Author
Jakob Arneth
11916206

Submission
Institute of Networks and Security

Thesis Supervisor
Dr. **Michael Roland**

Assistant Thesis Supervisor
Gerald Schoiber, MSc

May 2023

FIDO2 Token Authentication for Personal Identity Agent

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

This bachelor thesis aims to extend the Personal Identity Agent of the Digidow project by adding two new authentication methods with FIDO2 tokens. So far, users had to use a password for the authentication process. A method for authenticating with FIDO2 tokens has not been implemented yet. Therefore, the authentication process was enhanced by implementing the authentication with security keys. Initially, two-factor authentication with security keys as second factor was implemented. In addition to that, the application now fulfills the requirement of passwordless authentication. First, this bachelor thesis describes the theoretical background of FIDO2 token authentication. Second, it gives a detailed overview of the functionality of FIDO2 token authentication. Additionally, the design choices for the implementation and the individual implementation steps are outlined. Furthermore, an evaluation concerning the Tor Browser, the WebAuthn standard, the security key setup, and implementation options is done.

Kurzfassung

Das Ziel dieser Bachelorarbeit ist die Implementierung einer neuen Authentifizierungsmethode mit FIDO2-Tokens für den Personal Identity Agent des Digidow-Projektes. Bis jetzt mussten sich Benutzer mit einem Passwort anmelden und es stand keine Authentifizierungsmethode mit FIDO2-Tokens als Sicherheitsschlüssel zur Verfügung. Daher musste der Authentifizierungsprozess erweitert werden. Zu Beginn wurde die Zwei-Faktor-Authentifizierung mit Sicherheitsschlüsseln als zweiten Faktor implementiert. Zudem wurde Funktionalität hinzugefügt, damit sich Benutzer ohne Passwort und mit einem Sicherheitsschlüssel authentifizieren können. Zuerst geht diese Bachelorarbeit genauer auf den theoretischen Hintergrund von FIDO2 Token Authentifizierung ein. Danach wird die Funktionalität der FIDO2 Token Authentifizierung genauer erläutert. Zusätzlich werden die Entwurfsentscheidungen für die Implementierung und die einzelnen Implementierungsschritte beschrieben. Außerdem wird eine Evaluierung mit Bezug auf den Tor Browser, den WebAuthn Standard, die Einrichtung von Sicherheitsschlüsseln und Implementierungsoptionen vorgenommen.

Acknowledgements

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by ONCE (FFG grant FO999887054 in the program "IKT der Zukunft") and the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry for Digital and Economic Affairs (BMDW), the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

Contents

Chapter 1

Introduction

1.1 Digidow Project & Personal Identity Agent

The Digidow project from the Institute of Networks and Security is about private digital authentication in the physical world. Individuals should be able to use, for example, public transport or payment applications without carrying around a physical identity document or a trusted mobile device. In such a scenario, it is possible to save all users' data in a centralized database and track all actions of the users. However, the centralized approach cannot be realized without compromising the user's privacy. Another solution would be a decentralized approach, where the user is associated with a Personal Identity Agent (PIA). Each individual is represented by a PIA in the digital world.

The Institute of Networks and Security researches and develops this decentralized approach. In the current experimental setup, the Personal Identity Agents are hosted on a server at the institute. Each PIA runs as a separate process in a virtual machine. Whenever a new user wants to enroll the PIA instance, the user gets access to a unique onion service. Additionally, the user receives an enrollment PIN. Now, the user can pair with the PIA. Information about the user and identities are stored on the server side within the PIA. After the enrollment, users can log into their PIA.

1.2 Motivation

Thus far, it has only been possible to authenticate with a password at the Personal Identity Agent (PIA). Mere password-based authentication can lead to many security issues [22]. Some examples are weak passwords, insecure password storage, or password reuse. According to Srinivas et al. [22], it should be possible for online services to allow strong user authentication by leveraging native security features of end-user computing devices. This reduces the problems associated with creating and remembering many online credentials.

In the context of this bachelor thesis, this authentication procedure will be adapted, so that authentication is possible with FIDO2 tokens as security keys. These newly implemented features will increase the security of the whole authentication process for the PIA. According to Lyastani et al. [15], the FIDO2 standard has enormous potential to become the successor of mere password-based authentication in web applications. Alqubaisi et al. [1] also argue that the implementation of FIDO2 authentication gives each user the possibility to enhance privacy and security, as the FIDO2 standard is more secure than pure password-based authentication.

Additionally, all big players in the digital world (Facebook, Google, Microsoft, etc.) give their users the possibility to enhance the authentication process with two-factor authentication [15]. Therefore, implementing FIDO2 authentication for the PIA can be characterized as a state-of-the-art project.

1.3 Objectives and Goals

The goal of this bachelor thesis is to extend the Personal Identity Agent (PIA) of the Digidow project with new authentication methods that use FIDO2 tokens as security keys. The main reason for the enhancement of the authentication procedure of the PIA is, as mentioned above, the increase in security, as pure password-based authentication causes many security threats [1].

Therefore, this bachelor thesis aims to implement an authentication method, where FIDO2 tokens can be used as security keys for two-factor authentication. The security keys are the second factor to support password-based authentication with additional security. To use the FIDO2 token as second factor, the user must register the security key and link it to the PIA. After the successful registration, the user has to authenticate with the password and the FIDO2 token as second factor. This means that the user must authenticate with something he or she knows and possesses.

In addition to two-factor authentication, passwordless authentication will be implemented too. In this scenario, the user does not have to enter a password and is only asked to present a security key during authentication. As the password is omitted and the user verification is shifted to a secure FIDO2 token on the user end, not all security keys can be used to authenticate. Only security keys that enforce user verification, for example with a fingerprint or an additional PIN, should be allowed. This way security is not compromised [22, 26]. The registration process works the same way as with two-factor authentication. According to Lyastani et al. [15], users describe FIDO2 passwordless authentication as more usable and acceptable than traditional password-based authentication. Therefore, FIDO2 passwordless authentication is a very good option regarding user convenience.

Chapter 2

Background

2.1 FIDO Alliance

The FIDO (Fast Identity Online) Alliance was founded to change the nature of strong online authentication [22]. Their goal is to introduce an authentication standard that makes authentication easier and increases privacy and security. Moreover, the FIDO Alliance wants to reduce the importance of passwords on the web, as creating and remembering online credentials leads to many security issues [19].

Therefore, the FIDO Alliance has developed technical specifications for how users can be authenticated securely at online services. According to Pereira et al. [19], the goal is to avoid the use of server-side shared secrets in public key cryptography systems. In order to implement the new concept, many companies all over the world supported and integrated the FIDO Alliance services. The FIDO Alliance is an organization with 250 member companies worldwide. Among them are the Bank of America, Google, Facebook, Microsoft, and Amazon [15, 19].

According to Lyastani et al. [15], the FIDO2 authentication standard seems like a promising candidate for succeeding text-based password authentication in end-user web applications. The FIDO Alliance provides a concept that uses credentials that cannot be phished or replayed. In addition, the new standard is supported by almost all browsers and operating systems like Windows or Android. Furthermore, the concept offers a consistent user experience.

The FIDO Alliance also supports different technologies and devices to authenticate at an online service, including biometrics, Trusted Platform Modules (TPM), USB security tokens, smart cards, Bluetooth and near-field communication (NFC) [15].

2.2 Universal Second Factor (U2F)

The Universal Second Factor (U2F) protocol allows users to improve authentication security by adding a decisive second factor to the login procedure in addition to a password. U2F is the predecessor of the FIDO2 project, which is based on the Universal Second Factor protocol [15].

At first, the user has to register a second factor for the authentication process, also known as a security key. In general, the authentication procedure stays the same, as the user has to enter a password. However, the online service can now send a login challenge, which forces the user to present the registered second factor. The user's presence is confirmed by some interaction with the security key. The intent of user presence is not to identify a user, but to confirm that a user is physically present. This can be done, for example, by touching a sensor. Version 1.2 and later of the Universal Second Factor protocol allow to get a signature of a device without the user's presence [22]. This is not recommended and should be handled by the web application accordingly. After all, user presence is expected for the authentication process of the Personal Identity Agent.

Furthermore, user verification ensures that the user is authorized to use the authenticator. This, for example, can be done by entering a PIN for the security key or by presenting a saved fingerprint to the authenticator. If the user chooses to authenticate with a password and uses the security key as a second factor, it is recommended to omit user verification [26]. The reason for this is that the user enters a password and therefore a shared secret anyway. If user verification is enabled, the user would have to present a third factor during authentication, which is not recommended for standard web applications regarding convenience. In this case, only user presence is needed to authenticate the user.

2.3 Universal Authentication Framework (UAF)

According to Machani et al. [16], the core idea of the Universal Authentication Framework (UAF) protocol is to allow passwordless authentication at online services. During registration, the user registers a device or technology for authentication. For example, such a device or technology could be swiping a finger or looking at the camera. Then, the authentication action must be repeated and the user is successfully authenticated at the online service. However, implementing a single-factor, passwordless authentication creates new security issues. For this reason, the goal of UAF is to support the authentication process in web applications with an additional factor. Also, strong multi-factor authentication can be applied to web applications that rely greatly on a secure authentication process [16]. A typical example would be a web application that deals with online banking or finances.

2.4 WebAuthn

The Universal Authentication Framework protocol and the Universal Second Factor protocol were not deployed on a large scale. Web browsers like Safari and Microsoft Edge did not implement the U2F protocol and the UAF protocol was only available if the user installed a compatible FIDO client [1]. As a consequence, the FIDO Alliance developed, together with the World Wide Web Consortium (W3C), a new generation of the FIDO standards, also known as the FIDO2 protocol. The main goal was to offer the web community a standard for secure authentication services. The idea of establishing a new standard resulted in the creation of a new FIDO2 protocol composed of two sub-protocols.

First, there is the W3C Web Authentication (WebAuthn) protocol, which manages the communication between the client and the server. WebAuthn intends to standardize the authentication process for users at online services by public key cryptography. It is backward compatible with the U2F standard and WebAuthn can also perform passwordless authentication. For passwordless authentication, user verification is recommended. Otherwise, the security of the authentication process would not be enhanced. Using passwordless authentication allows the user to replace complicated passwords with, for instance, shorter 4-digit PINs or some biometrics. The main advantage is that security is not compromised, although the user's password is omitted [22]. Besides, if biometric matching is used, this is done locally. Hence, nothing about the biometric feature itself is being exchanged with the server. An example for this would be Windows Hello.

An authenticator (e.g. a USB security token) is required to do cryptographic operations during the authentication procedure. No matter which technology is used, the authenticator verifies the user locally. That way, sensitive authentication data is not shared with the web application. Additionally, several security keys can be registered for one user [2]. Another important aspect is that users can use the same device across multiple web applications since there are several virtual security keys generated on one physical device. Due to that, the user does not need to have a federated identity for different web applications but can simply use one authenticator [2].

The second sub-protocol of the FIDO2 protocol is the FIDO Client to Authenticator Protocol (CTAP). CTAP is an application layer protocol and aims to facilitate the communication between a cryptographic authenticator (e.g. a USB security token) and a client application (e.g. a web browser). This protocol is based on the U2F authentication standard. If the authenticator implements the second version of the Client to Authenticator Protocol (CTAP2), it is called a FIDO2 (WebAuthn) authenticator. The focus of the FIDO Alliance is now narrowed down to the communication between the client and the authenticator. W3C regulates the communication between the client and the server. With this adaptation of responsibilities, the popularity of FIDO2 has increased enormously [1].

2.5 Public Key Cryptography

The authentication process with FIDO2 tokens is based on asymmetric encryption, also called public key encryption. In this cryptographic system, there are always pairs of keys. Each key pair consists of a private key and a public key. There are different approaches how those key pairs can be created. There are, for example, RSA or Elliptic Curve Cryptography (ECC). The security of RSA encryption relies on the difficulty of factoring two large prime numbers [20] and the Elliptic Curve Cryptography is based on the algebraic structure of elliptic curves over finite fields [24]. The public key, as the name implies, can be known by the public, therefore, by everyone. On the contrary, the private key must only be known by the owner of the key. There are several practical use cases for how the key pairs can be used.

Encrypt and Decrypt Messages

Asymmetric encryption can be used to encrypt and decrypt messages (see Figure 2.1). If two stations want to communicate with each other, each of them must create a key pair, consisting of a private and a public key. Afterwards, the public keys are exchanged. With the exchanged public keys, the stations can encrypt messages and send them to the other station. Anybody who reads the encrypted message cannot decrypt it, as it can only be decrypted with the corresponding private key [3]. However, with the correct private key, the stations can decrypt messages from the other station and read the plaintext.

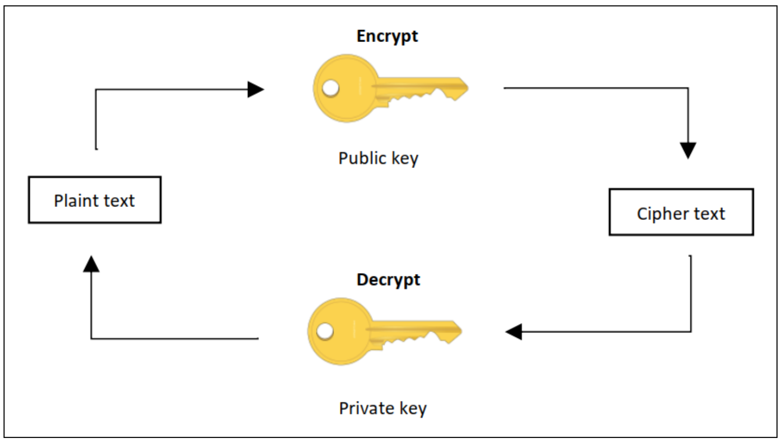


Figure 2.1: Encrypt and decrypt messages (Source: [3])

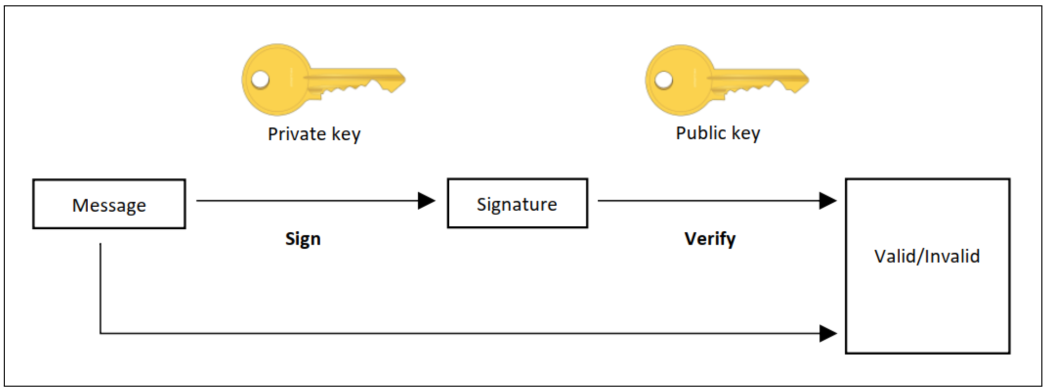


Figure 2.2: Sign and verify messages (Source: [3])

Sign and Verify Messages

Furthermore, asymmetric encryption can be used to sign and verify messages (see Figure 2.2). For this procedure, one public/private key pair must be created. Then, the sender can use the private key to create and sign messages. Afterwards, the recipient of a message can use the corresponding public key to verify the message against the signature. This procedure provides a secure way to ensure the authenticity and integrity of messages transmitted between sender and receiver [3].

Chapter 3

FIDO2 Token Authentication

3.1 Overview

For FIDO2 token authentication, asymmetric encryption is used. During the security key registration process, a key pair is created. If a user wants to authenticate with a registered FIDO2 token at a web application, the server can create a login challenge and send it to the client. The client uses the private key to sign the login challenge. During the signing process, the FIDO2 token never reveals the private key. Next, the signature is sent back to the server, where it is verified with the corresponding public key [1]. If the verification was successful, the user is authenticated at the web application.

The key pair created during the security key registration process is origin-specific. This means, that the created key pair during the registration is saved together with the origin of the web application [22]. For FIDO2 token authentication it is also possible to allow cross-origin authentication. However, by default this feature is not enabled as it brings potential security and privacy risks with it [23].

By using asymmetric encryption, security can be increased compared to mere password-based authentication. The reason for this is that the public key is stored on the server and the private key remains at the authenticator. Therefore, no shared secrets exist between users and websites that can be leaked by server-side attacks [15]. Depending on the cryptosystem, the private key is recreated by the FIDO2 token and does not have to be stored on the authenticator. Instead, a key handle, which is created during registration by the authenticator and sent by the server during authentication, is decrypted by the authenticator. The private key is then derived from the key handle during the signing process. For YubiKeys, the decryption of the key handle is done by a master key, which is unique for each security key[1].

During the registration, the client sends the origin of the web application to the authenticator. The FIDO2 token returns a fresh public key and a key handle, which are both stored on the server. An important aspect is that the FIDO2 token encodes the requesting origin into the key handle [22]. During the authentication process, the client sends the key handle created during registration and the origin of the web application to the authenticator. The FIDO2 token has to ensure that the origin of the registration and authentication match before performing any signing operation. If they match, the challenge will be signed and returned. Otherwise, no signature will be returned to the client. This origin check ensures that public keys and key handles cannot be reused by other online services or websites. Otherwise, the user would be traceable across different web applications.

According to Srinivas et al. [22], online services can therefore verify identities securely with FIDO2 tokens. Also, man-in-the-middle attacks can be detected

[1]https://developers.yubico.com/U2F/Protocol_details/Key_generation.html

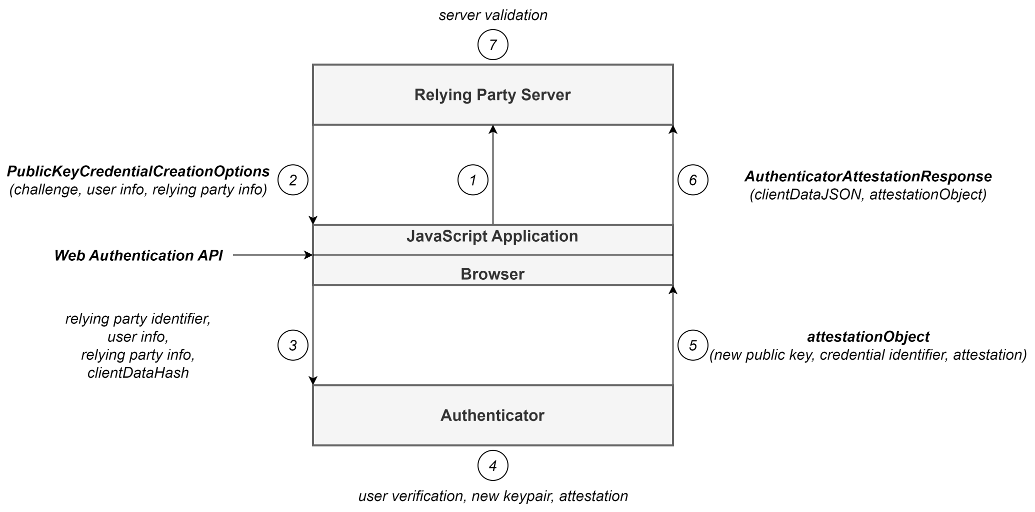

server validation
7
Relying Party Server
PublicKeyCredentialCreationOptions
(challenge, user info, relying party info)
2
1
6
AuthenticatorAttestationResponse
(clientDataJSON, attestationObject)
JavaScript Application
Web Authentication API
Browser
relying party identifier,
user info,
relying party info,
clientDataHash
3
5
attestationObject
(new public key, credential identifier, attestation)
Authenticator
4
user verification, new keypair, attestation

Figure 3.1: Register FIDO2 token (Source: [10])

in most situations, as credentials are saved with the origin of the web application. This means that the FIDO2 token would not even respond, since the origin of the man-in-the-middle would not match the stored origin.

Although, FIDO2 token authentication improves the security of the authentication process, some problems remain unresolved. According to Frymann et al. [9], it is currently under discussion how users can regain access to an account if the registered authenticator is damaged or lost. As a solution for this problem, Yubico proposes to register a backup authenticator in addition to the primary authenticator [14]. If the primary authenticator is lost, the backup authenticator could recover the private key to get access to the online service. According to Frymann et al. [9], no transfer or sharing of secrets is required. However, it is not resolved whether the security requirements are met due to a lack of analysis.

3.2 Registration Process

In order to use a security key, the FIDO2 token must be registered by the user (see Figure 3.1).

1. First, the web application sends a request to the server that the user wants to register a FIDO2 token as security key.
2. Second, the server responds by sending back a request (*PublicKeyCredentialCreationOptions*) to the client for a FIDO2 public key, which can be stored for authentication. This request includes a random challenge from the server, a user handle linked to the user account, a list of supported credential types, authenticator filtering criteria, a list of already registered credentials, and the server's preference for authenticator attestation [23]. In detail, the *PublicKeyCredentialCreationOptions* contain the following data fields for credential creation [10]:
   - **attestation:**
     Attestation is a built-in feature of the FIDO and WebAuthn protocols. It enables the web application to use a cryptographically verified chain of trust from the security key's manufacturer. Therefore, it can be chosen

which security keys can be trusted and which not. The Web Authentication API defines four types of attestation conveyance. First, the *"none"* type defines that the relying party is not interested in the attestation procedure of the authenticator. Second, the preference *"indirect"* allows the client to choose how the attestation statement is obtained. The *"direct"* option defines that the relying party wants to receive the attestation statement as it was generated by the authenticator. Last, the *"enterprise"* type specifies that the relying party wants to receive an attestation statement that may include unique identifying information.

- **attestationFormats:**
  This member specifies a preference for the attestation statement format that is used by the authenticator. The values for this parameter should be taken from the *IANA WebAuthn Attestation Statement Format Identifiers* registry[2]. One possible attestation statement format identifier could be "apple" which is used with Apple devices' platform authenticators.
- **authenticatorSelection:**
  With this field, restrictions concerning the authenticator's type can be defined. The value *"platform"* indicates a software token or rather a platform authenticator, like Windows Hello, and the value *"cross-platform"* indicates a hardware token or rather a roaming authenticator, like a USB security token. Furthermore, this attribute gives information on whether user verification is compulsory.
- **challenge:**
  The challenge parameter is a buffer of generated random bytes and is used to prevent replay attacks. Replay attacks are a form of network attacks that are used to deliberately delay or retransmit data. In order to prevent replay attacks, the challenge should contain enough entropy to make guessing as difficult as possible. Therefore, the challenge should have a length of at least 16 bytes which provides 2^{128} possible values and makes it hard enough for an attacker to guess the value of the challenge [10].
- **excludeCredentials:**
  This field is used to avoid identical credentials for the same user. If there is already a credential saved for the user (from the current authenticator), the creation of a new credential will fail.
- **extensions:**
  This member contains additional parameters to request extra processing by the authenticator or the client. One example could be that the relying party requests additional information from the client about the credential that was created. In detail, the values for setting this parameter should be obtained from the *IANA WebAuthn Extension Identifiers* registry[2].
- **pubKeyCredParams:**
  The *pubKeyCredParams* field specifies the encryption algorithms, which are supported by the web application during registration and authentication. A list of all possible algorithms can be obtained from the *IANA COSE Algorithms* registry[3]. The relying party should at least support the Ed25519, ES256, and RS256 algorithms.
- **rp:**
  rp stands for relying party. This term is used to refer to the server which

[2]https://www.iana.org/assignments/webauthn/webauthn.xhtml
[3]https://www.iana.org/assignments/cose/cose.xhtml

provides access to the web application. It consists of a name and an identifier. The identifier must be the website's origin or the origin must be a subdomain of the identifier.

- **timeout:**
  The timeout parameter specifies the time in milliseconds, the user has for presenting a FIDO2 token.
- **user:**
  The user field contains information about the user's account. It consists of a name, a display name, and an identifier.

3. To get a public key credential from the authenticator, an asymmetric FIDO2 key pair, consisting of a private and a public key, must be created by the authenticator. Hence, the request has to be forwarded from the client to the authenticator. Afterwards, the JavaScript client calls the function `navigator.credentials.create()` of the Web Authentication API with the above-defined parameters. Also, the web browser validates the relying party identifier against the origin of the web application, hashes the client data, and calls the `authenticatorMakeCredential` method to communicate with the authenticator [28]. The `authenticatorMakeCredential` method is defined in the specification of the Client to Authenticator Protocol (CTAP) to request the generation of a new credential in the authenticator. The following parameters are defined to call the method `authenticatorMakeCredential` to communicate with the authenticator [4]:
   - **clientDataHash**:
     The hash of the serialized client data which includes the challenge from the server, the origin of the web application, the type of operation, and a flag whether cross-origin credentials are allowed.
   - **excludeList:**
     This member contains a list of registered credentials to limit the creation of several credentials for the same user with the same authenticator.
   - **enterpriseAttestation:**
     Enterprise attestation provides the ability to configure the attestation statement of authenticators to return unique identifying information. This would allow enterprises to improve their security and monitor the authenticator management strategy by only allowing specific devices and attestation statements [25].
   - **options:**
     A parameter that includes several flags to influence the behavior of the authenticator. The flags define, for example, restrictions for user presence and user verification.
   - **pinUvAuthParam:**
     The pinUvAuthParam member contains a Message Authentication Code (MAC) of the clientDataHash.
   - **pinUvAuthProtocol:**
     The pinUvAuthProtocol indicates the PIN/UV protocol version which was chosen by the platform. The PIN/UV auth protocol (pinUvAuthProtocol) is used to avoid sending plaintext PINs to the authenticator. The protocol encrypts the entered PINs and replaces them with pinUvAuthTokens. These tokens are randomly-generated byte-strings which are effectively unguessable. They should have a length of least 16 bytes [4].
   - The **extensions**, **pubKeyCredParams**, **rp**, and **user** parameters from step 2 are included in the `authenticatorMakeCredential` call as well.

4. At the authenticator, an asymmetric key pair can be created with the received information. The key pair is unique for the combination of the local device, the user account and the online service [23]. The private key remains safe at the authenticator and the public key will become part of the attestation [28]. During the creation of the FIDO2 key pair, the FIDO2 token requests the user's presence. This can be done by, for example, pressing a button or touching a sensor. If the user wants to register a security key for passwordless authentication, the user must also be verified, for example, by a PIN or some biometrics.

5. If the creation of the key pair was successful, the authenticator will return the created public key, the credential identifier, and other attestation data to the web browser. In detail, the following data is sent back to the web browser [4, 10]:

   - **authData:**
     The authData field contains metadata about the registration process, the credential identifier, and the created public key.
   - **epAtt:**
     This parameter indicates whether an enterprise attestation was returned.
   - **fmt:**
     The attestation format is used to indicate how the server is supposed to parse and validate the attestation data.
   - **attStmt:**
     The attestation statement looks different depending on the given attestation format. In general, the attestation format defines the syntax of the attestation statement. The attestation statement is a specific type of signed data object containing statements about the credential object itself and the authenticator.
   - **largeBlobKey:**
     This field contains a random key that enables storing opaque data associated with a credential.

   At the web browser the received data becomes the *attestationObject* [28].

6. Next, the promise of the `navigator.credentials.create()` method of the Web Authentication API resolves to a public key credential object (*PublicKeyCredential*) which contains the attestation response of the authenticator (*AuthenticatorAttestationResponse*) [28]. Afterwards, the *PublicKeyCredential* is sent back to the server. In detail, it consists of the following fields [10]:

   - **id:**
     The identifier of the newly generated credential for the user as base64-encoded string.
   - **response:**
     The response contains the client data in JSON format, which is used to store data that was passed from the client to the authenticator. Furthermore, the response includes the received attestation object from the authenticator [28].
   - **type:**
     The type parameter defines the string representing the credential type. This could be, for example, *"public-key"*.

7. The received public key credential object must be validated by the server. To validate the registration, the WebAuthn specification describes a procedure with several steps which have to be done on the server side [10]:

- The type of the client data must match the string *“webauthn.create”*.
- The challenge of the client data must match the the base64-encoded challenge created in step 2.
- The origin of the client data must match the origin of the relying party identifier.
- The hashed relying party identifier of the authenticator data must match the hashed relying party identifier expected by the relying party.
- The user present bit of the authenticator data must be set.
- The user verification bit of the authenticator data must be set (if user verification is required for this registration event).
- The alg parameter of the authenticator data must match one of the items of pubKeyCredParams created in step 2.
- The attestation statement format must be correctly determined.
- The attestation statement must be correct and must convey a valid attestation signature. For this process, the attestation statement, the attestation statement format, the authenticator data, and the hashed client data are used. How this process looks like depends on the attestation statement format. There are, for example, different attestation statement formats for Android, Apple, or Microsoft authenticators. If the validation is successful, a list of attestation root certificates must be obtained from a trusted source[4]. Afterwards, the attestation trustworthiness is assessed. Therefore, the attestation type must be accepted by the relying party or the attestation public key must correctly chain up to an attestation root certificate [10].
- The credential identifier has to have a length of at most 1023 bytes.
- The credential identifier must not already be registered for any user.

If the credential object is validated successfully, the public key and the credential identifier will be stored on the server and used for future authentication requests by the associated user.

3.3 Authentication Process

After a successful security key registration, the user can authenticate at the web application using the FIDO2 token (see Figure 3.2).

1. At first, the user chooses to authenticate at the web application. The login request is forwarded to the server by the web application.
2. Then, the server creates a request (*PublicKeyCredentialRequestOptions*), including the login challenge, which is sent to the client [27]. Following parameters are defined in the *PublicKeyCredentialRequestOptions* [10]:
    - **allowCredentials:**
      In this array, the credential identifiers, which have been saved during registration, are set for the corresponding user. Also, a preferred authentication option like USB, Bluetooth, or NFC can be configured.
    - **attestation:**
      This parameter defines a preference for the attestation conveyance.

[4]https://www.iana.org/assignments/webauthn/webauthn.xhtml

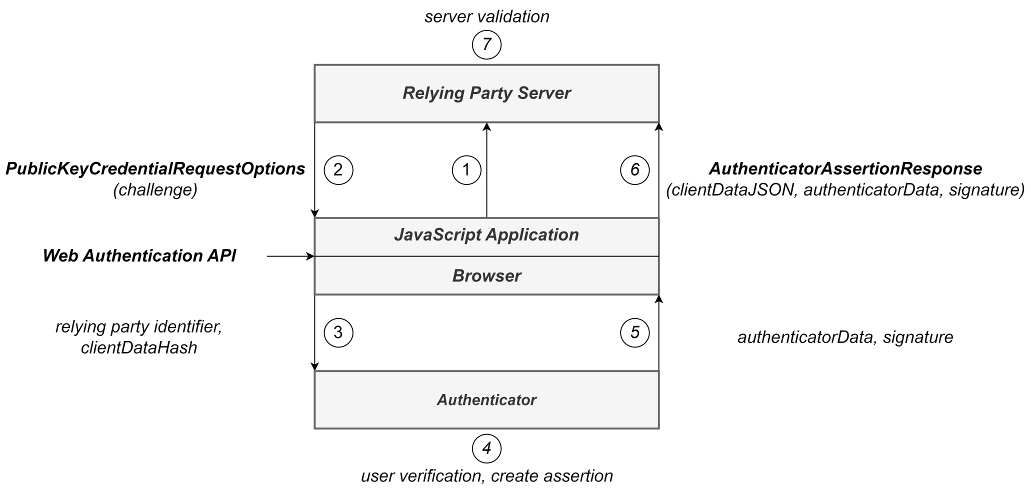

server validation
7
Relying Party Server
PublicKeyCredentialRequestOptions
(challenge)
2
1
6
AuthenticatorAssertionResponse
(clientDataJSON, authenticatorData, signature)
JavaScript Application
Web Authentication API
Browser
relying party identifier,
clientDataHash
3
5
authenticatorData, signature
Authenticator
4
user verification, create assertion

Figure 3.2: Authenticate with FIDO2 token (Source: [10])

- **attestationFormats:**
  This member works the same as during the registration. It specifies a preference for the attestation statement format that is used by the authenticator.
- **challenge:**
  This parameter works similarly to the registration process. It is a buffer that provides cryptographically generated, random bytes to prevent replay attacks.
- **extensions:**
  This member contains additional parameters to request extra processing by the authenticator or the client.
- **rpId:**
  The relying party identifier of the caller. It must match the relying party identifier which was used during registration.
- **timeout:**
  This field works similarly to the timeout parameter during the registration process. It specifies the time in milliseconds, the user has for presenting a FIDO2 token.
- **userVerification:**
  Defines whether user verification during the authentication process is mandatory. User verification is required for passwordless authentication.

3. On the client, the received information is used to communicate with the authenticator. The request of the server must be forwarded to the authenticator in order to sign the login challenge. Therefore, the client calls the JavaScript function `navigator.credentials.get()` of the Web Authentication API. Next, the browser validates the relying party identifier against the origin of the web application. Moreover, the client data is hashed and the `authenticatorGetAssertion` method is called [27]. This method is specified in the standard of the Client to Authenticator Protocol (CTAP) and is used to communicate with the authenticator [4]. In detail, the following parameters can be specified for `authenticatorGetAssertion`:

   - **allowList:**

This parameter defines a list that contains information, which credentials are allowed to be used for assertion by the authenticator.

- **clientDataHash:**
The hashed serialized client data. It includes the origin of the web application, the type of the operation, and the challenge created from the server [10].
- **extensions:**
The extensions parameter contains information to influence authenticator operations.
- **options:**
A parameter which includes several flags to influence the behavior of the authenticator. The flags define restrictions for user presence and user verification.
- **pinUvAuthParam:**
The pinUvAuthParam member contains the same information as during the registration process, a Message Authentication Code (MAC) of the clientDataHash.
- **pinUvAuthProtocol:**
The pinUvAuthProtocol indicates the PIN/UV protocol version which was chosen by the platform.
- **rpId:**
The relying party identifier of the server.

4. At the authenticator, the login challenge of the server is signed by the authenticator [23]. During the signing process, the FIDO2 token requests user presence and, if needed, user verification. Therefore, the user has to communicate in some way with the authenticator during the authentication process.
5. If the signing of the login challenge is successful, the authenticator will send back the credential identifier, the actual signature, the data structure used to generate the signature, the user handle, a signature counter, and several flags to the web browser [23]. The Client to Authenticator Protocol (CTAP) describes which fields are required to form a correct response [4]:
   - **authData:**
   The authenticator data stores information about the attempted authentication. This includes some flags, whether the user was present or verified during authentication. Additionally, the authenticator data contains the hashed relying party identifier and a signature counter of the authenticator.
   - **credential:**
   The credential field contains the credential identifier and credential type.
   - **largeBlobKey:**
   The parameter contains the content of the associated largeBlobKey created during registration.
   - **numberOfCredentials:**
   This member carries information about the total number of credentials for the relying party.
   - **signature:**
   This field contains the assertion signature, which was created from the authenticator.

- **user:**
  The user field contains information about the user name and the display name.
- **userSelected:**
  This parameter indicates whether the credential was selected by the user via direct communication with the authenticator.

6. At the web browser the promise of the `navigator.credentials.get()` method of the Web Authentication API is resolved to a public key credential object (*PublicKeyCredential*), which includes the authenticator assertion response (*AuthenticatorAssertionResponse*) from the authenticator. The received information is then sent back to the server to verify the signature and to finalize the authentication [27]. In detail, the *PublicKeyCredential* object contains the following fields:
   - **id:**
     The credential identifier that was used to generate the authentication assertion as a base64-encoded string.
   - **response:**
     The response object contains the client data in JSON format. Furthermore, the authenticator data is included in the response field. Moreover, the response object contains the signature object generated by the authenticator, which must be verified by the server. The signature object was generated by signing the login challenge from the server with the private key. The last field is the user handle, which stores the user identifier created during registration. If the authenticator supports attestation in assertions, also an attestation object will be created containing the attestation statement and attestation statement format [10].
   - **type:**
     This type field defines a string representing the credential type. This could be, for example, *"public-key"*.
7. Finally, the signature has to be verified by the server with the public key, which has been stored during the registration. In order to verify the signature, the WebAuthn specification describes a procedure with several steps which have to be done by the server [10]:
   - The credential identifier must match one of the options of the allowed credentials defined in step 2.
   - The user handle of the response object must be present.
   - The user identifier must match the identifier of the response object.
   - The type of the client data must match the string *"webauthn.get"*.
   - The challenge of the client data must match the base64-encoded challenge created in step 2.
   - The origin of the client data must match the origin of the relying party identifier.
   - The hashed relying party identifier of the authenticator data must match the hashed relying party identifier expected by the relying party.
   - The user present bit of the authenticator data must be set.
   - The user verification bit of the authenticator data must be set (if user verification is required).

- The signature must be a valid one over the binary concatenation of the authenticator data and the hashed client data. For this procedure, the stored public key is used.
- The bit which indicates that the attested credential data was included must be set.
- The public key and the credential identifier in the attestation object must match the public key and the credential identifier saved on the server.
- The attestation statement must be correct and must convey a valid attestation signature. If the validation is successful, a list of attestation root certificates must be obtained from a trusted source[5]. Afterwards, the attestation trustworthiness is assessed. Therefore, the attestation type must be accepted by the relying party or the attestation public key must correctly chain up to an attestation root certificate [10]. This process is called attestation in assertion. It is optional, must be supported by the authenticator, and must be requested by the relying party. Attestation in assertion could be helpful if the attestation statement format involves a third-party attesting to the state of the authenticator. Then, attestation in assertion can help to always check the authenticator to attest. Furthermore, it can also be useful for multi-device credentials where a generating authenticator (involved in the `authenticatorMakeCredential` operation) and a managing authenticator (involved in the authentication operation) are used. The generating authenticator and managing authenticator may not match. Therefore, attestation in assertion allows the relying party to observe such changes.

If all steps are verified successfully, the user will be logged in to the web application. Otherwise, the signature is not verified and access to the web application will be denied [1].

[5]https://www.iana.org/assignments/webauthn/webauthn.xhtml

Chapter 4

Design Choices

In order to implement FIDO2 token authentication, several design choices had to be made. There have already been some existing components of the web application that could be used to build upon. However, also new components have been added to the web application (see Figure 4.1).

4.1 Rocket

Most parts of the extension for FIDO2 token authentication are implemented in Rust, since the web application of the Personal Identity Agent is developed in Rust too. The web application is based on the Rocket web framework[1], which is used to develop secure web applications in Rust. The Rocket web framework has three philosophies: Primarily, Rocket is easy to use and takes a lot of measures to ensure that the developed web application is secure. Furthermore, all request handling is typed and self-contained. This means that Rocket handles type conversions and requests are handled by normal functions with parameters. Lastly, the Rocket web framework supports different, optional components and libraries.

4.2 Tera

For showing server-side data on web pages, Tera[2] templates are used. Tera is a template engine for Rust. The Tera templates are, for example, used to show

[1]https://rocket.rs/
[2]https://tera.netlify.app/

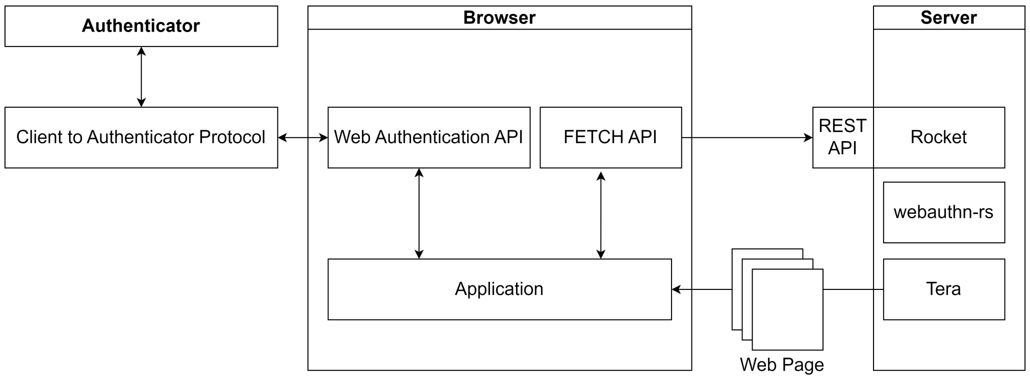


Figure 4.1: Component interaction diagram of the web application

the registered security keys to the user. There were already some existing Tera templates. Therefore, Tera templates are used for all new HTML pages which are not static. For the FIDO2 token authentication, several Tera templates were created:

- `add_security_key.html.tera`
- `check_password.html.tera`
- `start_authentication.html.tera`
- `start_registration.html.tera`

4.3 Web Authentication API

The communication between the authenticator and the server is done via the client side (= browser) and is, therefore, handled in JavaScript. The Web Authentication API[3] is used for communication between the client and the authenticator. It enables passwordless and two-factor authentication with public key cryptography. Generally, the Web Authentication API provides two methods for registering and authenticating with a security key:

- `navigator.credentials.create()`:
  This method is used to create new credentials (asymmetric key pair) associated with a user account.
- `navigator.credentials.get()`:
  The `navigator.credentials.get()` method is used for authenticating to a service with existing credentials.

4.4 Fetch API

The client uses the Fetch API[4], which creates request and response objects, in order to be able to communicate with the server. It is similar to XmlHttpRequest[5] but provides a more powerful feature set. To handle the requests of the Fetch API from the client, a REST API is implemented on the server side. All those functions transmit the data in JSON format. The method of all Fetch requests is *POST*. In the following sections, these functions will be described in detail.

4.5 WebAuthn Server

In order to create and verify credentials on the server side, a search and an evaluation of libraries for implementing WebAuthn into Rust web servers were done. In particular, the search was narrowed down to libraries that provide the relying party component of the FIDO2 protocol. Overall, three Rust crates[6] were found and evaluated to implement WebAuthn for Rust server web applications.

[3]https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API
[4]https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[5]https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[6]https://crates.io/

First, the slauth crate (version 0.5.0)[7] was evaluated. On Docs.rs all modules of the crate are documented [6]. However, the last release of the crate was in July 2020. By now, the crate was downloaded over 9,000 times. On GitHub the slauth repository has 10 contributors and according to the `README.md` server-side verification of WebAuthn is only partially implemented [17].

The second library which was evaluated was the libreauth crate (version 0.15.0)[8]. On Docs.rs all modules are documented and also examples are given in the documentation [7]. The last release of the crate was in April 2022. As of yet, the crate was downloaded over 120,000 times. On GitHub the repository has 6 contributors. However, the project is currently under development and the developers suggest that libreauth should not be used in production until the version 1.0.0 is released [5].

At last, the webauthn-rs crate (version 0.4.6)[9] was evaluated. The webauthn-rs crate uses OpenSSL and it follows the standards of the W3C Web Authentication. Besides, the webauthn-rs library was successfully audited by SUSE product security [11]. On Docs.rs the crate is documented and also examples for the implementation are given [8]. The last release of the crate was in November 2022. Thus far, the crate was downloaded over 85,000 times. On GitHub the repository has 18 contributors [11].

After comparing the three libraries, the decision to implement WebAuthn for Rust server web applications fell on the webauthn-rs crate. The library is currently in development, was audited, follows the W3C Web Authentication standards, is used by a broad community, is well documented, and is ready to be used in production. Furthermore, with the webauthn-rs library, the types and parameters described in section 3.2 and section 3.3 can be defined. Also, the validation of the public key and the verification of the signature are handled by the library. Hence, no cryptographic algorithms need to be implemented, as those functionalities are provided by the webauthn-rs library.

4.6 webauthn-rs Features

The webauthn-rs library is included in the web application via the *Cargo.toml* files of the Personal Identity Agent project. The library can be extended by the use of features, which provide additional functionalities compared to the default implementation. In order to be able to use all functionalities of the FIDO2 token authentication, the features must be activated accordingly.

Firstly, the *danger-user-presence-only-security-keys* feature is required to allow security keys without user verification. By default, user verification is required for security keys. However, if two-factor authentication is used, the user will have to enter a password and therefore, a knowledge factor anyways. With user verification enabled, the user would have to present an additional knowledge factor during the authentication process. Regarding user convenience, this is not a good option. For this reason, security keys which verify only user presence are allowed with two-factor authentication. If passwordless authentication is used, user verification will be mandatory. By enabling this feature, user verification will be prevented if it is not required. Nevertheless, newer keys will force user verification during the registration process but will not require a verification of the user during the authentication procedure. This function is implemented by design [8].

[7]https://crates.io/crates/slauth
[8]https://crates.io/crates/libreauth
[9]https://crates.io/crates/webauthn-rs

Secondly, the *preview-features* feature is enabled. This feature enables passwordless security key registration and authentication. A passwordless key is a cryptographic authenticator, which enforces multi-factor authentication. In other words, the security key verifies the user via, for example, a PIN or a biometric factor.

Lastly, as part of the *preview-features* feature, the *resident-key-support* must be activated too. The *preview-features* feature partly depends on functions and types that are implemented as part of the *resident-key-support* feature. Thus, if the feature *resident-key-support* is not activated, the above mentioned feature will not work properly.

Chapter 5

Implementation

5.1 Security Key Persistence

To ensure that FIDO2 token authentication works properly, the public key of the user must be stored safely on the server. There is already a working method for persisting server data, which has been extended for the FIDO2 token authentication.

5.1.1 Data

The Personal Identity Agent (PIA) has a struct `Data`, which holds information about configuration, identities and transactions. This is used to store and load information for the PIA. For the FIDO2 token authentication, an additional field called `security_keys` has been added to the struct for storing the user's security keys (see Listing A.1). This data structure is serialized into the `Data.json` file, which is stored on the server. During the start of the web application, all important data is read from the file.

5.1.2 SecurityKeys

The additional field in the `Data` struct is of type `SecurityKeys`, which is a new struct holding all security key information (see Listing A.6). This information is needed for the FIDO2 token authentication of the user. The `SecurityKeys` struct contains two fields. The `credentials` field, which stores the user credentials for authentication and the field `authentication_type`, which indicates whether two-factor authentication or passwordless authentication will be enabled.

The authentication type is stored in an enum (see Listing A.3). The default value is `NoType`, which means that the user does not use FIDO2 token authentication. `Password` means that two-factor authentication is enabled and `Passwordless` describes the fact that passwordless authentication is used. Depending on the authentication type, the functionality and appearance of the web application authentication will differ.

5.1.3 Credentials

As mentioned above, the security key credentials are stored on the server side. For this reason, the struct `Credentials` is used (see Listing A.2). It stores, depending on the authentication type, the `SecurityKey` or the `PasswordlessKey` object for the user. Both fields are annoted with `skip_serializing_if = "Option::is_none"` and will therefore be skipped during the serialisation process if they have no value. Additionally, the username and the name of the security key are stored. The `SecurityKey` and the `PasswordlessKey` objects store, for example, the credential identifier, the attestation certificate, the registration policy, the public key and some flags which were set during the registration process. This includes information about how the user was verified or the signature counter of the authenticator. Depending on the `SecurityKey` or the `PasswordlessKey` object, registration and authentication is handled differently in terms of allowed certificates and required user verification.

5.2 Temporary Data

All fields in the `WebauthnData` struct (see Listing A.7) are optional as those values are only stored temporarily during the registration and the authentication process. The `username` and `security_key_name` fields are stored during the security key registration.

`password_checked`, `registration_started` and `authentication_started` are used to store timestamp values. Those values are used by the request guards of the Rocket web framework to prevent erroneous or malicious access to web pages. A request guard is a powerful instrument of the Rocket web framework, which protects web pages from being accessed by mistake. This check is done based on the incoming request information.

For requesting a public key or a signature of the authenticator, the fields `creation_challenge_response` and `authentication_challenge_response` are used. Those values are stored on the server until they get sent to the client side.

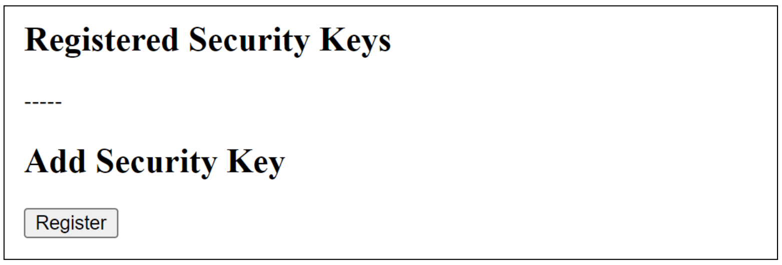
The last four fields of the `WebauthnData` struct are filled, depending on the authentication type, during the registration and authentication process. They store, for example, the user verification policy, encryption algorithms or login challenges.

5.3 Security Key Registration

In order to use security keys for the authentication process of the Personal Identity Agent (PIA), they must be registered by the user. This can be done on the pairing page (see Figure 5.1) after the user has finished authenticating at the PIA.

For the authentication process, the user does not need a username. Each PIA runs as a separate process in a virtual machine and the user gets access to a unique onion service. Therefore, the web application only asks for a password during the authentication process. However, the Web Authentication API needs a username for the FIDO2 token authentication. Therefore, the unique identifier of the user and the username are created randomly with a Universally Unique Identifier (version 4) and set during the registration.

The security key registration process on the server is split into the `start_registration()` and `finish_registration()` methods. The first function checks if the received security key name already exists. If not, a challenge for the security key registration will be created, which is used by the Web Authentication API. Also, a timestamp is created, which is used to restrict access to the *start_registration* page. The `finish_registration()` function is used to process the response of the credential object, created from the Web Authentication API.



Registered Security Keys

Add Security Key

Figure 5.1: Pairing page without security keys

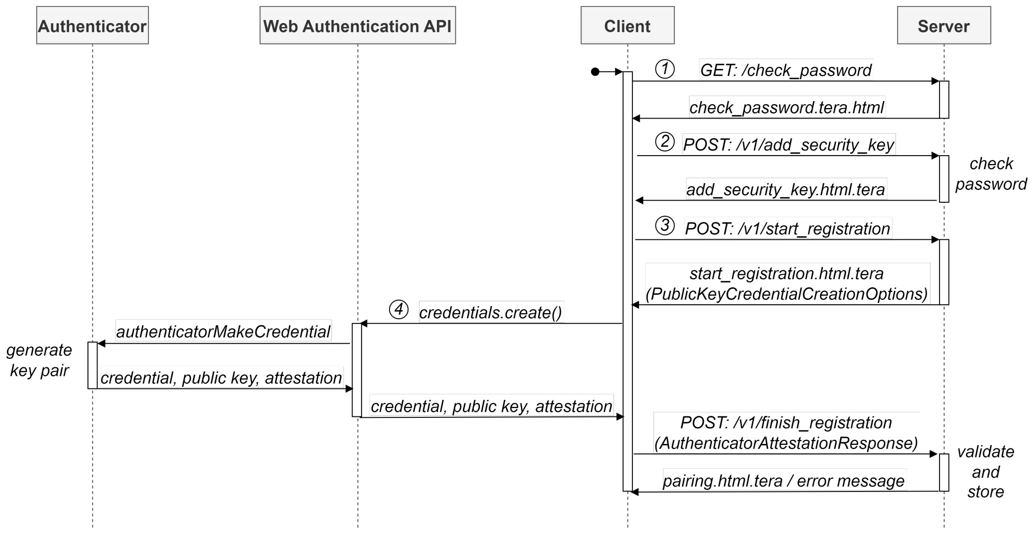

Authenticator
Web Authentication API
Client
Server
① GET: /check_password
check_password.tera.html
② POST: /v1/add_security_key
check password
add_security_key.html.tera
③ POST: /v1/start_registration
start_registration.html.tera (PublicKeyCredentialCreationOptions)
④ credentials.create()
authenticatorMakeCredential
generate key pair
credential, public key, attestation
credential, public key, attestation
POST: /v1/finish_registration (AuthenticatorAttestationResponse)
validate and store
pairing.html.tera / error message

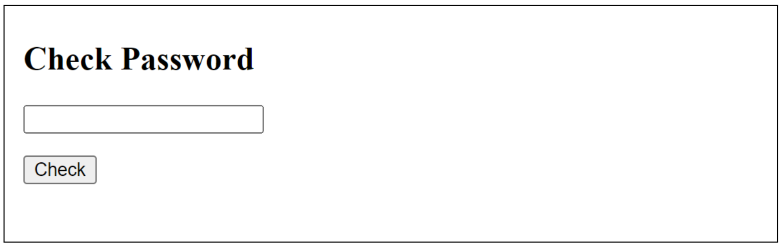
Figure 5.2: Sequence diagram for security key registration (Source: [23])

After the response has been verified, the credential of the registered security key is saved.

An overview of the security key registration process is depicted in Figure 5.2.

5.3.1 Check Password

If a user wants to register a security key, the *Register* button must be clicked. Afterwards, the user is redirected to the *check_password* page (see Figure 5.3). As the name implies, the *check_password* page is used to ask for the user's password. This happens every time the user wants to register a security key. Also, if the authentication process is already passwordless, the user must enter the password during the registration process of a new security key.



Check Password

Figure 5.3: *check_password* page

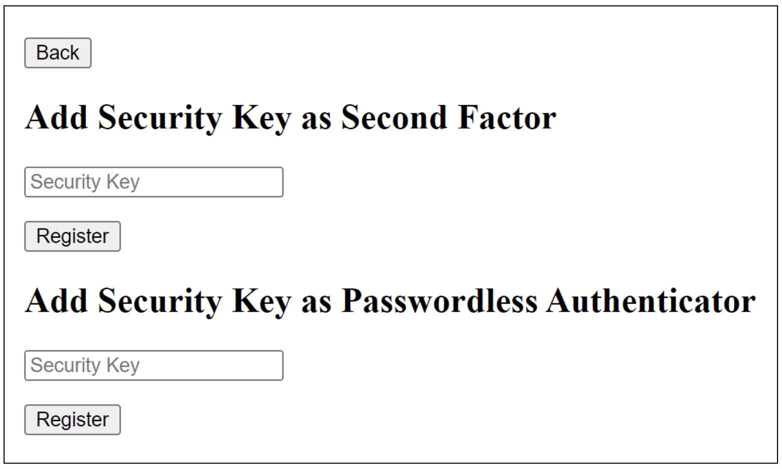

Back
Add Security Key as Second Factor
Security Key
Register
Add Security Key as Passwordless Authenticator
Security Key
Register

Figure 5.4: *add_security_key* page

To verify the password, the function `add_security_key()` has been created on the server. This function is called via the REST API before the *add_security_key* page is accessed (see Figure 5.2). It is used to verify the password that the user entered on the *check_password* page. If the password is verified, a timestamp will be created which is used to restrict access to the *add_security_key* page. After the user has entered the correct password, access to the *add_security_key* page is granted (see Figure 5.4).

Normally, this page is protected by a request guard and cannot be accessed without entering the correct password. In that case, the request guard will check if a password has been entered correctly in the last 300 seconds. The value for the time limit is stored in the constant `CHECK_PASSWORD` (see Listing A.5). This value is used by the request guard to protect the *add_security_key* page and the `start_registration()` method from erroneous access. If the password check has expired, the user gets an error message and can go back to the pairing or login page (see chapter 6).

5.3.2 Add Security Key

On the *add_security_key* page, the user can choose between two authentication types. First, two-factor authentication can be chosen to add a second factor to the current authentication process in addition to the password. The second option is used to add a security key for passwordless authentication. Depending on previously added security keys, the other option is hidden from the user. The authentication process of the PIA only allows one authentication type at a time. Therefore, only security keys with the same authentication type as the first registered security key can be added.

The main reason for only allowing one authentication type at a time, is to avoid design implications. Primarily, the login page and the authentication would be more difficult to implement. With only one allowed authentication type at a time, the implementation of the FIDO2 token authentication is facilitated. One problem could be, how the login page should look like, if both authentication options are allowed. A solution could be to show the password field every time, although passwordless authentication is used. This would not be a good option regarding user experience since the user cannot be 100 percent sure that no password is needed. The second problem is that the check of the password is done before the FIDO2 token authentication starts. Therefore, it is uncertain, how a passwordless authentication should be handled if the user entered a password. It would be difficult to implement the distinction of the authentication types.

In both cases, a unique name for the new security key must be entered. If the user tries to register a security key under a name that already exists, the registration process will fail.

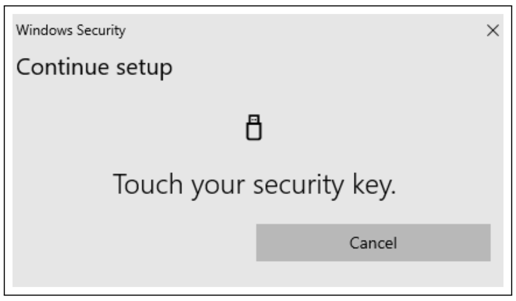
5.3.3 Start Registration

If the user clicks the *Register* button, the `start_registration` process will be initiated. At first, the `start_registration()` function on the server side is called from the client with the request *"POST: /v1/start_registration"* (see Figure 5.2). The server receives the security key name, the authentication type, the origin of the web application, and the relying party identifier in JSON format. At the server, this information is used to create a request for a public key.

Depending on the authentication type, the webauthn-rs library provides different functions:

- **Two-Factor Authentication:**
  To register a security key as second factor, the webauthn-rs function `start_securitykey_registration()` is used. For this function, the `attestation_ca_list` can be set as a parameter. This parameter contains a list of certificates from authenticator manufacturers, that the web application can trust. For example, if only Yubikeys should be allowed for the Personal Identity Agent, the Yubico Root certificate authority can be provided in this list to validate that all registered devices are manufactured by Yubico. For two-factor authentication, no constraints are set for authenticator manufacturers. Furthermore, the `start_securitykey_registration()` function checks if the feature *danger-user-presence-only-security-keys* is set (see section 4.6). If that is the case, user verification will not be enforced during the registration process. The webauthn-rs function returns a registration state object of the type `SecurityKeyRegistration`.
- **Passwordless Authentication:**
  The `start_passwordlesskey_registration()` function is used to register a security key as a passwordless authenticator. This function also includes the parameter `attestation_ca_list`. In contrast to two-factor authentication, the parameter needs a value if a passwordless authenticator is registered. This constraint is imposed by the webauthn_rs crate. It provides several lists, which can be passed over to the function. For example, the list can be limited to allow only Apple or Android certificate authorities. For the Personal Identity Agent, all certificate authorities, which are known to the webauthn-rs project, are allowed. This list contains certificate authorities from Google, Android, Apple, Microsoft, Yubico, and Nitrokey. Because a passwordless authenticator is registered, the `start_passwordlesskey_registration()` function enforces user verification. This function returns a registration state object of the type `PasswordlessKeyRegistration`.

For both functions, the user parameter must be defined. This includes a unique identifier, the username, and the display name. Since there is no user identifier required for the Personal Identity Agent, the unique identifier contains a version 4 Universally Unique Identifier (UUID). The username also contains a version 4 UUID but as a String representation and the display name contains the name of the security key.

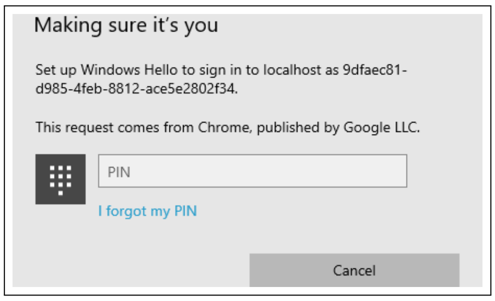

Windows Security

Continue setup

Touch your security key.

Cancel

(a) Request user presence


Making sure it's you

Set up Windows Hello to sign in to localhost as 9dfaec81-d985-4feb-8812-ace5e2802f34.

This request comes from Chrome, published by Google LLC.

PIN

I forgot my PIN

Cancel

(b) Request Windows Hello PIN

Figure 5.5: Register security key

Furthermore, both functions return either a `SecurityKeyRegistration` object or a `PasswordlessKeyRegistration` object. No matter which authentication type is used, the objects are stored on the server for later use. The registration state object is later used to verify the registration process.

Besides, both functions return an object of the type `CreationChallengeResponse` (see section 3.2) that is presented via a Tera template to the client by invoking the *start_registration* page.

The *start_registration* page is protected by request guards. In the first phase of the registration process, a timestamp is created and saved on the server. To grant or deny access to the *start_registration* page, the timestamp must not be older than the value stored in the `TIME_LIMIT` constant (see Listing A.4) in seconds. If the time limit has expired, the pages cannot be accessed by the user. They will show the HTTP error *403: Forbidden*. Additionally, the user must be logged in to access the *start_registration* page.

5.3.4 Create Key Pair

On the client side, the `CreationChallengeResponse` object is used to call the JavaScript function `navigator.credentials.create()` of the Web Authentication API. This function creates a key pair by requesting a FIDO2 token from the user (see Figure 5.5). The user must be present during the registration process. As mentioned in section 2.1, different technologies and devices can be used as FIDO2 tokens. Two examples of how to register a FIDO2 token can be seen in Figure 5.5. Figure 5.5a shows the registration of a USB security token. During this process, the Web Authentication API may prompt a dialog and ask for access to the USB security token. In the case of passwordless key registration, the user must also be verified by some biometric feature or a PIN. If the USB security token is used for the first time and was not configured by the user beforehand, it might be possible that a PIN must be set to use the security key. Since the entered PIN is required during the authentication process, the user should remember it. Figure 5.5b shows a registration with the Windows Hello PIN as a security key. In that case, the correct Windows Hello PIN of the user must be entered.

5.3.5 Finish Registration

After the Web Authentication API call, the `finish_registration()` server function is called via the REST API (see Figure 5.2). As described in section 3.2, the request contains the credential identifier, the attestation object, the client data

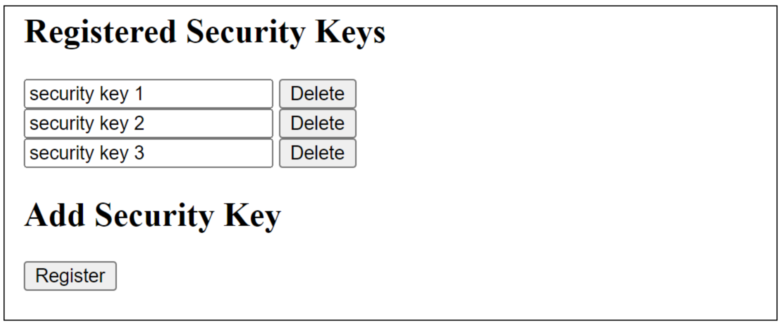


Figure 5.6: Pairing page with registered security keys

in JSON format, the origin of the web application, and the relying party identifier. Depending on the authentication type, the `finish_securitykey_registration()` function or the `finish_passwordlesskey_registration()` function of the webauthn-rs library is invoked. Those functions are used to complete the registration of the credential. They use, depending on the authentication type, the registration state object (`SecurityKeyRegistration` or `PasswordlessKeyRegistration`), which was stored during the *start_registration* process. The registration state object is used to validate the registration. Furthermore, the response object of the Web Authentication API is set as a parameter. The validation process is composed of several steps [10]. This part is fully provided by the webauthn-rs library.

After successful validation of the registration, the functions return a `SecurityKey` respectively a `PasswordlessKey` object. The objects contain the credentials, which are used for user authentication later. Therefore, the credentials are stored safely on the server. If the registration process is finished successfully, the user will be redirected to the pairing page and will see the newly added security key. Users can register multiple security keys on different devices so that they have fallback options for the authentication process. If there are several security keys registered for one user, they are shown one underneath the other (see Figure 5.6).

5.4 Security Key Removal

On the pairing page, all added security keys are shown to the user in the security keys section (see Figure 5.6). In this section, up to five keys are allowed. The number of security keys is defined by the constant `MAX_SECURITY_KEYS`, which is a configuration option of the server-side implementation. If the limit is reached, the option for adding security keys will be omitted from the pairing page. In addition to that, the registration of a security key will be canceled on the server, if the number of registered security keys is already higher or equal to the number stored in the constant `MAX_SECURITY_KEYS`.

Each security key can be deleted by clicking on the *Delete* button next to it if the user decides it is no longer needed. This invokes the `remove_security_key()` function on the server, which removes a security key from the user. The identifier of the security key is set as a parameter and used to search for the corresponding security key credential. If an entry is found in the credentials list of the `Data` struct, it will be removed and the user can no longer authenticate with it. Furthermore, if the list of the security key credentials is empty after the removal process, the authentication type of the user will be set to `NoType`. That way, the authentication procedure with FIDO2 tokens is disabled, as no registered security keys are left.

There is no best practice for the fallback option after FIDO2 token authentication is removed from a user account but it is suggested to warn the user that all security keys are removed [29]. How others implement the fallback option is discussed in chapter 6.

In order to change the identifier of a registered security key, the security key has to be deleted and registered again, as changing the identifier is not possible. The renaming of security keys has not been implemented in the context of this Bachelor thesis. In general, with the *Register* button, it is possible to add more security keys as long as there are not more than five keys registered. If there are already five registered security keys, a security key must be deleted before another one can be added.

5.5 Security Key Authentication

After the user has registered security keys, they can be used for the authentication process. Depending on the authentication type, the login page is different (see Figure 5.7).

If passwordless authentication is used, the text field for entering the password will be hidden, since no password is needed for the authentication (see Figure 5.7b). For two-factor authentication, the login page is not modified at all (see Figure 5.7a). The FIDO2 token authentication can be started directly by clicking on the *Login* button.

If two-factor authentication is used, the password must be entered in the text field by the user and checked by the server before the FIDO2 token authentication starts. However, if no security keys are registered for the user, the standard login page will be shown, since no changes were made for this case. Therefore, the user can log in by entering the correct password and clicking on the *Login* button.

The security key authentication process on the server is split into the `start_authentication()` and the `finish_authentication()` methods. The `start_authentication()` function checks and verifies the entered password of the user, if one was given. Also, a login challenge for the security key authentication is created, which is used by the Web Authentication API. The `finish_authentication()` function is used to process the response object of the Web Authentication API. This means that the signature is verified. After a successful verification, the user is authenticated. An overview of the security key authentication process is depicted in Figure 5.8.

5.5.1 Start Authentication

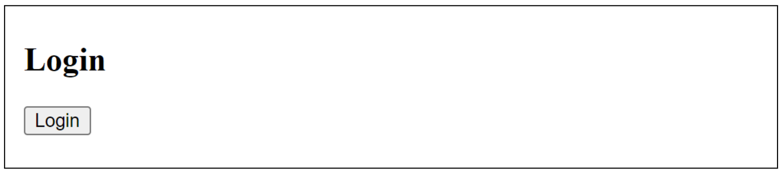
As soon as the authentication process is started, the web application calls the server function `start_authentication()` via the REST API (see Figure 5.8). The request includes the password and the authentication type. Furthermore, the origin and relying party identifier of the web application are included.

Depending on the authentication type, different functions of the webauthn-rs library are called:

- **Two-Factor Authentication:**
  To authenticate with a security key as second factor, the webauthn-rs function `start_securitykey_authentication()` is used. This function accepts a list of `SecurityKey` objects as a parameter. The parameter contains a

Login
Login

(a) Standard login page



(b) Passwordless login page

Figure 5.7: Login page

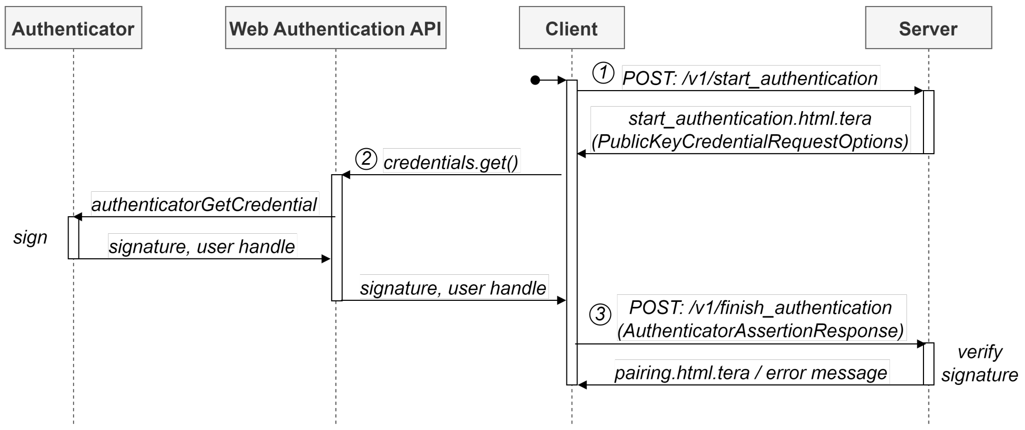

Authenticator
Web Authentication API
Client
Server
① POST: /v1/start_authentication
start_authentication.html.tera (PublicKeyCredentialRequestOptions)
② credentials.get()
authenticatorGetCredential
sign
signature, user handle
signature, user handle
③ POST: /v1/finish_authentication (AuthenticatorAssertionResponse)
verify signature
pairing.html.tera / error message

Figure 5.8: Sequence diagram for security key authentication (Source: [23])

list of all saved second-factor security keys of the user. Moreover, the `start_securitykey_authentication()` function checks if the feature *danger-user-presence-only-security-keys* is set (see section 4.6). If yes, user verification will not be required during the authentication process. The function `start_securitykey_authentication()` returns an authentication state object of the type `SecurityKeyAuthenication`.

- **Passwordless Authentication:**
  The `start_passwordlesskey_authentication()` function is used to authenticate with a security key as passwordless authenticator. It has one parameter, which contains a list of `PasswordlessKey` objects. Those objects have been saved during registration and are now used during passwordless authentication. The list contains all passwordless authenticators which can be associated with the user. Since passwordless authentication is used, the `start_passwordlesskey_authentication()` function enforces user verification. This function returns an authentication state object of the type `PasswordlessKeyAuthentication`.

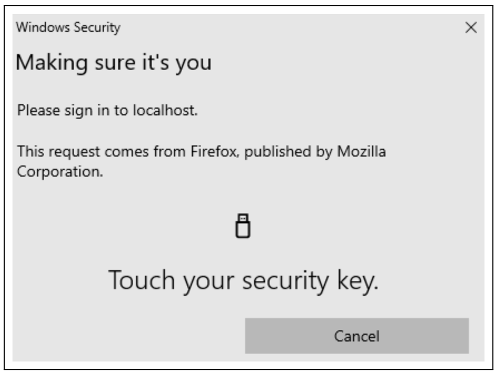
Both functions return either a `SecurityKeyAuthentication` object or a `PasswordlessKeyAuthentication` object. The authentication state objects are stored in their corresponding fields of the `SecurityKey` struct for later use during the authentication process. They contain the credential identifier, the user verification policy, and the challenge of the authentication attempt.

Moreover, both functions return a `RequestChallengeResponse` (see section 3.3) that is inserted in the Tera template of the *start_authentication* page and thus returned to the client. It contains the credentials, which are allowed to sign the login challenge. Due to this parameter, some privacy issues arise which are discussed in chapter 6. In addition to that, the object defines if user verification is needed for the authentication process. Furthermore, the timeout, the relying party identifier, and the login challenge itself are included.

The *start_authentication* page is protected by a request guard. In the first phase of the authentication process, a timestamp is created and saved on the server. Therefore, the *start_authentication* page can only be called if the timestamp is not older than the value stored in the `TIME_LIMIT` constant (see Listing A.4) in seconds. If the time limit has expired, the page cannot be accessed by the user. It will show the HTTP error *403: Forbidden*.

5.5.2 Create Signature

On the client side, the JavaScript function `navigator.credentials.get()` of the Web Authentication API, where the authentication challenge response is set as a parameter, is called. This function requests a registered FIDO2 token from the user (see Figure 5.9). During this request, the user's presence has to be confirmed. Figure 5.9 shows two examples of how to authenticate with a FIDO2 token. Figure 5.9a shows how the authentication works with a USB security token. If no security key is recognized, the web application will show a waiting dialog to the user. Moreover, the correct security key has to be presented during the authentication process. If the user presents a wrong USB security token, the web application will tell the user that the security key does not look familiar and that the correct USB security token must be inserted into a USB port. For passwordless key authentication, the user must also be verified by some biometric feature or a PIN. Therefore, the web application may ask for a PIN, for example, to use the USB security token. The user must enter the PIN configured during the registration process or in the operating system's settings. Otherwise, the authentication process will fail, as the user cannot be verified.


Windows Security

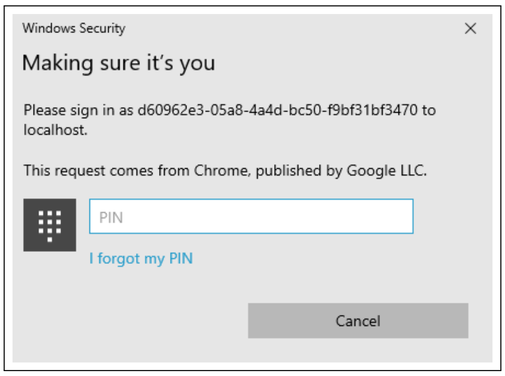
Making sure it's you

Please sign in to localhost.

This request comes from Firefox, published by Mozilla Corporation.

Touch your security key.

(a) Request user presence


Windows Security

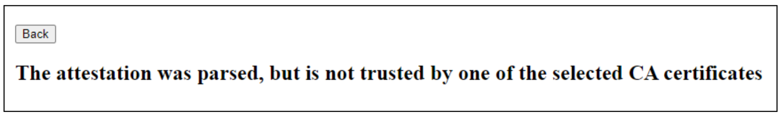
Making sure it's you

Please sign in as d60962e3-05a8-4a4d-bc50-f9bf31bf3470 to localhost.

This request comes from Chrome, published by Google LLC.

(b) Request Windows Hello PIN

Figure 5.9: Authenticate with a security key



The attestation was parsed, but is not trusted by one of the selected CA certificates

Figure 5.10: An exemplary error message

Figure 5.9b shows an authentication process with a registered Windows Hello PIN as security key. In this case, the correct Windows Hello PIN of the user must be entered so that the authentication succeeds.

5.5.3 Finish Authentication

After the interaction with the security key, the `finish_authentication()` function on the server side is called via the REST API (see Figure 5.8). As mentioned in section 3.3, the data which is sent back to the server contains the credential identifier, the response object, and the authenticator type. The response object includes the signature, the authenticator data, the user handle, and the JSON-encoded client data. On the server side, the function `finish_securitykey_authentication()` or `finish_passwordlesskey_authentication()` is called, which is determined by the authentication type. The authentication state object, created during the *start_authentication* process, and the response object are used as parameters. Those functions validate the authentication and verify the signature, which was created by the authenticator. The validation and verification process is fully provided by the webauthn-rs library. If the verification is successful, the user will be authenticated and redirected to the *pairing* page. Otherwise, the authentication process fails and the user will be redirected to the *login* page.

5.6 Error Handling

If an error occurs on the server, the response of the Fetch request is sent to the client in JSON format. The response includes the error message from the server. After the response has been received, the error message is shown on the current web page. The user can click on the *Back* button to get back to the pairing or login page, depending on the authentication status. Figure 5.10 shows an error message, which will occur if a wrong attestation format is used.

Chapter 6

Evaluation

6.1 Tor Browser

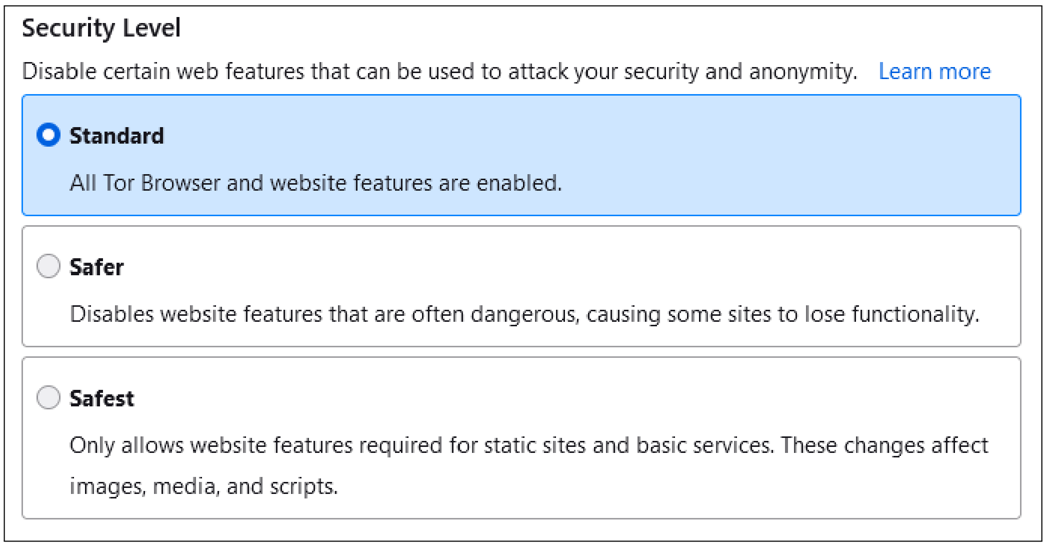
If the web application is used in the Tor Browser, some adaptions must be made so that the Web Authentication API will work correctly. That is because the Tor Browser has stricter security settings than a normal browser.

First of all, the security level of the Tor Browser must be set to *Standard* or *Safer* in the security level settings (see Figure 6.1). With the *Standard* option all Tor Browser and website features are enabled. The *Safer* option disables dangerous website features and therefore, some websites can lose their functionality. This option disables JavaScript for all non-HTTPS sites, some fonts and math symbols are disabled, and all media content is changed to click-to-play. The Tor Browser treats onion sites as HTTPS. For this reason, the *Safer* option is also sufficient for the functionality of the Web Authentication API. If the option *Safest* is selected, registration and authentication with FIDO2 tokens will not work correctly, as the Tor Browser forbids access to the Web Authentication API. After installation of the Tor Browser, the security level is set to *Standard*.

Additionally, one setting in the Tor Browser must be set manually so that the Web Authentication API calls work properly (see Figure 6.2). If the user enters *about:config* in the search bar of the Tor Browser, the preferences page will open. To allow Web Authentication calls for the Tor Browser, the entry *security.webauth.webauthn* must be set to true. This entry is set to *false*, as the default value. There was already a request for enabling the *security.webauth.webauthn* feature by default, but according to the Tor Project, the Web Authentication API must still be audited [21].

6.2 WebAuthn Standard

As mentioned in chapter 3, the registration and authentication of FIDO2 tokens are origin-specific. Therefore, the registration and authentication depend on the relying party identifier of the web application. The Personal Identity Agent is accessed in the Tor Browser via an Onion address. Thus, if the PIA is accessed via an Onion service, the Onion address will be used as relying party identifier. If the PIA is accessed directly, the relying party identifier will look different, since no Onion address is used. Therefore, a user's security key registered at the Onion address can only be used at the same Onion address to authenticate. Besides, if the registration is done over a non-Onion service but the authentication process is initiated via an Onion service, the authentication will fail. However, it is not intended that the Personal Identity Agent can be accessed without an Onion address. The current implementation uses the concept of ephemeral Onion addresses for callbacks into the Personal Identity Agent. But this implementation only affects machine-to-machine (M2M) communication. Hence,


Security Level

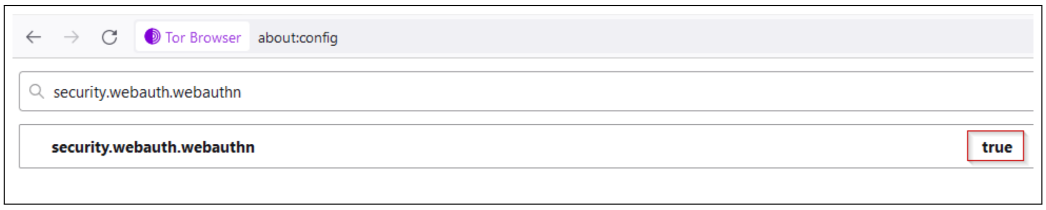
Disable certain web features that can be used to attack your security and anonymity. Learn more

[✓ Standard]
All Tor Browser and website features are enabled.

Safer
Disables website features that are often dangerous, causing some sites to lose functionality.

Safest
Only allows website features required for static sites and basic services. These changes affect images, media, and scripts.

Figure 6.1: Security level in Tor Browser


Tor Browser about:config

security.webauth.webauthn

security.webauth.webauthn true

Figure 6.2: Adaption of Tor Browser preferences

it can be assumed that the Personal Identity Agent is reachable for the user via an Onion service with a permanent Onion Address. This also means that the implementation of the Personal Identity Agent should have no problem with WebAuthn standard concerning origin-specific credentials.

6.3 Security Key Setup

In order to use the security keys appropriately in the web application, they have to be configured before the security key registration process. For platform authenticators like, for example, Windows Hello, the security key must be configured before it can be used in the web application. With USB security key tokens, it is possible to configure the PIN for the security key during the registration process. The biometric feature of a USB security token can only be used, if it was added to the security key beforehand. Unfortunately, registration of biometric factors is not possible in web applications. The Yubico webpage[1] gives an excellent introduction on how to set up a USB security token as security key on several operating systems and platforms.

6.4 Registration of Several Security Keys

If FIDO2 token authentication is chosen for the Personal Identity Agent, users should register several security keys to have several options to authenticate

[1]https://www.yubico.com/at/setup/yubikey-bio-series/

with. Thus far, functionality to restore security keys if they get lost has not been implemented. Therefore, it is recommended to register more than one security key to have a fallback option for the authentication process [14].

6.5 Fallback Option after FIDO2 Token Authentication

After FIDO2 token authentication was disabled, the fallback option generally depends on security and usability. By self-testing it could be determined, that Google, for instance, sends an e-mail to provide information about disabling two-factor verification, and mere password-based authentication is used again. Facebook, for example, just disables two-step verification and falls back to password-based authentication. The current implementation is discussed in chapter 5.

6.6 Privacy Considerations about Credential Identifiers

During the authentication process, the `RequestChallengeResponse` is sent to the client and contains a parameter that includes all allowed credentials. According to Lundberg [12], the parameter exposes the credential identifiers of the user to an unauthenticated caller. Therefore, there exists the risk that personal identifying information is leaked. However, this is only a problem for passwordless authentication since two-factor authentication asks for a password before FIDO2 token authentication is initiated. Thus, two-factor authentication prevents the information leak of credential identifiers [13]. In order to avoid the security risks of leaking credential information, passwordless authentication could be replaced by the use of discoverable credentials [10]. These credentials are stored on the authenticator or the client side and can be used during the authentication ceremony without the relying party providing any allowed credentials. This means that the relying party leaves the parameter containing the allowed credentials empty. Moreover, the authentication process can be done without identifying the user first. However, not all authenticators are able to work with discoverable credentials. The webauthn-rs crate provides four functions that support discoverable credentials. Nevertheless, on the documentation page of the webauthn-rs crate it is stated that this feature is currently in development [8].

6.7 Access Restriction of Start Registration Page

After the password check timed out, it is a good option regarding security to verify the password again instead of extending access to the page [18]. Long session timeouts should be avoided. Therefore, the user gets an error message that the password check has timed out. The timeout after checking the password was set to 300 seconds (see Listing A.5). The page access will not be extended if the password check expires. 300 seconds should be enough time for the user to finalize the registration. Furthermore, it would be a good option to limit access to the page by proper use of cookies instead of using several request guards in combination with server-side flags.

Chapter 7

Conclusion and Outlook

In the context of this Bachelor thesis, the Personal Identity Agent (PIA) of the Digidow project was enhanced by implementing safer authentication methods. For users, it is now possible to authenticate with FIDO2 tokens. There are different options for the authentication process. Firstly, the implemented extension allows two-factor authentication. In that case, the FIDO2 token is the second factor in addition to an existing password. Hence, the security of password-based authentication is enhanced by the additional second factor. Secondly, the extension enables passwordless authentication. The FIDO2 token is used as a passwordless authenticator. Although the password does not have to be entered anymore, security standards are not compromised. Here, the password is replaced by client-side authentication towards the FIDO2 token itself, for example with a short PIN or a biometric factor. Therefore, passwordless authentication is a very good option in terms of user convenience.

During the implementation, a very interesting aspect has been discovered for FIDO2 token authentication with different domains pointing to the same service. As described in chapter 3, the key pair created during registration and used for authentication is origin-specific. This functionality is important to guarantee that no two web applications use the same cryptographic key and thus, see the same user identity. Otherwise, the user would be traceable across different web applications. Therefore, it will not be possible for users to apply FIDO2 token authentication if the service is accessed via different domains.

In future, the Personal Identity Agent can be enhanced by implementing the recoverability of security keys. Currently, if a security key gets lost, the user has to have at least one security key as a fallback option to still be able to access the Personal Identity Agent. Moreover, if FIDO2 token authentication is disabled the user could be informed by e-mail, for instance, that mere password-based authentication is used again. Furthermore, the access protection of the *start_registration* page can be enhanced. Instead of using a time limit in combination with a request guard, access to the *start_registration* page can be restricted with the proper use of cookies. Additionally, passwordless key authentication can be enhanced by the support of discoverable credentials. This would avoid leaking credential identifiers during the passwordless authentication process.

Bibliography

[1] Fatima Alqubaisi, Ahmad Samer Wazan, Liza Ahmad, and David W. Chadwick. 2020. Should We Rush to Implement Password-less Single Factor FIDO2 based Authentication? In *2020 12th Annual Undergraduate Research Conference on Applied Computing (URC)*. IEEE, pp. 1–6. DOI: 10.1109/URC49805.2020.9099190.

[2] Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. 2021. Provable security analysis of FIDO2. In *Annual International Cryptology Conference*. Springer, pp. 125–156. DOI: 10.1007/978-3-030-84252-9_5.

[3] Elaine Barker. 2020. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. National Institute of Standards and Technology, Gaithersburg, MD, pp. 32–41. DOI: 10.6028/NIST.SP.800-175Br1.

[4] John Bradley, Jeff Hodges, Michael B. Jones, Akshay Kumar, Rolf Lindemann, and Johan Verrept. 2021. Client to Authenticator Protocol (CTAP). (2021). Retrieved 01/06/2023 from https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html.

[5] Rodolphe Bréard. 2023. libreauth. (2023). Retrieved 04/26/2023 from https://github.com/breard-r/libreauth.

[6] DOCS:RS. 2023. Crate slauth. (2023). Retrieved 04/26/2023 from https://docs.rs/slauth/0.5.0/slauth/.

[7] DOCS.RS. 2023. Crate libreauth. (2023). Retrieved 04/26/2023 from https://docs.rs/libreauth/0.15.0/libreauth/.

[8] DOCS.RS. 2023. Crate webauthn_rs. (2023). Retrieved 03/25/2023 from https://docs.rs/webauthn-rs/0.4.8/webauthn_rs/.

[9] Nick Frymann, Daniel Gardham, Franziskus Kiefer, Emil Lundberg, Mark Manulis, and Dain Nilsson. 2020. Asynchronous Remote Key Generation: An Analysis of Yubico's Proposal for W3C WebAuthn. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '20*). ACM, Virtual Event, USA, pp. 939–954. DOI: 10.1145/3372297.3417292.

[10] Jeff Hodges, J. C. Jones, Michael B. Jones, Akshay Kumar, and Emil Lundberg. 2021. Web Authentication: An API for accessing Public Key Credentials Level 2. (2021). Retrieved 02/01/2023 from https://www.w3.org/TR/2021/REC-webauthn-2-20210408/.

[11] James Hodgkinson. 2023. webauthn-rs. (2023). Retrieved 04/26/2023 from https://github.com/kanidm/webauthn-rs.

[12] Emil Lundberg. 2023. Add privacy considerations about credential IDs. Issue #1250. (2023). Retrieved 03/31/2023 from https://github.com/w3c/webauthn/pull/1250.

[13] Emil Lundberg. 2023. Privacy risk from revealing allowed credentials. Issue #1246. (2023). Retrieved 03/31/2023 from https://github.com/w3c/webauthn/issues/1246.

[14] Emil Lundberg and Dain Nilsson. 2023. Webauthn recovery extension. (2023). Retrieved 04/26/2023 from https://github.com/Yubico/webauthn-recovery-extension/.

[15] Sanam Ghorbani Lyastani, Michael Schilling, Michaela Neumayr, Michael Backes, and Sven Bugiel. 2020. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *IEEE Symposium on Security and Privacy*, pp. 268–285. DOI: 10.1109/SP40000.2020.00047.

[16] Salah Machani, Rob Philpott, Sampath Srinivas, John Kemp, and Jeff Hodges. 2020. FIDO UAF Architectural Overview. (2020). Retrieved 01/05/2023 from https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-overview-v1.2-ps-20201020.html.

[17] Richard Markiewicz, Benoît Cortier, and Michael Proulx. 2023. slauth. (2023). Retrieved 04/26/2023 from https://github.com/devolutions/slauth.

[18] OWASP. 2023. Session Timeout. (2023). Retrieved 04/03/2023 from https://owasp.org/www-community/Session_Timeout.

[19] Olivier Pereira, Florentin Rochet, and Cyrille Wiedling. 2018. Formal Analysis of the FIDO 1.x Protocol. In *Foundations and Practice of Security*. Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquin Garcia-Alfaro, (Eds.) Springer, Cham, pp. 68–82. DOI: 10.1007/978-3-319-75650-9_5.

[20] Abhishek Sachdeva. 2013. A Study of Encryption Algorithms AES, DES and RSA for Security. *Global Journal of Computer Science and Technology*, 13, E15, 32–40. ISSN: 0975-4172. https://computerresearch.org/index.php/computer/article/view/272.

[21] Mark Smith. 2018. audit the Web Authentication API. Issue #26614. (2018). Retrieved 01/02/2023 from https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/26614.

[22] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, and Alexei Czeskis. 2017. Universal 2nd Factor (U2F) Overview. (2017). Retrieved 01/05/2023 from https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html.

[23] Athanasios Vasileios Grammatopoulos, Ilias Politis, and Christos Xenakis. 2021. A Web Tool for Analyzing FIDO2/WebAuthn Requests and Responses. In (*ARES 21*), Article 125. ACM, Vienna, Austria, 10 pages. DOI: 10.1145/3465481.3469209.

[24] Yuhan Yan. 2022. The Overview of Elliptic Curve Cryptography (ECC). *Journal of Physics: Conference Series*, 2386, 1, Article 012019. DOI: 10.1088/1742-6596/2386/1/012019.

[25] Yubico Developers. 2023. Enterprise Attestation. (2023). Retrieved 04/26/2023 from https://developers.yubico.com/WebAuthn/Concepts/Enterprise_Attestation/.

[26] Yubico Developers. 2023. User Presence vs User Verification. (2023). Retrieved 01/02/2023 from https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/User_Presence_vs_User_Verification.html.

[27] Yubico Developers. 2023. WebAuthn Client Authentication. (2023). Retrieved 01/02/2023 from https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Authentication.html.

[28] Yubico Developers. 2023. WebAuthn Client Registration. (2023). Retrieved 01/02/2023 from https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Registration.html.

[29] Yubico Developers. 2023. WebAuthn Deployment Best Practices. (2023). Retrieved 02/13/2023 from https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/Best_Practices.html.

Appendix A

Selected Definitions

```
#[derive(Serialize, Deserialize)]
pub struct Data {
  ...
  /// Used to store security keys for the user
  #[serde(default)]
  pub security_keys: SecurityKeys,
}
```

Listing A.1: Field for storing security keys

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct Credentials {
  /// Stores the credential for the registered security key
  #[serde(skip_serializing_if = ”Option::is_none”)]
  pub security_key_credential: Option<SecurityKey>,

  /// Stores the credential for the registered passwordless security key
  #[serde(skip_serializing_if = ”Option::is_none”)]
  pub passwordless_key_credential: Option<PasswordlessKey>,

  /// Stores the username
  #[serde(default)]
  pub username: String,

  /// Stores the name of the security key
  #[serde(default)]
  pub security_key_name: String,
}
```

Listing A.2: Struct for user credentials

```
#[derive(Serialize, Deserialize, Clone, Debug, Default)]
pub enum AuthenticationType {
  Password,
  Passwordless,
  #[default]
  NoType
}
```

Listing A.3: Enum for authentication type

```
pub(crate) const TIME_LIMIT: i64 = 20;
```

Listing A.4: Constant for time limit

```
pub(crate) const CHECK_PASSWORD: i64 = 300;
```

Listing A.5: Constant for time limit to check the password

```
#[derive(Serialize, Deserialize, Debug, Clone, Default)]
pub struct SecurityKeys {
    /// Stores the credentials for user authentication
    pub credentials: Vec<Credentials>,

    /// Saves the authentication type of the user
    #[serde(default)]
    pub authentication_type: AuthenticationType,
}
```

Listing A.6: Struct for security keys

```
#[derive(Debug, Clone, Default)]
pub struct WebauthnData {
    /// Saves the username during registration
    pub username: Option<String>,

    /// Saves the security key name during registration
    pub security_key_name: Option<String>,

    /// Used for adding a security key
    pub password_checked: Option<i64>,

    /// Used for security key registration
    pub registration_started: Option<i64>,

    /// Used for security key authentication
    pub authentication_started: Option<i64>,

    /// Saves the response of the Webauthn API during registration
    pub creation_challenge_response: Option<CreationChallengeResponse>,

    /// Saves the response of the Webauthn API during authentication
    pub authentication_challenge_response: Option<RequestChallengeResponse>,

    /// Saves the registration state during security key registration
    pub security_key_registration: Option<SecurityKeyRegistration>,

    /// Saves the authentication state during security key authentication
    pub security_key_authentication: Option<SecurityKeyAuthentication>,

    /// Saves the registration state during passwordless security key registration
    pub passwordless_key_registration: Option<PasswordlessKeyRegistration>,

    /// Saves the authentication state during passwordless security key authentication
    pub passwordless_key_authentication: Option<PasswordlessKeyAuthentication>,
}
```

Listing A.7: Struct for temporary values