to launch the server container together with the required MongoDB database.

7.2 Tests with physical authenticators by using the tool

The last project objective was to use the developed tool to test different models and brands of physical authenticators, namely: two Yubico keys, the two Google Titan Security keys and a Solokey (see figure 6.1). All of them were tested in different use cases under the same conditions, on a Chromium browser running on Linux.

Firstly, some of the experiments, as said, were designed to conclude the most common characteristics of the authenticators. One of the performed tests have shown the attestation format used by each of the keys. Another test directly checked the supported signing algorithms, concluding that the most common is the ES256 algorithm (a.k.a ECDSA with SHA-256). Also, the ES256 algorithm is the only available algorithm in the Android platform, as seen in the tests with different browsers. Moreover, the RS256 algorithm, which is not recommended as seen in [71], is not present in any of the tested authenticators.

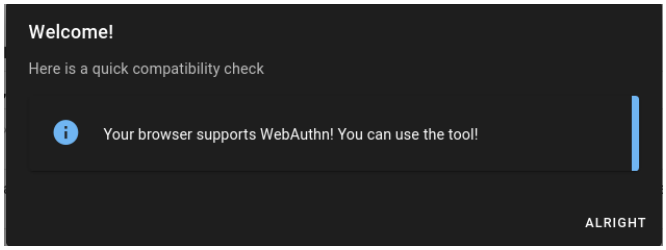
On the other hand, the authenticators were subjected to a resident credential compatibility test, as shown in figure 7.1. Resident credential is the feature that adds support for the first factor authentication method described by the FIDO2 project, as explained in section 2.2.6.

Testing results shown in the table 7.2 can be concluded in the division of the tested authenticators in two groups: authenticators using the old FIDO U2F authentication flow for second-factor authentication and those that are compatible with a passwordless authentication flow.

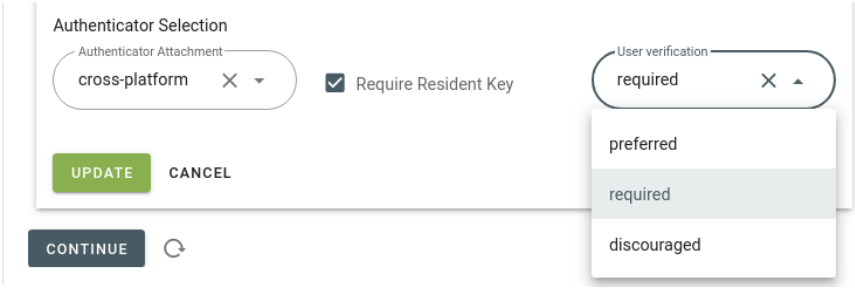
Both Google Titan keys use the old FIDO U2F flow while all the other tested keys were proven to support resident credentials, which are used in the first-factor authentication flow.

	Yubico Security Key	**Yubikey 5**	**Google Titan K9**	**Google Titan K13T**	**Solo Hacker**
Attestation format	"packed"	"packed"	"fido-u2f"	"fido-u2f"	"packed"
Transports	USB	USB, NFC	USB, NFC	USB, NFC, BLE	USB
ES256	Supported	Supported	Supported	Supported	Supported
EdDSA	Supported	Supported	Not supported	Not supported	Not supported
RS256	Not supported	Not supported	Not supported	Not supported	Not supported
Resident Cred. Support	YES	YES	NO	NO	YES

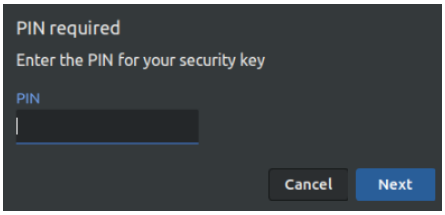
Table 7.2: Hardware authenticator testing results.



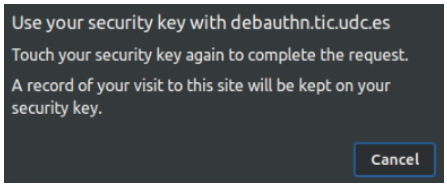
(a) Step 1: Feature detection verifying compatible browser.



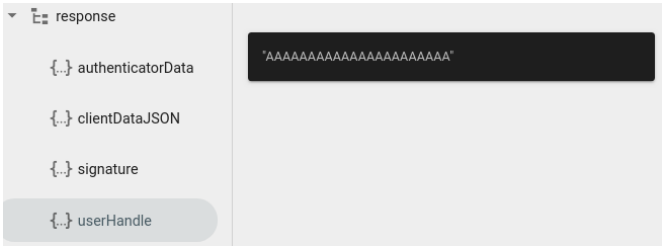
(b) Step 2: Configuration of the registration options.



(c) Step 3: browser asking for user verification (PIN).



(d) Step 4: browser asking for user presence.



(e) Step 5: explore details after authentication with the resident credential

Figure 7.1: DebAuthn usage example: testing resident credentials compatibility of the Solokey.

7.2.1 Extra results

On the other hand, other extrinsic features regarding the physical authenticators were checked, namely: configuration and personalization tools, additional features and the device firmware. All result details are grouped in the table 7.3.

After the experiments with different browsers and platforms like Android, it was found that depending on their implementation they may or may not be compatible with resident

	Yub. Security Key	Yubikey 5	Titan K9	Titan K13T	Solo Hacker
Can reset	YES	YES	NO	NO	YES
Modify PIN	YES	YES	NO	NO	YES
Provides ad hoc tool	YES	YES	NO	NO	YES
Main extra tool features	Enable and disable interfaces	Configure extra features of the key	-	-	'hmac-secret'. Update firmware. Random bytes. Monitor.
Additional features	-	Challenge-response. PIV: FIPS 201. OTP. OpenPGP smart card.	-	-	Random bytes generation
Firmware	Proprietary	Proprietary	Proprietary	Proprietary	Open Source

Table 7.3: Hardware authenticators extrinsic features results.

credentials. The main reason is that the client platform should implement user verification, usually the device PIN. This is an operation included in the FIDO CTAP2 protocol, so browsers or platforms not implementing it will not support FIDO2 resident credentials. Not supporting FIDO CTAP2 while supporting FIDO U2F (CTAP1) means that the only supported algorithm will be ES256 and the authenticator platform will force devices to use the "fido-u2f" attestation signature format.

Taking into account these characteristics, the tested browsers implementation of FIDO protocols and, therefore, its resident credentials support can be summerized in table 7.4.

Browser or platform	FIDO CTAP1	FIDO CTAP2
Firefox v77 on Linux	Supported	Not supported
Chromium v83 on Linux	Supported	Supported
Firefox and Chrome on Android v7.1.1 platform	Supported	Not supported

Table 7.4: Browser support for FIDO CTAP1 and FIDO CTAP2.

Chapter 8

Conclusion and future research

Implementing WebAuthn support is a complex task. This authentication method requires the server to perform several attestation checks to ensure the keys were actually generated in a trusted authenticator. Also, users need to own a trusted authenticator to be able to register it in the compatible web application.

WebAuthn has been designed according to the FIDO2 project, which seeks a first factor authentication mechanism that pretends to replace the old and traditional password method. This was the new concept of "passwordless" authentication flow, introduced by the FIDO CTAP2 protocol. The previous version, the FIDO U2F protocol, was designed to use authenticators as a second factor authentication mechanism which is not compatible with the new features. Although WebAuthn has backward support for FIDO U2F, authenticators and browsers only supporting this U2F standard will not be compatible with the new "passwordless" flows designed for FIDO CTAP2.

In the next few years, the adoption of the standard with hardware authenticators will possibly focus on second factor authentication methods, implemented as a security solution for users to strengthen their account. However, WebAuthn also contemplates the use of software authenticators, which do not involve the acquisition of a new product, as they are included in smartphones like Android. This makes software authenticators a more realistic scenario for users during the following years during an hypothetical transition to a "passwordless" authentication flow.

Finally, it is worth mentioning that the W3C is developing a second version for WebAuthn, with many changes. This makes DebAuthn an useful environment for checking use cases with browsers and authenticators, as proven in this degree thesis. Also, developers can test physical authenticators with browsers in order to guide their implementations, knowing beforehand the compatibility results with the different configurations.

8.1 Future research lines

Some of the future research lines may group different specific parts of WebAuthn and compatible technologies, resulting from this degree thesis work. This section includes some of them.

- **WebAuthn extensibility.** The standard defines optional registration and authentication extensions that add functionalities to suit particular cases. The research objective would be the analysis of the different existing extensions and the possible technologies it can work with, aside web applications. For example, authentication of an Operating System.
- **Updating the developed tool to WebAuthn L2.** The W3C is developing the new version of WebAuthn. It will include some differences, like changes in the values some options can have. For example, L1 version defines requiring resident credential as a true/false value. In L2, this option will take values like "preferred" or "required". Then, the developed tool should adapt to these changes to continue to be used as a debugging tool for this new version. The change would involve updating the frontend registration form and models and a backend implementation if necessary.
- **Roaming credentials for the developed tool.** The current implementation of the testing tool does not allow to use a registered credential from another device. Implementing it would require a new identifier not dependent on session information, making the information persistent. Also, the frontend should include a way to input this identifier and link user sessions.
- **FIDO Metadata Services.** The objective of this research line would be to analyse the distribution of MDS data and how this is used by Relying Parties to validate operations occurring in trusted authenticators. This research would focus on the security of the process and its potential vulnerabilities.
- **WebAuthn platforms and clients.** Windows Hello and the Android platform are client platforms of WebAuthn. They interface between the client, usually a browser, and the authenticator. The research would have as an objective to study the available platforms and analyze their main characteristics.

Appendices

Appendix A

Code base and installation

All the code developed has been published as an open source project called DebAuthn, under the GNU General Public License v3.0 license. All the code is available at the Github platform, hosted at the author's profile. It is composed of two repositories:

- **martinord/debauthn-backend** [81]. This is the main repository that implements the NodeJS server, the deployment explained at the Appendix B and contains a git submodule referencing the next repository, the frontend.
- **martinord/debauthn-frontend** [82]. The repository contains the implementation of the user interface in VueJS.

As explained in the Appendix B, the project can be started by using the `deploy.sh` script at the backend repository or deploy it using it Docker. However, it can be manually installed by running the following NPM scripts:

- `npm install`
- `npm run postinstall`

Once the server is installed, the frontend repository should be built by using the following NPM scripts:

- `npm install`
- `npm run build`

Finally, copy the `dist/` contents to the `src/public` folder at the backend server, so it can serve the static user interface.

Appendix B

Automating deployment with Docker

This appendix describes the automation of the deployment of the project by using Docker containers. It was used when delivering the code to production in the server. Also, a Docker image is generated when the code base (see Appendix A) is updated. This code base is divided in two projects, namely:

- **Frontend:** VueJS scripts and static user interface assets.
- **Backend:** NodeJS server code, that serves both the static frontend files and the REST endpoints.

B.1 Automating deployment

The process for deployment involved building the frontend project, that will produce the minified files to be served. This was done by running 'npm run build'. Then, the `dist/` folder is moved to the `src/public/` folder of the backend. The web server will then consider these files as static files. When navigating to the root of the web page, it will serve the `index.html` file, and all additional JS and CSS files are kept linked and served through the same web server.

However, the repositories are not linked. In order to deploy the application, the developer should work independently on both repositories and move these files around. This can be solved by linking the frontend project to the backend project with a git submodule. When the code is cloned, it will recursively clone the frontend project inside a folder.

Once this is done, the deployment can easily be automated, as the frontend code is now in the same project structure. In this case, a bash script with flags is used, serving as multi purpose. It automates the following operations:

- **Install.** Installs the backend dependencies and applies the patches.
- **Server.** Runs the backend server with NodeJS.
- **Front.** Builds the front and installs it inside the public folder of the backend.
- **All.** Runs install and front operations, ending by running the server with NodeJS.

By doing this, a developer could quickly set up the complete project, after configuring it by modifying the `src/config/` files, that set up the server configuration and the webauthn details, like the R.P. id and name. Nonetheless, there is a configuration option that is commonly used in NodeJS applications: the environment variables.

Then, the configuration files are changed to search for the correspondent environment variables, having a default value:

```
module.exports = {
    port: process.env.PORT || 5000,
    secret: process.env.SECRET || "SecretTestForDevelopment",
    mongoURI: "mongodb://"+process.env.MONGO+"/debauthn" ||
     "mongodb://localhost/debauthn",
    tlsEnabled: process.env.TLS == 'true',
    tls: {
        privateKey: "tls/private.key",
        certificate: "tls/certificate.crt"
    }
}
```

```
module.exports = {
    rpId: process.env.RP_ID || "localhost",
    rpName: process.env.RP_NAME || "DebAuthn"
}
```

Then, the server can be started by the following line:

```
$ RP_ID=debauthn.tic.udc.es npm run start
```

B.2 Deploying in Docker containers

Docker containers are a standardized unit of software [83]. This packs all dependencies into a container so it can be run without caring about installing missing pieces of software. Once this container image is built, it can be run by the Docker daemon in any O.S.

In order to build a container image, Docker needs some instructions, defined in a Dockerfile. In this case, it will use an existing container image loaded with NodeJS environment.

Then, it will copy the source code and run the install and front scripts mentioned in the previous section. By doing that, it will save all dependencies and build and install the frontend files. Lastly, a command is needed to start the server. For this, the aforementioned server script is going to be used.

```
FROM node:12
WORKDIR /usr/src/app
COPY . .
RUN ./deploy.sh --install
RUN ./deploy.sh --front
RUN rm -rf debauthn-frontend/ docs/
EXPOSE 5000
CMD ["./deploy.sh","--server"]
```

Having the project container image becomes interesting when it can be deployed with the MongoDB server, required for it to work. This can be done by using *docker-compose*, which can configure the deployment of several containers with a configuration file in YML format. For doing this, the configuration file includes the image name of MongoDB, which is pulled from DockerHub. Also, it will trigger the build of the project Docker image. Once this is done, both images can be deployed with some configuration. As now the configuration of the tool used environment variables, these can be included in the file:

```
version: '3'

services:
 debauthn:
   build: .
   ports:
     - "5000:5000"
   environment:
     - PORT=5000
     - SECRET="SecretForDeploymentPurposes!"
     - MONGO=mongo
     - TLS=false
     - RP_ID=localhost
     - RP_NAME=DebAuthn
 mongo:
   image: "mongo:4.2"
   ports:
     - "127.0.0.1:27017-27019:27017-27019"
```

Finally, with 'docker-compose up -d' the whole project can be built and deployed in Docker containers, exposing the server port that was configured, routing internally with MongoDB. This is also added to the deployment script under the flag '`--docker`'.

Appendix C

Public key cryptography

Before getting into WebAuthn and its details, this Appendix is aims to briefly review the basic ideas of cryptography. The following sections will provide the reader a very general idea, so further research on the topic is advisable.

One of the main authentication and cybersecurity concepts is cryptography. WebAuthn uses public key cryptography, which is designed to use different keys for encryption and decryption. Public key cryptography also provides the technology for digital signatures, a key concept in this project.

C.1 Symmetric and asymmetric cryptography

According to [75], cryptography "is the study of mathematical techniques related to aspects of in-formation security such as confidentiality, data integrity, entity authentication, and data origin authentication". Within the protocol studied in this research, both symmetric and asymmetric cryptography are used [84].

In 1976 with the publication of New Directions in Cryptography, by Diffie and Helman, the public-key cryptography was introduced. According to [75], it provided a new method for key exchange where only the private key must be kept secret while the public key can be accessible to anyone. This schema allows the owner, first, to sign information so it can be verified with the public key and, second, decipher information that has been encrypted with the public key (see fig C.1).

Public-key cryptography, or asymmetric cryptography, is the basis of digital signatures and digital certificates. Like hand-written signatures, by digitally signing some information, they provide verifiable authenticity of the information. Digital signatures also provide integrity proof, as they depend on the signer secret or private key but also on the message being signed. That is, it guarantees that the data has not been tampered with [85]. As they are based on asymmetric cryptography schema, this signature is verifiable by using only the

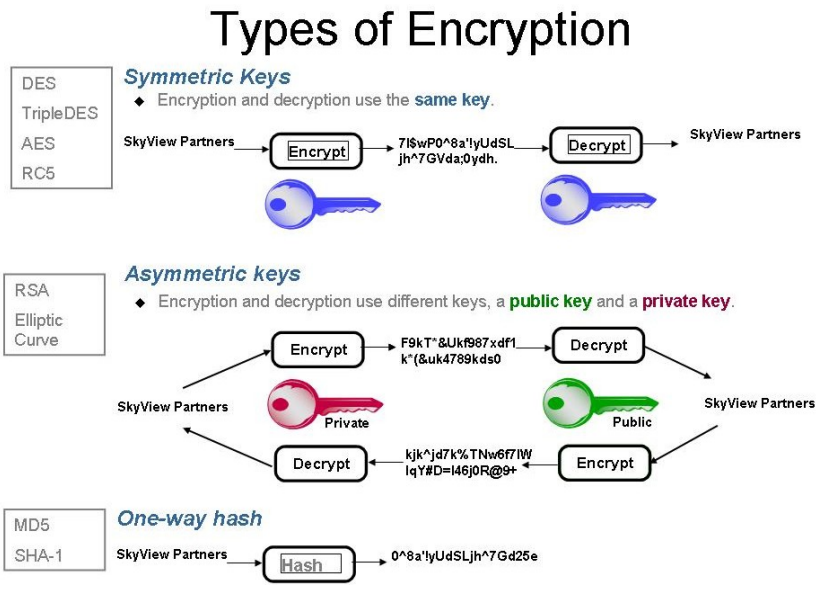


Figure C.1: Types of cryptography

public key, without requiring access to the private signing key.

Symmetric cryptography, however, uses the same key for both cipher and decipher data. As [75] explains, symmetric-key cryptography is currently used for ciphering data to provide confidentiality between communication entities. In modern systems, it is used in conjunction with asymmetric cryptography. The main reason is that asymmetric cryptography avoids the need of both ends sharing a secret before starting the communication. However, it is inneficient for large volumes of data in contrast to symmetric cryptography.

C.2 Public Key Infrastructure

As Smart [85] explains, in a public key system the system for distributing keys is not assumed to be secure. That means that there are no implicit means of ensuring that a public key belongs to someone and has not been spoofed.

This process of linking a public key to an entity or principal is called binding. The main binding tool is the digital certificate. This will include the public key and the identity binded to it, signed by a Certificate Authority, or CA. The CA is meant as a trusted authority by all stakeholders and it is assumed as a signature from the authority is secure.

Digital certificates are present in many systems nowadays, from digital signatures on documents to the TLS protocol that ciphers the HTTPS web connections along the web. All of them use a trust system. The most common is Public Key Infrastructure, or PKI, built from a distribution of trust on several CAs that depend on a common or root CA, constructing a hier-

archical tree of trust inside an organisation. Then, these root CAs from different organisations will be included in operating systems and browsers as trusted authorities.

Therefore, when a stakeholder needs to verify a signature, they will check it with the public key contained on a digital certificate. This certificate binds the key with the identity it belongs to. In order to ensure the public key is correct, the stakeholder may also verify the signature of the CA contained in the certificate with the CA public key. The corresponding CA public key is meant to be trusted via the PKI hierarchy system.

Appendix D

Android SafetyNet Attestation

Android SafetyNet is an Attestation format included in the WebAuthn standard. It was developed by Google to be used by the Android platform when using Android devices as authenticators.

This Appendix summarizes the schema of the Android SafetyNet API. During the degree thesis, support for this format was developed and added to DebAuthn (see section 5.6).

D.1 SafetyNet API protocol

In Android SafetyNet, a Relying Party needs to ensure the device is not compromised, as they explain in their API [18]. This anti-abuse system "assesses the device integrity" by providing "a cryptographically-signed attestation".

The Android Safetynet API protocol, as they explain in the documentation [18], has this structure (see figure D.1):

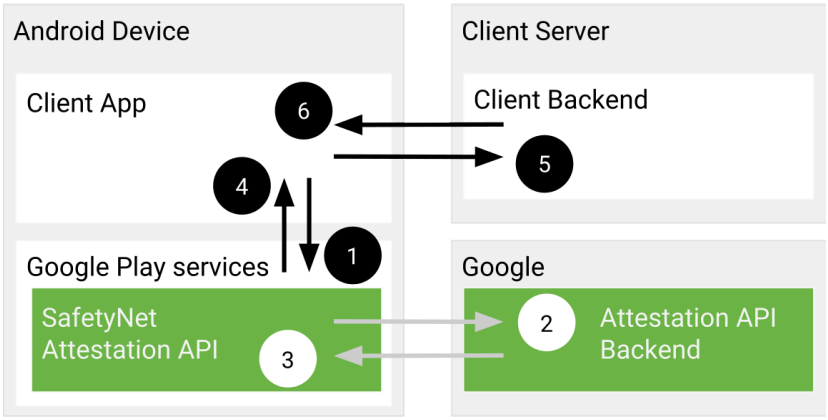


Figure D.1: Android SafetyNet API.

1. The SafetyNet Attestation API receives a call from your app. This call includes a nonce.

2. The SafetyNet Attestation service evaluates the runtime environment and requests a signed attestation of the assessment results from Google's servers.
3. Google's servers send the signed attestation to the SafetyNet Attestation service on the device.
4. The SafetyNet Attestation service returns this signed attestation to your app.
5. Your app forwards the signed attestation to your server.
6. This server validates the response and uses it for anti-abuse decisions. Your server communicates its findings to your app.

As we can see, this squema is very similar to the attestation process described in WebAuthn. Actually, the standard covers the attestation format this API is using in its 8th section [6].

Appendix E

Contributing to the fido2-lib library

The `fido2-lib` is an open source project and has been abandoned for two years. In fact, some of the last changes are not included in the released npm package, used for importing the project as a dependency in third parties code, like it is the case of this tool. One of the last changes was starting to work on Android SafetyNet attestation format, but as it was not validating the attestation, it was not working.

E.1 Pull Requests and forks

One option is to manually plug the new attestation format implementation in running time through `addAttestationFormat()` but this would require removing the existing failing module. For this reason, an issue was opened [86] together with a pull request including the implementation explained in section 5.6 [87].

However, as the maintainer has left the project, other developers from the community have worked on different forks. One of them is James Cullum, developer of the fido2-lib fork, called fido2-library [68]. Once contacted with him, the same pull request was rebased and adapted for their version [88], as seen in te figure. The fork also has a npm package so, once the pull request was merged into the master branch, it was deployed.

E.2 Unit tests and Continuous Integration

In order to better maintain the code, before a merge of the changes, some unit tests were coded with hardcoded data, to ensure future modifications of the aforementioned implementation do not provoke unexpected results. The project is using IstanbulJS suite for unit testing. Within its main test, the attestationResult is tested with some of the supported attestation formats, like 'none' and 'u2f'. Therefore, in order to test the new implementation, the attestation data was hardcoded in the test. This data was sampled from a production environment with

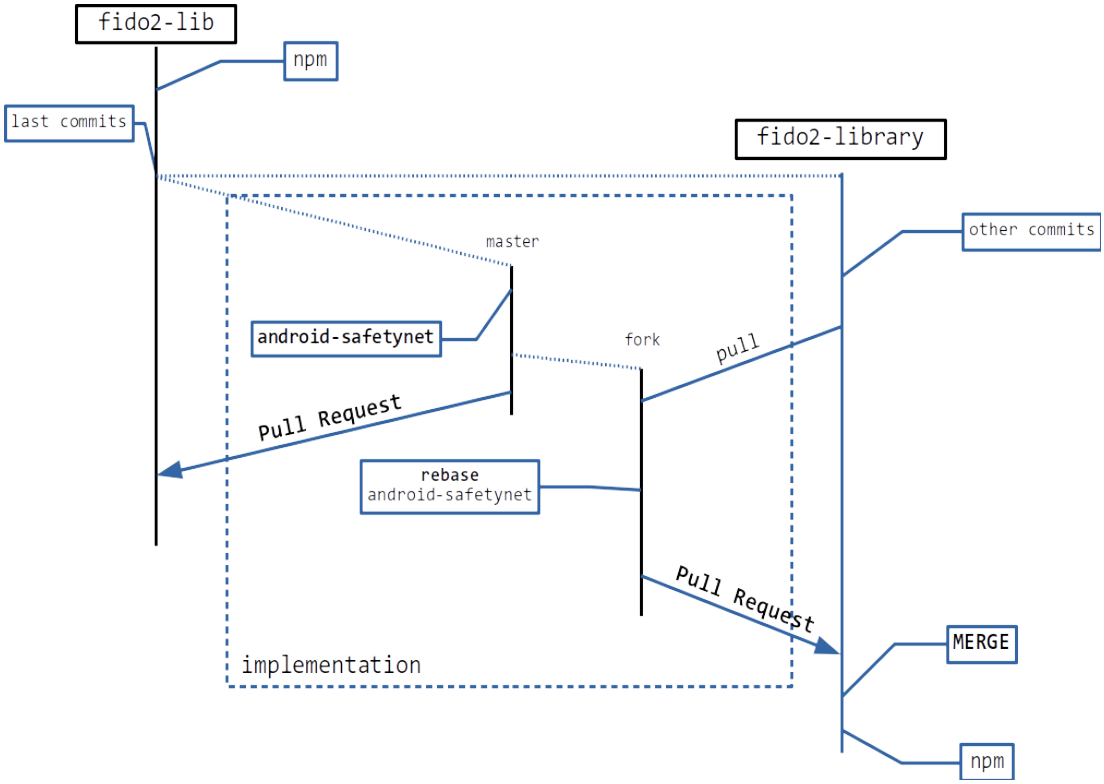


Figure E.1: Forks of the fido2-lib library: *implementation* (author's fork) and *fido2-library* from Cullum.

a CLI debugging tool for NodeJS, after an Android request from Chrome when using the credential platform with the screen-lock. This environment produced the required authenticator response with the "android-safetynet" attestation format.

Finally, when doing the Pull Request, these tests were triggered by the Github Actions, a Continuous Integration (CI) system with many test environments like IstanbulJS [89]. It also produced a test coverage report, ensuring most of the code has their respective tests. This Github actions configuration set by the fork maintainer will help them, along with the implemented tests, to detect any issues with future implementations related to this attestation format.

Appendix F

Code listings

This appendix includes some pieces of code that may be of interest for the reader. However, all developed code is open source and is publicly accessible, as explained in Appendix A.

F.1 Error handlers

This section includes the error handler of the backend with ExpressJS and the error catching and handling of the frontend with VueJS:

```
app.use(function (err, req, res, next) {
   console.error(err.stack)
   res.status(500).send(err.message)
   next()
})
```

```
navigator.credentials.create({ publicKey: this.options })
            .then((response) => {
                this.response = response
            })
            .catch((error) => {
                this.onError(error)
            })
...
onError(error) {
            if(error.response.data) this.error = error.response.data
            else this.error = error
            this.showError = true
        }
```

F.2 Encoding and decoding functions

The buffers are encoded and decoded using `ArrayBuffer`. The target code is base64url. Here are the functions:

```
encode(thing){
        // Array to Uint8Array
        if (Array.isArray(thing)) {
            thing = Uint8Array.from(thing);
        }

        // Uint8Array, etc. to ArrayBuffer
        if (thing.buffer instanceof ArrayBuffer && !(thing
    instanceof Buffer)) {
            thing = thing.buffer;
        }

        // ArrayBuffer to Buffer
        if (thing instanceof ArrayBuffer && !(thing instanceof
    Buffer)) {
            thing = Buffer.from(thing);
        }

        // Buffer to base64 string
        if (thing instanceof Buffer) {
            thing = thing.toString("base64");
        }

        if (typeof thing == "string") {
            // base64 to base64url
            // NOTE: "=" at the end of challenge is optional, strip
    it off here so that it's compatible with client
            thing = thing.replace(/\+/g, "-").replace(/\//g,
    "_").replace(/=*$/g, "");
        }

        return thing
    },
    decode(thing) {
         if (typeof thing === "string") {
             // base64url to base64
             thing = thing.replace(/-/g, "+").replace(/_/g, "/");
             // base64 to Buffer
             thing = Buffer.from(thing, "base64");
         }

```

```
38          // Buffer or Array to Uint8Array
39          if (thing instanceof Buffer || Array.isArray(thing)) {
40              thing = new Uint8Array(thing);
41          }
42
43          // Uint8Array to ArrayBuffer
44          if (thing instanceof Uint8Array) {
45              thing = thing.buffer;
46          }
47
48          return thing
49      }
```

List of Acronyms

R.P. *Relying Party.*

W3C *World Wide fddWeb Consortium.*

FIDO *Fast Identity Online.*

U2F *Universal Second-Factor.*

CA *Certificate Authority.*

PKI *Public Key Infrastructure.*

CTAP *Client to Authenticator Protocol.*

API *Aplication Programming Interface.*

E2E *End to End.*

ASD *Adaptive Software Development.*

SDLC *Software Development Life Cycle.*

IDE *Integrated Development Environment.*

CLI *Command-Line Interface.*

REST *Representational State Transfer.*

SPA *Single Page Application.*

AJAX *Asynchronous JavaScript and XML.*

JS *JavaScript.*

UI *User Interface.*

GUI *Graphical User Interface.*

NFC *Near-Field Communication.*

USB *Universal Serial Bus.*

BLE *Bluetooth Low Energy.*

PIN *Personal Identification Number.*

Glossary

WebAuthn Web Authentication protocol, developed by the W3C. It defines the whole authentication protocol. (a.k.a. WebAuthn API).

Authentication Act of verifying the identity of a computer system user.

Credential Piece of information used for verifying the identity of a user.

Relying Party The entity whose web application utilizes the Web Authentication API to register and authenticate users.

Authenticator A cryptographic entity used in WebAuthn by a browser to generate a public key credential during registration and cryptographically signing a challenge during authentication. When it is a hardware device it is usually called 'key'.

Attestation (WebAuthn) The process employed during registration to attest the provenance of an authenticator and the data it emits.

Assertion (WebAuthn) The cryptographically signed authenticator response returned by an authenticator during the authentication process.

Cryptography Study of mathematical techniques related to aspects of in-formation security such as confidentiality, data integrity, entity authentication, and data origin authentication.

Ceremony A network protocol with human-to-human communication, user interfaces and transfers of physical objects that carry data.

Application Programming Interface A computing interface which defines interaction between multiple software intermediaries by specifying the requests involved.

Resident Credential A WebAuthn credential that is stored in the authenticator memory.

DebAuthn The developed tool during this project. Available at https://debauthn.tic.udc.es.

Kanban board An agile project management tool for visualizing the work-in-progress.

Buffer (object) A JS object that represents a bulk of binary data.

Encode Represent binary data in another form or code. Decoding is the inverse process.

Firmware A specific class of software that provides the low-level control for a specific hardware of a device.

Backend The data access layer of a piece of software. Usually refers to the software running on the server side of a web application. (a.k.a. 'back-end').

Frontend The presentation layer of a piece of software. Usually refers to the software running on the client side of a web application, namely a browser. (a.k.a. 'front-end').

Debugging The process of finding and resolving defects or problems that prevent correct operation within software or systems.

Bibliography

[1] Y. A. Ahmed, M. A. Maarof, F. M. Hassan, and M. M. Abshir, "Survey of keylogger technologies," vol. 5, no. 2, pp. 25–31.

[2] ScatteredSecrets.com. How to crack billions of passwords? Library Catalog: medium.com. [Online]. Available: https://medium.com/@ScatteredSecrets/how-to-crack-billions-of-passwords-6773af298172

[3] B. Krebs. Google: Security keys neutralized employee phishing. [Online]. Available: https://krebsonsecurity.com/2018/07/google-security-keys-neutralized-employee-phishing/

[4] Discover YubiKeys | strong two-factor authentication for secure login. [Online]. Available: https://www.yubico.com/products/

[5] Titan security key fido u2 f usb c nfc ble. Library Catalog: cloud.google.com. [Online]. Available: https://cloud.google.com/titan-security-key

[6] W. , *Web Authentication: An API for accessing Public Key Credentials Level 1*. [Online]. Available: https://www.w3.org/TR/webauthn/

[7] F. Alliance, "Client to authenticator protocol (CTAP)." [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html

[8] N. Steele. A demonstration of the WebAuthn specification. Library Catalog: webauthn.io. [Online]. Available: https://webauthn.io

[9] auth0.com. Web authentication (WebAuthn) credential and login demo. Library Catalog: webauthn.me. [Online]. Available: https://webauthn.me

[10] Web authentication: An API for accessing public key credentials - level 2. [Online]. Available: https://w3c.github.io/webauthn/

[11] FIDO2: Moving the world beyond passwords using WebAuthn & CTAP. Library Catalog: fidoalliance.org. [Online]. Available: https://fidoalliance.org/fido2/

[12] Universal 2nd factor (u2f) overview. [Online]. Available: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html

[13] Credential management level 1. [Online]. Available: https://www.w3.org/TR/credential-management-1/

[14] FIDO metadata service. [Online]. Available: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-metadata-service-v2.0-id-20180227.html

[15] N. Amiet. FIDO2 deep dive: Attestations, trust model and security. Library Catalog: research.kudelskisecurity.com. [Online]. Available: https://research.kudelskisecurity.com/2020/02/12/fido2-deep-dive-attestations-trust-model-and-security/

[16] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*. ACM Press, p. 132. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1030083.1030103

[17] A. Enterprise. Android key attestation. Library Catalog: source.android.com. [Online]. Available: https://source.android.com/security/keystore/attestation

[18] SafetyNet attestation API. Library Catalog: developer.android.com. [Online]. Available: https://developer.android.com/training/safetynet/attestation

[19] Yubico | YubiKey strong two factor authentication. [Online]. Available: https://www.yubico.com/

[20] FIDO alliance - open authentication standards more secure than passwords. Library Catalog: fidoalliance.org. [Online]. Available: https://fidoalliance.org/

[21] FIDO dev (fido-dev) - google groups. [Online]. Available: https://groups.google.com/a/fidoalliance.org/forum/#!forum/fido-dev

[22] World wide web consortium (w3c). [Online]. Available: https://www.w3.org/

[23] public-webauthn@w3.org mail archives. [Online]. Available: https://lists.w3.org/Archives/Public/public-webauthn/

[24] Duo labs | duo security. [Online]. Available: https://duo.com/labs

[25] Auth0: Secure access for everyone. but not just anyone. [Online]. Available: https://auth0.com/

[26] Solo – SoloKeys. [Online]. Available: https://solokeys.com/

[27] "duo-labs/py_webauthn," original-date: 2017-11-10T18:02:28Z. [Online]. Available: https://github.com/duo-labs/py_webauthn

[28] "webauthn-open-source/fido2-lib," original-date: 2017-02-08T01:56:55Z. [Online]. Available: https://github.com/webauthn-open-source/fido2-lib

[29] A. Åberg, "abergs/fido2-net-lib," original-date: 2018-05-31T07:51:51Z. [Online]. Available: https://github.com/abergs/fido2-net-lib

[30] "cedarcode/webauthn-ruby," original-date: 2018-05-09T21:49:17Z. [Online]. Available: https://github.com/cedarcode/webauthn-ruby

[31] "duo-labs/webauthn," original-date: 2017-10-26T16:15:55Z. [Online]. Available: https://github.com/duo-labs/webauthn

[32] Titan security key | google cloud. [Online]. Available: https://cloud.google.com/titan-security-key/

[33] "solokeys/solo," original-date: 2018-09-13T22:42:08Z. [Online]. Available: https://github.com/solokeys/solo

[34] OnlyKey hardware password manager | one PIN to remember. [Online]. Available: https://onlykey.io/

[35] Thetis. Library Catalog: thetis.io. [Online]. Available: https://thetis.io/

[36] F. Henneke, "FabianHenneke/WearAuthn," original-date: 2019-10-27T12:42:48Z. [Online]. Available: https://github.com/FabianHenneke/WearAuthn

[37] "duo-labs/android-webauthn-authenticator," original-date: 2019-01-31T21:36:41Z. [Online]. Available: https://github.com/duo-labs/android-webauthn-authenticator

[38] "google/OpenSK," original-date: 2019-12-17T18:55:43Z. [Online]. Available: https://github.com/google/OpenSK

[39] WebAuthn demo. [Online]. Available: https://webauthn.org/

[40] Yubico demo website. [Online]. Available: https://demo.yubico.com/

[41] WebAuthn test app. [Online]. Available: https://webauthntest.azurewebsites.net/

[42] WebAuthn checker - w3c level 1 mode. [Online]. Available: https://webauthn.bin.coffee/

[43] J. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing Co Inc.,U.S.

[44] A. Cockburn, "Using both incremental and iterative development. 21. 27-30." vol. 21, pp. 27–30.

[45] M. Rehkopf. What is a kanban board? Library Catalog: www.atlassian.com. [Online]. Available: https://www.atlassian.com/agile/kanban/boards

[46] Chrome 70 beta: shape detection, web authentication, and more. Library Catalog: blog.chromium.org. [Online]. Available: https://blog.chromium.org/2018/09/chrome-70-beta-shape-detection-web.html

[47] N. Nguyen. Firefox gets down to business, and it's personal. Library Catalog: blog.mozilla.org. [Online]. Available: https://blog.mozilla.org/blog/2018/05/09/firefox-gets-down-to-business-and-its-personal

[48] BOE.es - documento BOE-a-2019-14977. [Online]. Available: https://www.boe.es/diario_boe/txt.php?id=BOE-A-2019-14977

[49] Express - node.js web application framework. Library Catalog: expressjs.com. [Online]. Available: https://expressjs.com/

[50] Node.js. Node.js. Library Catalog: nodejs.org. [Online]. Available: https://nodejs.org/en/

[51] Vue.js. Library Catalog: vuejs.org. [Online]. Available: https://vuejs.org/

[52] The most popular database for modern apps. Library Catalog: www.mongodb.com. [Online]. Available: https://www.mongodb.com

[53] "axios/axios," original-date: 2014-08-18T22:30:27Z. [Online]. Available: https://github.com/axios/axios

[54] authnrData rawAuthnrData should be ArrayBuffer · issue #23 · webauthn-open-source/fido2-lib. Library Catalog: github.com. [Online]. Available: https://github.com/webauthn-open-source/fido2-lib/issues/23

[55] fix: authnrData rawAuthnrData should be ArrayBuffer by goldenbearkin · pull request #25 · webauthn-open-source/fido2-lib. Library Catalog: github.com. [Online]. Available: https://github.com/webauthn-open-source/fido2-lib/pull/25

[56] Vue material design component framework — vuetify.js. Library Catalog: vuetifyjs.com. [Online]. Available: https://https://vuetifyjs.com/

[57] Getting started | vue router. [Online]. Available: https://router.vuejs.org/guide/

[58] "github/webauthn-json," original-date: 2019-07-01T18:52:54Z. [Online]. Available: https://github.com/github/webauthn-json

[59] S. Josefsson {\textless}simon@josefsson.org{\textgreater}. The base16, base32, and base64 data encodings. Library Catalog: tools.ietf.org. [Online]. Available: https://tools.ietf.org/html/rfc4648

[60] A. Takakuwa, "Moving from passwords to authenticators," p. 164.

[61] F. Skokan, "panva/jose," original-date: 2018-11-06T08:46:01Z. [Online]. Available: https://github.com/panva/jose

[62] CBOR object signing and encryption (COSE). [Online]. Available: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

[63] D. Sheldrick, "ds300/patch-package," original-date: 2017-05-08T20:31:20Z. [Online]. Available: https://github.com/ds300/patch-package

[64] Security key by yubico | two factor security key | USB-a. Library Catalog: www.yubico.com. [Online]. Available: https://www.yubico.com/product/security-key-by-yubico

[65] YubiKey 5.2.3 enhancements to FIDO 2 support. Library Catalog: support.yubico.com. [Online]. Available: https://support.yubico.com/support/solutions/articles/15000027138-yubikey-5-2-3-enhancements-to-fido-2-support

[66] YubiKey 5 NFC | two factor security key | USB-a & NFC. Library Catalog: www.yubico.com. [Online]. Available: https://www.yubico.com/product/yubikey-5-nfc

[67] Solo hacker - open source, FIDO2(for developers & makers). Library Catalog: solokeys.com. [Online]. Available: https://solokeys.com/products/solo-hacker

[68] YubiKey manager. Library Catalog: www.yubico.com. [Online]. Available: https://www.yubico.com/products/services-software/download/yubikey-manager/

[69] E. Lundberg, "FIDO2 - yubikey giving different attestation format in firefox and chrome." [Online]. Available: https://groups.google.com/a/fidoalliance.org/g/fido-dev/c/ILVVXClQN_8/m/j-U0PBlHCQAJ?pli=1

[70] FIDO u2f authenticator transports extension. [Online]. Available: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-authenticator-transports-extension-v1.2-ps-20170411.html

[71] Security concerns surrounding WebAuthn: Don't implement ECDAA (yet) - paragon initiative enterprises blog. [Online]. Available: https://paragonie.com/blog/2018/08/security-concerns-surrounding-webauthn-don-t-implement-ecdaa-yet

[72] OpenSSH - release notes 8.2. [Online]. Available: https://www.openssh.com/txt/release-8.2

[73] "solokeys/solo-python," original-date: 2019-02-15T22:09:47Z. [Online]. Available: https://github.com/solokeys/solo-python

[74] OnlyKey features | docs. [Online]. Available: https://docs.crp.to/features.html

[75] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 5th ed. [Online]. Available: http://cacr.uwaterloo.ca/hac/

[76] H. Krawczyk, R. Canetti, and M. Bellare. HMAC: Keyed-hashing for message authentication. Library Catalog: tools.ietf.org. [Online]. Available: https://tools.ietf.org/html/rfc2104

[77] B. Rush. KeeChallenge. [Online]. Available: http://richardbenjaminrush.com/keechallenge/

[78] Yubico login for windows configuration guide : Yubico support. [Online]. Available: https://support.yubico.com/support/solutions/articles/15000028729-yubico-login-for-windows-configuration-guide

[79] PIV compatible smart cards | yubico. [Online]. Available: https://www.yubico.com/authentication-standards/smart-card/

[80] Yubico developers. [Online]. Available: https://developers.yubico.com/

[81] martinord/debauthn-backend. Library Catalog: github.com. [Online]. Available: https://github.com/martinord/debauthn-backend

[82] martinord/debauthn-frontend. Library Catalog: github.com. [Online]. Available: https://github.com/martinord/debauthn-frontend

[83] What is a container? | app containerization | docker. Library Catalog: www.docker.com. [Online]. Available: https://www.docker.com/resources/what-container

[84] FIDO authenticator allowed cryptography list. [Online]. Available: https://fidoalliance.org/specs/fido-security-requirements-v1.0-fd-20170524/fido-authenticator-allowed-cryptography-list_20170524.html#allowed-cryptographic-functions

[85] N. Smart, *Cryptography: An Introduction*, 3rd ed. [Online]. Available: https://www.cs.umd.edu/~waa/414-F11/IntroToCrypto.pdf

[86] Android SafetyNet not working · issue #48 · webauthn-open-source/fido2-lib. Library Catalog: github.com. [Online]. Available: https://github.com/webauthn-open-source/fido2-lib/issues/48

[87] Complete android SafetyNet attestation (fix #48) by martinord · pull request #49 · webauthn-open-source/fido2-lib. Library Catalog: github.com. [Online]. Available: https://github.com/webauthn-open-source/fido2-lib/pull/49

[88] Complete android SafetyNet attestation by martinord · pull request #3 · FIDO-tools/fido2-library. Library Catalog: github.com. [Online]. Available: https://github.com/FIDO-Tools/fido2-library/pull/3

[89] Istanbul, a JavaScript test coverage tool. Library Catalog: istanbul.js.org. [Online]. Available: http://istanbul.js.org/