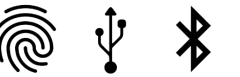
UNIVERSITY OF PIRAEUS

MASTER THESIS

FIDO2/WebAuthn implementation and analysis in terms of PSD2



Author:
Athanasios Vasileios GRAMMATOPOULOS

Supervisor:
Prof. Christos XENAKIS

A thesis submitted in fulfillment of the requirements for the postgraduate programme of Digital Systems Security

in the

Systems Security Laboratory
Department of Digital Systems

February 22, 2022

Declaration of Authorship

I, Athanasios Vasileios GRAMMATOPOULOS, declare that this thesis titled, "FIDO2/WebAuthn implementation and analysis in terms of PSD2" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

UNIVERSITY OF PIRAEUS

Abstract

Faculty Name
Department of Digital Systems

Digital Systems Security

FIDO2/WebAuthn implementation and analysis in terms of PSD2

by Athanasios Vasileios GRAMMATOPOULOS

FIDO is an alternative passoword-less authentication standard that can be used to replace traditional username and password authentication mechanics. FIDO leverage the use of public-private key cryptography in combination with the possession of personal authenticator devices (e.g. a laptop, a smartphone or a USB security key) to authenticate the user by requesting an additional verification through a biometric scan (e.g. a fingerprint scan) or a knowledge element (e.g. a PIN or an unlock Pattern). FIDO2 connects FIDO authenticators in the web environment, through the usage of the WebAuthn specification and thus making it ideal for providing strong client authentication (SCA) to meeting the requirements of Payment Services Directive (PSD2). In this work, we will look into how FIDO2/WebAuthn works, how FIDO can cover the SCA requirements and that issues may one face when doing so. Furthermore, FIDO2/WebAuthn solutions developed to enable the use of FIDO and ensuring strong user authentication in various application will be presented.

Το FIDO είναι ένα εναλλακτικό πρότυπο αυθεντικοποίησης χωρίς την χρήση κωδικών το οποίο μπορεί να χρησιμοποιηθεί για να αντικαταστήσει τις παραδοσιακές μεθόδους αυθεντικοποίησης οι οποίες βασίζονται στην χρήση username και password. Το FIDO χρησιμοποιεί κρυπτογραφία δημόσιου και ιδιωτικού κλειδιού σε συνδυασμό με την κατοχή προσωπικών συσκευών αυθεντικοποίησης (π.χ. ένα laptop, ένα έξυπνο κινητό ή ένα USB κλειδί ασφαλείας) για να αυθεντικοποιήσει τον χρήστη ζητώντας μια επιπλέων πιστοποίησή του μέσω βιομετρικών (π.χ. σάρωση δακτυλικού αποτυπώματος) ή κάποιο γνωστικό αντικείμενο (π.χ. ένα PIN ή ένα μοτίβο κλειδώματος). Το FIDO2 συνδέει τις FIDO συσκευές αυθεντικοποίησης στο περιβάλλον του διαδικτύου, μέσο της χρήσης της προδιαγραφής WebAuthn, και με αυτόν τον τρόπο το κάνει ιδανικό για την παροχή strong client authentication (SCA) για να καλύψει τις απαιτήσεις του Payment Services Directive (PSD2). Σε αυτήν την εργασία, κάνουμε μια ανασκόπηση στον τρόπο λειτουργείας των FIDO2/WebAuthn, στον τρόπο με τον οποίο το FIDO μπορεί να καλύψει τις απαιτήσεις του SCA αλλά και τι προβλήματα μπορεί να αντιμετωπίσει κάποιος κατά την χρήση του για τον λόγω αυτόν. Επιπλέον, παρουσιάζουμε FIDO2/WebAuthn λύσεις ανεπτυγμένες για την παροχή ισχυρής αυθεντικοποίησης μέσο FIDO σε εφαρμογές κάτω από διάφορα περιβάλλοντα.

Acknowledgements

I would like to thank my supervisor, professor Christos Xenakis, for the guidance and feedback he provided during my research. Furthermore, I would like to thank Ilias Politis as well as the rest of my colleagues at Systems Security Laboratory (SSL) for inspiring me, support me and providing me feedback. Lastly, I would like to thank my family and my friends for supporting me during my tight schedule (I owe you a couple of beers).

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

1 Introduction 1
1.1 Introduction . . . 1

2 FIDO2 / WebAuthn 3
2.1 FIDO2 in web applications through WebAuthn . . . 4
2.2 Authentication procedure . . . 4
2.3 Registration procedure . . . 6
2.4 Security Measures . . . 7
2.5 Types of authenticator devices . . . 9
2.6 Authenticator device attestation . . . 10

3 Payment Services Directive 2 11
3.1 The Directive (EU) 2015/2366 . . . 11
3.2 RTS on SCA Standards . . . 12
3.3 Opinion of the EBA on the elements of SCA . . . 13
3.3.1 Inherence element . . . 13
3.3.2 Possession element . . . 14
3.3.3 Knowledge element . . . 15
3.4 FIDO Alliance review of RTS for SCA . . . 15
3.5 Identified Issues and Problems . . . 19
3.5.1 Certification of Relying Party FIDO services . . . 19
3.5.2 Adoption and Compatibility . . . 20
3.5.3 Tolerance of Failed Authentications . . . 20
3.5.4 Selecting Trusted Authenticator Devices . . . 21
3.5.5 Availability Authenticator Devices . . . 21

4 Implementations 23
4.1 StrongMonkey . . . 23
4.1.1 Introduction . . . 23
4.1.2 Implementation . . . 24
4.1.3 Usage . . . 26
4.1.4 DEMO Application . . . 33
4.1.5 Conclusions . . . 35
4.2 StrongBee . . . 38
4.2.1 Introduction . . . 38
4.2.2 Implementation . . . 39
4.2.3 Usage . . . 40

4.2.4 Conclusions 46
4.3 FIDO2 authentication for OpenVPN 46
4.3.1 Introduction 47
4.3.2 Implementation 48
4.3.3 Usage 50
4.3.4 Conclusions 51

5 Conclusion **55**

A FIDO Metadata Filtering App **57**

B StrongMonkey Implementation Code **61**

C Configuring Keycloak for WebAuthn **75**

D VPN Implementation Code **85**

List of Figures

1.1 FIDO offers strong security and ease of use. 2

2.1 FIDO client authentication using challenge-response and public-private key cryptography. 3
2.2 The FIDO2 ecosystem and the technologies involved. 4
2.3 Step by step the FIDO2/WebAuthn authentication of a client. 5
2.4 Step by step the FIDO2/WebAuthn authentication of a client. 6
2.5 Android internal authenticator device asking user to touch fingerprint sensor in order to sign server challenge. 7
2.6 Windows 11 using Windows Hello internal authenticator device asking user to insert PIN in order to sign server challenge (registration process at the left, authentication process at the right). 8

3.1 PSD2 and FIDO terminology during authentication. 16

4.1 StrongMonkey, a PHP SDK for interacting with FIDO2 Server API v3.0.0. 23
4.2 Overview connection flow using StrongMonkey. 24
4.3 Detailed FIDO2/WebAuthn authentication flow using StrongMonkey. . 25
4.4 StrongMonkey main functions flow. 25
4.5 The user has initially to use the traditional username & password login. 33
4.6 User registers a Windows Hello authenticator device on his/her account. 34
4.7 User manage the security keys registered on the account listed under the Manage Keys page. 34
4.8 User uses WebAuthn sign in with the Windows Hello authenticator. . . 35
4.9 User use generates QR code to authenticate on PC using his/her smartphone's authenticator. 36
4.10 After scanning the QR code, user is asked to be authenticated in order to allow another device to sign into his/her account. 36
4.11 User was successfully authenticated on smartphone and his/her PC was signed in. 37
4.12 StrongBee, FIDO2/WebAuthn server in python. 38
4.13 Example usage of a VPN. 47
4.14 Traditional VPN authentication through username and password. . . . 48
4.15 Example usage of a VPN with FIDO2/WebAuthn. 48
4.16 VPN authentication through OIDC and FIDO2. 49
4.17 The user interface of the VPN client application. 50
4.18 The client launch a browser to authenticate the user through an OIDC service. 51
4.19 KeyCloak configured to authenticate the user using FIDO2/WebAuthn. 52
4.20 The user authenticates using the Windows Hello WebAuthn embedded authenticator using a PIN. 53
4.21 The user connected to the VPN service after successful authentication. 53

A.1 Found 11 certified FIDO2 authenticator devices protected by hardware mechanics with fingerprint detection capabilities. 58
A.2 Found 4 certified FIDO2 authenticator devices protected by hardware mechanics with face detection capabilities. 59

B.1 UML Class Diagram for the StrongMonkey PHP library 74

List of Tables

3.1 List of possible inherence elements provided by EBA. 14
3.2 List of possible possession elements provided by EBA. 15
3.3 List of possible knowledge elements provided by EBA. 16
3.4 Statistics of the authenticators under the FIDO Alliance metadata . . . 22

List of Abbreviations

2FA	**T**wo-**F**actor **A**uthentication
API	**A**pplication **P**rogramming **I**nterface
CTAP	**C**lient **t**o **A**uthenticator **P**rotocols
ECB	**E**uropean **C**entral **B**ank
FIDO	**F**ast **Id**entity **O**nline
HMAC	**H**ash-based **M**essage **A**uthentication **C**ode
HOTP	**H**MAC-based **O**One-**T**Time **P**Passwords
HTTP	**H**ypertext **T**ransfer **P**rotocol
HTTPS	**H**ypertext **T**ransfer **P**rotocol **S**ecure
IdM	**Id**entity **M**anagement
JWT	**J**SON **W**eb **T**oken
LGPL	GNU **L**esser **G**eneral **P**ublic **L**icense
MFA	**M**ulti-**F**actor **A**uthentication
NFC	**N**ear-**f**ield **c**ommunication
OIDC	**O**pen**ID** **C**onnect
OS	**O**perating **S**ystem
PoC	**P**oof-**o**f **C**oncept
PSD2	**P**ayment-**S**ervices **D**irective **2**
PSP	**P**ayment-**S**ervices **P**roviders
PSU	**P**ayment-**S**ervices **U**ser
QR	**Q**uick **R**esponse (code)
RP	**R**elying **P**arty
RTS	**R**egulatory **T**echnical **S**tandards
SCA	**S**trong **C**ustomer **A**uthentication
SDK	**S**oftware **D**evelopment **K**it
SMS	**S**hort **M**essage **S**ervice
SSO	**S**ingle **S**ign **O**n
TEE	**T**rusted **E**xecution **E**nvironment
TOTP	**T**ime-based **O**One-**T**Time **P**Passwords
USB	**U**niversal **S**erial **B**us
VPN	**V**irtual **P**rivate **N**etwork
WSGI	**W**eb **S**erver **G**ateway **I**nterface

Chapter 1

Introduction

1.1 Introduction

Authentication is the corner stone of all our digital services. Most of our systems are services rely in some for of authentication in order to control the access rights of each user into the system. The traditional mechanic for authenticating users is the username and password, though this is not the only one in existence.

> *"Authentication is a way to ascertain that a user is who they claim to be."*
>
> *—ENISA*

Based on their characteristics, authentication method can be divided into 3 main categories:

- Known secrets
  - Secrets that the user knows by hart
  - e.g. password, PIN
- Possession of secrets
  - Usually authenticators or big hashes (too big to remember)
  - e.g. serial number, unique token
- Unique characteristics
  - biometrics for humans or unique attributes for machines
  - e.g. fingerprint, IP address

During authentication, a system may perform more than one authentication method (multi factor authentication) in order to ensure the user's identity. The most popular mechanic is the use of secrets.

Nowadays, passwords are becoming a problem, as their use in many cases result in various issues, in many cases sourced from the way users handle them:

- **Lack of complexity & relative small size** - If not enforced by the system, user tend to use relative small passwords with a small complexity.
- **Repeated use of the same passwords** - In most cases since users have multiple accounts from various services, they reuse the same passwords.
- **Easily stolen through phishing attacks** - Phishing attacks can exploit relative easily the human factor and steal a user's credentials.

- **Insecurely stored as plaintext** - Many services don't follow the best security practices and store the password as it is inside their database, thus many passwords are exposed due to data breaches.

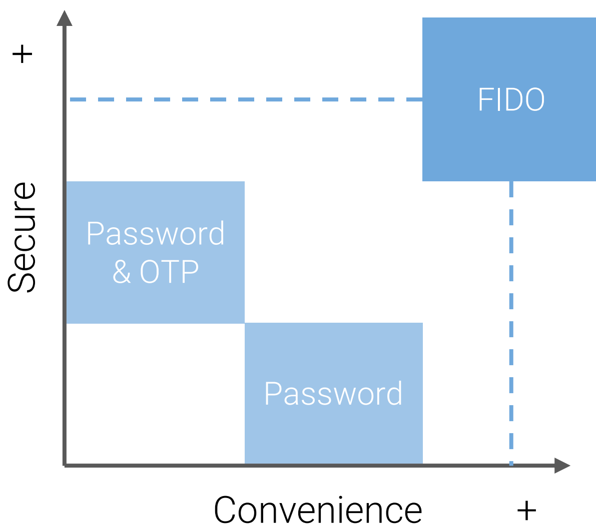


FIGURE 1.1: FIDO offers strong security and ease of use.

The FIDO is standardizing the authentication process, ensuring its security and hiding all the security complexity from the end user while improving his/her experience! FIDO is now supported by many modern devices. Nowadays, all smartphone devices can be used as a FIDO authenticator (i.e. Android, iPhone), all modern browsers support the WebAuthn specification (e.g. Chromium, Google Chrome, Firefox, Edge, Safari) and both Windows and Mac-OS feature platform authenticator solutions. Thus, even if one haven't bought a security USB token, still he/she can start using FIDO2/WebAuthn now.

A number of website adopted FIDO2/WebAuthn and their users can already start using them. The majority of the services introduce it as an additional second factor authentication (2FA) mechanic along with SMS and authenticator applications (usign TOTP and HOTP through QR codes). We have to note thought that FIDO is phishing resistant by design while the other mechanics are not.

Chapter 2

FIDO2 / WebAuthn

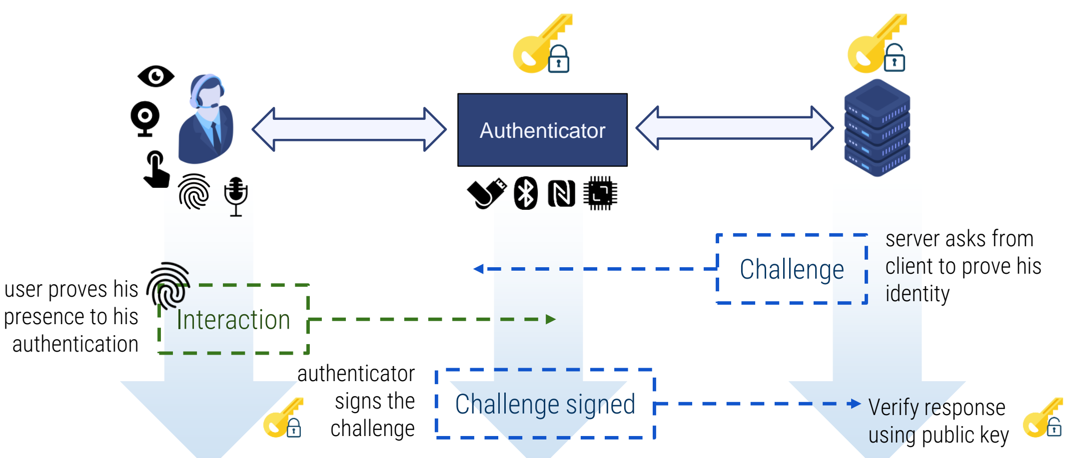


FIGURE 2.1: FIDO client authentication using challenge-response and public-private key cryptography.

FIDO defines a challenge-response scheme based on public key cryptography as depicted in Figure 2.1. The relying party (RP) server prepares a challenge in the form of a random value and forwards it to the client. The user, to prove its identity, must sign the challenge with a private key (through his/her authenticator device) and send the forwarded signature back to the server. The server will then need to verify the authenticity of the given signature using the public key of the authenticator device linked to the account the client is claiming to own. As one may notice, before the execution of the mentioned challenge response scheme, the server needs to have the authenticator device's public key. This simple conception ensures the security of the scheme, the compatibility, and ease of use of FIDO.

As shown in Figure 2.2, FIDO's ecosystem exapands from hardware devices (secure authenticator devices) to online services (FIDO servers and web applications), embedding a wide range of technologies. Looking at the user's side, FIDO's Client to Authenticator Protocols (CTAP), both the older CTAP1 (also known as U2F) and the new CTAP2, define how devices can communicate with FIDO compatible authenticators. FIDO Universal Authentication Framework (UAF) describes how a FIDO UAF server should communicate with client devices (such as a smartphone with a biometric sensor, such as a fingerprint sensor) to offer password-less authentication using only the user's biometrics. The more recent FIDO2, improves upon the older Universal 2nd Factor (U2F) authentication, and brings FIDO to the web environment and web services, through the WebAuthn specification, the javascript API and server side WebAuthn libraries or servers.

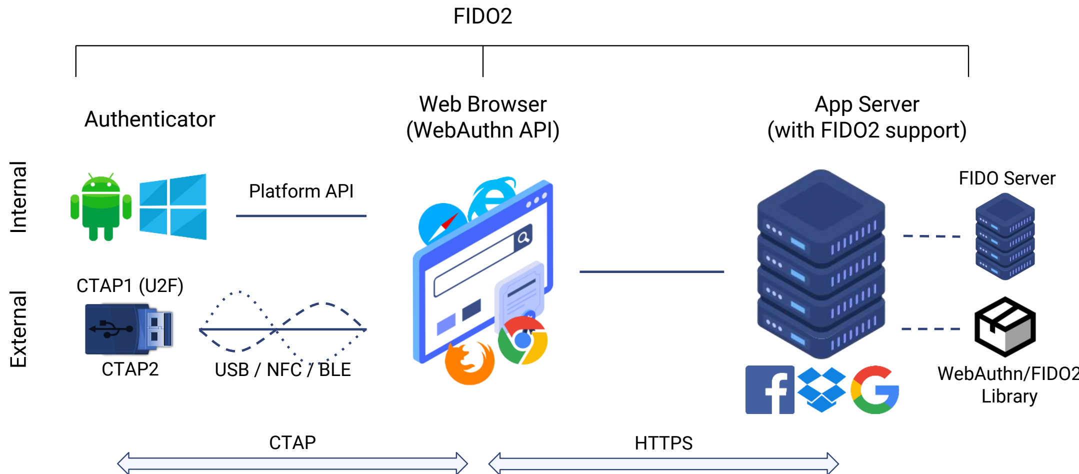


FIGURE 2.2: The FIDO2 ecosystem and the technologies involved.

2.1 FIDO2 in web applications through WebAuthn

Using the WebAuthn specification's javascript API, web applications can request from the client's browser (WebAuthn client) and the underlying operating system (OS) credentials creation (i.e., public key pair generation), as well as credentials retrieval (i.e., proof of secret key possession). The credentials creation method, which is accessible through the javascript `window.navigator.credentials.create`[1] method and the public key options, defined at the WebAuthn specifications[2], allow the creation of asymmetric cryptography keys (e.g. ECDSA key-pairs). These keys are bind to the caller web application's domain (RP id) and a user identifier (user handle) is linking the credentials with an account. Moreover, through the corresponding credential get method, which is accessible through the javascript `window.navigator.credentials.get`[3] and its public key options, the web applications can verify the client's possession of previously created credentials (key-pairs) by requesting the generation of a random challenge's signature. Thus, the identity of a user can be verified through a challenge-response scheme, as illustrated in Figure 2.3.

2.2 Authentication procedure

A typical use case of FIDO2, using the previously mentioned javascript methods, is an online password-less authentication. That is, the secure login of a user into a website without the use of a secret password. Figure 2.4 presents the authentication process as a diagram. To start the process, the user loads the website through a WebAuthn compatible browser (all major browsers are currently supporting the WebAuthen specification) and selects to login password-less. The website's back-end generates a cryptographically secure and high entropy random challenge (typically 128 bits or more, as suggested by the specification) and communicates it with the client device (the user's browser). The client is then able to invoke the WebAuthn `window.navigator.credentials.get` javascript method to request

[1] https://developer.mozilla.org/en-US/docs/Web/API/CredentialsContainer/create
[2] https://w3c.github.io/webauthn/
[3] https://developer.mozilla.org/en-US/docs/Web/API/CredentialsContainer/get

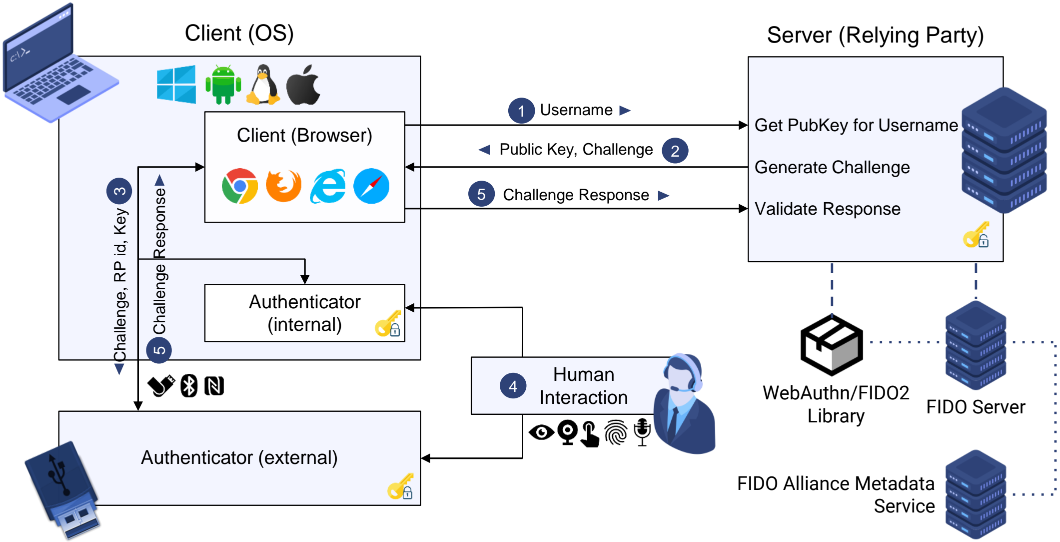


FIGURE 2.3: Step by step the FIDO2/WebAuthn authentication of a client.

from an authenticator device to sign the challenge. The browser then sends the challenge along with the website's domain name (RP id) and some more data (list of accepted credentials, list of excluded credentials, extensions etc.) to the available authenticator devices for signing, assuming that on of those authenticators possesses a key pair for the website in question (or one of the key pairs requested). Depending on the system and the support, the browser may contact the authenticator devices directly through the CTAP protocol [4] or call custom methods of OS specific WebAuthn Client implementations (such as Android's FIDO2 API [5]). Figure 2.5 shows how Android allows leverage's the internal platform authenticator to sign challenges and prove the user's identity using biometrics (usually a fingerprint sensor), while Figure 2.6 shows how Window Hello (on Windows 11) can sign challenges by asking a PIN form the user to verify the action.

Eventually, the browser gets a response that then parses and forwards to the corresponding javascript handler. The response includes the identifier of the key used for generating the signature (credentials id), the actual signature, the data structure used to generate the signature as reported by the authenticator, a user identifier (user handle), a signature counter and several flags. Then the website's front-end forwards the response data to the RP's back-end (the web application server) for verification. Upon successful verification, the user is logged in and his session (usually implemented using cookies) is updated.

The described use case assumes that the authenticator supports resident keys (i.e., discoverable credentials) and can report back the user identifier (user handle). To support older U2F authenticators, or to avoid storing information on the authenticator device (leveraging key wrapping techniques), web pages may store previously used account identifiers as cookies or at the local storage of the browser and include them at the initial challenge creation request. This way the web pages can provide the account's identifier to the server enabling the latter to return along with the challenge a list of credential IDs registered for this account.

[4]https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html

[5]https://developers.google.com/identity/fido/android/native-apps

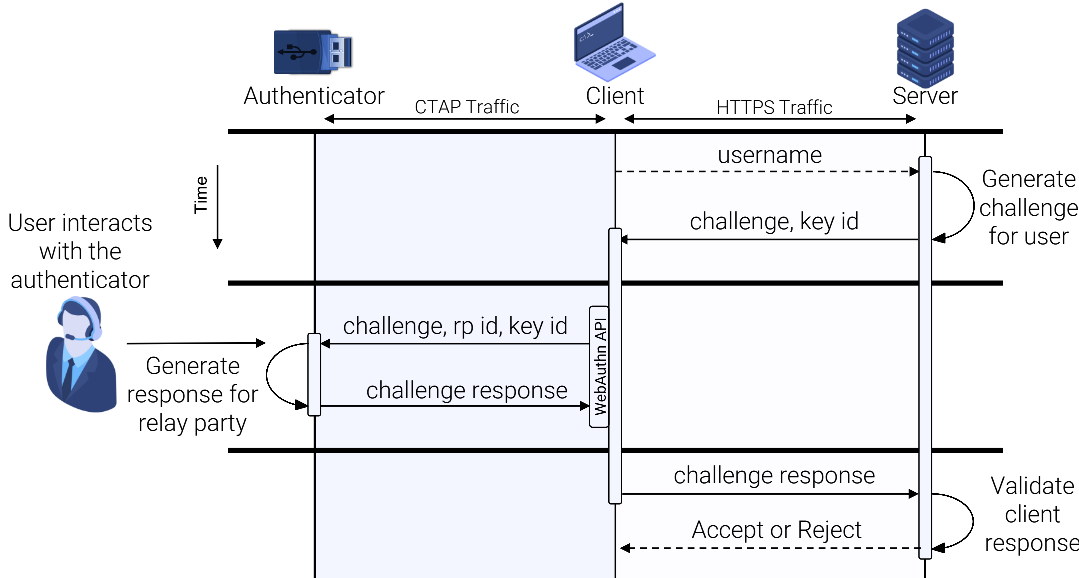


FIGURE 2.4: Step by step the FIDO2/WebAuthn authentication of a client.

This list of credentials could then be added on the invocation of the WebAuthn `window.navigator.credentials.get`. A similar process can be used for second factor authentication flows, as the user's identifier should already be known through the active session. This later method could also be used when there is a need to re-authenticate an already authenticated user, commonly used to renew sessions of returning users,by requesting a fingerprint or PIN authentication when relaunching a mobile application (usually found on banking related applications).

A true password-less authentication without the need to provide any username or password requires information to be stored on the authenticator device, which may increase the cost of the authenticator, limit the maximum number of keys that can be generated and need key management utilities to be able to remove stored keys not needed any more. On the other hand, the password-less authentication, which requires the knowledge of an account's identifier (e.g., a username) does not share such limitations, since the information can be wrapped securely and stored at the RP's server, through server-side credential storage modality.

2.3 Registration procedure

Prior to FIDO2's authentication process, an authenticator device will have to be registered with the RP server to the user's account. During the registration process, the client's authenticator device generates a public-private key pair (based on the supported algorithms by the RP server) and forward the public key along with a credentials identifier value to the server. The server will save the credentials and link them with the client's account. Hence, in order to register an authenticator device, the client should already be signed into the service, so that a session linked to an account would already be set up and the user account is know. As an extra security measure, the RP service may ask to re-authenticate the user to ensure the action is not performed by an unauthorised user. The registration process starts with a request for credentials creation by the client, asking the RP server to

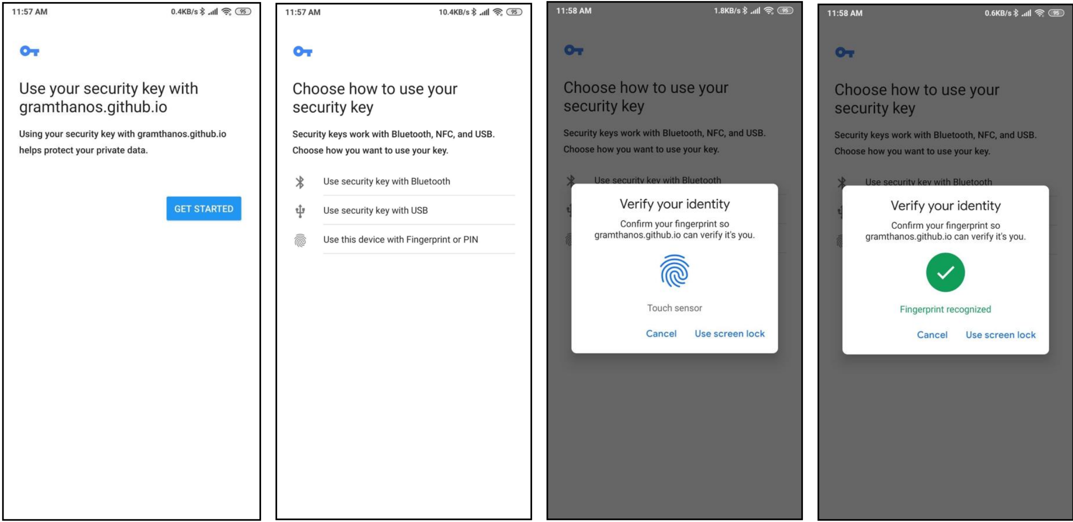


FIGURE 2.5: Android internal authenticator device asking user to touch fingerprint sensor in order to sign server challenge.

generate a challenge for this action. The server will return a cryptographically secure random challenge and a user handle, linked to the user's account, a list of supported credentials algorithms (e.g., ECDSA, RSASSA-PSS and/or RSASSA-PKCS1-v1.5), optionally a number of filtering criteria for the authenticator devices (e.g., external authenticators, platform authenticators), optionally a list of identifiers for already registered credentials (so that they can be excluded from the registration) as well as the server's preference for the authenticator's attestation response (asking the authenticator device to prove its identity). These parameters are used as options when calling the `window.navigator.credentials.create` javascript method to request the credentials creation from the authenicator device (as defined by the WebAuthn specification). After the successfully handling of the credentials creation procedure by the authenticator device, the browser will return the created credentials to the appropriate javascript handler as defined by the method caller. The response will include the generated credentials identifier, the generated public key, the challenge generated by the server and optional attestation information for the authenticator device (e.g., a device certificate). The received data will then be forwarded to the RP server. The server will then have to validate the provided information and then store at least the credentials identifier and the public key under the account the challenge was generated.

2.4 Security Measures

The main feature of FIDO2/WebAuthn, that makes it standout from other authentication methods, is its resistant to phishing and man-in-the-middle attacks. By design WebAuthn clients do not allow cross domain credentials access unless, they are created under the request came from a the target RP service. Thus fraudulent websites or applications are not able to request authentication for another RP website. This is achieved through the client's (your web browser) validation of the RP id of the request, which should be the domain name of the website. Authenticator devices link the credentials with the RP id and the user account (through the user handle) thus they will not return credentials sourced from other RP.

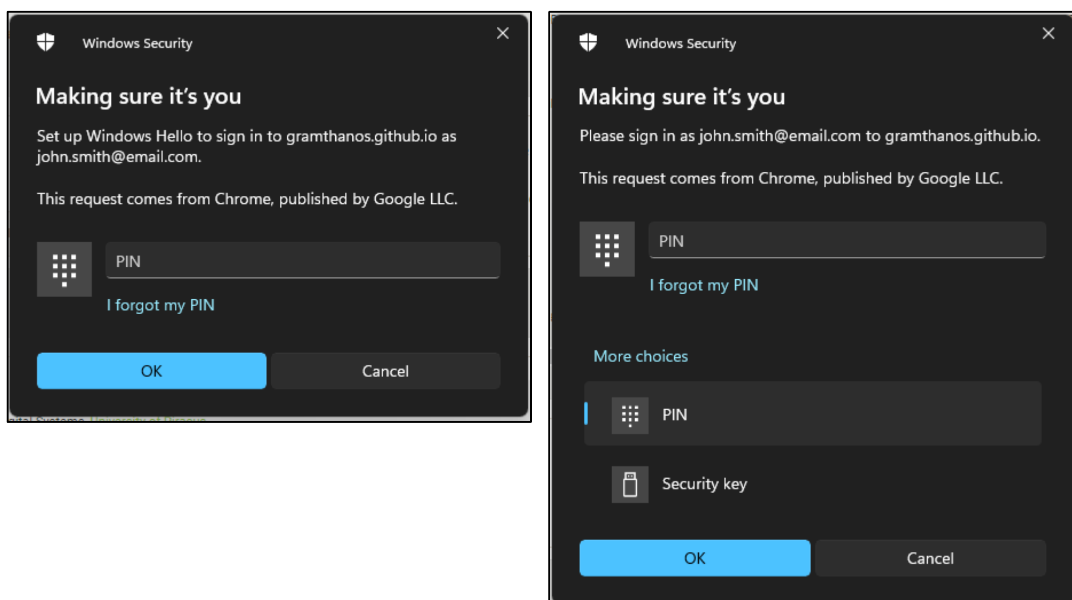


FIGURE 2.6: Windows 11 using Windows Hello internal authenticator device asking user to insert PIN in order to sign server challenge (registration process at the left, authentication process at the right).

To ensure the integrity and the confidentiality of the processes described above, web browsers expose the WebAuthn API only under secure context (web pages loaded under HTTPS), with the exception of "localhost" origins that are commonly used for development purposes. By requiring HTTPS, the browser can assure the authenticity of the server (by validating the server's certificate) and thus mitigate man-in-the-middle attacks at the network traffic level. In simple terms, to secure the schema, FIDO builds a trusted communication channel between the RP and authenticator device. We also have to note that Google defined an additional way to link Android applications with a domain name in order to authorize applications claiming to be linked to such domains. This is done through the `/.well-known/assetlinks.json`[6] and also be used to validate that an application should be given access to credentials storage for credentials under the claimed RP id.

Form the RP's side, in order to validate the correct execution of the authentication process, the server has to verify a number of information returned on the response apart from just the signature. In particular, the server has to:

- Check that the challenge returned match the one generated.
- Check if the origin of the response is the expected origins.
- Check if the Relay Party ID is the correct one.
- Check that the credentials identifier returned is already registered to the user account linked to the user handle return. (For password-less authentications)
- Check that the credentials identifier returned is already registered to the user account used to generate the challenge. (For second factor authentications)

[6] https://developers.google.com/digital-asset-links/v1/getting-started

- Ensure not to accept expired challenges, since the challenges are generated for use within a limited time frame.
- Save for each action save the signature counter for each credentials and ensure that for every authentication procedure this number increases, in order to protect users against cloned authenticator devices.
- Check that the all the flags returned by the authenticator are as expected (e.g. check user verification flag).
- Check if the algorithm of the credentials generated is among the supported signature algorithms provided by the server.
- Check the authenticator's attestation data
  - Ensure that the authenticator's attestation is valid (e.g. by checking the validity of the returned certificate).
  - Ensure the authenticator's AAGUID is inside the allow-list or is not inside the block-list (if the RP has such a list of authenticators).
  - Ensure that the authenticator is following the RP policy by checking the FIDO metadata service (e.g. check if the authenticator is certified).
  - Ensure that the authenticator is following the RP policy by checking the FIDO metadata service (e.g. check if the authenticator is certified, check if the authenticator features user identification through biometrics).
  - Ensure that the authenticator is not revoked or outdated by checking the FIDO metadata service.

As mentioned on the list above, the RP may request the authenticator's attestation data in order to assess the authenticator device information provided by the FIDO metadata service. For instance, a RP server may try to identify whether an authenticator device is among the devices approved by the service. The RP could also save the authenticator device's id so that it could check for revoked authenticators from time to time.

2.5 Types of authenticator devices

FIDO authenticators can be categorized based on their type into two categories, platform authenticators and cross-platform authenticators. Cross platform authenticators are external devices that connect with the system through USB, NFC, or Bluetooth (e.g., USB Keys or NFC Keys) and communicate through FIDO's CTAP1 and CTAP2 protocols. On the other hand, platform authenticators are embedded into the system (e.g., Android internal authenticator, Windows Hello authenticator) and may communicate with applications directly through the underlined system's calls and libraries (e.g., Microsoft WebAuthN Win32 headers [7]). Independent of the type of an authenticator, the device should be able to protect the private keys so that they cannot be extracted by an adversary that may have physical access to it.

Another practical characteristic that we can use to categorize the authenticator devices is the available methods they support to verify the user presence. Although the authenticator itself is a way the user to prove possession of the authenticator

[7] https://github.com/microsoft/webauthn

it self, in many cases the authenticator will have to verify the user's identity first before executing an FIDO/WebAuthn operation. Many authenticators (usually mobile or laptop devices) leverage access to bio-metrics sensors (e.g., fingerprint, face recognition, iris scan) to securely verify the user. Other simpler (usually USB cross-platform) authenticator devices features just a button, which the user press to verify its presence. To mitigate the risk of unauthorized use of such a FIDO2 device, an operation system may also ask the user for a PIN to authenticate him.

2.6 Authenticator device attestation

The WebAuthn requirement of a secure connection (through HTTPS) not only protects the information exchanged between the client and the server but also verifies the authenticity of the server (managed by the RP), through the trusted certificate issued to the domain name used by the service. In a similar way, depending on the application needs of a WebAuthn deployment, the RP may want to verify that the client's authenticator device is compliant with its policies. For example, the RP may have to verify before registering a new authenticator device, that this new device has the appropriate security level required by the service's security policy. To achieve this, the RP server may request additional attestation information from the authenticator device, during the registration phase, and assess them before finalising the registration process. The returned attestation statement would ideally prove the original identity of the authenticator device or verify the trustworthiness of the device. Depending on the attestation conveyance method, the authenticator may return its Authenticator Attestation GUID (AAGUID), exposing the authenticator's maker and model, as well as provide a way to verify its authenticity (e.g., by providing a certificate). Furthermore, using an authenticator's AAGUID, relying parties may query the FIDO Alliance Metadata Service (MDS) [8], to get more information about the authenticator device (e.g., authenticator security level, available user verification methods and combinations) and verify any attestation certificate returned.

Nevertheless, such an attestation of the authenticator device may expose too much information (e.g., Authenticator Model and Number), which may be used to track a user between multiple services. For this reason, an attestation conveyance preference can be defined, stating the RP's preference to, no attestation ("none"), anonymised attestation through a CA ("indirect"), or authenticator generated attestation ("direct" or "enterprise"). Thus, authenticator devices respond with different attestation responses based on the requested preference and their supported attestations or ignore the suggested attestation opting for user privacy. The RP from its side, may have to reject the authenticator registration if the needed attestation statement returned is not supported or the given or retrieved information does not satisfy its policies (e.g., due to failure of verifying any given certificate).

To allow the extension of the available attestation information, plug-able Attestation Statement Formats are supported by WebAuthn. Due to the nature of this scheme, the implementation of a RP may not support all of the attestation statement formats. The latest WebAuthn standard [9] describes the following attestation statement formats: None, Packed, TPM, Android Key, Android SafetyNet, FIDO U2F and Apple Anonymous. The corresponding Attestation Statement Format Identifier values are listed and maintained in the appropriate registry by IANA [10].

[8] FIDO Alliance Metadata Service, https://fidoalliance.org/metadata/

[9] https://www.w3.org/TR/webauthn-2/

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

Chapter 3

Payment Services Directive 2

This chapter will analyse the revised Payment Services Directive 2 (PSD2). Since we are looking into the directive from the FIDO2/WebAuthn perspective, we will focus on the sections related to the authentication of the user and how FIDO2 through WebAuthn can cover the need for Strong Customer Authentication (SCA).

3.1 The Directive (EU) 2015/2366

On November 25 of 2015, the Revised Payment Services Directive[1], also known as PSD2, of European Parliament and of the Council, was released. The directive came into force on January 16 of 2016 while its rules will take affect on January 13 of 2018, giving enough time to the market to adapt. The revised directive replaced the PSD1 [2] and amended 3 Directives[3] and 1 Regulation[4].

The PSD2 aims to regulate the payment transactions, focusing, among others, into ensuring the protection of customers when using online payment services. Many articles of the directive sets a number of requirements directly focusing on strengthening the authentication of the users. In particular, as defined in Article 97 (named "Authentication"), a strong customer authentication (SCA) may be required to authenticate when a) the user accesses an online account, b) initiates an electronic payment transaction or c) carries out an action online which may have a risk of payment fraud.

Specifically, the strong customer authentication (SCA) is defined by the PSD2 as:

> *"authentication based on the use of two or more elements categorised as knowledge (something only the user knows), possession (something only the user possesses) and inherence (something the user is) that are independent, in that the breach of one does not compromise the reliability of the others, and is designed in such a way as to protect the confidentiality of the authentication data"*

Based on the given definition, the directive defines strong customer authentication as an authentication mechanic which relies on at-least two elements. Those two authentication elements can be chosen from the following 3 categories of elements:

[1] Payment services (PSD2) - Directive (EU) 2015/2366, https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en

[2] Payment services (PSD 1) - Directive 2007/64/EC https://ec.europa.eu/info/law/payment-services-psd-1-directive-2007-64-ec_en

[3] Directives 2002/65/EC, 2009/110/EC and 2013/36/EU were amended by PSD2.

[4] Regulation (EU) No 1093/2010

- *knowledge, something only the user knows* —A piece of secret information that the users has to provide. Usually a password, a PIN or an answer to a security question.
- *possession, something only the user possesses* —A proof of possession of an items that the user has to provide. Usually a Smart Device like a smartphone, a Smart Card or a USB Security Key.
- *inherence, something the user is* —A biometric proof that can physically identify the user as an entity. Usually a fingerprint, face recognition or voice recognition.

We have to note here that the directive do not provide examples for each category of authentication elements. The above given examples are our own interpretation and were included for clarity.

The directive also defines other security-related requirements regarding monitoring and logging the appropriate transaction related actions and protecting transaction related data. Furthermore, transaction risk estimation functionalities to assess transitions are defined so that additional measures can be taken for high risk transactions. Lastly, the directive sets penalties for (article 97) for payment services that failed to implement the directive.

To facilitate the smooth implementation of the directive, the directive itself took advantage of the expertise and capabilities of the European Central Bank (ECB) and tasked them to release guidelines and draft regulatory technical standards (RTS) regarding the security of payment services, focusing particularly on the strong customer authentication.

3.2 RTS on SCA Standards

The latest version of ECB's Regulatory Technical Standards (RTS) on Strong Customer Authentication (SCA) and common and secure communication (CSC) [5], at the time of writing this thesis, was released on 23 February 2017. The RTS sets a number of general provisions, security measures, recommendations and analyses the exemptions from SCA.

The RTS were developed with 5 objectives in mind, defined by the PSD2:

- setting the appropriate level of security
- protecting the payment service user's (PSU's) funds and personal data
- ensuring fair competition between payment service providers (PSP)
- ensuring the neutrality of the technology and the business-model
- allowing the creation of new, inverted, user-friendly and accessible payment systems

The RTS try to set general requirements focusing on the security and avoid enforcing unnecessary requirements that may limit the development possibilities for

[5] Regulatory Technical Standards on strong customer authentication and secure communication under PSD2, https://www.eba.europa.eu/regulation-and-policy/payment-services-and-electronic-money/regulatory-technical-standards-on-strong-customer-authentication-and-secure-communication-under-psd2

payment services. For this reason, the standards do not set specific technological requirements, thus technologies such as FIDO are not mentioned.

The main SCA security measures requirements of the RTS are:

- *General authentication requirements* (article 2) —General requirements mostly focusing on the necessity of transaction monitoring mechanics, their analysis criteria of the transaction and their risk based approach they should follow.
- *Requirement for reviewing the security measures* (article 3) —Requirements over the auditing of the security measures, the frequency and the scope of the audit as well as the availability of the results in case they are requested by an authority.
- *Authentication code* requirements (article 4) —Describes the requirements for the secure generation of a unique authentication code per transaction upon successful authentication. Furthermore it defines the needed security measures to protect the services that generate the authentication codes.
- *Dynamic linking* requirements (article 5) —Defines the requirements to inform the user of the transaction amount bind to the authentication code. Ensuring the appropriate consent is given by the user and setting the appropriate measures to ensure confidentiality, authenticity and integrity of the transaction amount and information displayed to the user.
- *Requirements of the elements* (article 6, 7, 8 and 9) —Describing the requirements for each type of authentication element (knowledge, possession, devices and software linked to inherence elements) as well as requirements independent to the authentication element.

3.3 Opinion of the EBA on the elements of SCA

Since the RTS are technology independent, questions has been raised during the development, implementation, and adaption of the existing services to meet compliance with the new directive. The EBA released on an opinion on the SCA elements under PSD2[6] in order to share their view on the RTS and clarify which authentication approaches are compliant with the SCA requirements and make comments on the authentication elements.

3.3.1 Inherence element

Regarding inherence elements, EBA's view is that they are "biological and behavioural biometrics, relates to physical properties of body parts, physiological characteristics and behavioural processes created by the body, and any combination of these". Examples of inherence elements provided by the EBA include:

- retina scanning
- iris scanning
- fingerprint scanning

[6] https://www.eba.europa.eu/eba-publishes-an-opinion-on-the-elements-of-strong-customer-authentication-und

- vein recognition
- face geometry
- hand geometry
- voice recognition
- keystroke dynamics
- the angle the user holds the device
- the user's heart rate

Whether an inherence-based authenticator is compliant or not with the SCA is determined by it implementation, as the element should have a very low probability to falsely authenticate another user.

EBA provided an indicative list of compliant and non compliant inherence elements as an example past of which is shown in Table 3.1.

TABLE 3.1: List of possible inherence elements provided by EBA.

Element	is SCA compliant?
Fingerprint scanning	yes
Voice recognition	yes
Vein recognition	yes
Hand geometry	yes
Face geometry	yes
Retina scanning	yes
Iris scanning	yes
Keystroke dynamics	yes
Heart rate	yes
Memorised swiping path	no

3.3.2 Possession element

Regarding possession elements, EBA's view is that they don't have to be only physical possession but also possession of something that is not physical such as an application. In order to use a device as a possession element if a reliable method could be used to confirm the possession and provide evidence. As described by EBA, such evidence could be an OTP generated by the possession element device, a text message or a push notification. Regarding possession elements with evidence based on keys, in order to be conformant, there should be a binding process ensuring that only this device is linked to the evidence. Furthermore, digital signatures, such as a QR code of a card, could also be used as a possession element. Note that other information printed on cards, such as security codes, are not acceptable evidence of a possession element.

Once again, EBA provided an indicative list of compliant and non compliant possession elements as an example past of which is shown in Table 3.2.

TABLE 3.2: List of possible possession elements provided by EBA.

Element	is SCA compliant?
Possession of SIM card (SMS OTP as evidence)	yes
Hardware or software token generator (OTP as evidence)	yes
Hardware or software token (signature as evidence)	yes
Card with QR code (scan QR as evidence)	yes
App using security chip embedded into a device	yes
Card (evidence through card reader)	yes
Card (evidenced by dynamic security code)	yes
App installed on the device	no
Card (evidenced by printed card details)	no
Card (evidenced by printed OTP)	no

3.3.3 Knowledge element

Regarding knowledge elements, EBA's points out the need to have mitigation measures in place to block access to unauthorised parties as a result of the risk of disclosing those knowledge elements to 3rd parties. Once again, EBA provided an example list of knowledge elements:

- a password
- a PIN
- knowledge based response to challenges
- knowledge based response to questions
- a passphrase
- a memorised swiping path

Information printed on cards should not be considered a knowledge element. On the other hand, security codes delivered to the user separately form the card, can be used as knowledge elements. Furthermore, based on EBA's opinion, a user ID, such as a username or an email, is not to be considered as knowledge element. Tokens such as OTPs can not be considered knowledge are they were not existed before the authentication process began.

Yet again, EBA provided an indicative list of compliant and non compliant knowledge elements as an example past of which is shown in Table 3.3.

3.4 FIDO Alliance review of RTS for SCA

As a result of the PSD2, on December 20 of 20218, FIDO Alliance released a document providing a detained review of the Regulatory Technical Standards (RTS) for Strong Customer Authentication (SCA)[7]. In this document FIDO Alliance describes how the FIDO standard can cover these requirements and be compliant with the directive.

As shown in the Figure 3.1, FIDO can be used as as all types of authentication elements. FIDO authenticators, depending on their supporting mechanics and their

[7] https://fidoalliance.org/how_fido_meets_the_rts_requirements/

TABLE 3.3: List of possible knowledge elements provided by EBA.

Element	is SCA compliant?
Password	yes
PIN	yes
Knowledge based challenge questions	yes
Passphrase	yes
Memorised swing path	yes
Username	no
Email address	no
Card details printed on card	no
OTP generated	no
Printed list of OTPs	no

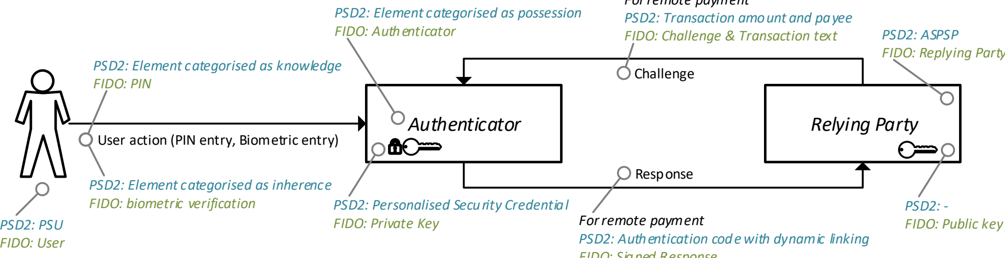


FIGURE 3.1: PSD2 and FIDO terminology during authentication.

implementation's security can be used as authentication elements as shown in the following list:

- inherence - Authenticator with biometric authentication capabilities.
- possession - The possession of the authenticator device itself.
- knowledge - Authenticator requesting a PIN from the user.

Apart from linking the terminology and presenting the basics of FIDO authenticators, the document also describes how FIDO covers each of the requirements set by the RTS. The following paragraphs will list in simplified terms these requirements and how FIDO covers them based on FIDO Alliance's analysis.

- **Requirement to document, periodically test, evaluate and audit the implementation** (Article 3.1) - FIDO Alliance offers a FIDO security certification program[8] through which the security level of each authenticator could be assessed.
- **Requirement to authenticate based on two or more elements** (Article 4.1) - FIDO authentication by default acts as a possession element authentication. For the second element, the authentication may use a password send to the server (when used as second factor authentication) or through an inherence (biometric) or knowledge (PIN) authentication locally verified by the authenticator device.

[8]https://fidoalliance.org/certification/

- **Ensuring no information for the element can leak from the authenticator code and a valid authenticator code can not be regenerated or forged** (Article 4.2) - By design in FIDO authentication, since it uses public-private key signatures for the signing of the challenges, it is infeasible to leak information for the private key.

- **Ensuring that one can not identify the incorrect element after an authentication failed, the number of consecutive failed authentication attempts should not exceed 5 and the communication session of the authentication data is protected** (Article 4.3) - In FIDO, since the authenticator device acts as a possession element and optionally in combination with another authentication element, one can not identify the element that failed the authentication. Furthermore, to ensure the limits for the consecutive failed authentication attempts one has to check the authenticator metadata in the FIDO Metadata Service. In addition, FIDO requires the communication with the relying party to be protected with TLS.

- **Requirement to make the user aware of the transaction amount, on which the authenticator code should be based on and upon a change on the amount the code should be invalidated** (Article 5.1) - FIDO supports these requirements in two ways. The message signed by the authenticator device may also include the transaction amount, payee ID and other related data. Alternatively, the FIDO also supports the "transaction confirmation" mechanic through which, depending on the authenticator support, a message can be displayed to the user for approval.

- **Ensuring the confidentiality, authenticity and integrity for the amount of the transaction and the information displayed to the payer during authentication** (Article 5.2) - The server may bind any transaction details to the server generated challenge and validate them after receiving the authenticator response. If a transaction confirmation message is given the authenticator may (depending on the support) display it to the user for confirmation.

- **Requirement to mitigate the risk of uncovering a knowledge element to an unauthorised party as well as ensuring that inherence elements has a very low probability of false positive identification** (Article 6/7/8) - Any knowledge code or inherence factor are safely stored internally in the authenticator during registration and verified again by the authenticator thus can not be leaked. Criteria like "False Acceptance Rate", "False Rejection Rate" and "Presentation Attack Detection" of an inherence authenticator are assessed by the FIDO Biometric Certification[9].

- **Ensuring that the breach of one authentication element can not compromise the other elements** (Article 9) - Since FIDO authenticator devices are by definition possession elements, we have two cases. Even if an adversary steals an authenticator device this does not compromise the PIN or biometric authentication needed. On the other hand, if an adversary steals an authenticator's PIN, he still needs access to also steal the authenticator device.

- **Requirement for multi purpose devices where any strong customer authentication element is used to be used with mitigation measures**

[9] https://fidoalliance.org/certification/biometric-component-certification/

including separated secure execution environments, security mechanisms to ensure that the software or device is not altered and if altered, measures to mitigate consequences (Article 9.2 and 9.3) - The FIDO certification, ensures the security of the authenticators and allows for 3 possible implementations of authenticators. First, L1+ security certification is given to devices with pure software implementations and hardened through security techniques. L2+ security certification is given to devices with Restricted Operating Environment such as TEE. Lastly, L3 o L3+ security certification is given to implementations with hardware components like Secure Elements.

- **Requirement to mask upon display, and not to store in plain text personalised security credentials, as well as to protect secret cryptographic material from unauthorised discosure** (Article 22.1 and 22.2) - FIDO authenticators's personalised credentials are key pairs, and the private key never leaves the authenticator device. Furthermore, the FIDO certification program assess the implementation and records the authenticator device's certificate level at the FIDO metadata service[10].

- **Requires the documentation of the process to manage cryptographic material associated with the personalised security credentials** (Article 22.3) - All FIDO-related protocols and specifications are fully documented and available online. Furthermore the FIDO metadata service provide additional information on the authenticator devices.

- **Ensure that personalised security credentials and the authentication codes are processed and routed under secure environments and according strong standards** (Article 22.4) - Personalised security credentials are generated inside the authenticator and are never disclosed. The generation and the protection of the keys is verified by the FIDO certification program and the requirements are based on recognised industry standards.

- **Ensure the creation of personalised security credentials under a secure environment and mitigate risks related to unauthorised use** (Article 23) - Personalised security credentials are generated inside the authenticator and are never disclosed. The generation and the protection of the keys is verified by the FIDO certification program. Without the knowledge of the PIN or the valid identification of the user by the biometric authentication an unauthorised party can not use the authenticator device.

- **Ensure that the association of the user with the personalised security credentials is performed under a secure environment and using SCA** (Article 24.1 and 24.2) - After the payment service conduct a proper user identity identification, the FIDO registration process can be conducted to generate a random new key pair for the user and associating it to the payment service and user identity. During this process the authenticator will verify the identity of the user by a PIN or a biometric authentication.

- **Ensure secure delivery mechanics for the delivery of the personalised security credentials, ensuring the authenticity of the software and ensuring the secure activation before use** (Article 25.1 and 25.2) - During the FIDO registration process, no private information is shared and after the process, the

[10] https://fidoalliance.org/metadata/

authenticator is automatically enabled. The private key never leaves the device and only requests from the relying party can be signed by it.

- **Ensure that the reactivation of the personalised security credentials follows the same secure procedures as their creation and delivery** (Article 26) - Since the relying party service has the public key of the created credentials, they can flag them as activated of disable based on their own needs and policies, without the need to exchange any information with the authenticator.
- **Ensure that the appropriate procedures to destroy, deactivate or revoke of personalised security credentials and related information as well as the secure reuse is established, documented and implemented if the authenticator devices can be reused** (Article 27) - Since the relying party service has the public key of the created credentials, they can flag them as activated of disable based on their own needs and policies, without the need to exchange any information with the authenticator. We have to note that the credentials are randomly generated during the registration process.

3.5 Identified Issues and Problems

FIDO2 and its extension to web sites, WebAuthn, can provide a strong authentication solution compatible with all the web, providing an solution to the password problem. Even though theoretically FIDO also meets the requirements set by PSD2 and the RTS for SCA, there are a number of practical issues that hold back its adoption by service providers.

In this section we will analyse the problems we identified and try to propose solutions where possible. Our focus will be on problems affecting both FIDO2 and WebAuthn.

3.5.1 Certification of Relying Party FIDO services

The FIDO Alliance created a certification program for both FIDO servers and FIDO authenticator devices. Although the certification program of the FIDO authenticators focus on their security, this is not the case with the FIDO server as the focus is on their compatibility. This means that payment service providers would have to develop their own FIDO2/WebAuthn implementation and ensure it meets their security requirements. This is not an easy task though as the community of FIDO2/WebAuthn experts is not mature enough due to the recent standardisation of WebAuthn.

To make things worst, the complexity of FIDO2 and WebAuthn, combined with the lack of tools, makes the implementation testing difficult and inefficient. This may deter many service providers from implementing FIDO2/WebAuthn due to the high risk of bad implementing the protocol or configuring the service.

To this end, we worked on providing a novel tool, presented on Section **??**, able to aid penetration testers and auditors into assessing FIDO2/WebAuthn relying party services, and also help developers learn how FIDO2 and WebAuthn works and debug or test their implementations.

FIDO Alliance should also enhance their FIDO server certification program by including an evaluation of the server's security. Tools similar to the one we released may aid in this process. As is the case with the authenticator devices certification

program, multiple security levels could be set for the server implementation, giving a better understanding of their security capabilities.

3.5.2 Adoption and Compatibility

WebAuthn adoption by browsers and operating systems has increased the last years, making the technology available to almost every user. All major operating systems, for both desktop and mobile devices support FIDO and in many case they also feature an embedded FIDO authenticator device inside. Furthermore, the latest industry trends which sets the existence of TMP chips on every computer as a requirement, will further increase the availability of FIDO authenticators with a strong security level.

Browser support is also here, with all major browser supporting WebAuthn. Though, browsers implement WebAuthn consciously, to avoid introducing new vulnerabilities, thus removing any part of the standard that may be used to compromise the browser's security or attack the user's privacy. For this reason, some FIDO features (such as the Transaction Message) although they were initially included on the first version of WebAuthn, they were later removed as no one implemented them (questioning their security).

These compatibility concerns may raise questions on whether an organisation should invest on WebAuthn or wait first until the standard is mature enough. Maybe this is ther reason why we don't see service providers with WebAuthn support. On the other hand, the current state of the compatibility shows that the support is already here and thus maybe this is the time to start adopting WebAuthn.

3.5.3 Tolerance of Failed Authentications

One critical measure, especially important for authentication based on knowledge element, to protect against attacks is the block (some times temporary) of the service after a number of failed authentication attempts. From the FIDO authenticator side, depending on its implementation, it may block or limit with timeouts the user interaction attempts when the user fails to authenticate successfully. For example, an authenticator may be blocked after the user entered 5 times a wrong PIN code. In this way it makes it harder for someone to uncover the PIN using brute force. Though we have to keep in mind that since such authenticator features are not mandatory, and as such their implementation it's up to their manufacturer.

In the same way, ideally, the relying party should also monitor and limit (if needed) the actually WebAuthn authentication. Though, since FIDO is not an one step verification process but a two step challenge response process, questions arise on what is a failed authentication. Of course an invalid response by the client can be flagged as a failed authentication, but what about an attempt where a challenge was requested but no reply was send to the server? On top of that, the WebAuthn process may take multiple seconds (even minutes) to be completed, as actions have to be done by human users (e.g. wait for a user to find his/her USB security key), thus making it important to link all info and timers with the challenge and the authentication session.

3.5.4 Selecting Trusted Authenticator Devices

As pointed out by FIDO Alliance review of the Regulatory Technical Standards (RTS) for Strong Customer Authentication (SCA)[11], not all authenticator devices are compliant with the PSD2 requirements. For example, based on their implementation, some authenticators may lack the appropriate security measures to protect the private key (personalised security credentials), may have a higher that accepted rate of false positive biometric identification or lack any block mechanics after a number of failed attempts.

To ensure that their users are using only authenticator devices approved by the relying party (in our case the service provider), there are 2 options. One may rely on the information given by the FIDO Alliance metadata service[12] and assess whether the given authenticator device meets the payment service's policy. Otherwise, one would have to create a list of accepted devices, tested or provided by the payment service itself, and accept only those for the authentication of the users. Both methodologies are based on the attestation provided by the devices thus the authenticity of the authenticator device is ensured.

3.5.5 Availability Authenticator Devices

Based on the security requirements set by PSD2 and the RTS, as also pointed out by the FIDO Alliance review of the Regulatory Technical Standards (RTS) for Strong Customer Authentication (SCA)[13], the FIDO Alliance metadata service[14] plays a vital role into assessing whether an authneticator device meets the security requirements set by PSD2 and the replying party's security policy.

The FIDO Alliance metadata service lists detailed information for all the authenticator register to it. The service can provide information regarding each authenticator device capabilities as well as certificates to verify the authenticity of the generated credentials during registration. By using this information, service providers can assess whether the credentials were generated by an authenticator device that meets their criteria (security requirements) and can be trusted.

For this reason we looked into the authenticators listed under the FIDO Alliance metadata service. Table 3.4 To filter the authenticators and count the available authenticators, we developed a tool[15] allowing though an easy to use user interface to pick the filters of the user's interest.

Based on the statistics we generated, one can see that there are limited availability in available authenticators as the security requirements increase. Most importantly, by looking at the certifications[16] of the available authenticators, there are only 5 L2 certified authenticators, while there is non L3 certified authenticator. Similarly, regarding the security level of the available authenticators, there are 4 authenticators with 256-bit cryptographic strength, while there isn't any authenticator with a cryptographic strength of 512-bit. To make mater worst, if a payment service provider wants to use a L2 certified authenticator as an possession and inherence element, there is only on authenticator available under these criteria at the metadata service.

[11] https://fidoalliance.org/how_fido_meets_the_rts_requirements/
[12] https://fidoalliance.org/metadata/
[13] https://fidoalliance.org/how_fido_meets_the_rts_requirements/
[14] https://fidoalliance.org/metadata/
[15] https://github.com/GramThanos/FIDO-Authenticator-Metadata-Filters
[16] https://fidoalliance.org/certification/authenticator-certification-levels/

TABLE 3.4: Statistics of the authenticators under the FIDO Alliance metadata

Criteria	Number
All authenticators	101
FIDO2 authenticators	49
U2F authenticators	35
UAF authenticators	17
L1 certified authenticators	54
L1+ certified authenticators	0
L2 certified authenticators	5
L2+ certified authenticators	0
L3 certified authenticators	0
L3+ certified authenticators	0
128-bit cryptographic strength authenticators	62
256-bit cryptographic strength authenticators	4
512-bit cryptographic strength authenticators	0
Inherence supporting authenticators	41
Knowledge supporting authenticators	32
L1/L2 Certified, FIDO2/U2F, inherence supporting, hardware/tee protected authenticators	17
L1/L2 Certified, FIDO2/U2F, knowledge supporting, hardware/tee protected authenticators	20
L2 Certified, FIDO2/U2F, inherence supporting authenticators	1
L2 Certified, FIDO2/U2F, knowledge supporting authenticators	1

To this end, even if FIDO can meet the requirements set by the PSD2, this does not means that appropriate authenticator device exists. Based on our analysis of the available authenticator devices on FIDO Alliance metadata service, there are not much authenticator options if your security requirements are over the basic level. We hope this to change in the future.

Chapter 4

Implementations

4.1 StrongMonkey

FIGURE 4.1: StrongMonkey, a PHP SDK for interacting with FIDO2 Server API v3.0.0.

This section looks into the StrongMonkey, an software development kit (SDK) we developed to assist in the implementation of custom FIDO2/WebAuthn services. The SDK is open-source released under the GNU LGPLv2.1 license and can be fund online at GitHub[1]. StrongMonkey provides to developers libraries through which they can interact with servers featuring the StrongKey's FIDO2 Server API[2].

4.1.1 Introduction

The authentication module is one of the most important parts of an application, as this module is responsible for managing the user access to the application's services. Developers have to possess knowledge on how to develop it securely using trusted authentication mechanics. However, even the simplest authentication mechanics, like the traditional username and password, are not so trivial to implement correctly (e.g. using salting, using password hashing algorithms).

FIDO2/WebAuthn authentication is more complex. Developers tasked to implement a WebAuthn authentication have to be familiar with the standard and implement it correctly to reach conformance. Hence, depending on the size of the project, it is easier, faster, cheaper and more secure to use an existing FIDO2/WebAuthn server or library. For small single application projects, the use of a WebAuthn library may the best option, while for bigger projects with multiple

[1] https://github.com/GramThanos/StrongMonkey

[2] https://demo4.strongkey.com/getstarted/#/openapi/fido

application and bigger numbers of active users, the use of a specialised FIDO2 server may be wiser.

StrongKey FIDO2 Server (SKFS) is an open source FIDO server. An application's back-end may contact the SKFS server through its REST (JSON) or SOAP (XML) API. The server's API allows an application to register new authenticator devices, authenticate users using already registered devices, as well as manage the already registered keys of a user. A description of the SKFS API is available on StrongKey's website[3]. The SKFS is free for use and the community edition of the server is available on GitHub[4] along with other proof-of-concept applications and utilities.

For the faster adoption of FIDO2 and WebAuthn, it is important to promote open source implementation and allow developers to use the WebAuthn technology seemingly with their application through an easy to implement way. For this reason we developed StrongMonkey, an SDK that developers can use to connect their application with an FIDO2 server that supports the SKFS API (e.g. StrongKey FIDO2 Server).

4.1.2 Implementation

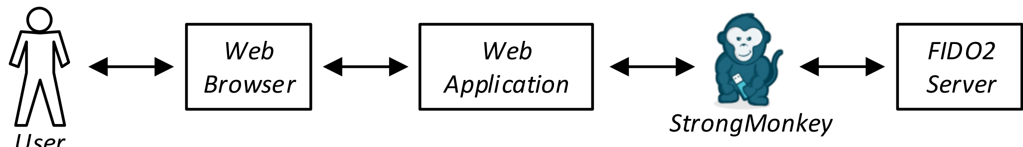


FIGURE 4.2: Overview connection flow using StrongMonkey.

StrongMonkey SDK comes in the form of a library, that can be included in the application code and convert the FIDO2 server's API into function calls, as shown on Figure 4.2, acting as a middle-ware to simplify the usage of the service. The SKD handles all the authentication procedures between the application server and the FIDO2 server as well as all the parsing needed to be done to pass the appropriate parameters to the FIDO2 server. A detailed flow of the authentication process using StrongKey is shown on Figure 4.3. The SDK is currently available in 2 languages, Python and PHP, thus is compatible with most web applications.

StrongMonkey offers 8 methods corresponding to the 8 main functions of the FIDO2 server:

- **Preregister** - Generate the request to be forwarded to the client for the registration of a new authenticator device.
- **Register** - Forward the authenticator response to the server to register the generated credentials.
- **Preauthenticate** - Generate the request to be forwarded to the client for the authentication with an authenticator device.
- **Authenticate** - Forward the authenticator response to the server to validate the challenge signature.

[3] https://demo4.strongkey.com/getstarted/#/openapi/fido
[4] https://github.com/StrongKey/fido2

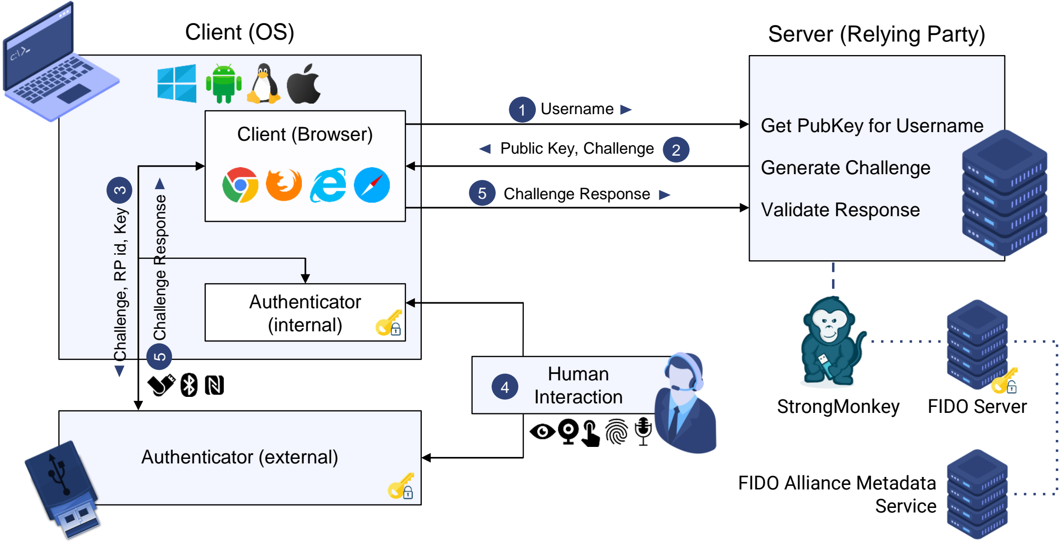


FIGURE 4.3: Detailed FIDO2/WebAuthn authentication flow using StrongMonkey.

- **Update key info** - Change the metadata saved for a particular authenticator (credentials).
- **Get key info** - Load the metadata information for a particular authenticator (credentials).
- **Deregister** - Remove a particular authenticator (credentials).
- **Ping** - Get the server information, used to test the connection with the server.

Each function formats the payload appropriate and calls the request function providing the endpoint to call and the payload to send, as shown in Figure 4.4. The request function will call prepare the request's authentication, by attaching the username and password or sign the request with HMAC, and create the request to the FIDO2 server. After receiving the request, the data provided will be parsed and returned to the application code.

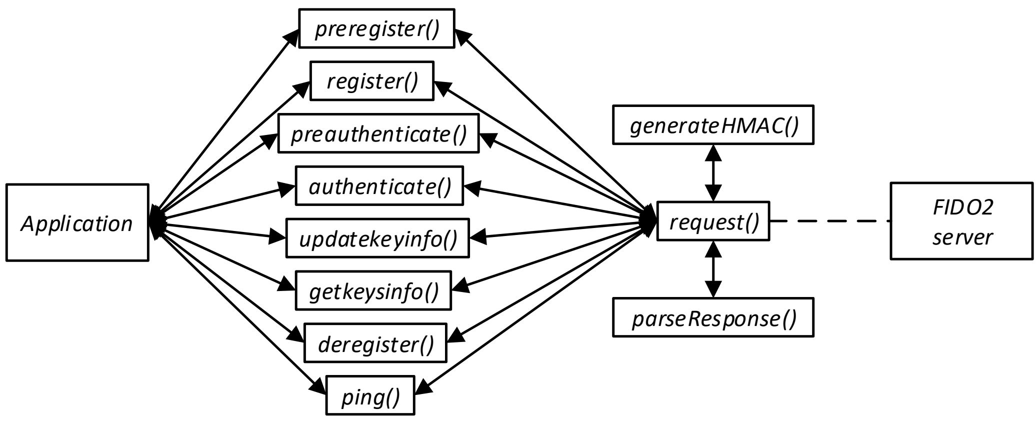


FIGURE 4.4: StrongMonkey main functions flow.

4.1.3 Usage

In this section we will explain briefly how developers may use the StrongMonkey SDK to connect their application with a supported FIDO service. As an example, the PHP version of the StrongMoneky will be used.

StrongMonkey Object

The developer has to first generate a StrongMonkey object specifying the FIDO2 server to connect to, the protocol to use as well as the authentication method and the credentials to authenticate with the server with.

```
1 new StrongMonkey(
2     string $hostport,
3     integer $did,
4     string $protocol,
5     string $authtype,
6     string $keyid,
7     string $keysecret
8 ) : StrongMonkey
```

To initialise the object, the following 6 parameters specifying the connection with the FIDO2 server has to be given:

- $hostport
  - Host and port to access the FIDO SOAP and REST formats
    - `"http://<FQDN>:<non-ssl-portnumber>"` or
    - `"https://<FQDN>:<ssl-portnumber>"`
- $did
  - Domain ID e.g. `1`
- $protocol
  - Web socket protocol; `'REST'` or `'SOAP'` (only REST is supported)
- $authtype
  - Authorization type; `'HMAC'` or `'PASSWORD'`
- $keyid
  - PublicKey or Username (keys should be in hex)
- $keysecret
  - SecretKey or Password (keys should be in hex)

The following code is an example of initialising the StongMonkey object using HMAC authentication:

```
1 // Prepare Object
2 $monkey = new StrongMonkey(
3     'https://strongkey.unipi.gr:8181', // URL of your FIDO2 server
4     1, // Domain ID (usually 1)
5     'REST', // Protocol ti use (currently always REST)
6     'HMAC', // Authentication method to use
```

```
7       '162a5684336fa6e7', // Default key ID (for HMAC)
8       '7edd81de1baab6ebcc76ebe3e38f41f4' // Default secret key (for HMAC)
9   );
```

The following code is an example of initialising the StongMonkey object using PASSWORD authentication:

```
1   // Prepare Object
2   $monkey = new StrongMonkey(
3       'https://strongkey.unipi.gr:8181', // URL of your FIDO2 server
4       1, // Domain ID (usually 1)
5       'REST', // Protocol ti use (currently always REST)
6       'PASSWORD', // Authentication method to use
7       'svcfidouser', // Default username (for PASSWORD)
8       'Abcd1234!' // Default password (for PASSWORD)
9   );
```

Preregister

To initialize a key registration challenge with the FIDO server, the preregister method has to be called.

```
1   $monkey->preregister(
2       string $username[,
3       string $displayname = null[,
4       array|string $options=null[,
5       array|string $extensions=null
6   ]]]) : integer|array
```

The preregister, method takes as an input 1 required parameter and 3 optional:

- $username
  - Username of the user
  - Based on the WebAuthn standard:
    - Human-palatable identifier for the user account, intended only for display, helping distinguish form other user
    - The relay party MAY let the user choose this value
    - ex. `john.smith@email.com` (email) or `+306901234567` (telephone)
- $displayname
  - Display name for the user
  - Based on the WebAuthn standard:
    - Human-palatable name for the user account, intended only for display
    - The relay party SHOULD let the user choose this value
    - ex. `J. Smith` (full name)
- $options
  - Object of options
  - Options support depend on your FIDO2 server
- $extensions
  - Object of extensions

- Extensions support depend on your FIDO2 server

The following code is an example call and the response is forwarded to the front-end:

```
// Request to start a key registration
$response = $monkey->preregister($username);
// Check for errors
if ($monkey->getError($response)) {
    die('Failed to start pre-registeration with the FIDO2 server.');
}
// Maybe save the challenge on the session so that you can match
// it when you receive the reply
$_SESSION['challenge'] = $response->Response->challenge;
// Prepare object for WebAuthn
$webauthn = $response->Response;
// Set your Replay Party info
// Relaying Party ID, a valid domain string that identifies the
// WebAuthn Relying Party (It should be the webpage domain
// or a subset of the domain)
$webauthn->rp->id = strtok($_SERVER['HTTP_HOST'], ':');
// Human-palatable identifier for the Relying Party,
// intended only for display
$webauthn->rp->name = 'StrongMonkey';
// Reply as JSON (assuming that the JavaScript will
// handle the request through ajax)
http_response_code(200);
header('Content-Type: application/json');
exit(json_encode($webauthn));
```

Register

To send a register response to the FIDO server, the register method has to be called.

```
$monkey->register(
    array|string $response[,
    array|string $metadata=null
]) : integer|array
```

The register, method takes as an input 1 required parameter and 1 optional:

- $response
  - Response data from the authenticator
- $metadata
  - Additional meta data
  - Meta data needed depend on your FIDO2 server
  - e.g. StrongKey FIDO2 Server requires an object as shown on the example bellow

The following code is an example call:

```
// Assuming that the reply from the client is at the
// variable $authenticator_response
// Before forwarding the request to the server,
// you may want to check the challenge
$clientDataJSON = json_decode(base64_decode(
```

```
6       $authenticator_response['response']['clientDataJSON']
7   ));
8   if (
9       !$clientDataJSON ||
10      $_SESSION['challenge'] !== $clientDataJSON->challenge
11  ) {
12      die('Authentication failed due to challenge mismatch.');
13  }
14  // Request to register Key
15  $response = $monkey->register($authenticator_response, array(
16      'version' => '1.0',
17      'create_location' => 'webapp',
18      'username' => $username,
19      'origin' => 'https://' . $_SERVER['HTTP_HOST']
20  ));
21  // Check for errors
22  if ($monkey->getError($response)) {
23      die('Failed to register key to the FIDO2 server.');
24  }
25  // Print response message
26  die($response->Response);
```

Preauthenticate

To initialize a key authentication challenge with the FIDO server, the preauthenticate method has to be called.

```
1   $monkey->preauthenticate(
2       string $username[,
3       array|string $options=null[,
4       array|string $extensions=null
5   ]]) : integer|array
```

The preauthenticate, method takes as an input 1 required parameter and 2 optional:

- $username
  - Username of the user
- $options
  - Object of options
  - Options support depend on your FIDO2 server
- $extensions
  - Object of extensions
  - Extensions support depend on your FIDO2 server

The following code is an example call and the response is forwarded to the front-end:

```
1   // Request to start an user authentication
2   $response = $monkey->preauthenticate($username);
3   // Check for errors
4   if ($monkey->getError($response)) {
5       die('Failed to start pre-authenticate with the FIDO2 server.');
6   }
```

```
7  // Maybe save the challenge on the session so
8  // that you can match it when you receive the reply
9  $_SESSION['challenge'] = $response->Response->challenge;
10 // Prepare object for WebAuthn
11 $webauthn = $response->Response;
12 http_response_code(200);
13 header('Content-Type: application/json');
14 exit(json_encode($webauthn));
```

Authenticate

To send the authenticate response to the FIDO server, the authenticate method has to be called.

```
1 $monkey->authenticate(
2     array|string $response[,
3     array|string $metadata=null
4 ]) : integer|array
```

The authenticate, method takes as an input 1 required parameter and 1 optional:

- $response
  - Response data from the authenticator
- $metadata
  - Additional meta data
  - Meta data needed depend on your FIDO2 server
  - e.g. StrongKey FIDO2 Server requires an object as shown on the example bellow

The following code is an example call:

```
1  // Assuming that the reply from the client
2  // is at the variable $authenticator_response
3  // Before forwarding the request to the server,
4  // you may want to check the challenge
5  $clientDataJSON = json_decode(base64_decode(
6      $authenticator_response['response']['clientDataJSON']
7  ));
8  if (
9      !$clientDataJSON ||
10     $_SESSION['challenge'] !== $clientDataJSON->challenge
11 ) {
12     die('Authentication failed due to challenge mismatch.');
13 }
14 // Here you may also want to check if
15 // the username provided exists as a user
16 // Request to authenticate user
17 $response = $monkey->authenticate($authenticator_response, array(
18     'version' => '1.0',
19     'last_used_location' => 'webapp',
20     'username' => $username,
21     'origin' => 'https://' . $_SERVER['HTTP_HOST']
22 ));
23 // Check for errors
24 if ($monkey->getError($response)) {
25     die('Failed to authenticate user with the FIDO2 server.');
```

```
}
// Print response message
die($response->Response); // This will be blank
```

Update key info

To update key information, the updatekeyinfo method has to be called.

```
$monkey->updatekeyinfo(
    string $status,
    string $modify_location,
    string $displayname,
    string $keyid
) : integer|array
```

The updatekeyinfo, method takes as an input 4 required parameter:

- $status
  - The status of the key (Active, Inactive)
- $modify_location
  - Modify location
- $displayname
  - Display name of the key
- $keyid
  - Id of the key to change

The following code is an example call:

```
// Request to update Key
$response = $monkey->updatekeyinfo(
    'Inactive',
    'webapp',
    'Text Display Name',
    $keyid
);
// Check for errors
if ($monkey->getError($response)) {
    die('Failed to update key to the FIDO2 server.');
}
// Print response message
$keys = $response->Response;
```

Get key info

To get user's keys information from the FIDO server, the getkeysinfo method has to be called.

```
$monkey->getkeysinfo(string $username) : integer|array
```

The getkeysinfo, method takes as an input 1 required parameter:

- $username

- Username of the user

The following code is an example call:

```
// Request to get Keys from user
$response = $monkey->getkeysinfo($username);
// Check for errors
if ($monkey->getError($response)) {
    die('Failed to get keys from the FIDO2 server.');
}
// Retrieve keys from the response
$keys = $response->Response->keys;
```

Deregister

To delete user's key information from the FIDO server, the deregister method has to be called.

```
$monkey->deregister(string $keyid) : integer|array
```

The deregister, method takes as an input 1 required parameter:

- $keyid
  - Id of the key to deregister

The following code is an example call:

```
// Request to delete key
$response = $monkey->deregister($keyid);
// Check for errors
if ($monkey->getError($response)) {
    die('Failed to deregister key on the FIDO2 server.');
}
// Print response message
die($response->Response);
```

Ping

To send a ping to the FIDO server to check if up, the ping method has to be called.

```
$monkey->ping() : boolean|string
```

The ping, method doesn't take any parameter. The following code is an example call:

```
// Ping request
$response = $monkey->ping();
// Check for errors
if ($monkey->getError($response)) {
    die('Failed to ping FIDO2 server.');
}
// Print response
die($response);
```

4.1.4 DEMO Application

Along with the StrongMonkey SDK, a DEMO web application[5] was developed to showcase how FIDO2/WebAuthn can be deployed. The application is based on StrongMonkey and StrongKey FIDO2 Server to and allows users to create a DEMO temporal account, register FIDO authenticator devices using WebAuthn and experience password-less authentication with FIDO. Additional, an experimental FIDO sign in with a QR code and a smartphone as an authenticator device was implemented to show how the technology can be extended even further to provide innovated solutions.

Use case example

After creating an account, a user is able to sign into the DEMO web application using a traditional username and password, as shown in Figure 4.5. Right now there are no FIDO authenticator devices registered on the account, and thus a password authentication is the only option.

FIGURE 4.5: The user has initially to use the traditional username & password login.

After the initial login, the user can visit his/her account "Manage Keys" settings page to register authenticator devices and enable secure password-less login with FIDO/WebAuthn. Figure 4.6 shows a user initialising the WebAuthn credentials registration process by clicking the "Add Key". On this particular case, the user's computer featured a Windows 11 OS, and thus Windows Hello (which can be used as a FIDO platform authenticator) handled the request. Since the user's device didn't had a biometric sensor, the user had to authenticate using his Windows Hello PIN, to allow the internal authenticator to generated the requested credentials and forward them to the FIDO service. We have to note that the Windows Hello mechanics are based on the TPM chip of the device and latest Windows 11 compatible computers require by default a TPM chip to operate.

Through the "Manage Keys" page, a user is also able to remove a registered FIDO device from his/her account. Just like on a laptop, a user can use the browser of his/her smartphone to register its internal authenticator device (Android and iOS do support their own internal FIDO authenticator devices). Figure 4.7 shows registered keys of the user, one key from his/her Windows Hello authenticator and one key from his/her Android smartphone device.

[5] https://github.com/GramThanos/StrongMonkey/tree/a17fc78ad4b5d445ca402c9c40eb1fd0ca980bcf/php/demo-app

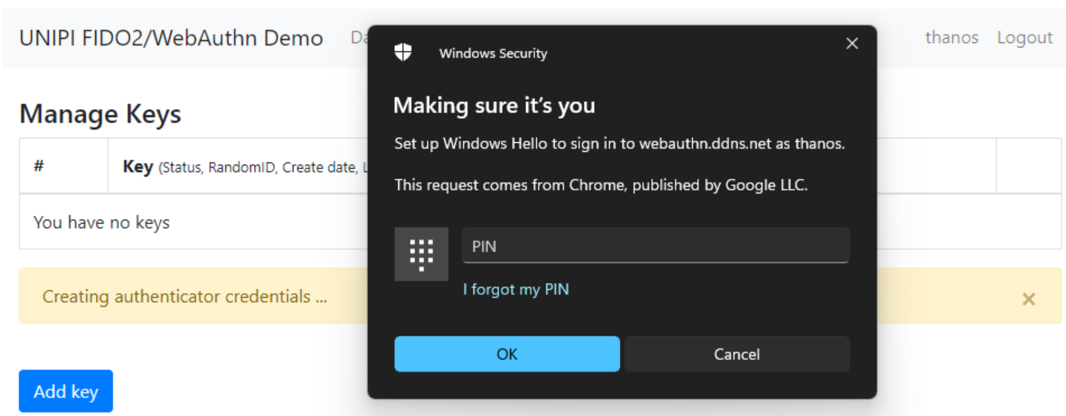


FIGURE 4.6: User registers a Windows Hello authenticator device on his/her account.

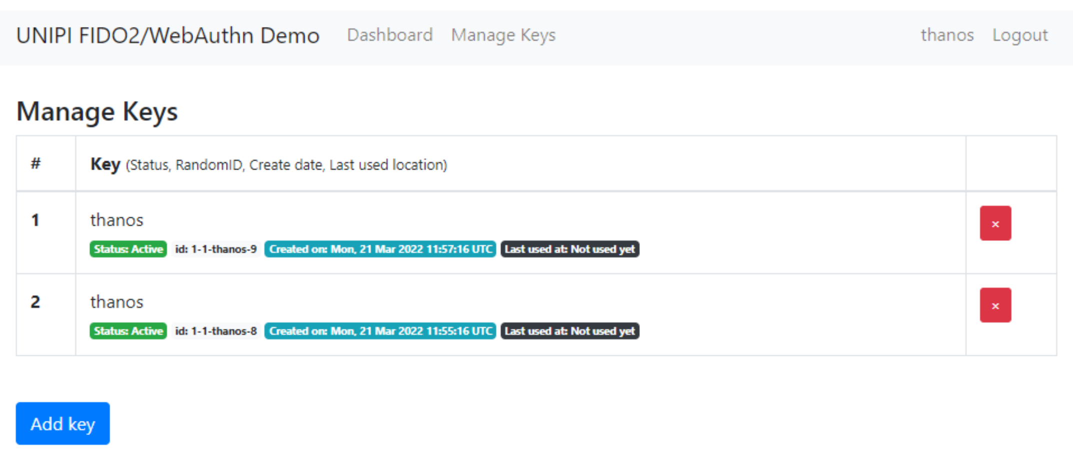


FIGURE 4.7: User manage the security keys registered on the account listed under the Manage Keys page.

After having registered an authenticator device on his/her account, the user is now able to use FIDO login and sign in password-less. After providing the account username and clicking the "Login with FIDO" button, the WebAuthn credentials get process. Figure 4.8 shows the user using his/her device Windows Hello to authenticate to the service. Optionally, through the user interface the user may also select to use a Security Token (a FIDO compatible USB, Bluetooth or NFC authenticator). Depending on the authenticator's features, the user may use a PIN, a button, or his biometrics (e.g. touch the fingerprint sensor) to approve the action.

As mentioned, earlier, an experimental QR code authentication flow was developed and implemented on the DEMO application to showcase the possibilities of the technology. The QR code authentication flow works similar as the normal FIDO2/WebAuthn authentication flow, with the exception that the authentication session between the server and the client is extended to a 3rd device that has access to the authenticator device registered to the user account. Assuming that a user initiates the FIDO QR code authentication flow on his/her laptop device, he can scan the QR code, shown in Figure 4.9, with his/her smartphone device and open the website on a mobile Web browser that supports WebAuthn. As presented on Figure 4.10, for security reasons the user will then see the information of the client device requesting permission to sign into his/her account.

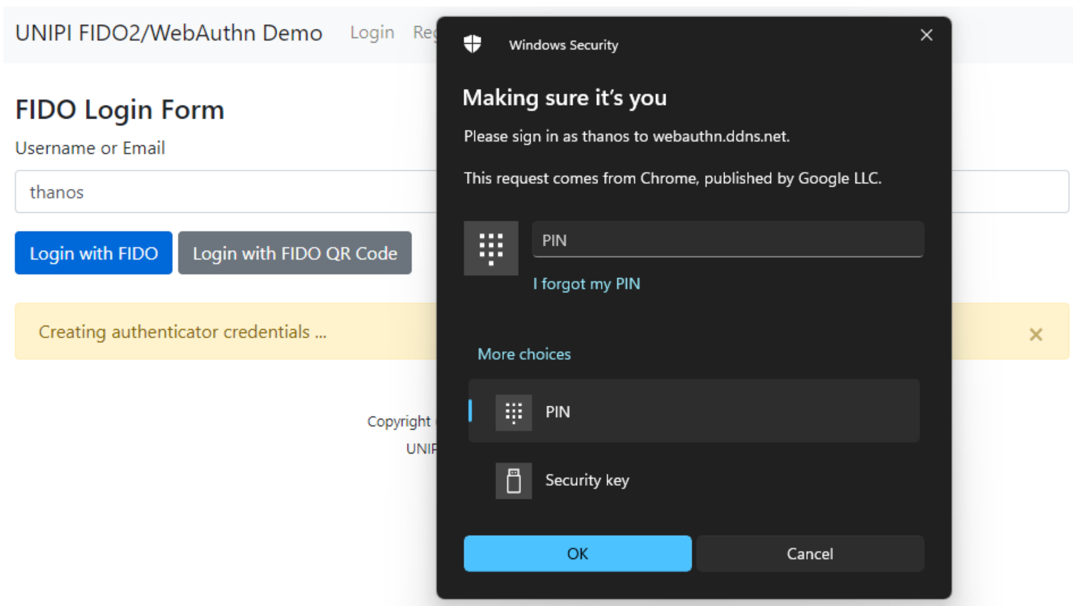


FIGURE 4.8: User uses WebAuthn sign in with the Windows Hello authenticator.

By selecting the "Authenticate" options, the normal WebAuthn authentication flow will start and request authentication using the device's internal authenticator device. After successful authentication, as shown on Figure 4.10, the laptop device of the user will be also authenticated and signed into the account.

4.1.5 Conclusions

To make the development of FIDO2/WebAuthn enabled secure authentication services easier, developers need more libraries and servers. Furthermore, to make the implementation easier tools and libraries are needed to handle the trivial but complex procedures needed to connect with such services.

StrongMonkey is a step forward into simplifying the implementation of FIDO2/WebAuthn into applications of all kinds. Our developed SDK provides a middleware for both Python and PHP applications connecting them with the StrongKey FIDO2 Server and other services featuring the same API. On top of that, this allows an organisation to implement FIDO2/WebAuthn authentication across multiple application and websites, linking them into a single FIDO2 authentication server.

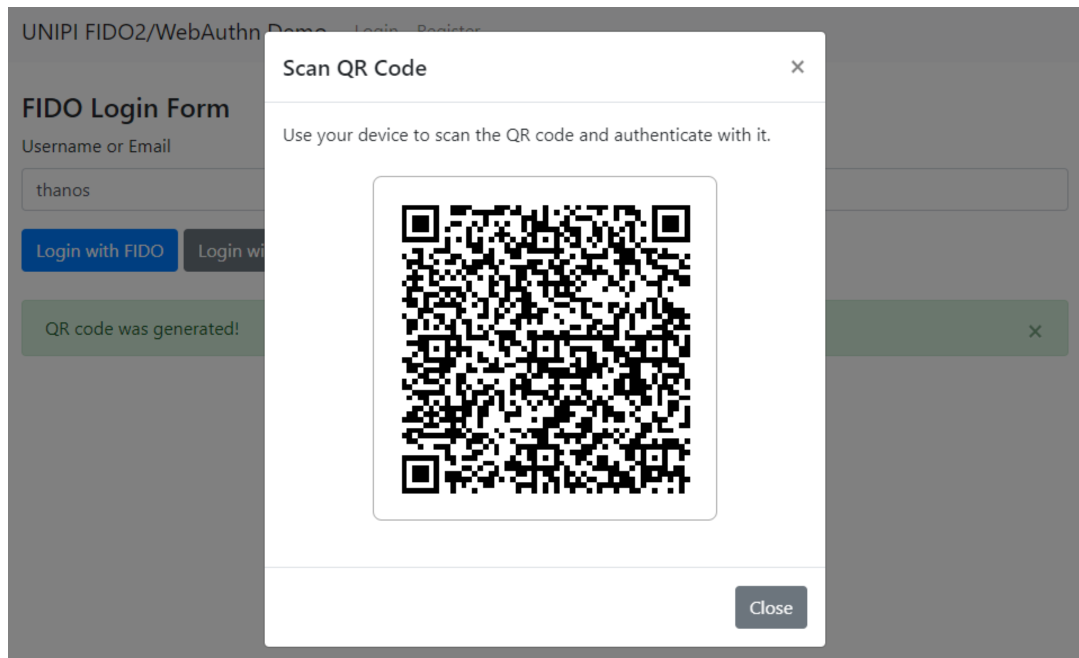


FIGURE 4.9: User use generates QR code to authenticate on PC using his/her smartphone's authenticator.

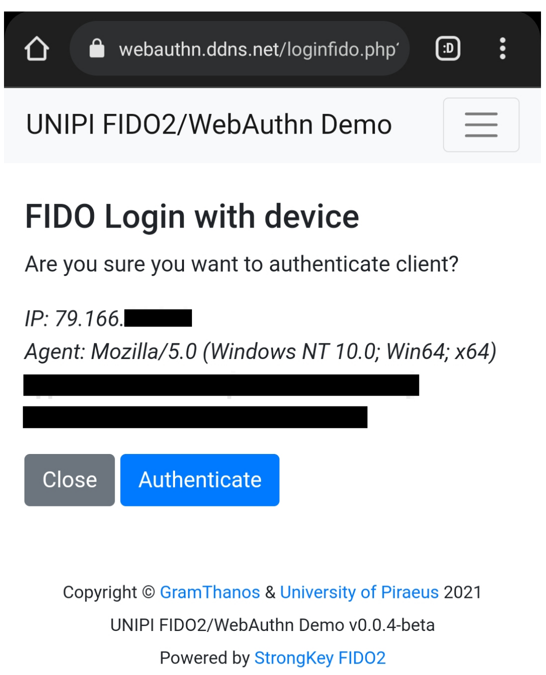


FIGURE 4.10: After scanning the QR code, user is asked to be authenticated in order to allow another device to sign into his/her account.

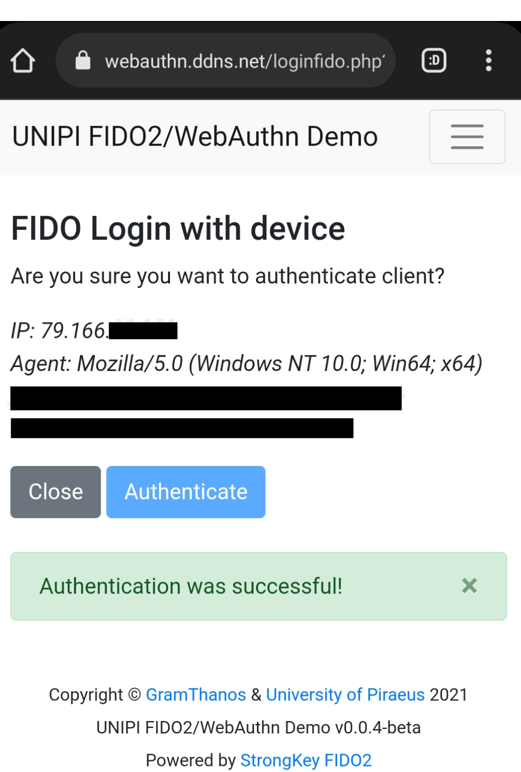


FIGURE 4.11: User was successfully authenticated on smartphone and his/her PC was signed in.

4.2 StrongBee

FIGURE 4.12: StrongBee, FIDO2/WebAuthn server in python.

This section looks into the StrongBee, FIDO2/WebAuthn server in python, based on Yubico's FIDO 2.0[6] library and Flask[7], we developed as an asset to enable the implementation of FIDO2/WebAuthn services managed through a single server. The server is open-source released under the GNU LGPLv2.1 license and can be fund online at GitHub[8]. StrongBee supports the StrongKey's FIDO2 Server API[9] and thus can be used as a light alternative to the popular StrongKey's FIDO2 Server[10].

4.2.1 Introduction

Implementing a password authentication securely does require a number of things such as salting and hashing passwords but can be considered relative easy when compared with FIDO2/WebAuthn authentication implementation. Indeed, implementing FIDO2 and WebAuthn authentication requires deep knowledge of the related specifications and knowledge over various different technologies often found both on the front-end and on the back-end of a service. Hence, implementing a FIDO authentication requires knowledge possessed by specialised experts.

To mitigate the need of specialists on the area and also minimise implementation issues and vulnerabilities, libraries and specialised fully built servers can be used instead or re-implementing everything from scratch. As is the case with cryptography, also with FIDO2/WebAuthn authentication it easier and more efficient to use WebAuthn libraries and ready to use FIDO2 servers.

One open source solution for implementing FIDO2 authentication is the StrongKey FIDO2 Server (SKFS)[11]. SKFS is written in Java and by inspecting it code, it is based in an server initially intended for FIDO U2F authentication. The server also defines an API, through which application can connect and use the services of the server. Additionally, it has many features making it ideal for deploying a central single FIDO2 server system for managing the FIDO authentication across an organisation's users. On the other hand, one may go more technical and look into

[6] https://github.com/Yubico/python-fido2
[7] https://github.com/pallets/flask
[8] https://github.com/GramThanos/StrongBee
[9] https://demo4.strongkey.com/getstarted/#/openapi/fido
[10] https://github.com/StrongKey/fido2
[11] StrongKey FIDO2 Server. Open-source FIDO server, featuring the FIDO2 standard, https://github.com/StrongKey/fido2

Yubico's FIDO 2.0[12] library which apart from the main functions developed to interact with FIDO compatible authenticator device, it also supports a basic implementation of a FIDO2/WebAuthn server.

By taking the best parts of both of these projects, we developed a new FIDO2/WebAuthn server, named StrongBee, targeting smaller projects that want to provide strong authentication solutions through FIDO2 and WebAuthn. Using the low level FIDO server functionalities of Yubico's FIDO 2.0 library and combining it with StrongKey FIDO2 Server's API, we released a modern, lighter and easiest to use server, featuring true password-less authentication (without the need to provide not even a username).

4.2.2 Implementation

Our server implementation is based on Python[13] and Flask[14] making it easily deplorable almost any device. The following list provides a brief description of each file of the server code:

- **strongbee** - Server source code folder
  - ↪ **api.py** - API endpoints code
  - ↪ **authen.py** - Service authentication code
  - ↪ **config.py** - Configuration code
  - ↪ **models.py** - Database models code
  - ↪ **server.py** - Main server code
  - ↪ **utilities.py** - Helpful functions code
  - ↪ **database.api.db** - SQLite database with service credentials (configurable path)
  - ↪ **database.keys.db** - SQLite database with users and authenticator credentials (configurable path)
  - ↪ **certificate.private.pem** - Private SSL certificate for HTTPS (configurable path)
  - ↪ **certificate.public.pem** - Public SSL certificate for HTTPS (configurable path)
- **tests** - Service tests folder
  - ↪ **StrongMonkey.py** - StrongMonkey library to connect with the service
  - ↪ **test-client.py** - Fake web application client to test service
  - ↪ **requirements.txt** - Python packages required for the test code

↪ **run.py** - Code to execute StrongBee server flask application

↪ **requirements.txt** - Python packages required for the StrongBee server

The server implements the StrongKey FIDO2 API and expose the following endpoints for authorised web applications to use to offer FIDO2/WebAuthn services:

[12] https://github.com/Yubico/python-fido2
[13] Python, https://www.python.org/
[14] Flask, https://flask.palletsprojects.com/

- **/sbfs/rest/ping** - Endpoint to check if the server is running and check its status
- **/sbfs/rest/preregister** - Endpoint to generate options and challenged for registering new credentials
- **/sbfs/rest/register** - Endpoint to send the authenticator's reply generated based on the preregister options and if valid register the generated credentials
- **/sbfs/rest/preauthenticate** - Endpoint to generate options and challenged for authenticating a user using one of the credentials registered
- **/sbfs/rest/authenticate** - Endpoint to send the authenticator's reply generated based on the preauthenticate options and if valid authenticate the user
- **/sbfs/rest/updatekeyinfo** - Endpoint to update the metadata saved with specific credentials
- **/sbfs/rest/getkeysinfo** - Endpoint to get the metadata saved with specific credentials
- **/sbfs/rest/deregister** - Endpoint to remove specific registered credentials

For compatibility reasons the above mentioned endpoints are not only available at `/sbfs/rest/...` but also also available under `/skfs/rest/...`, hence the server may also be used as a replacement for the StrongKey FIDO2 Server.

4.2.3 Usage

Preparing your system

To execute StrongBee, your system has to have `Python 3` and `pip` already installed installed. You will also have to get the latest version of StrongBee from the GitHub repository. You can download the repository in a `zip`[15] or clone it using `git` by running:

```
git clone https://github.com/GramThanos/StrongBee
```

After downloading the latest version (and extracting it if you got the zip file), you will have to install all the required packages (e.g. `flask`, `fido2`). This can be done by entering the StrongBee directory and using `pip` to install all the required packages listed inside the `requirements.txt` by executing:

```
cd StrongBee
python3 -m pip install -r requirements.txt
```

All the dependencies will now be installed and now you will have to configure the StrongBee server before running it.

[15] https://github.com/GramThanos/StrongBee/archive/refs/heads/main.zip

Configuring the server

Before launching your FIDO2 server, you will have to change its configuration to match your preferences. Thus, open the `strongbee/config.py` with your favorite text editor and change the parameters appropriately. Listing 4.1 shows the main parts of the default configuration. One may change the host or the port number of the server as well as disable the use of SSL (used for the HTTPS).

```
1  # Server Options
2  PORT = 8181
3  HOST = '0.0.0.0'
4
5  # SSL
6  SSL = True
7  CERT_PUBLIC = os.path.join(BASEDIR, 'certificate.public.pem')
8  CERT_PRIVATE = os.path.join(BASEDIR, 'certificate.private.pem')
9
10 # SQL Alchemy
11 SQLALCHEMY_BINDS = {
12   'keys':  'sqlite:///' + os.path.join(BASEDIR, 'database.keys.db'),
13   'api':  'sqlite:///' + os.path.join(BASEDIR, 'database.api.db'),
14   'cache': 'sqlite:///:memory:'
15 }
```

LISTING 4.1: Server side authentication code in Python.

To use the service through an HTTPS, one may link the server with some valid public and private certificate keys. Alternative for testing one may also generate self-signed certificates using `openssl` and place them under the `strongbee` directory:

```
openssl req -x509 -newkey rsa:4096 -nodes\
-out certificate.public.pem -keyout certificate.private.pem -days 365
```

Execution

Running the StrongBee server is relative simple, as one can just execute the provided `run.py` to run the flask server.

```
python3 run.sh
```

Unfortunately the web server inside Flask is not suggested for production. For production it is suggested to you a Web Server Gateway Interface (WSGI) server. For example, you may deploy StrongBee using `gunicorn`[16] with the following command:

```
gunicorn --bind 0.0.0.0:8181 run:server --workers 2\
```
certfile=strongbee/certificate.public.pem –keyfile=strongbee/certificate.private.pem|

Examples

Let's look into the requests created by the `test-client.py` and the responses generated by the StrongBee server. Note that for the execution of the tests, the `soft-webauthn` package[17] is needed to simulate the authenticator responses. The

[16] https://gunicorn.org/
[17] https://pypi.org/project/soft-webauthn/

tests also make use of the StrongMonkey SDK in python to contact the StrongBee server.

For each test contacted by the test client script, we will show the request send by the client (endpoint URL, headers, data and server response).

The first action done by the test client script is to test the ping endpoint, used to get the server information and also check if ther server is running. The endpoint is located at:
`https://localhost:8181/sbfs/rest/ping`

```
1 Accept: application/json
2 Content-Type: application/json
3 User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
4 strongbee-content-sha256:
5 Date: Mon, 21 Jan 2022 15:46:42 UTC
6 strongbee-api-version: SK3_0
7 Authorization: HMAC b55938050b05019a:0cfIT+TUjM55YqpQsElLSIcERuc9jnMzj5tb1VCkHUo=
```

LISTING 4.2: Headers send with the ping request.

```
1 {"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}}
```

LISTING 4.3: Data send with the ping request.

```
1 StrongBee, StrongBee FIDO2 Server v0.0.3-beta
2 Hostname: localhost:8181
3 Current time: Mon Mar 21 15:46:42 UTC 2022
4 Up since: Mon Mar 21 15:46:35 UTC 2022
5 FIDO Server Domain 3 is alive!
```

LISTING 4.4: Response returned to the ping request.

Then the test client script starts the registration by contacting the preregister endpoint, used to get the a authenticator registration challenge from the server. The endpoint is located at:
`https://localhost:8181/sbfs/rest/preregister`

```
1 Accept: application/json
2 Content-Type: application/json
3 User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
4 strongbee-content-sha256: GzOIPpYxaEkY2g/Tt1yN+6z+cjMTM4oOLIGPeChs30s=
5 Date: Mon, 21 Jan 2022 15:46:42 UTC
6 strongbee-api-version: SK3_0
7 Authorization: HMAC b55938050b05019a:RmJuPhiQ5LIRDl3EaONdyxOZts7/7zgHNME7fUHuroE=
```

LISTING 4.5: Headers send with the preregister request.

```
1 {"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}, "payload"
  ↪ : {"username": "gramthanos@gmail.com", "displayname": "gramthanos@gmail.com",
  ↪ "options": "{}", "extensions": "{}"}}
```

LISTING 4.6: Data send with the preregister request.

```
1 {
2     "Response": {
3         "rp": {
4             "id": "unipi.gr",
5             "name": "unipi.gr"
6         },
7         "user": {
8             "id": "Z3JhbXRoYW5vc0BnbWFpbC5jb20",
```

```
9             "name": "gramthanos@gmail.com",
10            "displayName": "gramthanos@gmail.com"
11        },
12        "challenge": "S1aq3Fmz1qaxzL5rv0dMxlHgzLGjYVWlDezbAPbASEI",
13        "pubKeyCredParams": [
14            {"type": "public-key","alg": -7},
15            {"type": "public-key","alg": -8},
16            {"type": "public-key","alg": -37},
17            {"type": "public-key","alg": -257}
18        ],
19        "excludeCredentials": [],
20        "timeout": 60000
21    }
22 }
```


LISTING 4.7: Response returned to the preregister request.

The test client script used the provided parameters and challenge from the server to generate new credentials and a valid authenticator response.


```
1 {
2     "id": "5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY",
3     "rawId": "5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY",
4     "response": {
5         "clientDataJSON": "
              ↪ eyJ0eXBlIjogIndlYmF1dGhuLmNyZWF0ZSIsICJjaGFsbGVuZ2UiOiAiUzFhcTNGbXoxcWF4ekw1cnYwZE14bEhnekxHal
              ↪ ",
6         "attestationObject": "
              ↪ o2NmbXRkbm9uZWdhdHRTdG10oGhhdXRoRGF0YVikgh8LzXo6LfwnYQzR6rDVRNpPzX7RkQQxuk9PNiWa4qRBAAAAAAAAAA
              ↪ -aJXyWq5GgItjSvvI94oR3KgbNrWdS3oWX2N-c"
7     },
8     "type": "public-key"
9 }
```


LISTING 4.8: Credentials creation, authenticator response generated by test client.

Then the test client script returns the authenticator response to the server by contacting the register endpoint. The endpoint is located at:
`https://localhost:8181/sbfs/rest/register`

```
1 Accept: application/json
2 Content-Type: application/json
3 User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
4 strongbee-content-sha256: KE/eiJeUs1wc35244DRdw51JckxB5Lk6VlCtmiyLoPc=
5 Date: Mon, 21 Jan 2022 15:46:42 UTC
6 strongbee-api-version: SK3_0
7 Authorization: HMAC b55938050b05019a:mesgnVGRMEaPEQG2M+TVx5yYBdhqnE1K92RTlfBa2NA=
```

LISTING 4.9: Headers send with the register request.


```
1 {"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}, "payload"
    ↪ : {"response": "{\"id\": \"5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY\", \"
    ↪ rawId\": \"5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY\", \"response\": {\"
    ↪ clientDataJSON\": \"
    ↪ eyJ0eXBlIjogIndlYmF1dGhuLmNyZWF0ZSIsICJjaGFsbGVuZ2UiOiAiUzFhcTNGbXoxcWF4ekw1cnYwZE14bEhnekxHallWV2xEZ
    ↪ \", \"attestationObject\": \"
    ↪ o2NmbXRkbm9uZWdhdHRTdG10oGhhdXRoRGF0YVikgh8LzXo6LfwnYQzR6rDVRNpPzX7RkQQxuk9PNiWa4qRBAAAAAAAAAAAAAAAAA
    ↪ -aJXyWq5GgItjSvvI94oR3KgbNrWdS3oWX2N-c\"}, \"type\": \"public-key\"}", "
    ↪ metadata": "{\"version\": \"1.0\", \"create_location\": \"testing\", \"
    ↪ username\": \"gramthanos@gmail.com\", \"origin\": \"https://unipi.gr\"}"}}
```


LISTING 4.10: Data send with the register request.

```
1 {"Response": "Successfully processed registration response"}
```

LISTING 4.11: Response returned to the register request.

The next action for the test client, now that it possess valid credentials, is to test the authentication, string by contacting the preauthenticate endpoint. The endpoint is located at:
`https://localhost:8181/sbfs/rest/preauthenticate`

```
1 Accept: application/json
2 Content-Type: application/json
3 User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
4 strongbee-content-sha256: f4L+ea4Dpm/gGjNbUJ9N0O0G+z1JfYNf+2COTYk/6rA=
5 Date: Mon, 21 Jan 2022 15:46:42 UTC
6 strongbee-api-version: SK3_0
7 Authorization: HMAC b55938050b05019a:y5vE/EPeEIxZPJd7g4w93Vkt3MLzAJyb6xtoDGvtxbk=
```

LISTING 4.12: Headers send with the preauthenticate request.

```
1 {"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}, "payload"
    ↪ : {"username": "gramthanos@gmail.com", "options": "{}", "extensions": "{}"}}
```

LISTING 4.13: Data send with the preauthenticate request.

```
1  {
2      "Response": {
3          "challenge": "_duBFDoIn-GY9I8tkUa-OtyWe-KRbmLGjnaIljmTACc",
4          "rpId": "unipi.gr",
5          "allowCredentials": [{
6              "type": "public-key",
7              "id": "5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY"
8          }],
9          "timeout": 60000
10     }
11 }
```

LISTING 4.14: Response returned to the preauthenticate request.

The test client script used the provided challenge from the preauthenticate endpoint to sign it and generate valid authenticator response.

```
1  {
2      'id': '5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY',
3      'rawId': '5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY',
4      'response': {
5          'authenticatorData': 'gh8LzXo6LfwnYQzR6rDVRNpPzX7RkQQxuk9PNiWa4qQBAAAAAQ',
6          'clientDataJSON': '
               ↪ eyJ0eXBlIjogIndlYmF1dGhuLmdldCIsICJjaGFsbGVuZ2UiOiAiX2R1QkZEb0luLUdZOUk4dGtVYS1PdHlXZS1LU
               ↪ ',
7          'signature': 'MEQCIGM49HHrIkBDVL-8
               ↪ G4UEJIblJ40ce_4z61D1rIHHX8LSAiA7vRfwINaiBaPJ48bTgSyXDWy3y-
               ↪ O6YVExAWPUOUik9A',
8          'userHandle': 'Z3JhbXRoYW5vc0BnbWFpbC5jb20'
9      },
10     'type': 'public-key'
11 }
```

LISTING 4.15: Credentials get, authenticator response generated by test client.

Now the test client has to send the generated response to the server through the authenticate endpoint, in check the user's identity. The endpoint is located at:
`https://localhost:8181/sbfs/rest/authenticate`

```
Accept: application/json
Content-Type: application/json
User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
strongbee-content-sha256: wz71VYcqw3ExpLzL/QfAoMs4vz6xoY9jRtOroHzfNWg=
Date: Mon, 21 Jan 2022 15:46:42 UTC
strongbee-api-version: SK3_0
Authorization: HMAC b55938050b05019a:MM1aayY8mRDRL+7RYhWSwRI6VaMgim6gzOkV84AL4eo=
```

LISTING 4.16: Headers send with the authenticate request.


```
{"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}, "payload"
    ↪ : {"response": "{\"id\": \"5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY\", \"
    ↪ rawId\": \"5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY\", \"response\": {\"
    ↪ authenticatorData\": \"gh8LzXo6LfwnYQzR6rDVRNpPzX7RkQQxuk9PNiWa4qQBAAAAAQ\",
    ↪ \"clientDataJSON\": \"
    ↪ eyJ0eXBlIjogIndlYmF1dGhuLmdldCIsICJjaGFsbGVuZ2UiOiAiX2R1QkZEb0luLUdZOUk4dGtVYS1PdHlXZS1LUmJtTEdqbmFJb
    ↪ \", \"signature\": \"MEQCIGM49HHrIkBDVL-8
    ↪ G4UEJIblJ40ce_4z61D1rIHHX8LSAiA7vRfwINaiBaPJ48bTgSyXDWy3y-O6YVExAWPUOUik9A\",
    ↪ \"userHandle\": \"Z3JhbXRoYW5vc0BnbWFpbC5jb20\"}, \"type\": \"public-key\"}",
    ↪ "metadata": "{\"version\": \"1.0\", \"last_used_location\": \"testing\", \"
    ↪ username\": \"gramthanos@gmail.com\", \"origin\": \"https://unipi.gr\"}"}}
```


LISTING 4.17: Data send with the authenticate request.


```
{"Response": "Successfully authenticated key"}
```


LISTING 4.18: Response returned to the authenticate request.

The test client continues by testign the getkeysinfo endpoint, which returns the credentials bind to a user account. The endpoint is located at:
`https://localhost:8181/sbfs/rest/getkeysinfo`

```
Accept: application/json
Content-Type: application/json
User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
strongbee-content-sha256: bGNXLnxROcNpSaTiqAh3kfykOFKYOrPtfzANvuhv/IE=
Date: Mon, 21 Jan 2022 15:46:42 UTC
strongbee-api-version: SK3_0
Authorization: HMAC b55938050b05019a:UAe18/JGZlsvMJwZTMFXqigDgXkXaSiW9WiOhfwadi8=
```

LISTING 4.19: Headers send with the getkeysinfo request.


```
{"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}, "payload"
    ↪ : {"username": "gramthanos@gmail.com"}}
```


LISTING 4.20: Data send with the getkeysinfo request.


```
{
    "Response": {
        "keys": [{
            "randomid": "5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY",
            "randomid_ttl_seconds": "9999",
            "fidoProtocol": "FIDO2_0",
            "fidoVersion": "FIDO2_0",
            "createLocation": "testing",
            "createDate": "1647877602918",
            "lastusedLocation": "testing",
            "modifyDate": "1647877602918",
            "status": "Active",
            "displayName": "gramthanos@gmail.com"
        }]
```

```
15     }
16 }
```

LISTING 4.21: Response returned to the getkeysinfo request.

Finally, the test client test the deregister endpoint, which removes credentials from a user account, thus deregistering the linked authenticator device. The endpoint is located at:
`https://localhost:8181/sbfs/rest/deregister`

```
1 Accept: application/json
2 Content-Type: application/json
3 User-Agent: StrongMonkey-Agent/v0.0.4-beta.strongbee
4 strongbee-content-sha256: qdi0Whdrq8X9NRVoJnNFcZ6qS7+zj+zFnwYGEfNSNNU=
5 Date: Mon, 21 Jan 2022 15:46:42 UTC
6 strongbee-api-version: SK3_0
7 Authorization: HMAC b55938050b05019a:cUSY274uTxDIcTWrvKqHslArzM4ot4RqzB4qVPi9eEs=
```

LISTING 4.22: Headers send with the deregister request.

```
1 {"svcinfo": {"did": "unipi.gr", "protocol": "FIDO2_0", "authtype": "HMAC"}, "payload"
    ↪ : {"keyid": "5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY"}}
```

LISTING 4.23: Data send with the deregister request.

```
1 {
2     "Response": {
3         "randomid": "5A1IafTPRQuURIWGmDm_SwQcjjq_y7E3D9S4TzTRbiY",
4         "randomid_ttl_seconds": "9999",
5         "fidoProtocol": "FIDO2_0",
6         "fidoVersion": "FIDO2_0",
7         "createLocation": "testing",
8         "createDate": "1647877602918",
9         "lastusedLocation": "testing",
10        "modifyDate": "1647877602918",
11        "status": "Active",
12        "displayName": "gramthanos@gmail.com"
13    }
14 }
```

LISTING 4.24: Response returned to the deregister request.

4.2.4 Conclusions

Our implementation was found to be a light and easy to use fully featured FIDO2/WebAuthn server, ideal for small projects. Since it is written in Python, based on modern web technologies, we expect that more expect developers will be able to use it. Furthermore, as the implementation is open sourced and publicly available in GitHub, anyone from the community can further extend the server's functionalities and improve it.

4.3 FIDO2 authentication for OpenVPN

Implementing strong authentication methods should not be limited only within the web environment. Thus, in this section, we will present how one can improve both the security and the usability of a Virtual Private Network (VPN) service by

allowing the users to authenticate through FIDO2/WebAuthn. Moreover, our solution moved the client's authentication to the web which opens up more possibilities for user authentication. By doing that, apart from the usage of FIDO/WebAuthn, an organisations would be able to leverage existing Single-Sign-On (SSO) services or implement Multi Factor Authentication (MFA) with the VPN service.

4.3.1 Introduction

One of the most popular methods for establishing a secure and trusted communication channel is the deployment of a VPN. As shown in Figure 4.13, by setting up a remote access VPN, organisations can safely provide access to internal services (located inside an organisation's intranet) to employees working remotely, as if they were located at the office. At the same time VPN protects the user's network traffic by encrypting it, thus its use is of high importance when the user is accessing the internet from a public wireless network. To offer these services, VPN solutions usually support multiple encryption algorithms to protect their traffic and typical authentication mechanics for mutual authentication of both the server and the client.

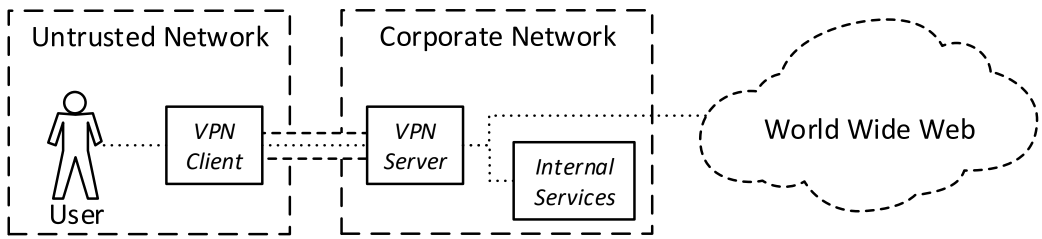


FIGURE 4.13: Example usage of a VPN.

During the COVID-19 pandemic, VPN services played an important role, as they allowed organisations to continue their operation by having employees working remotely in a secure way. As suggested also by ENISA in its press release regarding teleworking tips[18], the use of VPN is recommended so as to protect the information exchanged when working from home and over untrusted networks.

To access such a VPN service, the client has to authenticate herself/himself with the server, so that only the authorised users can use the service. This is usually done through a username and password or a password protected certificate. Furthermore, the client has to authenticate the VPN server in order to mitigate man in the middle attacks. This is also done using certificates. This user authentication should be adequately secure in order to protect the services and data to which the VPN provides access. Hence, a password authentication does not always meets the security requirements needed. For this reason, the last years, second factor authentication mechanics has started appearing on VPN servers, though sometimes in expense of usability.

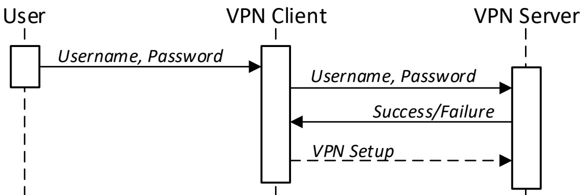


FIGURE 4.14: Traditional VPN authentication through username and password.

4.3.2 Implementation

Traditionally, as shown on Figure 4.14, to authenticate with a VPN service, a client has to send 2 values to the server, a username (to specify the account bind to the user's identity) and then send a secret password to prove its identity. We altered this authentication mechanic, and mixed it with the OpenID Connect (OIDC)[19] flow (widely used in web services) so that we can authenticate the user on the web and then share his identity to the VPN server. Furthermore, through OIDC, as presented on Figure 4.15 we can deploy a WebAuthn password-less authentication and eliminate the need for passwords or use it as a Two Factor Authentication (2FA) to strengthen security.

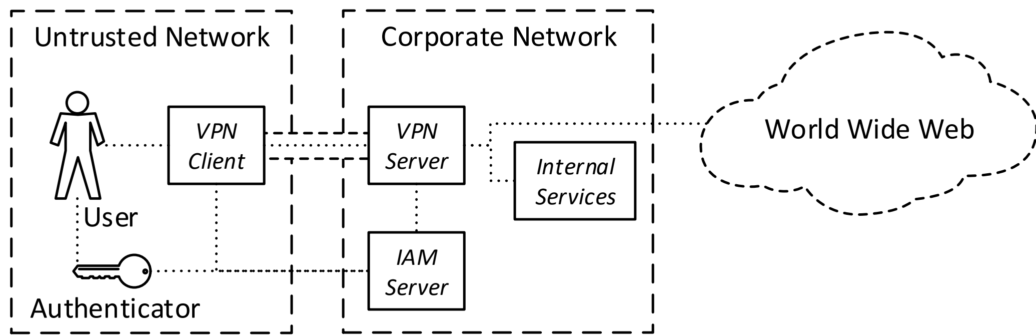


FIGURE 4.15: Example usage of a VPN with FIDO2/WebAuthn.

As shown on Figure 4.16, the user starts by launching the VPN client, inserting the account username and clicking to authenticate. The VPN client then launches a browser and redirects the user to the organisation's identity management service that supports OIDC and FIDO2/WebAuthn authentication. The user uses a FIDO authenticator device to authenticate and then the service sends back the authentication OIDC response back to the VPN client. The VPN client then forwards the received information as a password to the VPN service. The VPN server retrieves the OIDC response and contacts the OIDC SSO service endpoints to recover the client's identity and assess whether the authentication was successful. In this authentication scenario, the VPN server is treated as a 3rd party application registered to the OIDC service and the whole process is based on the OAuth 2.0[20] protocol for authentication.

[18] Tips for cybersecurity when working from home, https://www.enisa.europa.eu/tips-for-cybersecurity-when-working-from-home

[19] https://openid.net/connect/

[20] https://oauth.net/2/

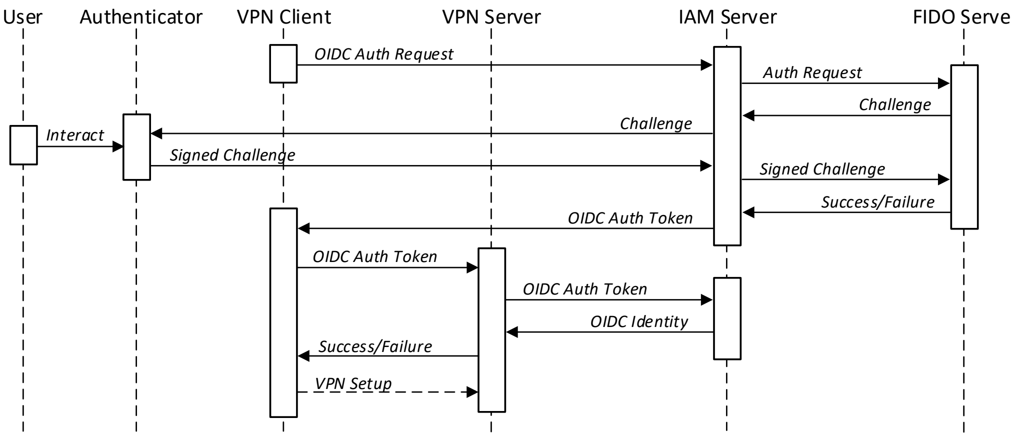


FIGURE 4.16: VPN authentication through OIDC and FIDO2.

To create a proof-of-concept, we developed a VPN client desktop application based on Electron interacting with an OpenVPN community edition[21] client to set up the service. Furthermore, for the backend, we used an OpenVPN server and a custom authentication script was developed in python to complete the OIDC flow and assess the authentication response by contacting the SSO service. As an identity management service, the KeyCloak open source server was used, configured for password-less authentication with FIDO2/WebAuthn. This implementation is publicly available on GitHub[22] as open source software. Additionally the solution can be configured to work with more multi factor authentication mechanics to strengthen the security.

Our implementation can be divided into 2 parts, the server side extension of the OpenVPN server with the use of an authentication Python script and the client application. Appendix D lists the 2 most important codes of the implementation, the server authentication Python code (Listing D.1) as well as the main client application JavaScript code (Listing D.2). Their functionality will be explained briefly in the following paragraphs.

The server side Python script is called by OpenVPN when the client sends the authentication credentials, in the form of a username and a password. By configuring such alternative authentication methods[23], one can extend the functionalities of OpenVPN and this is what we have done. The script takes from the environment the password, expected to be a JSON code encoded in base64 and decodes it. Then it retrieves the service id (e.g. keycloak-oidc) and the authentication code. Using these info, the script contacts the service's endpoint passing the given code and retrieving the user's identity information. The script could be further extended to check whether the authenticated user has permissions to use the service.

The JavaScript script is part of the Electron application. The script is responsible for handling both the UI (through interactions with the DOM) as well as the interaction with the system's OpenVPN client and the embedded browser (used to communicate with the identity management server). The script will start by loading the appropriate configuration files which they hold the endpoint information of the identity management servers supported.

[21] https://openvpn.net/community/
[22] https://github.com/GramThanos/vpn-oidc
[23] https://openvpn.net/community-resources/using-alternative-authentication-methods/

Upon a user initiation of the authentication process, the script will create a new embedded browser window and start the OIDC process. To retrieve the authentication code, the return URL of the OIDC will be hijacked by the application. Then the authentication code will be formatted along with the service id as JSON and encoded to base64 which will be sent to the server as a password. As a username, a random string in combination with the service id will be used. We have to note that since we are using the password to pass the information, we may be limited by the maximum password length supported by the server. In the future, we may improve the packing of the information to ensure the limit is not reached.

4.3.3 Usage

After launching our VPN client application, shown on Figure 4.17, the user has to select the appropriate authentication method of his/her preference. In our implementation we added our custom Keycloak authentication service and 2 additional commercial SSO services (Google and Facebook).

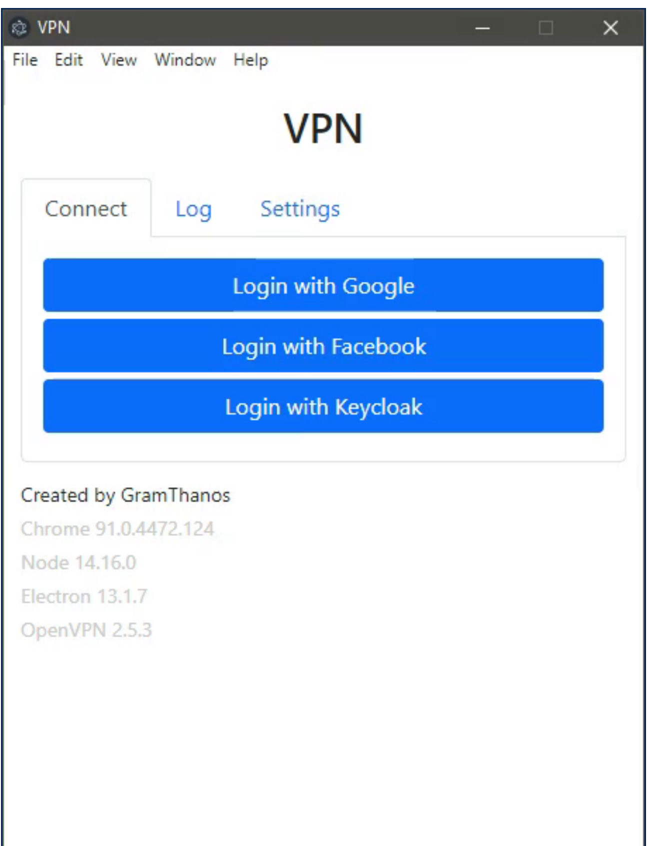


FIGURE 4.17: The user interface of the VPN client application.

Following the selection of the authentication service (in our example usecase, our custom KeyCloak service) a new web browser window will open, as shown in Figure 4.18. Depending on the authentication flow of the server, the user will have to provide the appropriate credentials to be authenticated. Our KeyCloak server was configured to authenticate the user password-less through WebAuthn. Thus, the service will ask the user to fill in the user's username. In Figure 4.19 is shown how the request from the server to authenticate the user using his/her FIDO security key.

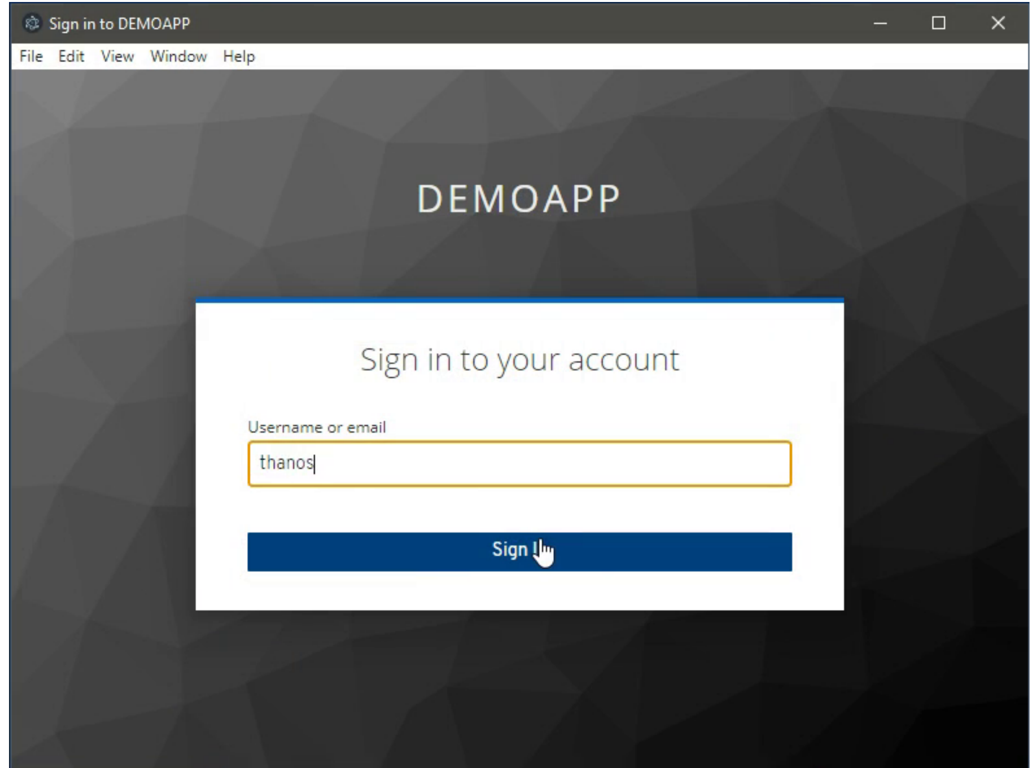


FIGURE 4.18: The client launch a browser to authenticate the user through an OIDC service.

The user will then have to plug in his/her authenticator device, or use the embedded one of their platform. Figure 4.20 shows the user providing a PIN to allow Windows Hello authenticator device to sign the request, and which will be checked by the server. Upon successful authentication the client will forward the returned access token to the VPN server and after if valid, as shown on Figure 4.21, the user will be connected to the VPN service.

Through the user interface the user may close the connection and start a new one on demand.

4.3.4 Conclusions

VPN services are nowadays essential, especially since they are an important component to ensure the business continuity of organisations. By leveraging OIDC our VPN authentication can authenticate users securely through SSO technologies, without the need to create special VPN accounts. By enabling users to authenticate using FIDO2/WebAuthn we improve even more the authentication mechanic by increasing the security and user experience eliminating the need for passwords.

Due to security and usability concerns, it is expected that at the future similar authentication flows will be included out of the box on most of the VPN solutions, as it is a more secure, practical and user friendly approach to authenticate users. FIDO authentication mechanics availability will increase as its specification is adopted by more and more solutions.

Our implementation, could be further developed to improve its configuration, graphical interface, installation process and cross platform support. One of the main future steps of this work could be porting the application in other operating systems.

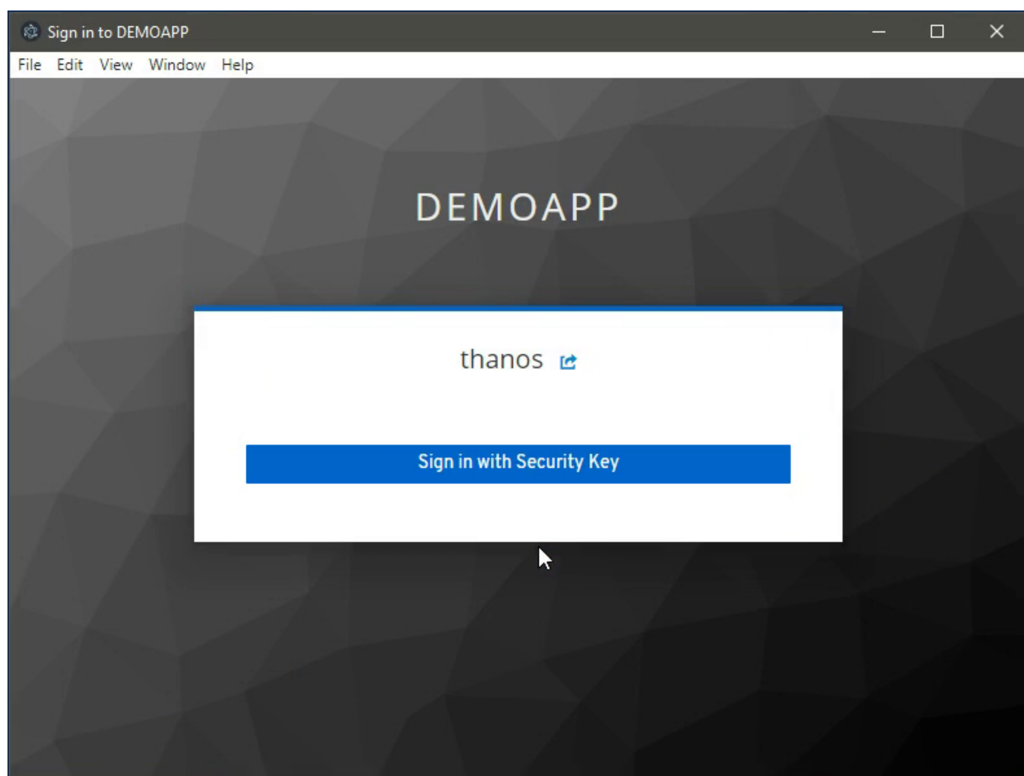


FIGURE 4.19: KeyCloak configured to authenticate the user using FIDO2/WebAuthn.

In my opinion, targeting mobile platforms (i.e. Android, iOS) to enable the use of bio-metric authentication on VPN is of high importance.

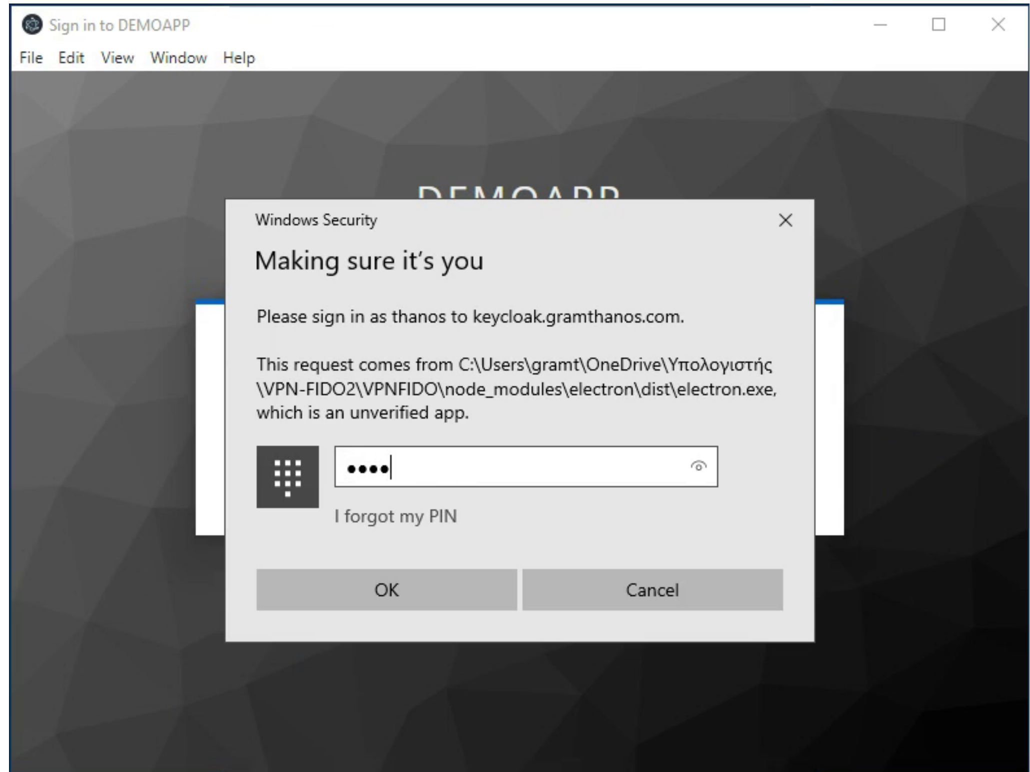


FIGURE 4.20: The user authenticates using the Windows Hello WebAuthn embedded authenticator using a PIN.

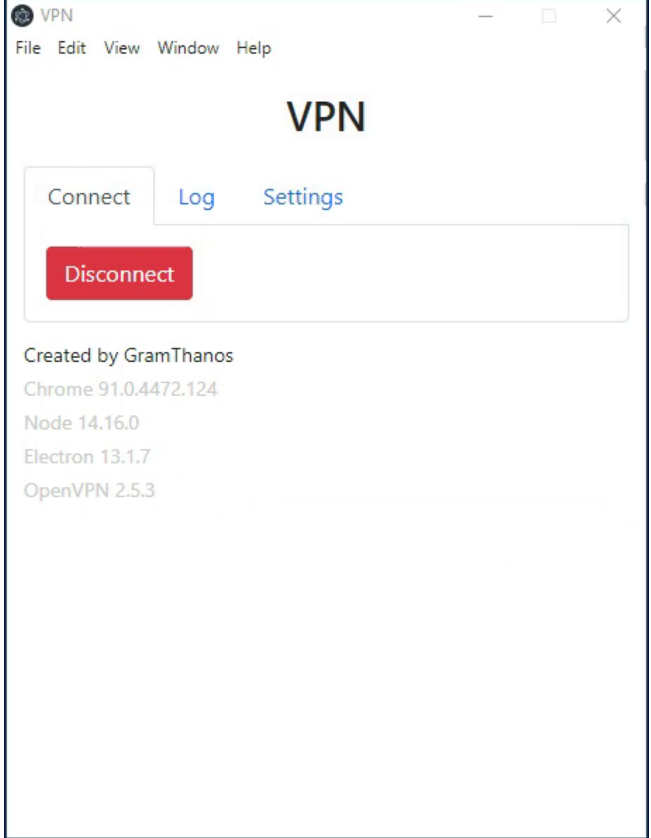


FIGURE 4.21: The user connected to the VPN service after successful authentication.

Chapter 5

Conclusion

FIDO2 through WebAuthn brought secure password-less authentication on the web. Through easy to use mechanics, users are now able to use secure public private key cryptography to authenticate on the web. The technology was found to be quite superior to that of username and password as it more secure and easier to use especially when combined with biometric authentication.

One of the most important FIDO2/WebAuthn features is that it offers something that other authentication mechanics are lacking, it is phishing resistant. This feature makes it ideal for use when interacting with online payment services. This will mitigate attacks exploiting the human factor such as phishing attacks, that is currently one of the weak points on online banking services and PSD2 failed to address.

On the other hand, as one can understand, FIDO2/WebAuthn's complexity makes it difficult to develop FIDO2 services without the use of a specialised FIDO2 server or a FIDO2/WebAuthn library, and even with the use of 3rd party software to handle the verification process, still, services may be left vulnerable due to faulty configuration of the solution.

In this work, we showed how FIDO works internally and how FIDO2 through WebAuthn is able to offer secure and easy to use authentication on the web. Furthermore, we analysed the strong client authentication (SCA) requirements introduced by the PSD2 and how FIDO covers these needs, but also what problems may one face when trying to apply FIDO2/WebAuthn for SCA. Based on our analysis, the FIDO2/WebAuthn is compiant with the PSD2 but only when used with compliant authenticator devices, thus payment services should leverage the FIDO Alliance Metadata Service in order to identify authenticator devices that meet their policy requirements. Furthermore, in an effort to support the faster adoption of WebAuthn (and thus FIDO2) we released a number of open source related solutions.

Our StrongMonkey SDK allows applications written in PHP or Python to connect easily to FIDO2 servers supporting the StrongKey FIDO2 Server API. Additionally, we also presented StrongBee, our own light implementation of the StrongKey FIDO2 Server API serving as an alternative FIDO2/WebAuthn solution for smaller projects or for use on testing environments. Lastly, we presented a novel PoC implementation of a VPN service secured through a FIDO2/WebAuthn authentication based on OIDC.

All the developed solutions were released as open source software on GitHub, allowing anyone to further advance them. We hope the community to be benefit from our implementations and leverage them to provide secure authentication mechanics and services to the general public.

Appendix A

FIDO Metadata Filtering App

The FIDO Alliance maintains a metadata services that includes a wide variety of FIDO authenticator devices. The service lists for each authenticator information and their characteristics. Such metadata are essential for authenticating an authenticator device and building trust between the authenticator device and the relying party service.

The metadata service offers a JSON Web Token (JWT) file with all the information encoded and signed inside. In order to look into what authenticators are available based on their characteristics, a simple web application able to filtered them was developed. The source code of the application is available on GitHub[1] while a live version of the webpage is also hosted on GitHub[2].

The application automatically loads the JWT file form provided from the FIDO Alliance metadata service and decode it on the background. As presented in Figure A.1 and Figure A.2, the user is able to select filters based on the authenticator's family protocol, latest status report (e.g. certification), cryptographic strength, protection mechanics for the key generation and matching, as well as the available user verification method.

The application automatically updates the results at the bottom as soon as the user alters the filters. For each authenticator the authenticator's AAGUID, KeyIdentifiers or AAID is shown along with the authenticator's description and any given image (usually the logo of the manufacturer).

[1] https://github.com/GramThanos/FIDO-Authenticator-Metadata-Filters
[2] https://gramthanos.github.io/FIDO-Authenticator-Metadata-Filters/

FIDO Authenticator Metadata Filters

nextUpdate	2022-04-01 up-to-date	get
protocolFamily	☐ uaf ☐ u2f ☑ fido2	clear
statusReport (latest)	☐ NOT_FIDO_CERTIFIED ☐ FIDO_CERTIFIED ☐ USER_VERIFICATION_BYPASS ☐ ATTESTATION_KEY_COMPROMISE ☐ USER_KEY_REMOTE_COMPROMISE ☐ USER_KEY_PHYSICAL_COMPROMISE ☐ UPDATE_AVAILABLE ☐ REVOKED ☐ SELF_ASSERTION_SUBMITTED ☑ FIDO_CERTIFIED_L1 ☑ FIDO_CERTIFIED_L1plus ☑ FIDO_CERTIFIED_L2 ☑ FIDO_CERTIFIED_L2plus ☑ FIDO_CERTIFIED_L3 ☑ FIDO_CERTIFIED_L3plus	clear
cryptoStrength	☐ 128 ☐ 256 ☐ 512	clear
keyProtection	☐ software ☐ hardware ☑ tee ☑ secure_element ☐ remote_handle OR mode	clear
matcherProtection	☐ software ☑ tee ☑ on_chip OR mode	clear
userVerificationMethod	☐ presence_internal ☑ fingerprint_internal ☐ passcode_internal ☐ voiceprint_internal ☐ faceprint_internal ☐ location_internal ☐ eyeprint_internal ☐ pattern_internal ☐ handprint_internal ☐ passcode_external ☐ pattern_external ☐ none ☐ all AND mod	clear
Results (11)	[39a5647e-1853-446c-a1f6-a79bae9f5bc7] Vancosys Android Authenticator v2 [820d89ed-d65a-409e-85cb-f73f0578f82a] Vancosys iOS Authenticator v2 [d821a7d4-e97c-4cb6-bd82-4237731fd4be] Hyper FIDO® Bio Security Key v1 HYPERSECU [b93fd961-f2e6-462f-b122-82002247de78] Android Authenticator with SafetyNet Attestation v1 [9ddd1817-af5a-4672-a2b9-3e3dd95000a9] Windows Hello VBS Hardware Authenticator v1 [12ded745-4bed-47d4-abaa-e713f51d6393] Feitian AllinOne FIDO2 Authenticator v1 FEITIAN [83c47309-aabb-4108-8470-8be838b573cb] YubiKey Bio Series (Enterprise Profile) v328965 [8c97a730-3f7b-41a6-87d6-1e9b62bda6f0] FT-JCOS FIDO Fingerprint Card v1 FEITIAN [a1f52be5-dfab-4364-b51c-2bd496b14a56] OCTATCO EzFinger2 FIDO2 AUTHENTICATOR v5 [d41f5a69-b817-4144-a13c-9ebd6d9254d6] ATKey.Card CTAP2.0 v2 [77010bd7-212a-4fc9-b236-d2ca5e9d4084] Feitian BioPass FIDO2 Authenticator v1 FEITIAN	

Copyright © 2022, Athanasios Vasileios Grammatopoulos

FIGURE A.1: Found 11 certified FIDO2 authenticator devices protected by hardware mechanics with fingerprint detection capabilities.

FIDO Authenticator Metadata Filters

nextUpdate	2022-04-01 up-to-date	get
protocolFamily	☐ uaf ☐ u2f ☑ fido2	clear
statusReport (latest)	☐ NOT_FIDO_CERTIFIED ☐ FIDO_CERTIFIED ☐ USER_VERIFICATION_BYPASS ☐ ATTESTATION_KEY_COMPROMISE ☐ USER_KEY_REMOTE_COMPROMISE ☐ USER_KEY_PHYSICAL_COMPROMISE ☐ UPDATE_AVAILABLE ☐ REVOKED ☐ SELF_ASSERTION_SUBMITTED ☑ FIDO_CERTIFIED_L1 ☑ FIDO_CERTIFIED_L1plus ☑ FIDO_CERTIFIED_L2 ☑ FIDO_CERTIFIED_L2plus ☑ FIDO_CERTIFIED_L3 ☑ FIDO_CERTIFIED_L3plus	clear
cryptoStrength	☐ 128 ☐ 256 ☐ 512	clear
keyProtection	☐ software ☐ hardware ☑ tee ☑ secure_element ☐ remote_handle OR mode	clear
matcherProtection	☐ software ☑ tee ☑ on_chip OR mode	clear
userVerificationMethod	☐ presence_internal ☐ fingerprint_internal ☐ passcode_internal ☐ voiceprint_internal ☑ faceprint_internal ☐ location_internal ☐ eyeprint_internal ☐ pattern_internal ☐ handprint_internal ☐ passcode_external ☐ pattern_external ☐ none ☐ all AND mod	clear
Results (4)	[39a5647e-1853-446c-a1f6-a79bae9f5bc7] Vancosys Android Authenticator v2 [820d89ed-d65a-409e-85cb-f73f0578f82a] Vancosys iOS Authenticator v2 [b93fd961-f2e6-462f-b122-82002247de78] Android Authenticator with SafetyNet Attestation v1 [9ddd1817-af5a-4672-a2b9-3e3dd95000a9] Windows Hello VBS Hardware Authenticator v1	

Copyright © 2022, Athanasios Vasileios Grammatopoulos

FIGURE A.2: Found 4 certified FIDO2 authenticator devices protected by hardware mechanics with face detection capabilities.

Appendix B

StrongMonkey Implementation Code

In Section 4.1 we presented our StrongMonkey SDK that can be used to connect an application with a FIDO2 server in order to support WebAuthn authentication easily. As we mentioned the full code is publicly available at GitHub[1].

In this section we will list the 2 libraries developed, the Python library's code (Listing B.1) as well as the Python PHP's code (Listing B.2). Additionally, a UML diagram for the StrongMonkey PHP library if provided in Figure B.1.

```
1  #!/usr/bin/python3
2  #
3  # StrongMonkey v0.0.4-beta
4  # Python SDK for interacting with FIDO2 Server API v3.0.0
5  # Copyright (c) 2020 Grammatopoulos Athanasios-Vasileios
6  #
7
8  import os
9  import json
10 import base64
11 import hmac
12 import hashlib
13 import datetime
14 import requests
15 import sys
16
17 STRONGMONKEY_VESION = 'v0.0.4-beta';
18 STRONGMONKEY_DEBUG = False;
19 STRONGMONKEY_CONNECTTIMEOUT = 10;
20 STRONGMONKEY_TIMEOUT = 30;
21 STRONGMONKEY_USERAGENT = 'StrongMonkey-Agent' + '/' + STRONGMONKEY_VESION;
22
23 class StrongMonkey:
24
25     # Static variables
26     api_protocol = 'FIDO2_0';
27     api_version = 'SK3_0';
28     api_url_base = '/skfs/rest';
29     version = STRONGMONKEY_VESION;
30     useragent = STRONGMONKEY_USERAGENT;
31
32     # ERRORS
33     PARSE_ERROR = 1001;
34     SUBMIT_ERROR = 1002;
35     AUTHENTICATION_FAILED = 1003;
36     RESOURCE_UNAVAILABLE = 1004;
37     UNEXPECTED_ERROR = 1005;
```

[1] https://github.com/GramThanos/StrongMonkey

```
38      UNUSED_ROUTES = 1006;
39      UNKNOWN_ERROR = 1007;
40
41      # Authorization Methods
42      AUTHORIZATION_HMAC = 'HMAC';
43      AUTHORIZATION_PASSWORD = 'PASSWORD';
44      # Protocol Methods
45      PROTOCOL_REST = 'REST';
46
47
48      def __init__ (self, hostport, did, protocol, authtype, keyid, keysecret):
49          # TODO: Test inputs? No?
50          # Save information
51          self.hostport = hostport
52          self.did = did
53          self.protocol = protocol
54          self.authtype = authtype
55          self.keyid = keyid
56          self.keysecret = keysecret
57
58          # Check if not supported
59          if (authtype != StrongMonkey.AUTHORIZATION_HMAC and authtype != StrongMonkey.
              ↪ AUTHORIZATION_PASSWORD):
60              print('The provided authorization method is not supported')
61          if (protocol != StrongMonkey.PROTOCOL_REST):
62              print('The provided protocol is not supported')
63
64      def preregister (self, username, displayname=None, options=None, extensions=None)
            ↪ :
65          # Init parameters
66          if (displayname is None):
67              displayname = username
68          options = self.jsonStringPrepare(options, {})
69          extensions = self.jsonStringPrepare(extensions, {})
70
71          # Create data
72          payload = {
73              'username' : username,
74              'displayname' : displayname,
75              'options' : options,
76              'extensions' : extensions
77          };
78
79          # Make preregister request
80          return self.request(payload, '/preregister');
81
82      def register (self, response, metadata=None):
83          # Init empty parameters
84          response = self.jsonStringPrepare(response)
85          metadata = self.jsonStringPrepare(metadata, {})
86
87          # Create data
88          payload = {
89              'response' : response,
90              'metadata' : metadata
91          }
92
93          # Make register request
94          return self.request(payload, '/register')
95
96      def preauthenticate (self, username=None, options=None, extensions=None):
97          # Init empty parameters
```

```
98          options = self.jsonStringPrepare(options, {})
99          extensions = self.jsonStringPrepare(extensions, {})
100 
101         # Create data
102         payload = {
103             'username' : username,
104             'options' : options,
105             'extensions' : extensions
106         }
107 
108         # Make preauthenticate request
109         return self.request(payload, '/preauthenticate')
110 
111     def authenticate (self, response, metadata=None):
112         # Init empty parameters
113         response = self.jsonStringPrepare(response)
114         metadata = self.jsonStringPrepare(metadata, {})
115 
116         # Create data
117         payload = {
118             'response' : response,
119             'metadata' : metadata
120         }
121 
122         # Make authenticate request
123         return self.request(payload, '/authenticate')
124 
125     def updatekeyinfo (self, status, modify_location, displayname, keyid):
126         # Create data
127         payload = {
128             "status" : status,
129             "modify_location" : modify_location,
130             "displayname" : displayname,
131             "keyid" : keyid
132         }
133 
134         # Make updatekeyinfo request
135         return self.request(payload, '/updatekeyinfo')
136 
137     def getkeysinfo (self, username):
138         # Create data
139         payload = {
140             "username" : username
141         }
142 
143         # Make getkeysinfo request
144         return self.request(payload, '/getkeysinfo')
145 
146     def deregister (self, keyid):
147         # Create data
148         payload = {
149             "keyid" : keyid
150         }
151 
152         # Make deregister request
153         return self.request(payload, '/deregister')
154 
155     def ping (self):
156         # Make ping request
157         response = self.request(None, '/ping', False)
158         # If no error
159         if (response['code'] == 200):
```

```
160             return response['body']
161 
162         # Return error code
163         return self.parseResponse(response['code'], response['body'])
164 
165     def request (self, payload, action_path, parse=True):
166         global STRONGMONKEY_DEBUG, STRONGMONKEY_CONNECTTIMEOUT, STRONGMONKEY_TIMEOUT
167         # Create data
168         body = {
169             "svcinfo" : {
170                 "did" : self.did,
171                 "protocol" : StrongMonkey.api_protocol,
172                 "authtype" : self.authtype
173             }
174         }
175         # Prepare payload
176         if not (payload is None):
177             body['payload'] = payload
178 
179         # Generate path
180         path = StrongMonkey.api_url_base + action_path
181 
182         # Prepare Request Headers
183         headers = {
184             'Accept': 'application/json',
185             'Content-Type': 'application/json',
186             'User-Agent': StrongMonkey.useragent
187         }
188 
189         # HMAC
190         if (self.authtype == StrongMonkey.AUTHORIZATION_HMAC):
191             # Get date
192             date = datetime.datetime.now(datetime.timezone.utc).strftime("%a, %-d %b %
                 ↪ Y %H:%M:%S %Z")
193 
194             # Prepare hashes
195             payload_hash = ''
196             mimetype = ''
197             if not (payload is None):
198                 payload_string = json.dumps(body['payload'],separators=(',', ':'))
199                 payload_hash = hashlib.sha256(payload_string.encode()).digest()
200                 payload_hash = base64.b64encode(payload_hash).decode()
201                 mimetype = 'application/json'
202 
203             # Generate HMAC authentication
204             authentication_hash = self.generateHMAC('POST', payload_hash, mimetype,
                 ↪ date, path)
205 
206             # Add authorization Headers
207             headers['strongkey-content-sha256'] = payload_hash
208             headers['Date'] = date
209             headers['strongkey-api-version'] = StrongMonkey.api_version
210             headers['Authorization'] = authentication_hash
211         # Credentials
212         else:
213             body['svcinfo']['svcusername'] = self.keyid
214             body['svcinfo']['svcpassword'] = self.keysecret
215 
216         # Create request
217         reqOptions = {
218             'url' : self.hostport + path,
219             'verify' : True,
```

```
220             'data' : json.dumps(body),
221             'headers' : headers,
222             'timeout' : STRONGMONKEY_TIMEOUT
223         }
224         if (STRONGMONKEY_DEBUG):
225             requests.packages.urllib3.disable_warnings()
226             reqOptions['verify'] = False
227         ch = requests.post(
228             reqOptions['url'],
229             verify = reqOptions['verify'],
230             data = reqOptions['data'],
231             headers = reqOptions['headers'],
232             timeout = reqOptions['timeout']
233         )
234         response = ch.text
235         response_code = ch.status_code
236 
237         if (parse):
238             return self.parseResponse(response_code, response)
239         else:
240             return {
241                 'code' : response_code,
242                 'body' : response
243             }
244 
245     def parseResponse (self, code, response):
246         # 200: Success
247         if (code == 200):
248             try:
249                 response = json.loads(response)
250                 return response
251             except ValueError:
252                 return StrongMonkey.PARSE_ERROR
253         # 400: There was an error in the submitted input.
254         if (code == 400):
255             return StrongMonkey.SUBMIT_ERROR;
256         # 401: The authentication failed.
257         if (code == 401):
258             return StrongMonkey.AUTHENTICATION_FAILED;
259         # 404: The requested resource is unavailable.
260         if (code == 404):
261             return StrongMonkey.RESOURCE_UNAVAILABLE
262         # 500: The server ran into an unexpected exception.
263         if (code == 500):
264             return StrongMonkey.UNEXPECTED_ERROR
265         # 501: Unused routes return a 501 exception with an error message.
266         if (code == 501):
267             return StrongMonkey.UNUSED_ROUTES
268         return StrongMonkey.UNKNOWN_ERROR
269 
270     def getError (self, error):
271         # If not error
272         if (not isinstance(error, int)):
273             return False
274         # Resolve error code
275         if error == StrongMonkey.PARSE_ERROR:
276             return 'StrongMonkey: Response parse error.'
277         if error == StrongMonkey.SUBMIT_ERROR:
278             return 'StrongMonkey: There was an error in the submitted input.'
279         if error == StrongMonkey.AUTHENTICATION_FAILED:
280             return 'StrongMonkey: The authentication failed.'
281         if error == StrongMonkey.RESOURCE_UNAVAILABLE:
```

```
282             return 'StrongMonkey: The requested resource is unavailable.'
283         if error == StrongMonkey.UNEXPECTED_ERROR:
284             return 'StrongMonkey: The server ran into an unexpected exception.'
285         if error == StrongMonkey.UNKNOWN_ERROR:
286             return 'StrongMonkey: Unused routes return a 501 exception with an error
                    ↪ message.'
287         return 'StrongMonkey: Unknown error code.'
288 
289     def generateHMAC (self, method, payload, mimetype, datestr, path):
290         # Assembly hash message
291         message = [
292             method,
293             payload,
294             mimetype,
295             datestr,
296             StrongMonkey.api_version,
297             path
298         ]
299         message = "\n".join(message)
300         # Generate HMAC
301         digest = hmac.new(bytes.fromhex(self.keysecret), msg = bytes(message , 'latin
                ↪ -1'), digestmod = hashlib.sha256).digest()
302         # Return header
303         return 'HMAC ' + self.keyid + ':' + base64.b64encode(digest).decode()
304 
305     def jsonStringPrepare (self, vjson, ifnull=None):
306         if ((vjson is None) and not (ifnull is None)):
307             vjson = ifnull
308         if (isinstance(vjson, str)):
309             return vjson
310         return json.dumps(vjson)
```

LISTING B.1: StrongMonkey SDK library for Python applications.

```
1  <?php
2  /**
3   * StrongMonkey v0.0.4-beta
4   * PHP SDK for interacting with FIDO2 Server API v3.0.0
5   * Copyright (c) 2020 Grammatopoulos Athanasios-Vasileios
6   */
7  
8  define('STRONGMONKEY_VESION', 'v0.0.4-beta');
9  if (!defined('STRONGMONKEY_DEBUG')) define('STRONGMONKEY_DEBUG', false);
10 if (!defined('STRONGMONKEY_CONNECTTIMEOUT')) define('STRONGMONKEY_CONNECTTIMEOUT',
       ↪ 10);
11 if (!defined('STRONGMONKEY_TIMEOUT')) define('STRONGMONKEY_TIMEOUT', 30);
12 if (!defined('STRONGMONKEY_USERAGENT')) define('STRONGMONKEY_USERAGENT', '
       ↪ StrongMonkey-Agent' . '/' . STRONGMONKEY_VESION);
13 
14 class StrongMonkey {
15 
16   // Static variables
17   private static $api_protocol = 'FIDO2_0';
18   private static $api_version = 'SK3_0';
19   private static $api_url_base = '/skfs/rest';
20   private static $version = STRONGMONKEY_VESION;
21   private static $useragent = STRONGMONKEY_USERAGENT;
22 
23   // ERRORS
24   private static $PARSE_ERROR = 1001;
25   private static $SUBMIT_ERROR = 1002;
26   private static $AUTHENTICATION_FAILED = 1003;
```

```
27    private static $RESOURCE_UNAVAILABLE = 1004;
28    private static $UNEXPECTED_ERROR = 1005;
29    private static $UNUSED_ROUTES = 1006;
30    private static $UNKNOWN_ERROR = 1007;
31
32    // Authorization Methods
33    private static $AUTHORIZATION_HMAC = 'HMAC';
34    private static $AUTHORIZATION_PASSWORD = 'PASSWORD';
35    // Protocol Methods
36    private static $PROTOCOL_REST = 'REST';
37
38    // Private variables
39    private $hostport;
40    private $did;
41    private $wsprotocol;
42    private $authtype;
43    private $keyid;
44    private $keysecret;
45
46    /**
47     * Create a StrongMonkey object which can later be used to communicate with the
          ↪ StrongKey FIDO2 Server
48     * @param string $hostport Host and port to access the FIDO SOAP and REST formats
49     * http://<FQDN>:<non-ssl-portnumber> or
50     * https://<FQDN>:<ssl-portnumber>
51     * @param integer $did Domain ID
52     * @param string $wsprotocol Web socket protocol; REST or SOAP
53     * @param string $authtype Authorization type; HMAC or PASSWORD
54     * @param string $id PublicKey or Username (Keys should be in hex)
55     * @param string $secret SecretKey or Password (Keys should be in hex)
56     */
57    function __construct ($hostport, $did, $protocol, $authtype, $keyid, $keysecret) {
58      // TODO: Test inputs? No?
59      // Save information
60      $this->hostport = $hostport;
61      $this->did = $did;
62      $this->protocol = $protocol;
63      $this->authtype = $authtype;
64      $this->keyid = $keyid;
65      $this->keysecret = $keysecret;
66
67      // Check if not supported
68      if ($authtype != StrongMonkey::$AUTHORIZATION_HMAC && $authtype != StrongMonkey::
           ↪ $AUTHORIZATION_PASSWORD) {
69        die('The provided authorization method is not supported');
70      }
71      if ($protocol != StrongMonkey::$PROTOCOL_REST) {
72        die('The provided protocol is not supported');
73      }
74    }
75
76    /**
77     * Initialize a key registration challenge with the FIDO server
78     * @param string $username Username of the user
79     * @param string $displayname Display name for the user
80     * @param array|string $options Object of options
81     * @param array|string $extensions Object of extensions
82     * @return integer|array
83     */
84    public function preregister ($username, $displayname=null, $options=null,
         ↪ $extensions=null) {
85      // Init parameters
```

```
86      if (is_null($displayname)) $displayname = $username;
87      $options = $this->jsonStringPrepare($options, new stdClass);
88      $extensions = $this->jsonStringPrepare($extensions, new stdClass);
89
90      // Create data
91      $payload = array(
92        'username' => $username,
93        'displayname' => $displayname,
94        'options' => $options,
95        'extensions' => $extensions
96      );
97
98      // Make preregister request
99      return $this->request($payload, '/preregister');
100   }
101
102   /**
103    * Send register response to the FIDO server
104    * @param array|string $response Response data from the authenticator
105    * @param array|string $metadata Additional meta data
106    * @return integer|array
107    */
108   public function register ($response, $metadata=null) {
109     // Init empty parameters
110     $response = $this->jsonStringPrepare($response);
111     $metadata = $this->jsonStringPrepare($metadata, new stdClass);
112
113     // Create data
114     $payload = array(
115       'response' => $response,
116       'metadata' => $metadata
117     );
118
119     // Make register request
120     return $this->request($payload, '/register');
121   }
122
123   /**
124    * Initialize a key authentication challenge with the FIDO server
125    * @param string $username Username of the user
126    * @param array|string $options Object of options
127    * @param array|string $extensions Object of extensions
128    * @return integer|array
129    */
130   public function preauthenticate ($username=null, $options=null, $extensions=null) {
131     // Init empty parameters
132     //if (is_null($username)) {
133     // if (is_null($options)) $options = array();
134     // $options['Residentkey'] = 'req';
135     //}
136     $options = $this->jsonStringPrepare($options, new stdClass);
137     $extensions = $this->jsonStringPrepare($extensions, new stdClass);
138
139     // Create data
140     $payload = array(
141       'username' => $username,
142       'options' => $options,
143       'extensions' => $extensions
144     );
145
146     // Make preauthenticate request
147     return $this->request($payload, '/preauthenticate');
```

```
148 }
149
150 /**
151  * Send authenticate response to the FIDO server
152  * @param array|string $response Response data from the authenticator
153  * @param array|string $metadata Additional meta data
154  * @return integer|array
155  */
156 public function authenticate ($response, $metadata=null) {
157   // Init empty parameters
158   $response = $this->jsonStringPrepare($response);
159   $metadata = $this->jsonStringPrepare($metadata, new stdClass);
160
161   // Create data
162   $payload = array(
163     'response' => $response,
164     'metadata' => $metadata
165   );
166
167   // Make authenticate request
168   return $this->request($payload, '/authenticate');
169 }
170
171 /**
172  * Update key information
173  * @param string $status The status of the key (Active, Inactive)
174  * @param string $modify_location Modify location
175  * @param string $displayname Display name of the key
176  * @param string $keyid Id of the key to change
177  * @return integer|array
178  */
179 public function updatekeyinfo ($status, $modify_location, $displayname, $keyid) {
180   // Create data
181   $payload = array(
182     "status" => $status,
183     "modify_location" => $modify_location,
184     "displayname" => $displayname,
185     "keyid" => $keyid
186   );
187
188   // Make updatekeyinfo request
189   return $this->request($payload, '/updatekeyinfo');
190 }
191
192 /**
193  * Get user's keys information from the FIDO server
194  * @param string $username Username of the user
195  * @return integer|array
196  */
197 public function getkeysinfo ($username) {
198   // Create data
199   $payload = array(
200     "username" => $username
201   );
202
203   // Make getkeysinfo request
204   return $this->request($payload, '/getkeysinfo');
205 }
206
207 /**
208  * Delete user's key information from the FIDO server
209  * @param string $keyid Id of the key to deregister
```

```
210      * @return integer/array
211      */
212     public function deregister ($keyid) {
213       // Create data
214       $payload = array(
215         "keyid" => $keyid
216       );
217
218       // Make deregister request
219       return $this->request($payload, '/deregister');
220     }
221
222     /**
223      * Send a ping to the FIDO server
224      * @return boolean/string
225      */
226     public function ping () {
227       // Make ping request
228       $response = $this->request(null, '/ping', false);
229       // If no error
230       if ($response['code'] === 200) {
231         return $response['body'];
232       }
233       // Return error code
234       return $this->parseResponse($response['code'], $response['body']);
235     }
236
237     /**
238      * Create a request to the FIDO server
239      * @param array $payload Payload to send
240      * @param string $action_path API path for the action
241      * @param boolean $parse Automatically parse response
242      * @return integer/array
243      */
244     public function request ($payload, $action_path, $parse=true) {
245       // Create data
246       $body = array(
247         "svcinfo" => array(
248           "did" => $this->did,
249           "protocol" => StrongMonkey::$api_protocol,
250           "authtype" => $this->authtype
251         )
252       );
253       // Prepare payload
254       if (!is_null($payload)) {
255         $body['payload'] = $payload;
256       }
257
258       // Generate path
259       $path = StrongMonkey::$api_url_base . $action_path;
260
261       // Prepare Request Headers
262       $headers = array(
263         'Accept: application/json',
264         'Content-Type: application/json',
265         'User-Agent: ' . StrongMonkey::$useragent
266       );
267
268       // HMAC
269       if ($this->authtype === StrongMonkey::$AUTHORIZATION_HMAC) {
270         // Get date
271         $date = date('D, j M Y H:i:s e');
```

```
272
273       // Prepare hashes
274       $payload_hash = '';
275       $mimetype = '';
276       if (!is_null($payload)) {
277         $payload_string = json_encode($body['payload'], JSON_UNESCAPED_SLASHES);
278         $payload_hash = base64_encode(hex2bin(hash('sha256', $payload_string)));
279         $mimetype = 'application/json';
280       }
281
282       // Generate HMAC authentication
283       $authentication_hash = $this->generateHMAC('POST', $payload_hash, $mimetype,
              ↪ $date, $path);
284
285       // Add authorization Headers
286       $headers[] = 'strongkey-content-sha256: ' . $payload_hash;
287       $headers[] = 'date: ' . $date;
288       $headers[] = 'strongkey-api-version: ' . StrongMonkey::$api_version;
289       $headers[] = 'Authorization: ' . $authentication_hash;
290     }
291     // Credentials
292     else {
293       $body['svcinfo']['svcusername'] = $this->keyid;
294       $body['svcinfo']['svcpassword'] = $this->keysecret;
295     }
296
297     // Create request
298     $ch = curl_init();
299     curl_setopt($ch, CURLOPT_URL, $this->hostport . $path);
300     if (STRONGMONKEY_DEBUG) {
301       curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, 0);
302       curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 0);
303     } else {
304       curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, 1);
305       curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 2);
306     }
307     curl_setopt($ch, CURLOPT_POST, 1);
308     curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($body, JSON_UNESCAPED_SLASHES));
309     curl_setopt($ch, CURLOPT_HTTPHEADER, $headers);
310     curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
311     curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, STRONGMONKEY_CONNECTTIMEOUT);
312     curl_setopt($ch, CURLOPT_TIMEOUT, STRONGMONKEY_TIMEOUT);
313     $response = curl_exec($ch);
314     $response_code = curl_getinfo($ch, CURLINFO_RESPONSE_CODE);
315     curl_close($ch);
316
317     if ($parse) {
318       return $this->parseResponse($response_code, $response);
319     }
320     else {
321       return array(
322         'code' => $response_code,
323         'body' => $response
324       );
325     }
326   }
327
328   /**
329    * Parse response from the FIDO server
330    * @param integer $code HTTP code returned
331    * @param string $response Response body
332    * @return integer|array
```

```
333    */
334   private function parseResponse ($code, $response) {
335     // 200: Success
336     if ($code === 200) {
337       $response = json_decode($response);
338       if ($response) {
339         return $response;
340       }
341       return StrongMonkey::$PARSE_ERROR;
342     }
343     // 400: There was an error in the submitted input.
344     if ($code === 400) {
345       return StrongMonkey::$SUBMIT_ERROR;
346     }
347     // 401: The authentication failed.
348     if ($code === 401) {
349       return StrongMonkey::$AUTHENTICATION_FAILED;
350     }
351     // 404: The requested resource is unavailable.
352     if ($code === 404) {
353       return StrongMonkey::$RESOURCE_UNAVAILABLE;
354     }
355     // 500: The server ran into an unexpected exception.
356     if ($code === 500) {
357       return StrongMonkey::$UNEXPECTED_ERROR;
358     }
359     // 501: Unused routes return a 501 exception with an error message.
360     if ($code === 501) {
361       return StrongMonkey::$UNUSED_ROUTES;
362     }
363     return StrongMonkey::$UNKNOWN_ERROR;
364   }
365
366   /**
367    * Check and get error string if any
368    * @param mixed $error Response returned from an action
369    * @return boolean|string
370    */
371   public function getError ($error) {
372     // If not error
373     if (!is_numeric($error)) {
374       return false;
375     }
376     // Resolve error code
377     switch ($error) {
378       case StrongMonkey::$PARSE_ERROR:
379         return 'StrongMonkey: Response parse error.';
380       case StrongMonkey::$SUBMIT_ERROR:
381         return 'StrongMonkey: There was an error in the submitted input.';
382       case StrongMonkey::$AUTHENTICATION_FAILED:
383         return 'StrongMonkey: The authentication failed.';
384       case StrongMonkey::$RESOURCE_UNAVAILABLE:
385         return 'StrongMonkey: The requested resource is unavailable.';
386       case StrongMonkey::$UNEXPECTED_ERROR:
387         return 'StrongMonkey: The server ran into an unexpected exception.';
388       case StrongMonkey::$UNKNOWN_ERROR:
389         return 'StrongMonkey: Unused routes return a 501 exception with an error
              ↪ message.';
390       default:
391         return 'StrongMonkey: Unknown error code.';
392     }
393   }
```

```
394
395    /**
396     * Generate HMAC authentication header value
397     * @param string $method Request method to be used
398     * @param string $payload Payload Hash to be used
399     * @param string $mimetype Mime-type to be used
400     * @param string $datestr Date string to be used
401     * @param string $path Path to be used
402     * @return string
403     */
404    private function generateHMAC ($method, $payload, $mimetype, $datestr, $path) {
405      // Assembly hash message
406      $message = array(
407        $method,
408        $payload,
409        $mimetype,
410        $datestr,
411        StrongMonkey::$api_version,
412        $path
413      );
414      // Generate HMAC
415      $digest = hash_hmac('sha256', implode("\n", $message), pack('H*', $this->
           ↪ keysecret));
416      // Return header
417      return 'HMAC ' . $this->keyid . ':' . base64_encode(pack('H*', $digest));
418    }
419
420    /**
421     * Convert JSON to string
422     * @param array|string $json JSON value to be converted
423     * @param array|string $ifnull Default falue if value is null
424     * @return string
425     */
426    private function jsonStringPrepare ($json, $ifnull=null) {
427      if ($json === null && $ifnull !== null) {
428        $json = $ifnull;
429      }
430      if (is_string($json)) {
431        return $json;
432      }
433      return json_encode($json, JSON_UNESCAPED_SLASHES);
434    }
435
436  }
```

LISTING B.2: StrongMonkey SDK library for PHP applications.

FIGURE B.1: UML Class Diagram for the StrongMonkey PHP library

```
StrongMonkey
-----------------------------------------------------------------
-api_protocol:string
-api_version:string
-api_url_base:string
-version:string
-useragent:string
-hostport:string
-did:integer
-wsprotocol:string
-authtype:string
-keyid:string
-keysecret:string
-----------------------------------------------------------------
+StrongMonkey (
   hostport:string, did:integer, protocol:string,
   authtype:string, keyid:string, keysecret:string
)
+preregister (
   username:string, displayname:string|null,
   options:array|string|null, extensions:array|string|null
):integer|array
+register (
   response:array|string, metadata:array|string|null
):integer|array
+preauthenticate (
   username:string|null, options:array|string|null, extensions:array|string|null
):integer|array
+authenticate (
   response:array|string, metadata:array|string|null
):integer|array
+updatekeyinfo (
   status:string, modify_location:string,
   displayname:string, keyid:string
):integer|array
+getkeysinfo (username:string):integer|array
+deregister (keyid:string):integer|array
+ping():integer|string
+request (
   payload:array, action_path:string, parse:boolean
):integer|array
-parseResponse (code:integer, response:string):integer|array
+getError (error:mixed):boolean|string
-generateHMAC (
   method:string, payload:string,
   mimetype:string, datestr:string, path:string
):string
-jsonStringPrepare (json:array|string, ifnull:array|string|null):string
```

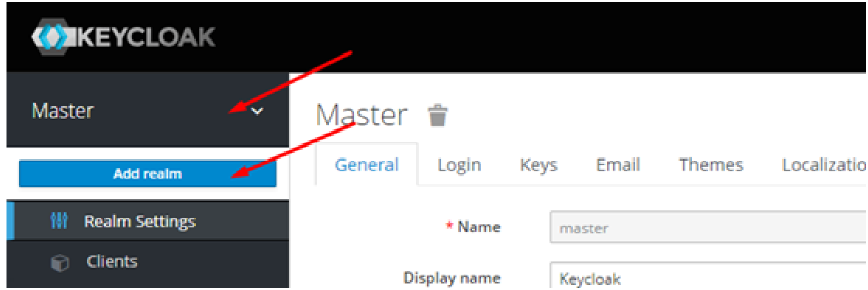
Appendix C

Configuring Keycloak for WebAuthn

This section is a guide on how to configure FIDO2/WebAuthn on Keycloak[1] IdM server.To follow the guide you one will have to log into your Keycloak server as an admin. The following configuration was performed on an Ubuntu 20.04.1 server running Keycloak v12.0.4.

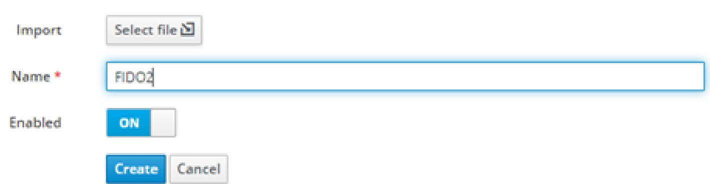
Create a new realm (Optional)

Fist, let's create a new realm for our FIDO2/WebAuthn testing. Click the dropdown next to the realm name (here "Master") and click the "Add realm" button:



Fill in the realm name of preference (we inserted "FIDO2") and then click the "Create" button:

Add realm



Note that users will now be able to login to the realm account though:
`https://<domain>:<port>/auth/realms/<realm-name>/account`

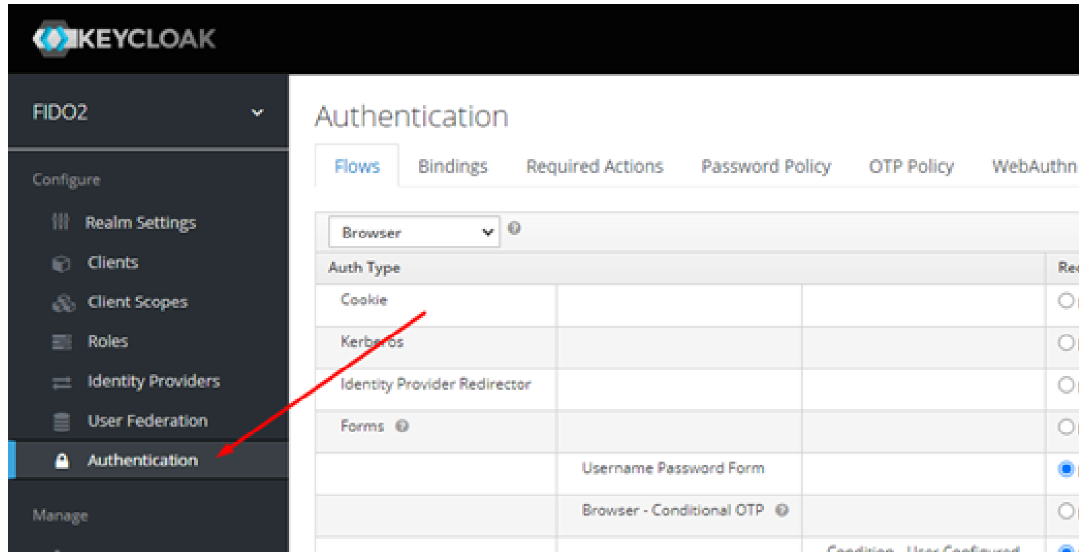
Thus in our example lab case:
`https://keycloak.gramthanos.com:8443/auth/realms/FIDO2/account`

Enable WebAuthn Registers

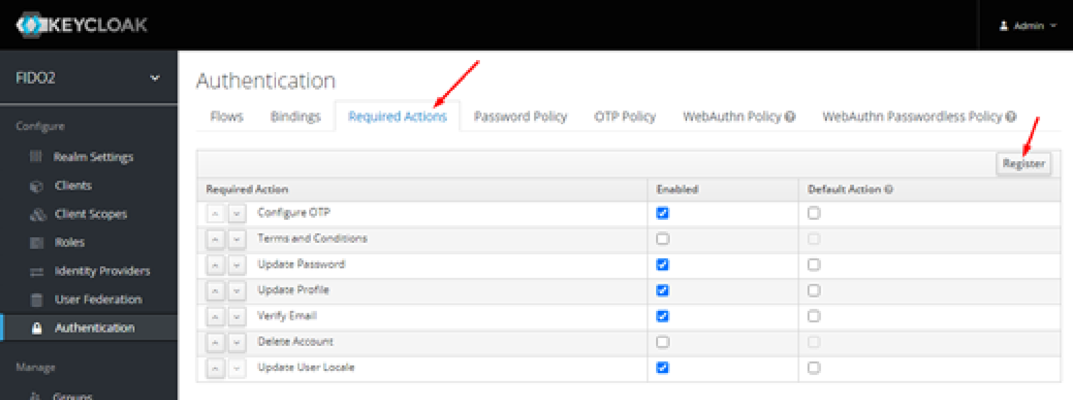
On your new realm, navigate to the "Authentication" menu on the sidebar.

---

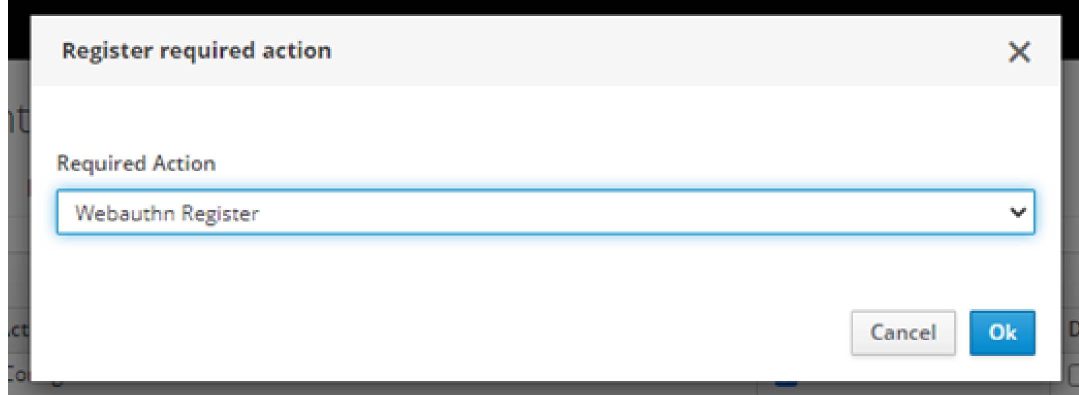
[1] https://www.keycloak.org/



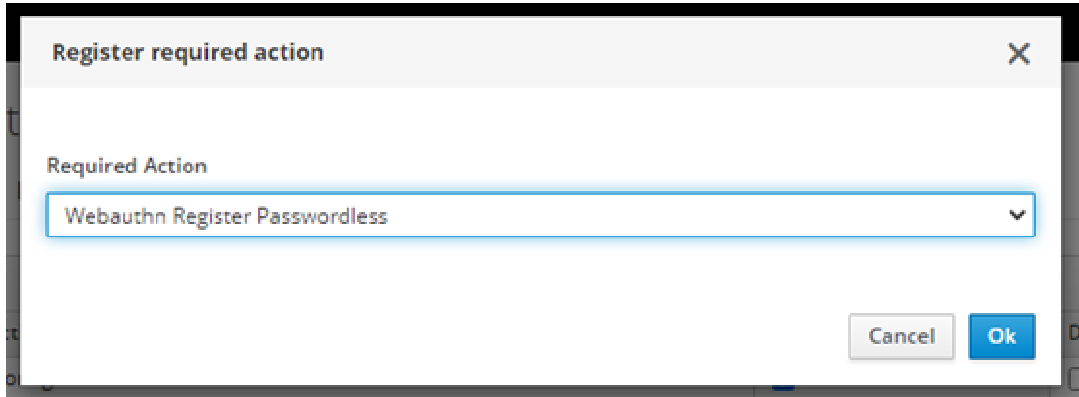
Then select the “Required Actions” tab at the top. You will now be able to insert 2 WebAuthn related “Registers”.



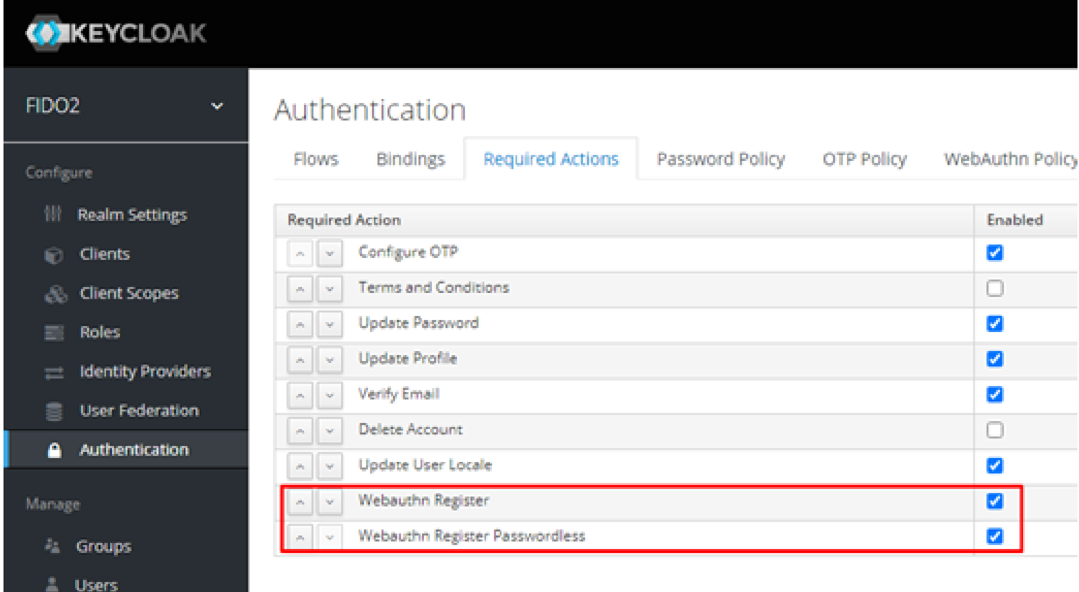
Click the register button and insert the “WebAuthn Register” used to manage the WebAuthn keys to be used as 2nd Factor Authenticators under the user’s “Personal Info > Account Security > Signing In > Two-Factor Authentication” panel (when a user logs in on the Keycloak realm panel).



Then, you can also add the “WebAuthn Register Passwordless” that is used to manage WebAuthn keys for password-less authentication. These keys are listed under another section on the user’s panel.

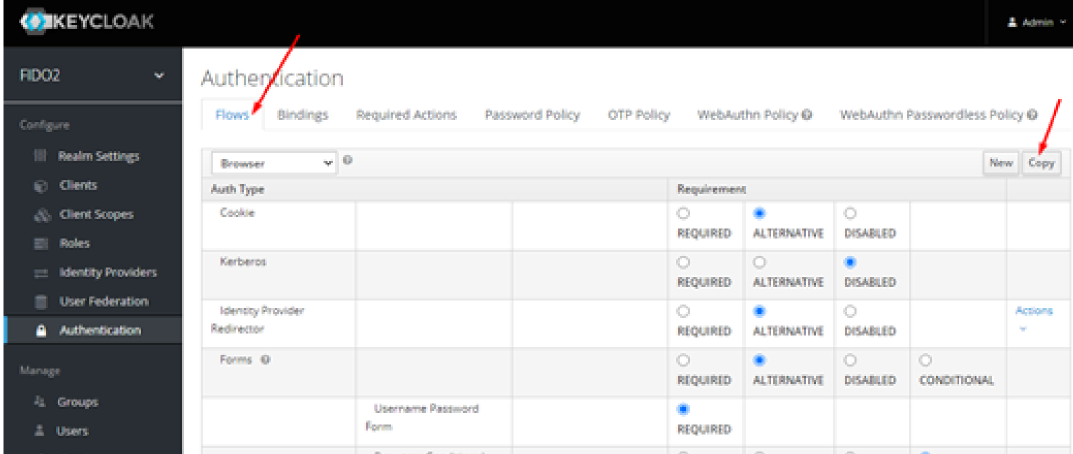


The two WebAuthn Registers can now be seen in the list, and their "Enabled" checkbox should be checked.



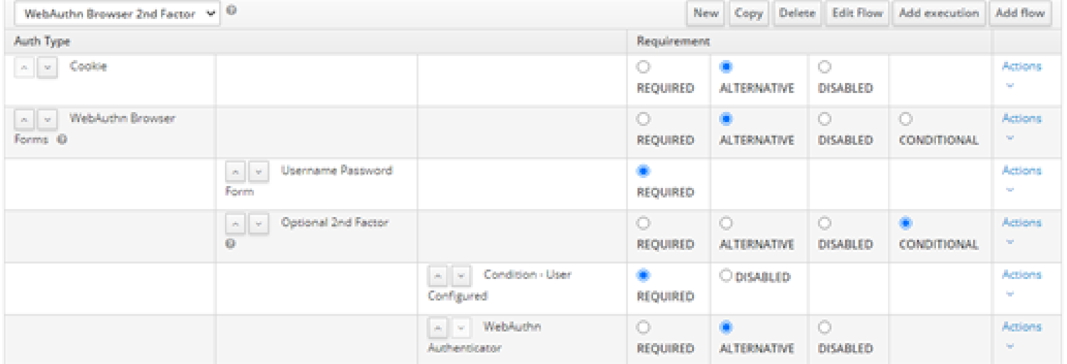
Create WebAuthn Authentication Flows

Now that we enabled WebAuthn registers we are able to create new Authentication flows that also include authentication through WebAuthn. Go to the "Flows" tab and click to clone the default "Browser" flow:



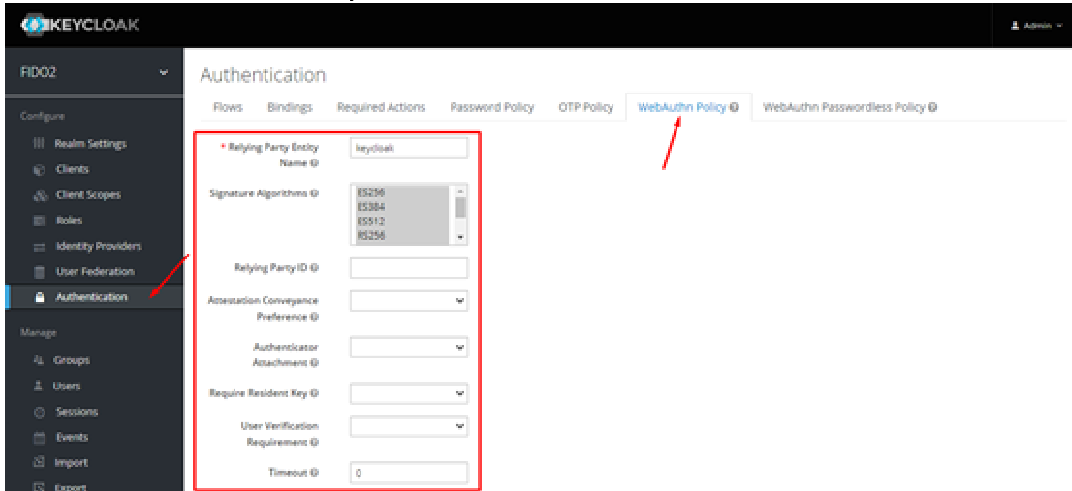
Second Factor Authentication Flow (U2F)

In the image below you can see how we configured a traditional Username & Password Authentication flow with an additional WebAuthn step serving as a 2nd factor authentication.

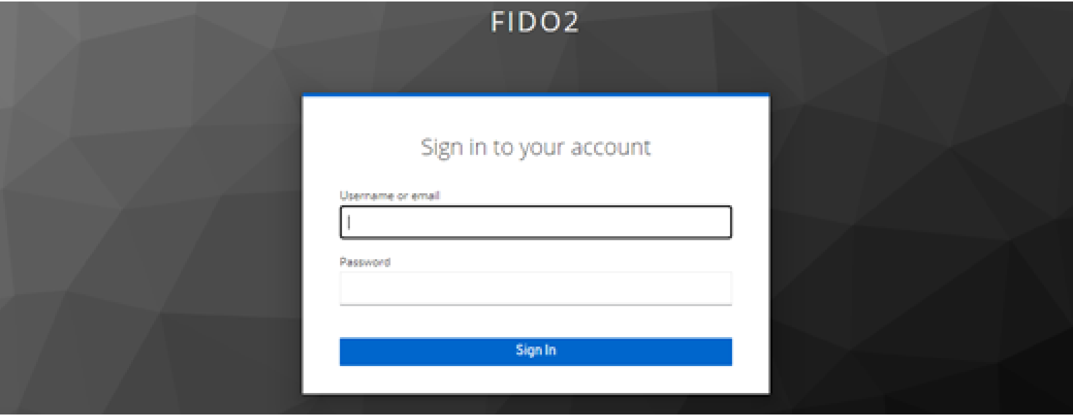


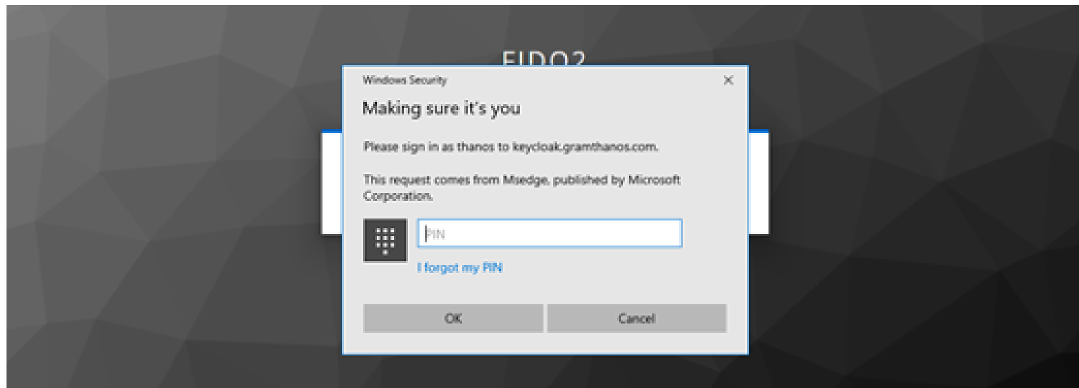
Note that we set the “Optional 2nd Factor” flow to “CONDITIONAL” thus the 2nd factor will only appear for users that have already registered a FIDO key for 2nd factor authentication. Otherwise the authentication flow will only be based on the Username & Password.

This flow uses the “WebAuthn Authenticator” linked to the “WebAuthn Register” that we added in a previous section. The policy of this authenticator can be edited under the “WebAuthn Policy” tab on Authentication.

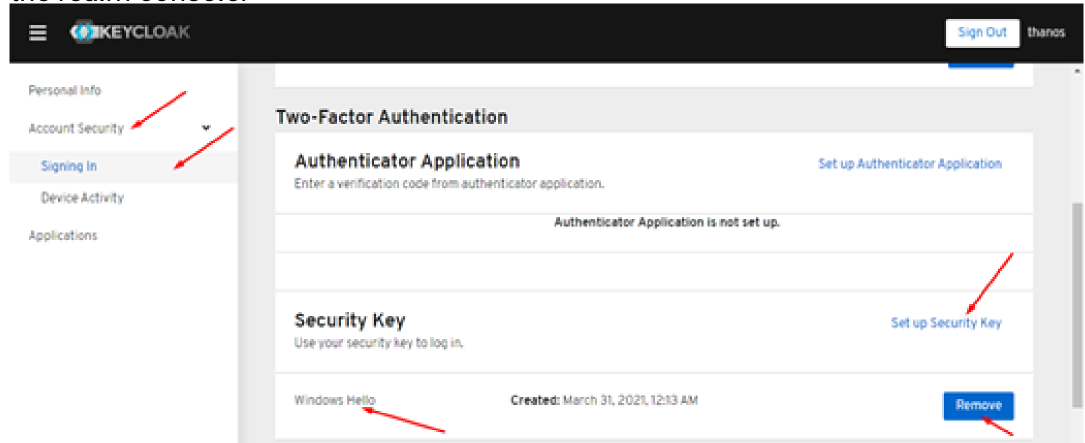


Here is the login flow as the user experience it:





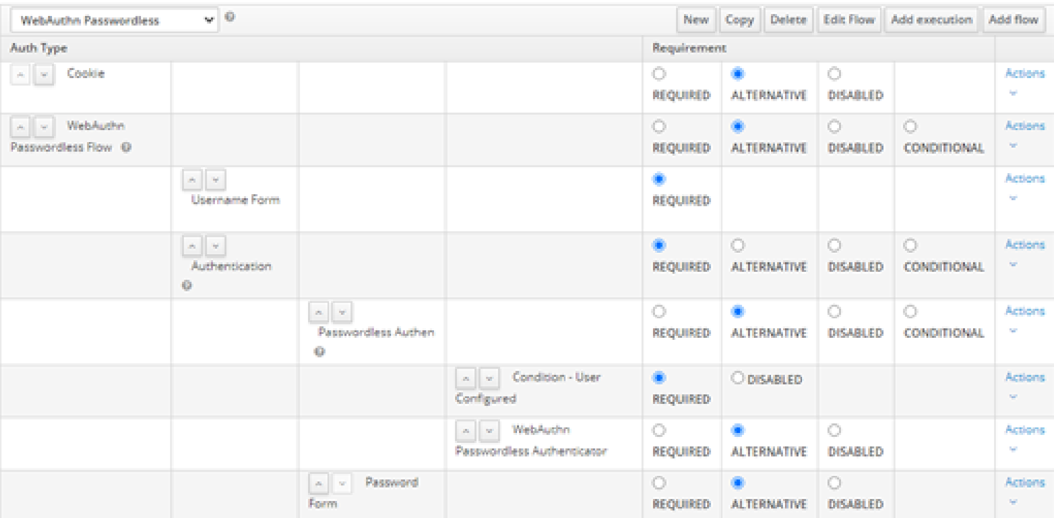
The user will be able to register keys from their sign in options after they login on the realm console:



In the image above you can see that we registered a Windows Hello using a PIN as an authenticator device for this user.

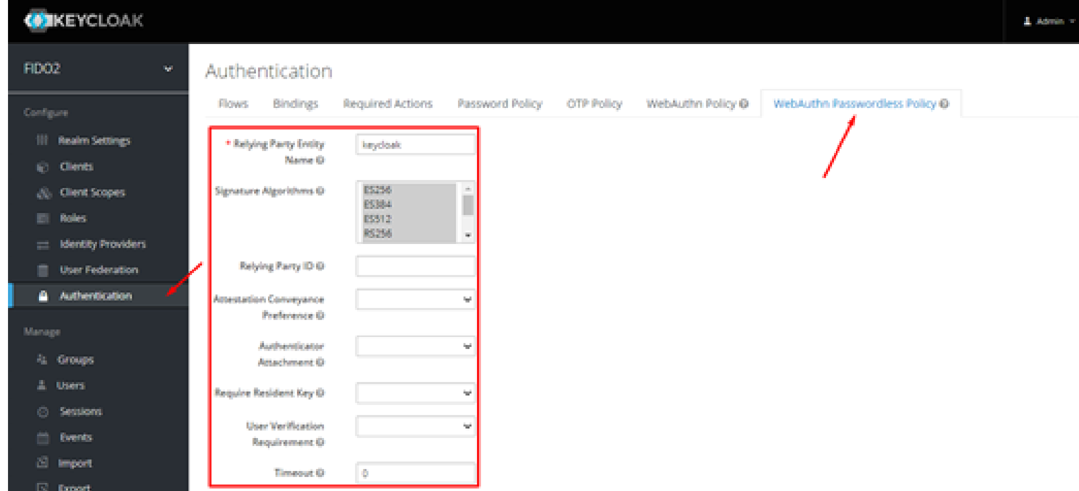
Password-less Authentication only with username

In the image below you can see how we configured a traditional Username & Password Authentication flow with an alternative login option using an authenticator device for password-less login (provided that the user has registered such an authenticator key under password-less).

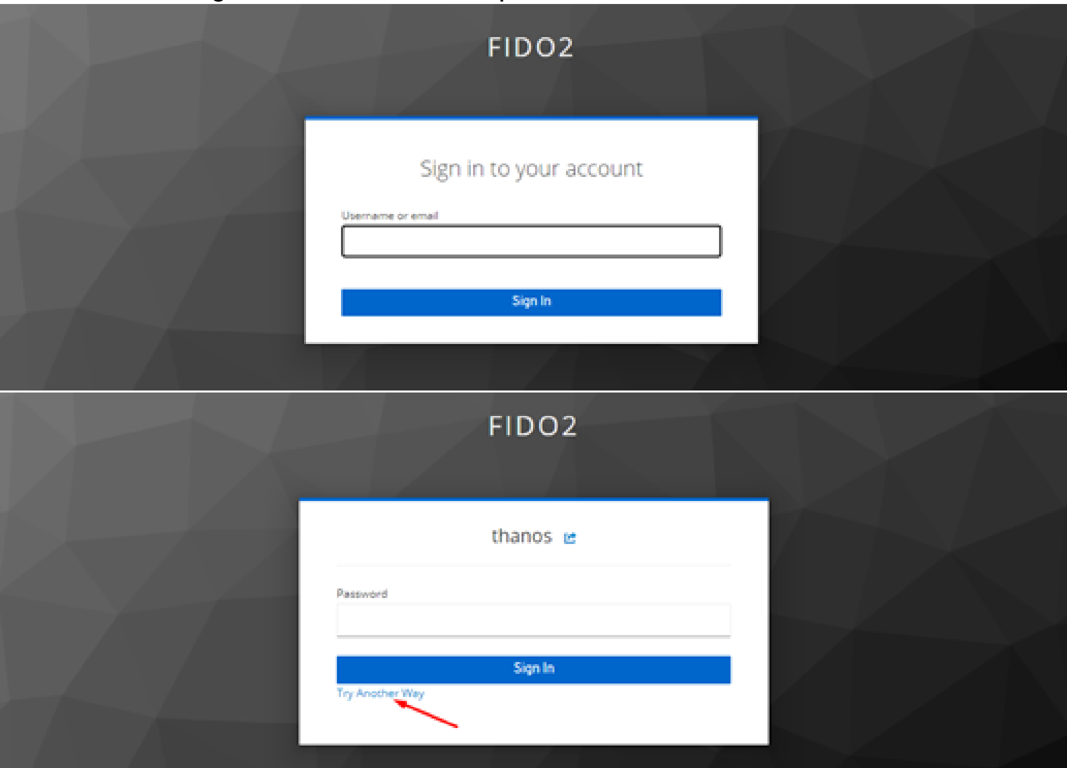


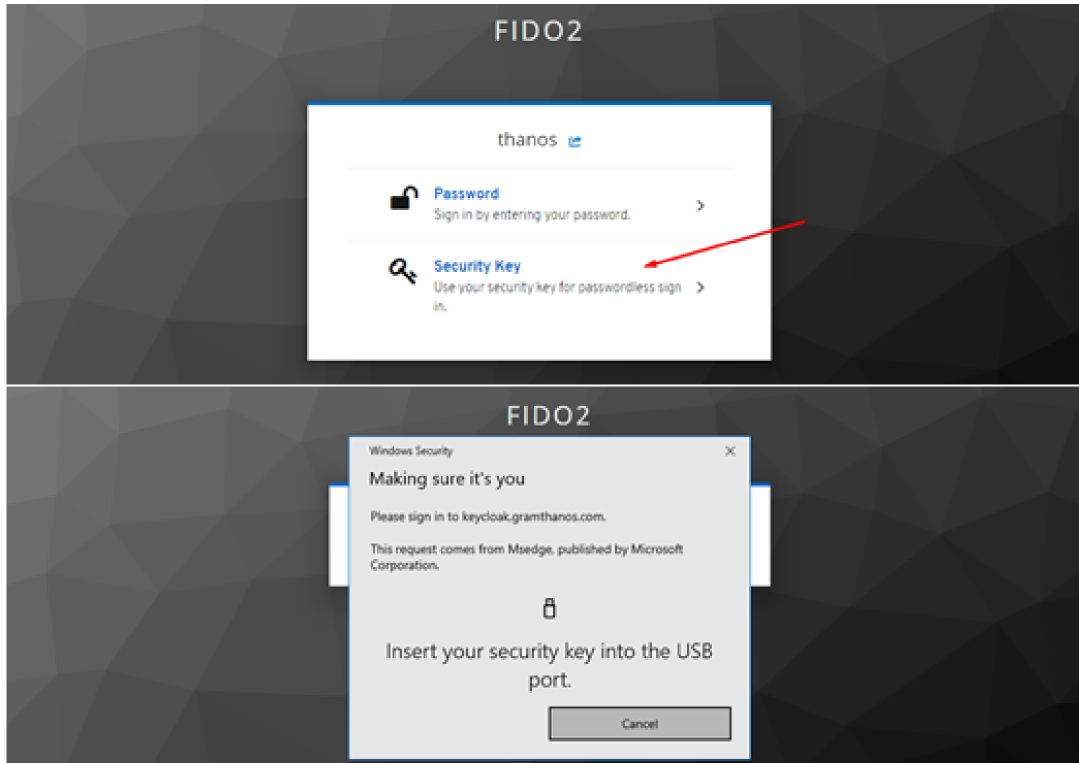
Note that since we used the "Condition - User Configured" and we set the "Passwordless Authen" flow to "ALTERNATIVE", the option will only be available if the user already registered a FIDO key for password-less login. Otherwise the authentication flow will be based on the Username & Password.

This flow uses the "WebAuthn Passwordless Authenticator" linked to the "WebAuthn Register Passwordless" that we added in a previous section. The policy of this authenticator can be edited under the "WebAuthn Passwordless Policy" tab on Authentication.

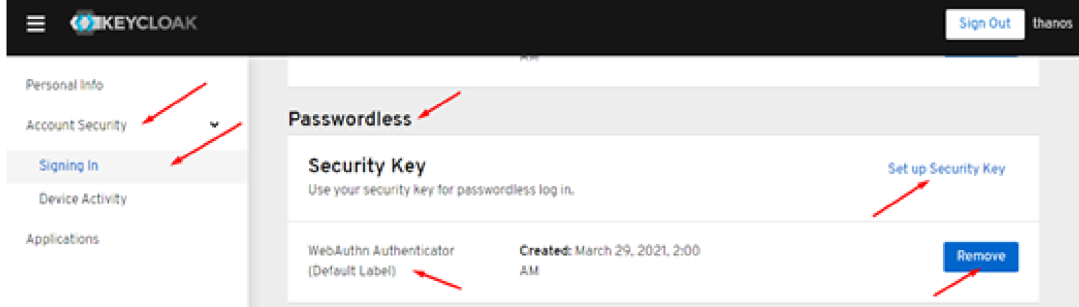


Here is the login flow as the user experience it:





The user will be able to register keys from their sign in options after they login on the realm console:

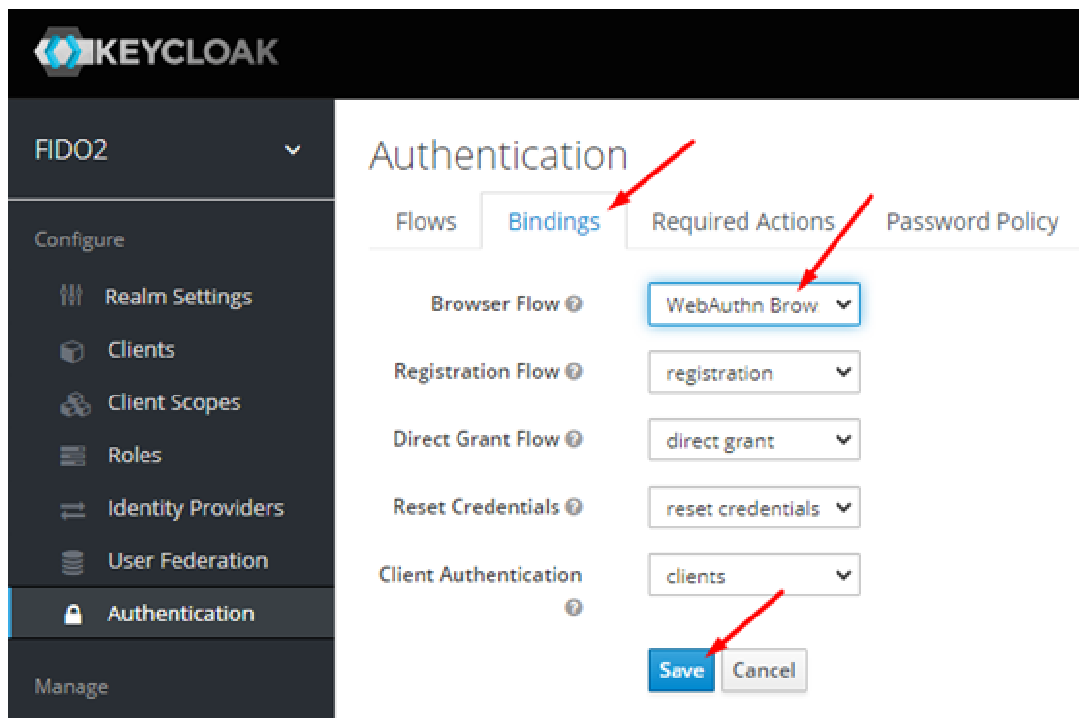


In the image above you can see that we registered an authenticator device under password-less login.

Binding an Authentication flow

After creating the authentication flow, you will be able to set our authentication flow of preference under the “Bindings” tab:

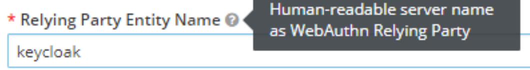
Keycloak provides two configuration options. The idea is that one (the less strict one) can been used for second factor authentication and the other one (with stronger security) to be used for password-less authentication.



Configuring FIDO Policy

Keycloak provides two configuration options. The idea is that one (the less strict one) can be used for second factor authentication and the other one (with stronger security) to be used for password-less authentication.

Relying Party Name
The relying party name is used only to be displayed to humans. Depending on the platform this name may appear on the UI of the authentication or registration.

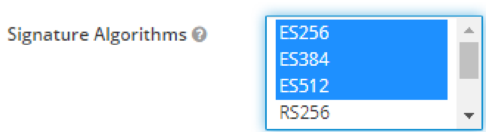


Signature Algorithms
From the available signature algorithms only the 3 ECDSA-based are recommended for use.

Code Name	Number	Description	Recommended
ES256	-7	ECDSA w/ SHA-256	Yes
ES384	-35	ECDSA w/ SHA-384	Yes
ES512	-36	ECDSA w/ SHA-512	Yes
RS256	-257	RSASSA-PKCS1-v1_5 using SHA-256	No
RS384	-258	RSASSA-PKCS1-v1_5 using SHA-384	No
RS512	-259	RSASSA-PKCS1-v1_5 using SHA-512	No
RS1	-65535	RSASSA-PKCS1-v1_5 using SHA-1	Deprecated

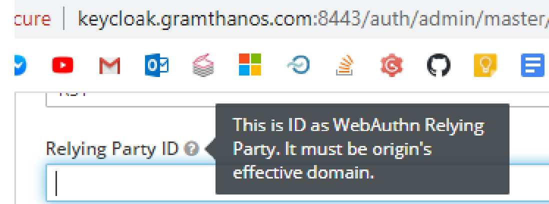
More information about these specific algorithms as well as an up-to-date usage recommendation can be found at IANA's CBOR Object Signing and Encryption (COSE)[2] website.

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



Relying Party ID

The Relying Party ID should match the domain name of the website or match a higher level domain. In our case, since our domain name of the website is “keycloak.gramthanos.com” valid Relying Party ID are the “keycloak.gramthanos.com” and “gramthanos.com”. Leaving the input blank, the website’s domain name will be used.



Note that if the Relying Party ID changes, the already generated authenticators should not work anymore as they are bound to that.

Attestation Conveyance Preference

Value	Description
	Blank value means do not set, which is the same as “none”
none	No attestation is needed from the authenticator.
indirect	Indirect attestation is requested.
direct	Direct attestation is requested.

Authenticator Attachment

Value	Description
	Blank value means don’t filter authenticator.
platform	Filter platform authenticators (e.g. Windows Hello).
cross-platform	Filter cross-platform authenticators (e.g. External USB Authenticator).

Require Resident Key

This option requests the authenticator to create only resident keys. For now this option seems not to have any effect from the server side. Normally, this option is supposed to be used to login without providing a username which seems not to be supported.

User Verification Requirement

Value	Description
	A blank value will be treated as "preferred".
required	Requires the user to interact with the authenticator.
preferred	The user interaction is preferred.
discouraged	The authenticator is discouraged from asking the user to interact with it.

Timeout

This option sets the timeout in seconds for the registration of an authenticator device. In our tests this does not work for the authentication.

Avoid Same Authenticator Registration

This option will exclude already registered authenticators during the registration process. We don't see a reason for this not to be on.

Acceptable AAGUIDs

Essentially this is a white-list of the accepted authenticator devices. The AAGUID values can be found on the webpage of the manufacturer or the authenticators (e.g. YubiKey Hardware FIDO2 AAGUIDs[3]). Alternatively a tool has to be used to check the AAGUID of an authenticator (e.g. using Yubico's Python FIDO2 library and get-info.py[4]).

The AAGUID the authenticator device reports is usually linked to the requested attestation, thus if this is to be used, the Attestation Conveyance Preference should be set to direct or maybe indirect depending on the authenticator. If for example attestation "none" is used, the authenticator may choose to send an all zeros AAGUID.

[3] https://support.yubico.com/hc/en-us/articles/360016648959-YubiKey-Hardware-FIDO2-AAGUIDs
[4] https://github.com/Yubico/python-fido2/tree/master/examples

Appendix D

VPN Implementation Code

In Section 4.3 we presented our implementation of FIDO2 on OpenVPN. As we mentioned the full code is publicly available at GitHub[1]. In this section we will list the 2 most important parts of the implementation, the server authentication Python code (Listing D.1) as well as the main client application JavaScript code (Listing D.2).

```
1  #!/usr/bin/env python3
2  import os
3  import sys
4  import json
5  import base64
6  import requests
7
8  ssl_verify = True
9
10 # This is only for development
11 #requests.packages.urllib3.disable_warnings()
12 #ssl_verify = False
13
14 authservices = {
15   "google-oidc" : {
16     "wellknown" : "https://accounts.google.com/.well-known/openid-configuration",
17     "clientid" : "<google-client-id>",
18     "secret" : "<google-secret>",
19     "redirect" : "https://vpnapp.electron.gramthanos.com/oidc"
20   },
21   "keycloak-oidc" : {
22     "wellknown" : "<keycloak-app-url>/.well-known/openid-configuration",
23     "clientid" : "<keycloak-client-id>",
24     "secret" : "<keycloak-secret>",
25     "redirect" : "https://vpnapp.electron.gramthanos.com/oidc"
26   }
27 };
28
29 # Parse authen info given
30 authentication = json.loads(base64.b64decode(os.getenv('password')).decode())
31
32 # Check if correct response
33 if not 'service' in authentication.keys() or not 'code' in authentication.keys():
34   sys.exit(1)
35 # Check if service in list of services
36 if not authentication['service'] in authservices.keys():
37   sys.exit(1)
38
39 # Get service info
40 service = authservices[authentication['service']]
41
42 # Get configuration
```

[1] https://github.com/GramThanos/vpn-oidc

```
43 response = requests.get(service['wellknown'], verify = ssl_verify);
44 # Check if request failed
45 if response.status_code != 200:
46   sys.exit(1)
47 # Parse data
48 discovery = response.json()
49 # Check if endpoints dont exists
50 if not 'token_endpoint' in discovery.keys():
51   sys.exit(1)
52 if not 'userinfo_endpoint' in discovery.keys():
53   sys.exit(1)
54
55 # Request data
56 response = requests.post(discovery['token_endpoint'], data = {
57   'code': authentication['code'],
58   'client_id' : service['clientid'],
59   'client_secret' : service['secret'],
60   'redirect_uri' : service['redirect'],
61   'grant_type' : 'authorization_code'
62 }, verify = ssl_verify);
63 # Check if request failed
64 if response.status_code != 200:
65   sys.exit(1)
66 # Parse data
67 data = response.json()
68 # Check if endpoints dont exists
69 if not 'access_token' in data.keys() or not 'id_token' in data.keys():
70   sys.exit(1)
71
72 jwt = data['id_token'].split('.')
73 info = json.loads(base64.b64decode(jwt[1] + '=' * (-len(jwt[1]) % 4)).decode())
74 print('Welcome ' + info['email'] + ' !')
75 sys.exit(0)
```

LISTING D.1: Server side authentication code in Python.

```
1  // preload.js
2  const fs = require('fs');
3  const path = require('path');
4  const crypto = require('crypto');
5  const isDevelopment = false;
6  const axios = isDevelopment ?
7    require('axios').create({
8      httpsAgent: new require('https').Agent({rejectUnauthorized: false}),
9      adapter: require('axios/lib/adapters/http')
10   }) :
11   require('axios').create({
12     adapter: require('axios/lib/adapters/http')
13   });
14 const child_process = require('child_process');
15
16 const $app = {
17   connected : false,
18
19   configPath : path.join(__dirname, '..', 'config.json'),
20   openvpnPath : null,
21
22   // Load configuration file
23   loadConfig : function() {
24     // Default config
25     this.config = {authservices: []};
26
```

```
27      try {
28        // Load config file
29        let tmp = JSON.parse(fs.readFileSync(this.configPath, 'utf8'));
30        // Validate config
31        if (
32          !tmp.hasOwnProperty('authservices') ||
33          !(tmp.authservices instanceof Array)
34        ) {
35          throw('Invalid config.');
36        }
37        // Config loaded
38        this.config = tmp;
39        this.log('Config loaded.');
40      } catch (e) {
41        this.log('Failed to loaded config:\n' + e.toString());
42      }
43    },
44
45    // Display available authentication services
46    showAuthServices : function() {
47      // UI wrapper for services
48      const wrapper = document.getElementById('connect-form');
49
50      // If no services available
51      if (this.config.authservices.length == 0) {
52        let msg = document.createElement('span');
53        msg.textContent = 'No service found.';
54        wrapper.appendChild(msg);
55        this.log('No services available.');
56        return;
57      }
58
59      let length = 0;
60      // Display each service on screen
61      this.config.authservices.forEach(auth => {
62        let button = document.createElement('input');
63        button.setAttribute('type', 'submit');
64        button.setAttribute('value', auth.name);
65        button.setAttribute('title', auth.description);
66        button.className = 'btn btn-primary';
67        button.dataset.authservice = (length + 1);
68        wrapper.appendChild(button);
69        length++;
70      });
71      this.log(length + ' services available.');
72    },
73
74    // Disable all authentication services buttons
75    disableAuthServices : function() {
76      [...document.getElementById('connect-form')
77      .getElementsByTagName('input')].forEach(input => {
78        input.setAttribute('disabled', 'disabled');
79      });
80    },
81
82    // Enable all authentication services buttons
83    enableAuthServices : function() {
84      [...document.getElementById('connect-form')
85      .getElementsByTagName('input')].forEach(input => {
86        input.removeAttribute('disabled', 'disabled');
87      });
88    },
```

```
89
90    // Change View
91    enableConnectedView : function() {
92      document.getElementById('disconnected').style.display = 'none';
93      document.getElementById('connected').style.display = 'block';
94    },
95    enableDisconnectedView : function() {
96      document.getElementById('connected').style.display = 'none';
97      document.getElementById('disconnected').style.display = 'block';
98    },
99
100   // Add authentication services buttons handlers
101   addAuthServicesEventListeners : function() {
102     // Handle form submit event
103     document.getElementById('connect-form')
104     .addEventListener('submit', (e) => {
105       e.preventDefault();
106
107       // Get authentication service to use
108       // (detect which button was pressed)
109       const authservice = this.config.authservices[Math.round(
110         parseInt(document.activeElement.dataset.authservice, 10)
111       ) - 1];
112       if (!authservice) return false;
113
114       // Disable authentication services buttons
115       this.disableAuthServices();
116       // Get authentication services information
117       this.getAuthServiceEndpoint(authservice)
118       .then((endpoint) => {
119         // Start authentication
120         this.launchAuthService(authservice, endpoint)
121         .then((authentucationResponse) => {
122           // Generate username and password
123           let username =
124             crypto.randomBytes(16).toString('base64') +
125             '@' + authservice.id; // Random Username
126           // Save auth response json as password
127           let password = Buffer.from(
128             JSON.stringify(authentucationResponse)
129           ).toString('base64');
130
131           // Connect on VPN
132           this.vpnConnect(authservice, username, password)
133           .catch((error) => {
134             this.enableAuthServices();
135           });
136         })
137         .catch((error) => {
138           reject('Failed authenticate.');
139           this.enableAuthServices();
140         });
141       })
142       .catch((error) => {
143         this.log(error);
144         this.enableAuthServices();
145       })
146
147       return false;
148     });
149
150     // Handle disconnect
```

```
151     document.getElementById('disconnect-form')
152     .addEventListener('submit', (e) => {
153       if (!this.openvpnProcess) {
154         this.openvpnProcess.kill('SIGINT');
155       }
156       this.killOpenVPNClients();
157     });
158   },
159
160   // Load authentication service endpoint
161   getAuthServiceEndpoint : function(authservice) {
162     // Load authentication service OIDC info
163     this.log('Loading "' +
164       authservice.name + '" connection information...' );
165     return new Promise((resolve, reject) => {
166       axios({
167         method: 'get',
168         url: authservice.wellknown,
169         responseType: 'json'
170       })
171       .then((response) => {
172         // Check for errors
173         if (
174           !response.data ||
175           !response.data.authorization_endpoint ||
176           !response.data.userinfo_endpoint
177         ) {
178           reject('Failed to recover OIDC endpoints');
179           return;
180         }
181         if (
182           !response.data.response_types_supported ||
183           !response.data.response_types_supported.includes('code')
184         ) {
185           reject('OIDC configuration does not support code response type');
186           return;
187         }
188         if (
189           !response.data.scopes_supported ||
190           !response.data.scopes_supported.includes('openid') ||
191           !response.data.scopes_supported.includes('email')
192         ) {
193           reject('OIDC configuration does not support needed scopes');
194           return;
195         }
196
197         resolve(response.data.authorization_endpoint);
198       })
199       .catch((error) => {
200         reject('Failed to load OIDC configuration');
201       });
202     });
203   },
204
205
206   launchAuthService : function(authservice, endpoint) {
207     this.log('Starting authentication with "' +
208       authservice.name + '"...' );
209     return new Promise((resolve, reject) => {
210       let state = 'security_token' + ':' +
211         crypto.randomBytes(64).toString('base64') + ':' +
212         authservice.redirect;
```

```
213       let nonce = crypto.randomBytes(64).toString('base64');
214
215       // Prepare URL
216       let serviceURL = new URL(endpoint);
217       serviceURL.searchParams.append('client_id', authservice.clientid);
218       serviceURL.searchParams.append('response_type', 'code');
219       serviceURL.searchParams.append('scope', 'openid email');
220       serviceURL.searchParams.append('redirect_uri', authservice.redirect);
221       serviceURL.searchParams.append('state', state);
222       serviceURL.searchParams.append('nonce', nonce);
223       serviceURL = serviceURL.toString();
224
225       // Open new window for authentication
226       const { remote } = require('electron');
227       const win = new remote.BrowserWindow({
228         title: 'Authenticate',
229         show: false,
230         width: 800,
231         height: 600,
232         backgroundColor: '#ccc',
233         webPreferences: {
234           nodeIntegration: false,
235           enableRemoteModule: false,
236           sandbox: true
237         },
238         parent: remote.getCurrentWindow(),
239         modal: true
240       });
241       //win.setMenuBarVisibility(false);
242
243       win.once('ready-to-show', () => {
244         win.show();
245       });
246
247       win.once('closed', () => {
248         reject('Authentication aborted.');
249       });
250
251       win.loadURL(serviceURL);
252       const {session: {webRequest}} = win.webContents;
253
254       // Catch callback URL
255       webRequest.onBeforeRequest({
256         urls: [authservice.redirect + '*']
257       }, (details, callback) => {
258         let url = new URL(details.url);
259         // Validate response state
260         if (url.searchParams.get('state') !== state) {
261           reject('Authentication failed, invalid response.');
262           return;
263         }
264
265         // Prepare response
266         let response = {
267           service : authservice.id,
268           //session_state : url.searchParams.get('session_state'),
269           code : url.searchParams.get('code')
270         };
271         win.close();
272
273         // Return authentication results
274         resolve(response);
```

```
275       });
276     });
277   },
278 
279   findOpenvpn : function() {
280     // List of possible locations for openvpn
281     let possiblePaths = [
282       path.join('C:/Program Files', 'OpenVPN/bin', 'openvpn.exe'),
283       path.join('C:/Program Files (x86)', 'OpenVPN/bin', 'openvpn.exe'),
284     ];
285 
286     // Check paths for openvpn
287     for (let path of possiblePaths) {
288       try {
289         if (fs.existsSync(path)) {
290           this.openvpnPath = path;
291           break;
292         }
293       } catch(err) {}
294     }
295 
296     // If openvpn was found
297     if (this.openvpnPath) {
298       try {
299         // Try to load version
300         let version = child_process.execFileSync(
301           path.basename(this.openvpnPath),
302           ['--version'],
303           {cwd: path.dirname(this.openvpnPath)}
304         ).toString().trim();
305         version = version.match(/OpenVPN\s*(\d*\.*\d*\.*\d*\.*\d*)/i);
306         version = version[1] || 'Unknown';
307         this.versionsInfo.push('OpenVPN' + ' ' + version);
308 
309         this.log('Found OpenVPN version ' + version);
310       } catch (e) {
311         // Failed to load version
312         this.openvpnPath = null;
313         this.log('No OpenVPN installation found.');
314       }
315     }
316   },
317 
318   vpnConnect : function(authservice, user, pass) {
319     return new Promise((resolve, reject) => {
320       // Run OpenVPN Client
321       this.runOpenVPNClient(authservice, (line, process) => {
322         // Handle output
323         if ((/Enter Auth Username:/i).test(line)) {
324           process.stdin.write(user);
325           return;
326         }
327         else if ((/Enter Auth Password:/i).test(line)) {
328           process.stdin.write(pass);
329           return;
330         }
331         else if ((/Initialization Sequence Completed/i).test(line)) {
332           this.enableConnectedView();
333         }
334 
335         //2021-09-12 21:18:53
336         line = line.replace(
```

```
337          /^\s*\d\d\d\d-\d\d-\d\d\s*\d\d:\d\d:\d\d\s*/i, '');
338        this.log('[OpenVPN] ' + line);
339      })
340      .catch((error) => {
341        this.enableDisconnectedView();
342        reject();
343      });
344    });
345  },
346
347  runOpenVPNClient : function(authservice, handler) {
348    return new Promise((resolve, reject) => {
349      // Check if OpenVPN was found
350      if (!this.openvpnPath) {
351        reject('OpenVPN was not found.');
352        return;
353      }
354
355      this.killOpenVPNClients();
356
357      // Note
358      this.log('Running OpenVPN Client.');
359
360      // Create OpenVPN process
361      var child = child_process.spawn(path.basename(this.openvpnPath), [
362        '--config',
363        path.join(__dirname, '..', 'profiles', authservice.profile)
364      ], {
365        encoding: 'utf8',
366        shell: true,
367        cwd: path.dirname(this.openvpnPath)
368      });
369
370      // You can also use a variable
371      // to save the output for when the script closes later
372      child.on('error', (error) => {
373        console.log('OpenVPN Client Error', error.toString());
374      });
375
376      child.stdin.setEncoding('utf8');
377
378      child.stdout.setEncoding('utf8');
379      child.stdout.on('data', (data) => {
380        handler(data.toString(), child);
381      });
382
383      child.stderr.setEncoding('utf8');
384      child.stderr.on('data', (data) => {
385        handler(data.toString(), child);
386      });
387
388      child.on('close', (code) => {
389        // Here you can get the exit code of the script
390        //switch (code) {
391        // case 0:
392        //  console.log('Client Close', 'Ended');
393        //  break;
394        //}
395        reject(code);
396      });
397
398      this.openvpnProcess = child;
```

```
    });
  },

  killOpenVPNClients : function() {
    try {
      child_process.execFileSync(
        'taskkill.exe',
        ['/F', '/IM', 'openvpn.exe']
      );
    } catch (e) {}
  },

  // Load node info
  versionsInfo : [],
  loadVersionsNode : function() {
    const capitalize = (word) => {
      return word[0].toUpperCase() + word.substring(1).toLowerCase();
    }
    // List versions
    for (let dependency of ['chrome', 'node', 'electron']) {
      this.versionsInfo.push(
        capitalize(dependency) + ' ' + process.versions[dependency]
      );
    }
  },

  // Show versions on GUI
  showVersions : function() {
    // Clear placeholder
    document.getElementById('version').textContent = '';
    // List versions
    for (let version of this.versionsInfo) {
      document.getElementById('version').appendChild(
        document.createTextNode(version)
      );
      document.getElementById('version').appendChild(
        document.createElement('br')
      );
    }
  },

  // Log information function
  log : function(data, date=true, newline=true) {
    if (!this.logElement) {
      this.logElement = document.getElementById('log-textarea');
    }
    this.logElement.value += (newline ? '\n' : '') +
      (date ? '[' + new Date().toISOString() + '] ' : '') +
      data;
  }
};

// When DOM is ready
window.addEventListener('DOMContentLoaded', () => {
  // Log app start
  $app.log('Client started.', true, false);
  // Load some version info
  $app.loadVersionsNode();
  // Find OpenVPN installation
  $app.findOpenvpn();
  // Load app configuration file
  $app.loadConfig();
```

```
461   // Display available authentication services
462   $app.showAuthServices();
463   // Attach handlers
464   $app.addAuthServicesEventListeners();
465   // Show versions on GUI
466   $app.showVersions();
467 });
```

LISTING D.2: Client side application code in JavaScript