MUniCS

Captive Portal Network Authentication Based on WebAuthn Security Keys

Martiño Rivera-Dourado

Supervisors: José M. Vázquez-Naya and Marcos Gestal

Academic year 2021/2022

Universidade Vigo

UNIVERSIDADE DA CORUÑA

Martiño Rivera-Dourado
Captive Portal Network Authentication Based on WebAuthn Security Keys
Master's Thesis. Academic year 2021/2022
Supervisors: José M. Vázquez-Naya and Marcos Gestal

Máster Inter-Universitario en Ciberseguridade
Universidade da Coruña
Facultade de Informática
Campus de Elviña
15071, A Coruña

Abstract

Network authentication is performed via different technologies, which have evolved together with authentication systems in other environments. In all these environments, the authentication paradigm during the last decades has been the well known password. However, passwords have some important security problems, like *phishing* or *keylogging*. In 2019, the WebAuthn standard from the W3C started a new authentication paradigm based on hardware devices known as security keys. Although they are already being used in many web authentication services, they have not yet been integrated with network authentication mechanisms. This work successfully developed and integrated an authentication server based on WebAuthn security keys with a captive portal system. With this solution, users can be authenticated using security keys within a web-based captive portal network authentication system that gives clients access to network resources. The resulting authentication server is compatible with major operating systems like Windows 10 and Ubuntu 20.04, browsers like Firefox and Google Chrome and security keys like the Solokey and the Yubikey.

***Keywords* —** Network authentication, captive portal, WebAuthn, FIDO, passwordless authentication

Resumo

A autenticación de rede realízase a través de diferentes tecnoloxías, que evolucionaron xunto con sistemas de autenticación noutros escenarios. En todos estes escenarios, o paradigma de autenticación durante as últimas décadas foi o coñecido contrasinal. Porén, os contrasinais teñen algúns problemas de seguridade importantes, como o *phishing* ou o *keylogging*. En 2019, o estándar WebAuthn da W3C comezou un novo paradigma da autenticación baseado en dispositivos físicos coñecidos como chaves de seguridade. Aínda que estas xa se están usando en moitos servizos de autenticación web, aínda non foron integradas en mecanismos de autenticación de rede. Este traballo desenvolveu e integrou con éxito un servidor de autenticación baseado en chaves de seguridade WebAuthn cun sistema de portal cativo. Con esta solución, os usuarios poden autenticarse usando chaves de seguridade nun sistema de autenticación de rede con portal cativo baseado en web que da acceso aos clientes a recursos de rede. O servidor de autenticación resultante é compatible con sistemas operativos relevantes como Windows 10 ou Ubuntu 20.04, navegadores como Firefox e Google Chrome e chaves de seguridade como a Solokey e a Yubikey.

***Palabras clave* —** Autenticación de rede, portal cativo, WebAuthn, FIDO, autenticación sen contrasinais

Acknowledgements

This Master's Thesis has been possible thanks to the support of many people. Firstly, it is important to mention the RNASA-IMEDIR research group, where the thesis directors and myself have been working during the last months, welcoming me with many opportunities and guidance in this first steps towards research. In this line, the CITIC research centre from the Universidade da Coruña has been an important help, for their financial and infrastructure resources but also for the unconditional help from their staff and research members.

I would also like to personally thank Jose for their direction during previous works, this thesis and, hopefully, during my next steps within the academic field. Additionally, I want to acknowledge the colleagues that have also form part of the daily work, sharing their knowledge and help everyday, and being always interested in my progress.

Finally, all my studies and the future professional career could not be achieved without the support from my family, who have always been putting faith in my future. There are also other names of real friends that would be worth listing here. They can already identify themselves. They have always been the first ones to be happy for my achievements. *Sen máis, grazas*.

Contents

1. Introduction **1**

1.1. Motivation and study area 1

1.1.1. Passwords and their vulnerabilities 1

1.1.2. WebAuthn: a new standard for web authentication 2

1.1.3. Network authentication and captive portals 3

1.2. Project objectives and hypothesis 4

1.3. Thesis organisation 5

2. State of the Art **7**

2.1. WebAuthn, FIDO and Security Keys 7

2.1.1. WebAuthn: web authentication 8

2.1.2. Cryptographic details 11

2.1.3. Discoverable and non-discoverable credentials 12

2.2. WebAuthn existing technology 13

2.2.1. Security keys 14

2.2.2. Browser and Operating System support 15

2.2.3. WebAuthn server libraries 16

2.3. Network authentication and captive portals 17

2.3.1. Network authentication standards 17

2.3.2. Captive portals 17

2.3.3. Advantages and disadvantages of captive portals 20

2.4. Captive portal existing technology 20

2.4.1. Proprietary firmware 21

2.4.2. Open-source custom firmware 21

3. Materials and methods **23**

3.1. Project planning 23

3.1.1. Project phases and timing 23

3.1.2. Estimated and monitored cost 24

3.2. Required materials 26

3.2.1. Virtualisation software 26

3.2.2. Security keys 26

3.2.3. User equipment: browsers and operating systems 27

3.3. Methodology 28
3.3.1. Initial research 29
3.3.2. Authentication protocol design 29
3.3.3. Authentication server development 30
3.3.4. System integration 31
3.3.5. Captive portal testing 31

4. Authentication development and integration 33
4.1. Authentication protocol design 33
4.1.1. WebAuthn registration and authentication 34
4.1.2. Captive portal integration 35
4.1.3. The scenarios and the use cases 36
4.2. Environment setup 39
4.2.1. Virtual networking topology 39
4.2.2. OpenWRT router installation and configuration 40
4.2.3. Development environment 43
4.3. Authentication server development 44
4.3.1. WebAuthn with discoverable credentials 45
4.3.2. User database and application architecture 45
4.3.3. Administration interface: registration and information 48
4.3.4. Securing the web application 49
4.3.5. ExpressJS and API management 49
4.3.6. Session management: expiration and multiple sessions 50
4.3.7. WebAuthn with non-discoverable credentials 51
4.4. System integration 53
4.4.1. Selection of a captive portal solution 53
4.4.2. Reverse engineering of OpenNDS 54
4.4.3. Cryptographic details of FAS in OpenNDS 56
4.4.4. Extending the API for Authmon integration 57
4.4.5. Adding support for multiple gateways 60
4.4.6. Webpage automatic redirection 60
4.5. Captive portal testing 61

5. Results 63
5.1. WebAuthn Authentication Server 63
5.2. Analysis of OpenNDS FAS 65
5.3. WebAuthn authentication integrated in OpenNDS 68
5.4. Discussion 71

6. Conclusions and future work **73**
6.1. Conclusions 73
6.2. Future work 74

Bibliography **77**

List of Figures **81**

List of Tables **85**

A. Deployment and installation of TLS certificates **87**

B. Reverse proxy deployment **89**

C. Reverse engineering sniffing and logging **91**
C.1. Traffic sniffing with Wireshark 91
C.2. OpenNDS verbose logging 93

1 Introduction

> *If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions.*
>
> — **Albert Einstein**
> (Theoretical physicist)

The present document aims to describe the final project accomplished by the author during his Master's Degree in Cybersecurity. In this chapter, the motivation and project objectives of the Master's Thesis are presented (section 1.1 and 1.2 respectively). Finally, section 1.3 outlines the organisation of the thesis itself.

1.1 Motivation and study area

The society often associates the concept of cybersecurity to passwords. They are omnipresent in information systems all around the world for quite a long time. For a user, they are easy to use and understand and, without a doubt, passwords have become a well-accepted paradigm as an authentication standard.

1.1.1 Passwords and their vulnerabilities

Passwords as an authentication method have become vulnerable to numerous attacks. The most famous is phishing, where an attacker misleads the victim to reveal their credentials [Gup+17]. This attack usually involves techniques like website mirroring, used for creating a fake web portal that looks similar to a real web service that is familiar for the user. Then, the attacker approaches the victim impersonating the familiar service to mislead them to follow a URL link pointing to the fake website. This social engineering technique makes the victim potentially trusting the website and introduce their passwords as required, giving the attacker access to them.

Phishing is not the only existing threat the passwords are subject to. Keyloggers are hardware or software tools that attackers use to log the user input in a system [SC09]. Some hardware keyloggers can be physically placed between the keyboard and the computer to log keystrokes that are directly sent to the attacker. This attack effectively targets passwords that are introduced by the user via the keyboard. Besides, if the keylogger is placed in form of malware, it could even target passwords copied to the clipboard of the Operating System.

Finally, it is a reality that information leaks often appear in the news. These leaks can threaten passwords if they contain password hashes used by authentication servers. Although an attacker is not able to directly use password hashes in the authentication portal, the password can be obtained from them with different techniques. Password cracking techniques are based on automated trial and error methods that produce password hashes from different proposed passwords. If a proposed password produces the same password hash than the stolen one, then the proposed password will be accepted by the compromised authentication server. New cracking techniques are continuously appearing, like shown in [Tir+18].

In conclusion, all phishing, password stealing, and cracking attacks entail a security risk for password-based authentication methods that leave the user unprotected and without control over their credentials. This makes the attacker able to access these systems and, therefore, to steal the information and data they hold.

1.1.2 WebAuthn: a new standard for web authentication

Taking into account that information systems protected by passwords are heavily threatened, some relevant technology companies like Google, Microsoft, Meta or Yubico have started to develop a new authentication method. Their original idea is to use independent hardware devices known as security keys. These tokens can validate the user identity once they are registered in the system, only requiring the user to plug the device in and pressing a button.

For this purpose, back in 2014, these companies created the FIDO Alliance, which has published two main standards during the last few years: FIDO Universal Second Factor (U2F) [@All17] and the FIDO Client To Authenticator Protocol (CTAP) [@All18]. Since then, some security keys compliant with these standards were developed, like the Yubikey [@Yub], the Solokey [@Sol] or the Google Titan Security Keys [@Goo].

In this context, the W3C Consortium developed a new standard compatible with these security keys. WebAuthn is a new W3C standard that aims to complement or even replace passwords as a web authentication method. Relying on the FIDO standards, WebAuthn [@W3C21] defines a browser API that has already been implemented in famous browsers like Firefox and Chrome. In addition, it describes a protocol that allows its usage as an authentication method in web application servers. Consequently, some relevant web applications have already added WebAuthn security keys as an authentication method, to be used together with passwords or even to replace them. These services include Google and Microsoft accounts, some cloud provider portals like Amazon AWS but also in social networks like Facebook.

This new authentication paradigm powered by the FIDO2 Project [@Allb] opens new possibilities for authentication mechanisms in multiple systems. Although the efforts were directed to web application user authentication, there are other computer systems where it could be integrated. One of them is authentication in computer networks, where access to the network or its resources is controlled.

1.1.3 Network authentication and captive portals

There are different ways to provide access control to a network. Some solutions like EAP with 802.1X can control the connectivity at the link level, but others like captive portal systems perform packet filtering to restrict access to network resources after a client attachment. After authentication, both systems aim to authorise the end device to send and receive traffic on the network.

Wireless home network authentication Home personal networks can use different network authentication protocols for controlling connectivity. Although there has been an evolution in the protocols, from WEP to the last WPA3-Personal, they still rely on long-term passwords to authenticate users.

EAP and 802.1X authentication Corporate networks mainly use the Extensible Authentication Protocol (EAP) together with the 802.1X standard for authenticating users in wired or wireless networks. These standards are used for **controlling the connectivity to the networks at link level**. The most common EAP methods still use passwords as an authentication mechanism, like EAP-MD5 and LEAP or some tunneled methods based on PEAP or EAP-TTLS, like MSCHAPv2 or PAP.

Captive portal authentication Controlling access to the network resources can be done by **filtering the traffic at network or link layer** after a successful connection to the network. Captive portals are websites displayed in end devices after network attachment to an Access Point (AP) before granting access to network resources or the Internet.

Captive portals directly fit with the new Web Authentication (WebAuthn) standard. In this Master's Thesis, the author has developed a captive portal that handles user authentication with WebAuthn security keys.

1.2 Project objectives and hypothesis

The thesis hypothesis is that WebAuthn can be used for network authentication in access control with captive portal systems, strengthening these authentication mechanisms and making them resistant to common attacks for which passwords are vulnerable.

For proving this hypothesis, the main project objective is to add an authentication method based on WebAuthn to a captive portal installed on an access layer device of a network. This integration has been done as a Proof of Concept (PoC) guided by the following secondary objectives:

- **OBJ1:** Provide an analysis of a captive portal technology for the integration with the new authentication method.
- **OBJ2:** Develop a functional web authentication application that interacts with the WebAuthn browser API for authenticating security keys.
- **OBJ3:** Integrate the server authentication validation with the captive portal access control system for client authorisation.
- **OBJ4:** Validate the resulting technology with different compatible devices, browsers and operating systems.

1.3 Thesis organisation

The organisation of the Master's Thesis is structured in chapters, dividing the content as follows:

- **Chapter 2** introduces the existing related work, the *State of the Art* (SoA), both for WebAuthn authentication and captive portals.
- **Chapter 3** describes the project planning, together with the required materials and the followed methodology.
- **Chapter 4** explains the core work executed during the project, from the design to the validation of the integrated system.
- **Chapter 5** groups all relevant results of the Master's Thesis.
- **Chapter 6** draws some conclusions for the work accomplished and identifies some future work.

State of the Art

2

Chapter 2 includes the related work. As explained in the section 1.2, the project aims to integrate security keys in a web captive portal. Therefore, in this Master's Thesis, the most relevant technology are the security keys of the WebAuthn standard and the captive portals used for network authentication.

2.1 WebAuthn, FIDO and Security Keys

Since 2014, big technology companies like Google, Microsoft, Meta or Yubico have joined efforts to create a new authentication method based on hardware independent security keys. As shown in figure 2.1, during the last few years there has been a constant evolution of these standards. Firstly, the FIDO Alliance developed a communication protocol to connect security keys to end devices like personal computers or smartphones. Then, the W3C Consortium created a new authentication protocol for web applications, that relied on FIDO for using security keys as tokens.

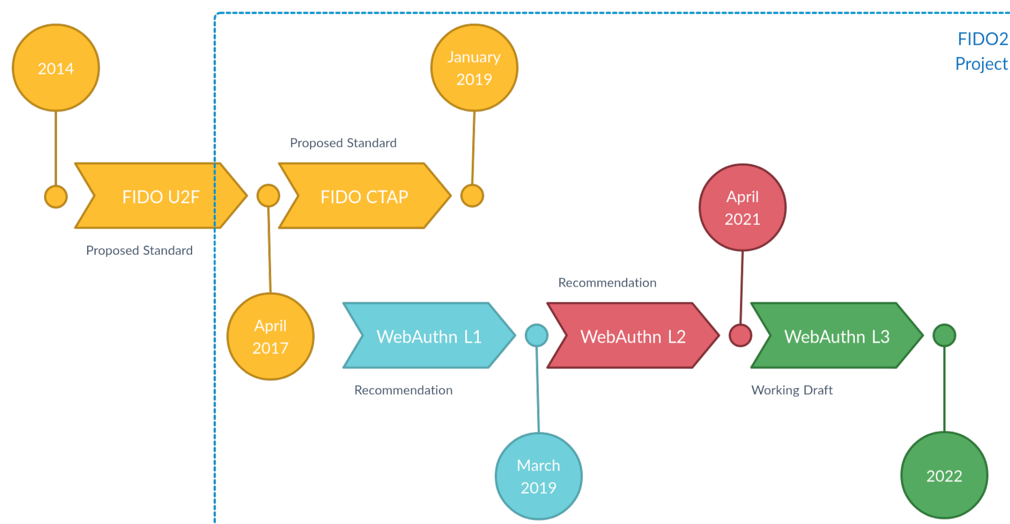

Fig. 2.1.: Evolution of the FIDO and WebAuthn standards. FIDO and WebAuthn standards are wrapped arround the FIDO2 Project, which focuses in the new authentication method based on security keys.

2.1.1 WebAuthn: web authentication

WebAuthn [@W3C21] has evolved during the last few years. However, the main idea of this web authentication scheme has not changed. This section is intended to provide some insight on the fundamentals of this standard. The main involved entities in the protocol are:

- **Relying Party**: the web application that performs authentication, understood as the web server and client-side interface.
- **Client browser and platform**: the browser used to access the web application has to support the WebAuthn API, thus implementing FIDO CTAP communication with the authenticator.
- **Authenticator**: an authenticator is a security key compliant with the authenticator model of WebAuthn, thus implementing the FIDO CTAP communication.
- **User**: the user will interact with the client browser and the authenticator, usually declaring *user presence* by pressing a button.

The figure 2.2 shows the main communication protocol of WebAuthn. We can describe the following flow:

1. The Relying Party serves the client-side web interface, that is, the web authentication portal.
2. The user selects to start an operation, either **registration or authentication**.
3. The browser performs a request to the authentication server of the Relying Party.
4. The authentication server sends a **challenge** to the web browser.
5. Using the **WebAuthn API**, the browser sends a request to the security key.
6. The user interacts with the security key, usually **pressing a button**, to confirm the operation.
7. The authenticator performs the corresponding **cryptographic operations** and sends the response to the web browser, who sends it to the authentication server.
8. The authentication server verifies the operation.
9. Once verified, the authentication or registration is confirmed.

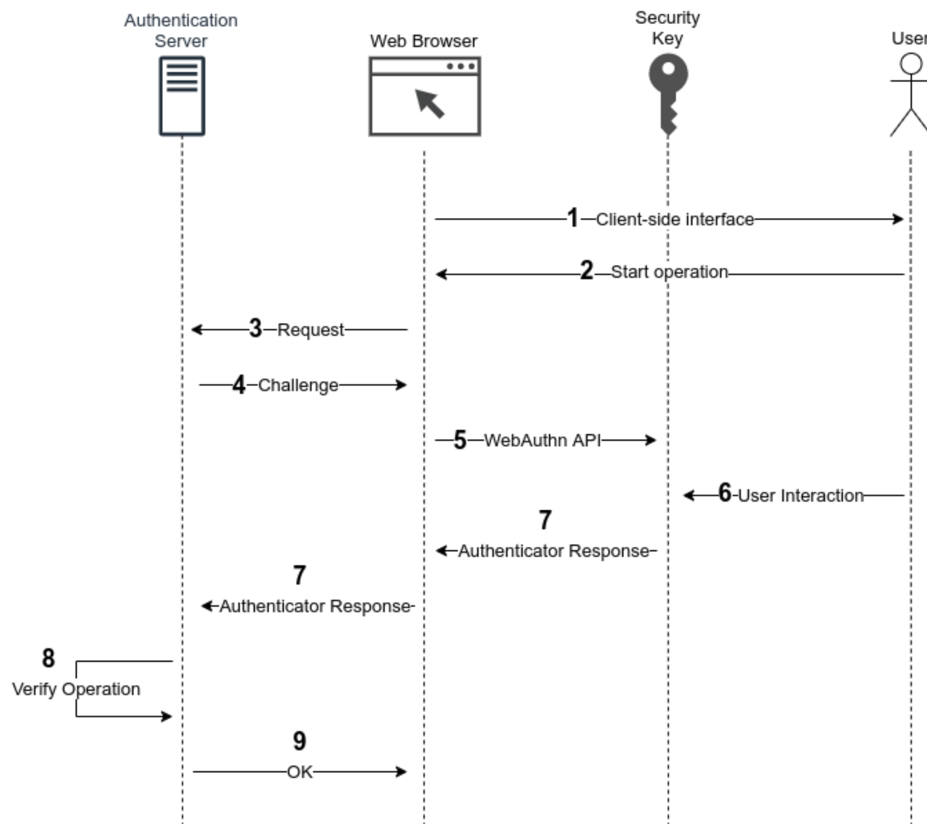

Fig. 2.2.: General communication protocol of a WebAuthn operation.

WebAuthn defines two main ceremonies: attestation (or registration) and assertion (or authentication). Both of them share the same fundamentals in the communication process (see figure 2.2), but they reperesent two different operations. During attestation, the security key is registered in the system and, during assertion, the security key will be used for authentication.

Attestation or registration ceremony (figure 2.3) During attestation, the security key is instructed to create a new public-key credential. In this process, the server generates a random challenge and the registration configuration. When the security key receives the request through the browser, it generates a new credential (see credential types in section 2.1.3), signs the challenge and performs attestation. The attestation process involves extra cryptographic procedures that the Relying Party can later use to validate the security key manufacturer. Finally, the Relying Party saves the generated credential associated with the user in the database.

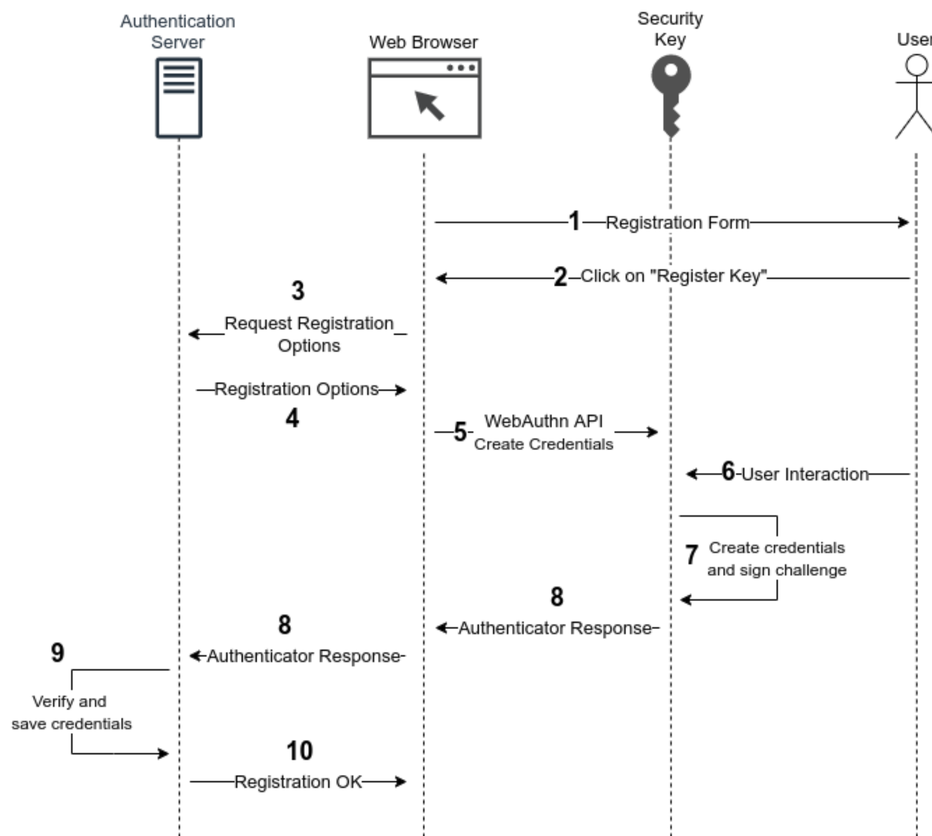

Fig. 2.3.: Communication protocol of a WebAuthn registration. The operation is also known as *attestation ceremony*.

Assertion or authentication ceremony (figure 2.4) During assertion, the security key is challenged to verify it is the same that has been registered before. In this process, the server generates a random challenge and the authentication configuration. When the security key receives the request though the browser, it identifies the credential to be used and solves the challenge (signs the challenge, see section 2.1.2). Finally, the Relying Party receives the solve and verifies it with the stored credential of the database, successfully authenticating the security key.

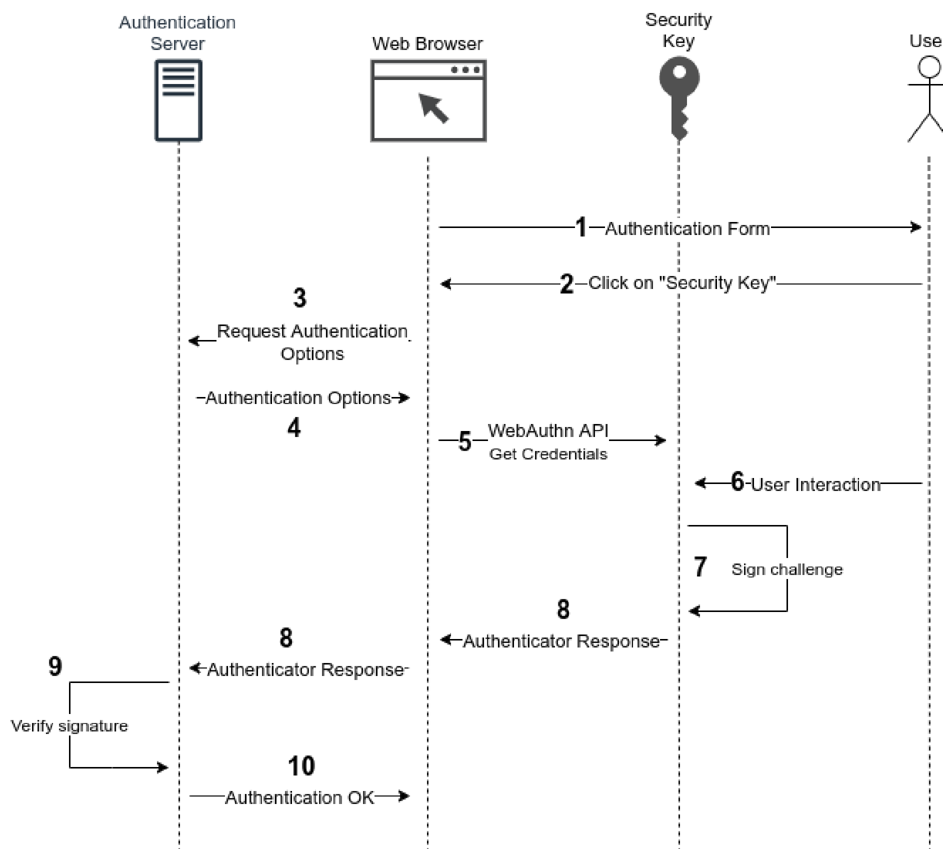

Fig. 2.4.: Communication protocol of a WebAuthn authentication. The operation is also known as *assertion ceremony*.

2.1.2 Cryptographic details

WebAuthn is based on public-key cryptography. The credentials used by the security keys are private keys, while the credentials stored in the Relying Party server are public keys. Authentication occurs when the Relying Party server can verify the signature of a challenge by the security key. In this schema, when the server receives the authentication request, it will generate a random challenge for sending it to the

security key. Then, the security key is in charge to sign the challenge and return it to the server. Finally, using the corresponding public key, the server validates the signature.

Regardless of the type of credential, whether discoverable or not (see section 2.1.3), the security key is the only device able to successfully sign the challenge. In the case of discoverable credentials, the private key is stored within the security key, while in non-discoverable credentials, the private key is received encrypted with another symmetric key, unique to the security key.

2.1.3 Discoverable and non-discoverable credentials

The security keys can be registered with two types of credentials: discoverable and non-discoverable credentials. It is important to notice that every security key compatible with WebAuthn will support non-discoverable credentials, but not all of them support discoverable ones. In other words, non-discoverable credentials have wider support than discoverable ones.

Discoverable credentials (figure 2.5) The credentials are stored into the physical memory of the security key. They are also known as *resident credentials*. When the security key receives a request, it signs the challenge and returns the corresponding user identifier (or handle). This identifier is used by the Relying Party server to identify the user associated with the security key and, therefore, to correctly verify the authentication process. For this reason, using discoverable credentials does not require to identify the user before authentication.

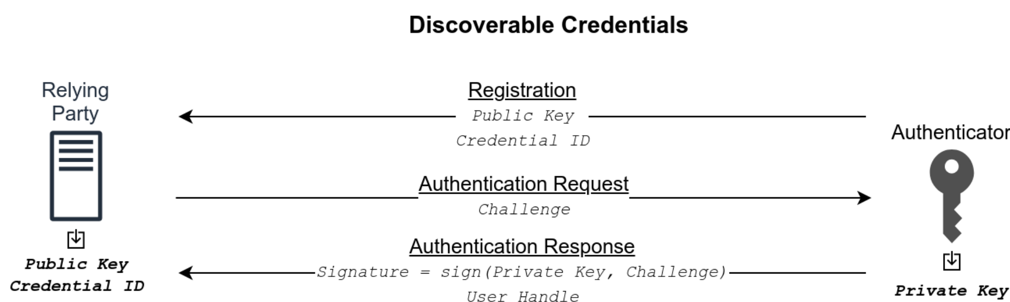

Fig. 2.5.: Discoverable credentials (resident credentials): the private key is saved at the authenticator memory. The public key is sent and stored in the server.

Non-discoverable credentials (figure 2.6) The credentials are not stored into the security key are also known as *non-resident credentials*. In contrast with discoverable credentials, the non-discoverable credentials are sent encrypted to the server during registration. The encryption key which encrypts the credentials is unique to the security key, so only the original device can decrypt them. During authentication, the Relying Party server should identify the user and send their list of encrypted credentials, previously registered, to the security key. Although the credentials are not stored into the security key, they can be decrypted once received during authentication.

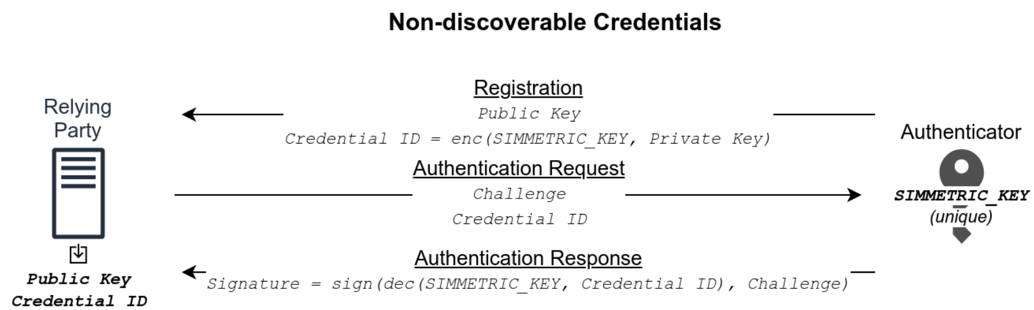

Fig. 2.6.: Non-discoverable credentials (non-resident credentials): the private key is sent encrypted to the server with a symmetric key. This symmetric key is unique to the device. The credential identifier is the result of the encryption, so it is sent during authentication back to the authenticator for decryption. The public key is sent and stored in the server.

In brief, non-discoverable credentials have a wider support, but they require to identify the user previously to the authentication request. For example, this can be done specifying a username. On the other hand, discoverable credentials require physical memory in the authenticator, but they will identify the authenticated user in the authentication response. Therefore, discoverable credentials can be used as a first factor authentication method without username. Non-discoverable credentials, on the contrary, are usually present as a second factor authentication method.

2.2 WebAuthn existing technology

WebAuthn has been already adopted in several services [Riv+21]. Therefore, there are some functional security keys in the market, browsers and Operating Systems have added support and some libraries enable the implementation for web applications.

This section collects some of the most relevant existing WebAuthn technology. Some of them will also be enumerated in chapter 3 as part of the technology stack of the Master's Thesis project presented here.

2.2.1 Security keys

Security keys are the core element of this new authentication protocol. From the first standardisation approach in 2014 with FIDO U2F [@All17], there have been several manufacturers involved in building compatible hardware devices. In 2019, the WebAuthn standard [@W3C21] together with the second version of FIDO CTAP (FIDO CTAP2 [@All18]) added few changes to the security key requirements. The most relevant one is the compatibility with resident credentials, also known as discoverable credentials (see section 2.1.3).

This change improved the authentication standard for allowing the usage of security keys as a first factor authentication method. However, systems that require using discoverable credentials in the authentication process now need compatible security keys. In contrast, devices compatible with discoverable credentials are also backwards compatible with old standards and non-discoverable credentials.

One of the most relevant companies is Yubico, involved in the FIDO Alliance from the beginning. Nowadays, their catalogue of products [@Yub] reveals their expertise in the field. From their popular *Yubikey 5 series* products and their cheaper *Security Key series* devices to more modern innovation with the *Yubikey Bio series*, devices that use biometric fingerprint scanning for user verification.

The *Yubikey 5 series* of Yubico implements other technologies in their devices apart from the features of a WebAuthn authenticator. These include the generation of temporal codes commonly used as second factor, like HOTP or TOTP; the implementation of smart card protocols like PIV, which can store digital certificates securely; OpenPGP encryption standard compatibility to store private keys; or even storing a limited amount of static passwords.

Yubico is not the only manufacturer that includes extra technologies. In fact, there are other open-source projects that develop their firmware in community, that implement some additional features. Solokeys [@Sol] offers open-source cheap devices compatible with WebAuthn whose firmware can be updated. In fact, their *Solokey Hacker* security key allows developers to install a custom firmware, which contributes to the open-source community they have. Another open-source key is the *Onlykey* [@Onl] by CryptoTrust.

In this context, Google opted for create their own security keys to be offered with their online services: the *Google Titan Security Keys* [@Goo]. Google was one of the first companies to implement security keys as an authentication method internally. Nowadays, they offer in their online store a kit of two security keys, for using one as a backup. These devices are compatible only with non-discoverable credentials [Riv20], which fulfils the requirements of Google services.

Finally, the FIDO standards that define the low-level hardware requirements define a set of communication transports for the security keys. Although most of them use the *Universal Serial Bus* (USB), there are other technologies, like *Near Field Communication* (NFC) or *Bluetooth Low Energy* (BLE).

Finally, it is worth mentioning that WebAuthn authenticators can also be implemented as software tokens. Although hardware security keys are the most popular implementation, there exist some initiatives that implement FIDO security keys with software. The most relevant solution so far are Android smartphones. These allow using the Android KeyStore as a key management for managing WebAuthn credentials and, therefore, use the smartphone as security key.

2.2.2 Browser and Operating System support

Web browsers are client software that run web applications that implement WebAuthn authentication. These browsers have to implement the W3C WebAuthn API to be compatible with the standard. However, as explained before, FIDO U2F was created before WebAuthn, so some browsers traditionally implemented the old protocol.

The web of Mozilla [@Moz] includes a detailed table of the compatibility of the WebAuthn API operations with different web browsers. According to these tables, the major web browsers are supporting the complete set of WebAuthn operations, excluding *Internet Explorer*, *Samsung Internet* or the *WebView Android*. In the case of *Firefox*, the current implementation is not compatible with discoverable credentials. In contrast, one of the most updated browsers is *Google Chrome*, actively involved in the protocol development. These differences are taken into account during this Master's Thesis for developing and testing the implementations.

As analysed in [Riv+21], Operating Systems have started to support WebAuthn and FIDO standards for other authentication mechanisms, further than web applications. Yubico has developed a *Pluggable Authentication Module* (PAM) for using FIDO authenticators as a token to authenticate users on Linux-based Operating Systems. In Windows, Yubico has also created Yubico Login, which allows using Yubikeys

to sign-in. In order to use other FIDO authenticators, Microsoft has created a business solution for FIDO2 authentication through their Azure Active Directory Multi-Factor feature, which uses Kerberos tickets to authorize users with on-premise Active Directory controllers.

The Windows Operating System has developed their own Windows' native WebAuthn API. In contrast with Linux-based Operating Systems, browsers running in Windows have to interact with security keys through this API, limiting their functionality to the specific implementation level of the API. Similarly, the Android Operating System has a native WebAuthn platform for interacting with security keys. However, this platform is not yet compatible with discoverable credentials. This makes the non-discoverable credentials the only type of credentials that all applications, including web browsers, implement in their WebAuthn authentication processes.

In conclusion, all these concerns have to be taken into account to develop technology solutions that are compatible with different environments. The Firefox browser and the Android platform, for example, are not compatible nowadays with discoverable credentials, which can limit their usability with authentication processes configured with this type of credentials.

2.2.3 WebAuthn server libraries

For a web application to support an authentication process based on WebAuthn security keys, its server has to implement the required server functionality. As analysed in [Riv20], this implementation is not trivial. However, there exist several software libraries that have been developed during the last few years.

There are several projects for different programming languages. In Python, `py_webauthn` [@Lab22a] is a project developed by *Duo Labs* to be used with web servers like *Flask* or *Django*. The same organisation is also developing `webauthn`, a implementation for Go [@Lab22b]. There exist other projects for traditional programming languages like .NET, with `fido2-net-lib` library [@Lib22]; or Java, with `webauthn4j` [@web22]. Finally, there exist libraries like `SimpleWebAuthn` [@Mil22] that implement both server and client libraries, in Typescript and JavaScript respectively.

2.3 Network authentication and captive portals

Network protection is a key area in information systems security. In this field, access control is one of the most important issues to address. Being inside a network helps an intruder in leveraging attacks such as eavesdropping data or Denial of Services (DoS), so it it is important to take a preventive approach restricting access to the network with security policies.

This section summarises the existing network authentication standards that help enforcing network access control policies, focusing in captive portals.

2.3.1 Network authentication standards

Enforcing access control policies, on how end devices connect to a network, is key to avoiding internal threats and, therefore, to contribute to the cybersecurity of wired and wireless networks. The approach for enforcing access control policies in networks are the Network Access Control (NAC) systems. These ensure that, within a LAN, connectivity is only granted to authenticated and authorised devices or users.

Theoretically, there are three main tasks to perform when a user accesses a network or system: authentication, authorization, and accounting (AAA). Access to systems protected by NAC is controlled by an authentication process that verifies the user or device identity. The most widespread standard for access control authentication in enterprise LANs is the Port-Based Network Access Control (IEEE 802.1X)(PNAC). This technology is implemented in a considerable number of networks that society uses every day, i.e., Wi-Fi networks with WPA2-Enterprise (IEEE 802.11i) or Ethernet ones with MACsec (IEEE 802.1AE).

Authentication methods based on PNAC often require enterprise-level authentication mechanisms or network equipment. On the other hand, captive portals have also being used for authentication, mostly in wireless networks.

2.3.2 Captive portals

A captive portal is a web page served after the connection to a network and before granting the client access to network resources. There are different uses for a captive

portal, like the acceptance of a end-user license agreement, survey completion, payment or authentication.

Captive portals are placed in many different networks, but they have been popular on open wireless networks in hotels, airports or shopping centres. Their traditional use also included marketing purposes for collecting user data before granting access. In this line, Ali et al. [Ali+19] analysed the privacy issues of 67 unique public Wi-Fi hotspots that use captive portals, which revealed the collection of a significant amount of privacy-sensitive personal data.

However, captive portals can be used to enable web authentication for verifying the user identity before granting access to the network. In this scenario, the user accesses the captive portal through a web browser and, after the authentication, the network operator server will release the client from the imposed traffic limit, i.e., the network captivity.

Captive portal architecture

According to RFC 8952 [LDL20], which proposes a standard captive portal architecture (see figure 2.7), the implementations of captive portals generally require:

1. **A web server**, which is responsible of hosting the captive portal.
2. **A method to allow/block network traffic**, which enforces the captivity or network filtering to restrict access to the network resources.
3. **A method to alert the user**, displaying the captive portal in the user interface and allowing to perform the required action, like authentication.

For allowing and blocking network traffic, captive portal policies usually implement a packet filtering technique. The enforcement device blocks all unwanted traffic until the user has successfully authenticated. For adding such policies, as stated in the RFC [LDL20], a user equipment identifier is needed. Examples of these identifiers that are often used in these scenarios are link-level MAC addresses or network-level IP addresses. Despite they often used in captive portals, both of these identifiers are vulnerable to different types of identifier spoofing [AT10]. For this reason, [LDL20] recommends the usage of the physical interface as a user device identifier, when possible.

The aforementioned RFC [LDL20] also states that current captive portals implement some variations of DNS or HTTP forging for alerting the user and, therefore, for displaying the captive portal. This has been the most common solution so far. Soon after this RFC was published in 2020, there have been published related works

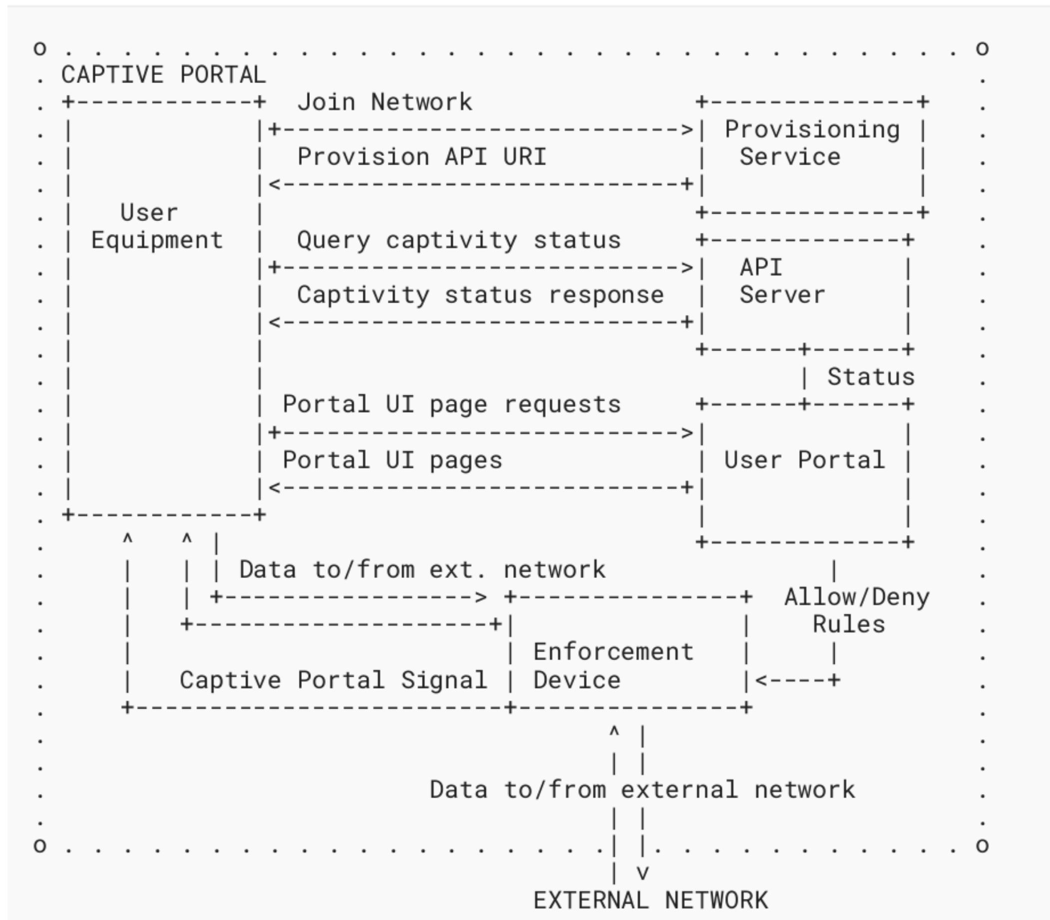

Fig. 2.7.: Captive portal architecture as proposed in RFC 8952 [LDL20]. The captive portal architecture includes the user equipment, working as a client, and the elements of the captive portal implementations.

like RFC 8910 [KK20] which describes methods for captive portal identification in DHCPv4/v6 and Router Advertisements (RAs).

2.3.3 Advantages and disadvantages of captive portals

As explained in [MZB20] there are several disadvantages of using authentication with captive portals. The current implementations of most captive portals rely on a web-based Man-in-the-Middle (MitM) hack that can cause redirection problems due to certificate mismatches, among other problems. In addition, it is important to notice that captive portal authentication occurs at the application level. Therefore, this type of authentication is not cryptographically related with encryption keys in Wi-Fi with 802.11i, leaving the link level unprotected.

It is remarkable that there have some improvements in this field. The RFC 8910 [KK20] published in 2020 adds a DHCP code to improve the captive portal detection. In wireless networks, the appearance of WPA3, a new Wi-Fi Alliance standard, adds encryption to open Wi-Fi networks with the Opportunistic Wireless Encryption (OWE) protocol [@Alla]. This feature mitigates eavesdropping attacks on open wireless networks and, therefore, makes them more secure to use a captive portal on them.

On the other hand, captive portals are a flexible solution to implement new types of authentication that can be based on web technology. The reason lies on that the only requirement for the client is to have a compatible web browser. Besides, web applications are familiar to everyday Internet users, so the authentication process should be more user friendly than configuring other network authentication in their devices.

2.4 Captive portal existing technology

There are several ways to implement a captive portal. Most of the solutions nowadays are proprietary implementations delivered in business routers, but there are other open-source or custom router firmwares that offer this solution. This section includes a brief relation of the existing captive portal technology.

2.4.1 Proprietary firmware

Proprietary business solutions that implement captive portal often have built-in software in their product firmware. From the router configuration, a network administrator can directly setup a captive portal in their wireless network. This router implements both the web server and the enforcement device to implement the correct access policy.

There are many solutions in this field. For example, the company NETGEAR names their product *NETGEAR Instant Captive Portal* [@NET], an enterprise-level captive portal that can be setup in minutes. It allows authentication and even configuring an external RADIUS server. Some Cisco *Small Business* wireless equipment add the possibility to configure a captive portal with similar characteristics, including the external authentication [@Cis17]. In this line, other companies like Linksys offer this solution in their business access points [@Lin]. Finally, ASUS implements guest networks in some of their domestic routers to allow setting a web-based static authentication [@ASU21].

All these solutions showcase the usages of captive portals in existing networking products, for business and some domestic usages. However, there are companies that have developed specific networking products for captive portals. WAVER is a company of network equipment that has focused on guest wireless networks, captive portals and marketing. Their products are oriented to business where captive portals are typically found, like airports, coffee shops, hotels or restaurants. As shown in their website [@Wav] they implement different authentication methods in their captive portal for enabling access to Internet, including username and password authentication or even scanning temporal codes issued by the business.

2.4.2 Open-source custom firmware

Apart from the solutions built-in the proprietary network equipment developed by the manufacturers, there exist other open-source firmwares compatible with captive portals. These pieces of software can often be installed on compatible proprietary equipment for their usage.

pfSense is one of the most popular open-source firmwares available in the market, mainly developed by Netgate. Similar to other proprietary solutions mentioned in section 2.4.1, pfSense offers authentication based on local configured accounts or an external RADIUS server [@pfS]. Their firmware is compatible with several routers, including Netgate but also others like some TP-Link and Protectli models.

On the other hand, OpenWRT is a Linux-based operating system that targets embedded systems [@Pro16]. This firmware can be installed in some routers from Lynksis, Asus, TP-Link and Netgear, among others. OpenWRT has many available add-on packages that add features to the core light firmware. One of the available packages, built as a separate open-source project, is OpenNDS. OpenNDS [@Whi22], based on NoDogSplash, is a captive portal software developed in C compatible with this firmware. Among their options, it is worth mentioning it adds the possibility to externalise the web server used in the captive portal. This configuration is known as *Forwarding Authentication Service* (FAS).

As explained in section 4.4.1, OpenNDS is used to integrate WebAuthn authentication with a captive portal in this work. The main reason to choose OpenNDS installed on OpenWRT is the ability to configure a remote web server with the FAS configuration, while using open-source solutions for the firmware and the captive portal software.

3 Materials and methods

The work accomplished during this Master's Thesis was planned according to the specific objectives. In this chapter, the methodology of each project phase is described, together with the engineering planning and cost estimation. Finally, the materials and technology required for this work are listed.

3.1 Project planning

This section describes the engineering project planning. First, the project phases are described with the final timing gantt diagram, complemented with the time delays of the project execution. Then, the cost estimation and monitored cost taking into account the project delays are summarised in tables.

3.1.1 Project phases and timing

According to the objectives enumerated in section 1.2, the work of the project has been divided in five different phases. Each of them has a specific methodology in their own, described later in section 3.3. Executed as a cascade project, the main project phases and subphases are:

- **PH1:** Initial research.
  - **PH1.1:** Captive portal implementations.
  - **PH1.2:** WebAuthn existing libraries.
  - **PH1.3:** Environment setup.
- **PH2:** Authentication protocol design.
  - **PH2.1:** WebAuthn registration and authentication.
  - **PH2.2:** Captive portal integration design.

- **PH3:** Authentication server development.
  - **PH3.1:** WebAuthn authentication development.
  - **PH3.2:** End-to-end authentication testing.
- **PH4:** System integration.
  - **PH4.1:** Captive portal integration analysis.
  - **PH4.2:** Authentication server integration with captive portal.
  - **PH4.3:** Integration testing.
- **PH5:** Captive portal testing.

This project was planned with a time duration of 7 months. However, taking into account the time delays, the project had a total duration time of 9.5 months. Figure 3.1 shows a representation of the phases planning in a Gantt diagram.

Major delays were caused by PH4.1 and PH4.2 of *PH4: System integration*, as it is explained in section 4.4. Other aspects related to the development of the authentication server also delayed the iterative work of the project phase *PH3: Authentication server development*.

3.1.2 Estimated and monitored cost

The presented project costs are mainly defined by the human resources involved in the project. Specifically, two project managers (Ph.D.) and one research assistant hired part-time time during the project execution. The estimated and final costs of table 3.1 were computed with University of A Coruña parameters with prorated payments with the correspondent monthly brute salary.

Description	Salary month	Salary hour	Estimated	Monitored	Estimated cost	Real cost
Manager (x2)	2,787.25 €	17.42 €	210 h	285 h	7,316.53 €	9,918.65 €
Developer	2,379.54 €	14.87 €	1120 h	1520 h	16,656.80 €	22,580.35 €
				TOTAL	23,973.33 €	32,498.99 €

Tab. 3.1.: Human resources estimated and monitored cost in euros.

Taking into account the materials and human resources costs, table 3.2 shows the final estimated and monitored project costs. In conclusion, the delays in the project caused an increment of the 24.27% in the final cost with respect to the forecasted cost.

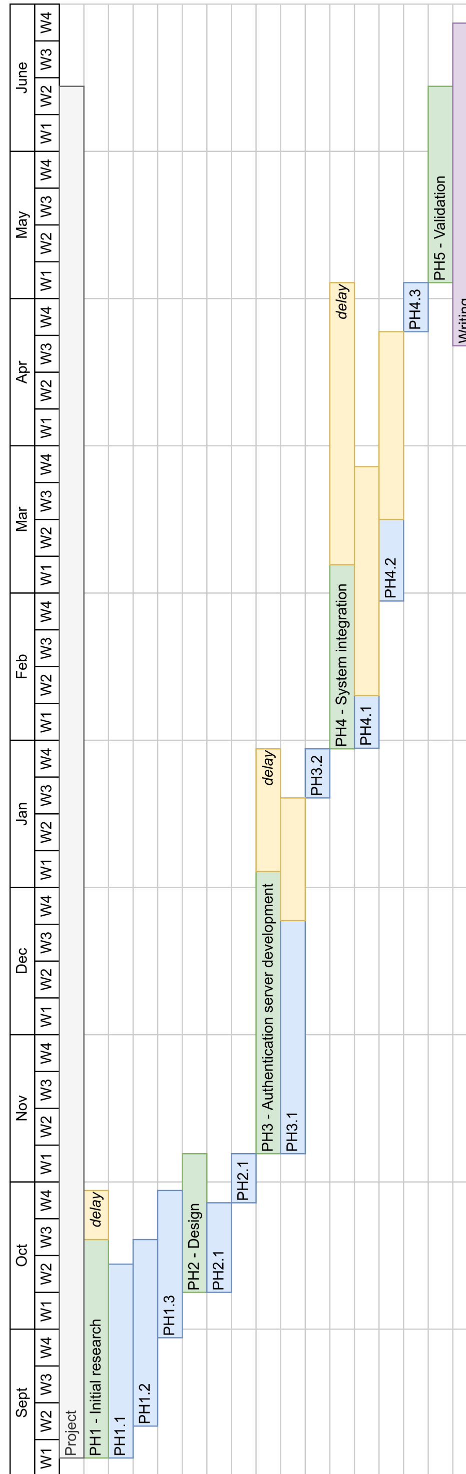

Fig. 3.1.: Gantt diagram of the Master's Thesis project. In green, the main project phases and in blue the corresponding subphases. Yellow boxes represent the unexpected final delays of the project, with respect to the planned timing of each phase and subphase.

Description	Forecasted cost / unit	Real cost / unit	Units	Forecasted cost	Real cost
Human resources	23,973.33 €	32,498.99 €	1	23,973.33 €	32,498.99 €
Security keys	50 €	50 €	4	200.00 €	200,00 €
Laptop	1.500,00 €	1.200,00 €	1	1.500,00 €	1.200,00 €
			TOTAL	25,673.33 €	33,898.99 €

Tab. 3.2.: Total estimated and monitored cost in euros, including human resources and materials

3.2 Required materials

This section includes the required materials for the execution of the Master's Thesis project. According to the project phases and the methodology (see section 3.3), the required materials can be summarised in:

1. Virtualisation software.
2. WebAuthn security keys.
3. Compatible user equipment.

3.2.1 Virtualisation software

In this project, we have used a virtual environment (see section 4.2). It is important to highlight that all this work could be done in a physical environment with networking and user equipment, which reliably represents the real scenario where the results of this Master's Thesis could be applied (see section 6.2). However, a virtual environment can reproduce the physical scenario and provide quicker deployment processes for the development and validation tasks.

There are several virtualisation software alternatives. In this project, we have used *VirtualBox 6.1* for setting up the environment, which allows to create different Virtual Machines (VMs) that can be attached to different virtual networks. Also, it easily integrates with the host hardware for interacting with USB security keys.

3.2.2 Security keys

WebAuthn security keys, also known as WebAuthn authenticators, are devices used to authenticate users in a WebAuthn-compatible authentication system. As explained

in section 2.2.1, there are two main types of security keys: (1) hardware security keys and (2) software security keys.

According to a previous work done in [Riv20], security keys that support WebAuthn differ in some characteristics. The most relevant difference for this work is their ability to manage discoverable (a.k.a resident) credentials (see section 2.1.3). In this work, both discoverable and non-discoverable credentials are supported, as shown in the results (see section 5.1).

Table 3.3 lists the security keys used during this project (see figure 3.2). They are classified according to their support with discoverable credentials and the type of authenticator. The support of discoverable credentials is also restricted by the support of browsesrs and operating systems (see section 3.2.3).

Fig. 3.2.: Security keys used in this project. From left to right in the same order than in table 3.3.

Security key	Type	Discoverable Credentials
Solokey Hacker	Hardware	Supported
Yubico Yubikey 5 NFC	Hardware	Supported
Yubico Security Key	Hardware	Not supported
Google USB-A/NFC Titan Security Key (K9)	Hardware	Not supported

Tab. 3.3.: Security keys used during this project. They are classified according to their support with discoverable credentials.

3.2.3 User equipment: browsers and operating systems

The client equipment required for the environment includes a personal computer that can be attached to the network. In the virtual environment, the user equipment is based on a Virtual Machine (VM). The client VM operating systems used are:

1. Canonical Ubuntu Desktop 20.04.4 LTS.
2. Microsoft Windows 10 Enterprise v1909.
3. Kali Linux 2021.4 (kali-rolling).
4. Manjaro KDE 21.3.2.

Kali Linux was used during the development phase, while Ubuntu, Windows and Manjaro were mainly used for validation end-to-end tests after some features were implemented. As seen in the last project phase, Ubuntu, Windows and Manjaro have implemented a captive portal detection mechanism for alerting the user themselves, without the need for the user to generate web traffic.

Moreover, for the client to be able to authenticate using security keys, the web browser should have a compatible implementation of the WebAuthn API. As tested in [RGV20] and with the additional tests performed in this Master's Thesis, the browsers used during the project on Windows and Linux are:

1. Google Chrome.
2. Chromium.
3. Mozilla Firefox[1].

During the captive portal testing (see section 4.5), different browsers and user equipment were subject to test. Other scenarios would involve setting up a hardware environment with networking equipment (see section 6.2).

3.3 Methodology

As described in section 3.1.1, the project is divided in several phases. The project follows a cascade methodology, in which the main project phases are executed sequentially. As seen in the Gantt diagram in figure 3.1, the main phases are initiated in sequence. This rule excludes the initial research (PH1) and design (PH2) phases, which have finally overlapped at the end of phase PH1. That is, the initial research did not end before the design phase was started.

During the whole project, the phases and all details were documented in a logbook. For this purpose, the author used the *Notion* software. Each project phase was documented separately according to its specific methodology. After the completion

[1] Mozilla Firefox in version 101.0 or lower does not yet support WebAuthn discoverable credentials.

of a project phase, the next one is initiated. The following subsections describe the methodology of each main project phase. Some methodologies used in different project phases may be similar, but they are adapted to the specific purposes of the phase.

3.3.1 Initial research

The first project phase is a necessary research on the state of the art of the Master's Thesis related technology and related previous work. The results of this phase are documented in chapter 2, which includes a review of:

1. WebAuthn, FIDO and security keys technology: the new authentication standards and existing technology related to them.
2. Network authentication and captive portals: the usage of captive portals in network authentication and other network authentication standards.

The initial research was based in a scientific literature review complemented by the review of (1) informal tutorials published in technology blogs, (2) open source projects and (3) market of related technology solutions. For each of the aforementioned specific knowledge areas, the scientific literature review guides the researcher to find existing standards and scientific articles. Then, using these standards and the available informal tutorials, the open source projects are researched together with a market analysis.

For example, when researching WebAuthn and the security keys, a market research of the available security keys brands, like Yubico or the Solokeys eventually guides the research towards other existing technology and standards. These complement the literature review which finally results in a deep dive on the specific knowledge area. Once all this is gathered, it is documented.

3.3.2 Authentication protocol design

As stated in the introduction of the methodology, this phase is related with the initial research. When the existing standards and technology solutions were analysed, the design of the integration of WebAuthn with the captive portal is documented. Then, this design will be used as a reference during the next project phases that involve the development work. The design is included in section 4.1.

For designing the authentication protocol, first a generic scenario is considered. In this scenario, a simple WebAuthn web application implements the registration and authentication with security keys. Finally, this scenario is modified and extended to consider the captive portal use cases. That is, registering and authenticating users for network authentication in a captive portal compatible with security keys.

3.3.3 Authentication server development

The purpose of this project phase is to implement the generic scenario of a WebAuthn web application. Prior to this phase, the environment is setup taking into account the initial research and protocol design, as explained in section 4.2.

During the development itself, an adaptation of the Scrum methodology is used. In this case, the Scrum Master and Product Owner are roles played by the complete team. Therefore, the sprint backlog is defined according to a set of requirements identified from the design phase but also from the review of each sprint. The development tasks in each sprint are tracked using a Kanban board in *Notion*, with the following columns: (1) not started, (2) in progress, (3) bug, (4) completed. The third column is related with development tasks derived from a bug detected when testing the system.

Regarding the software version control, the author used *Git*. The software commits were grouped in a separate branch for each sprint. At the end of the sprint and after its review, the branch is merged into the main branch. Additionally, for achieving a detailed version control history, commits represent a minimal software change. They are named according to a naming convention:

- **feat**: A new feature is introduced with the commit changes.
- **fix**: A bug fix occurs.
- **chore**: Changes that do not relate with a feature nor a fix, and do not update the source files. For example, updating dependencies.
- **docs**: Updates to the documentation.
- **style**: Changes to the style and format.
- **refactor**: Refactored code that does not fix a bug or adds a feature.

3.3.4 System integration

Once the generic authentication server was developed, the system integration phase aims to integrate this server with a captive portal, according to the final scenario design.

Although the work done in this phase is related with development tasks similar to the previous phase, the nature of the problem is different. The iterative methodology and the version control system used in this phase will follow the same idea. However, during integration with an existing captive portal system, the implementation of software increments should be based on frequent testing in the final system. End-to-end (E2E) testing introduced in each development task of each sprint aims to validate the integration implementations in the final environment, which ensures the final increment is working integrated in the captive portal.

Finally, it is important to mention that during this integration a reverse engineering methodology helped analysing the specific working of the captive portal system for the integration with the authentication server. In this process, the network traffic was captured and manually analysed. Together with the system logs, the findings were documented to help to identify the system requirements that guided the implementation. This process is explained in detail in section 4.4.2.

3.3.5 Captive portal testing

At the end of the project design and development phases, the final integrated system was validated with different operating systems and browsers. According to their WebAuthn support, as explained in section 2.2.2, the registration and authentication processes are tested in the virtual environment.

For this purpose, the virtual environment was setup with different client machines, according to the operating system to be tested. First, different security keys were registered under the same conditions. Then, the different operating systems and browsers are used to perform authentication via the captive portal with these registered security keys. Once the authentication process is validated, the registration process is also tested for each environment. All these tests are documented in section 4.5.

Authentication development and integration

4

As stated at the project planning in section 3.1, the work done in this Master's Thesis was divided in five main phases. This work can be summarised in the sections included in this chapter. The tasks here described resulted in a functional system integration serving as a Proof of Concept (PoC), as shown in chapter 5.

In short, the project began with an initial research of the existing technology, which resulted in the analysis included in the State of the Art (SoA, see chapter 2). During this research, the author configured and deployed the environment for the later project phases.

The first section of this chapter (section 4.1) describes the authentication design phase, which included the scenario analysis, the WebAuthn protocol design and the final integration analysis. Regarding the development, the first implementation covered the WebAuthn authentication server, iteratively developed (see section 4.3). Once this development fulfilled the requirements, it was integrated with the captive portal system during the system integration phase (see section 4.4). Finally, the system was validated during the captive portal testing (see section 4.5) with different browsers, operating systems and security keys.

4.1 Authentication protocol design

Before any implementation, the project started with a design phase. After the initial research of the existing technology, the design phase was based in an authentication scenario and use cases, described hereafter. Accordingly, this section also includes the design of the server operations according to WebAuthn registration and authentication ceremonies. Finally, the integration with the captive portal scenario has driven to a more complete design to be used in the last implementation phase of the project.

4.1.1 WebAuthn registration and authentication

Figure 4.1 shows an overview of the generic scenario where WebAuthn registration and authentication is placed: a web application. In this scenario, the user interacts with the web application through a web browser. The web browser, compatible with the WebAuthn API, interacts with the security key the user has plugged in. Both registration and authentication operations are then handled at the web application server, that implements WebAuthn support.

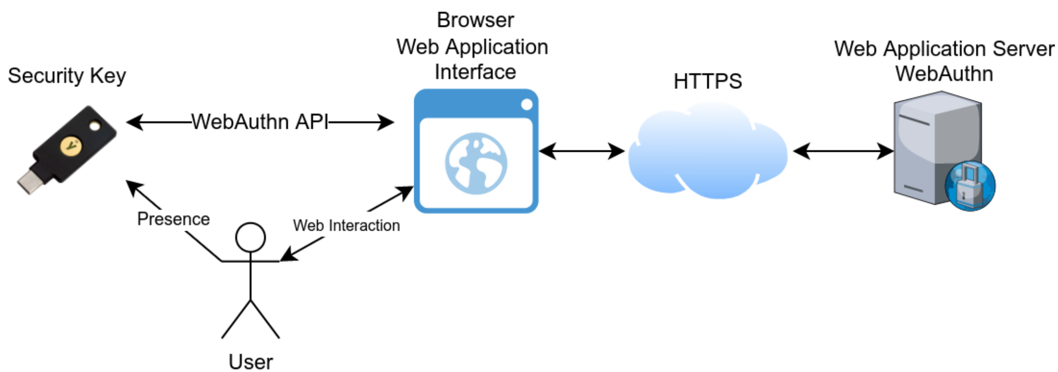

Fig. 4.1.: Overview of the general WebAuthn registration and authentication scenario.

In this context, the web application server should implement verification of both WebAuthn operations: registration (*attestation*) and authentication (*assertion*).

The design proposed for the authentication server of this project is based on sessions. Therefore, as represented in the statechart diagram of figure 4.2, after the verification of WebAuthn authentication, the web application should create and store a new user session to manage a list of authenticated users, for an eventual use as a source list for authorisation, as it explained in section 4.1.2.

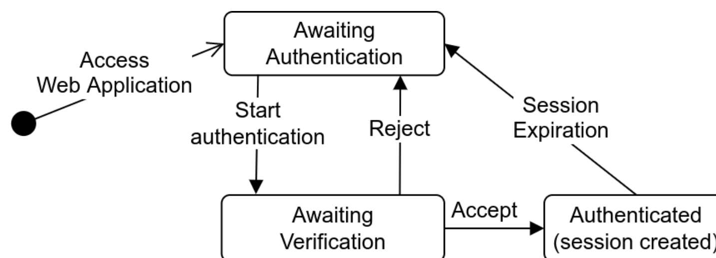

Fig. 4.2.: Statechart of WebAuthn authentication and the basic session management.

4.1.2 Captive portal integration

As proposed in the hypothesis of this Master's Thesis, WebAuthn could also be used to authenticate users in the context of captive portal network authentication. In this proposed environment, an access point or router redirects the client to a captive portal. After the web authentication, the router changes the filtering rules to authorise the client access to network resources.

Figure 4.3 overviews the scenario. In the proposed design, the router redirects to a captive portal serving a web application that implements WebAuthn authentication. This server also implements a module that interacts with the router for the proper authorisation of clients.

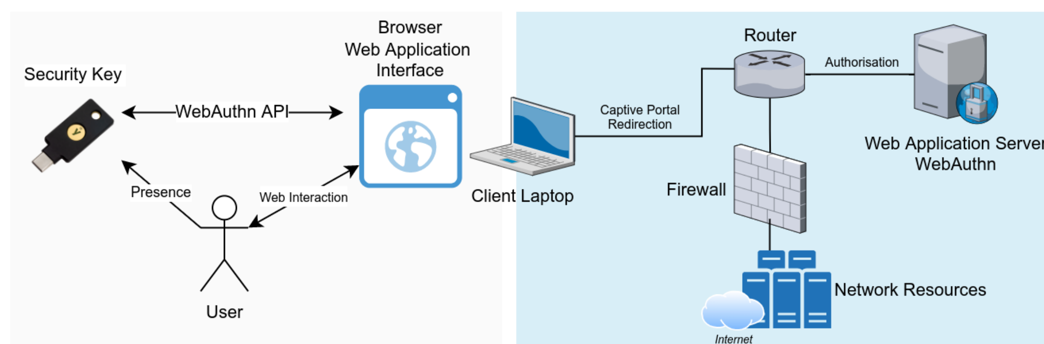

Fig. 4.3.: Overview of the captive portal integration with WebAuthn.

According to the design proposed in the WebAuthn registration and authentication design (see section 4.1.1), the integration will be based on the user verified active sessions for authorisation. That is, after successful client authentication, the client session is created. Then, the router interacts with the web application server to retrieve a valid list of clients to be authenticated. Figure 4.4 completes the previous statechart of session management, integrating the router interaction.

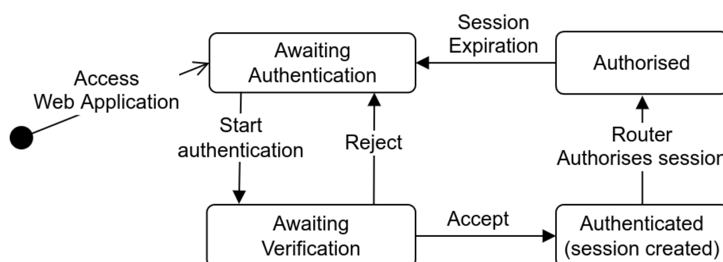

Fig. 4.4.: Statechart of the session management integrated with the router serving a captive portal that authorises the client.

4.1.3 The scenarios and the use cases

The objective of the project is to create a WebAuthn authentication server that can be integrated into a captive portal technology. Therefore, we can describe two main software increments, that result in two general scenarios. The first one (the *generic scenario*) includes as a web application authentication server that allows arbitrary registration and authentication of WebAuthn security keys. The second one (the *final scenario*) represents the complete scenario of a captive portal network authentication system with WebAuthn, which is based on the first scenario. This last scenario represents the complete scenario where the contribution of this Master's Thesis is based on.

Generic scenario: WebAuthn web application

The first scenario includes the basic functionality of the project. That is, registering and authenticating a WebAuthn security key. This scenario aims to represent a generic view of web authentication using WebAuthn security keys, present nowadays in several production web services.

Considering the generic nature of this scenario, but also in view of the future integration with a captive portal, the users and their security keys are only registered by an administrator. For this Proof of Concept, this restriction adds control on the user registration, and can be seen as a simplification of other scenarios where the user management can be more complex. For example, other scenarios involving external systems that provide a centric user management, like LDAP.

Consequently, two main use cases are defined:

1. **Registration of a WebAuthn security key by the administrator**. The administrator register security keys associated to users. For adding new users, the administrator should specify a new username. If a registered username is specified, the security key is added to the corresponding existing user.
2. **Authentication with a WebAuthn security key by the user**. The user authenticates by using a security key that has been previously registered for that user. This use case creates a new web session associated to the user.

These two main use cases, together with the secondary ones, are represented in figure 4.5. The *administrator* and *user* roles represent a simple general Role-Based Access Control (RBAC) that restricts the user registration operation to actors with the *administrator* role.

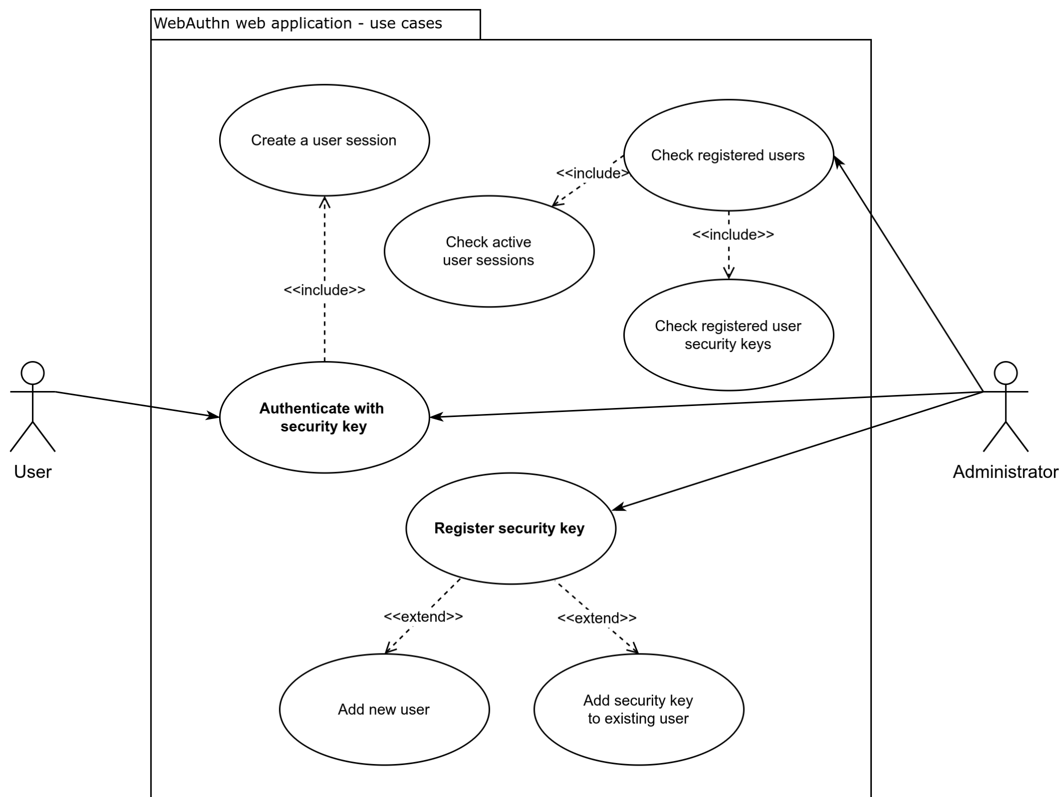

Fig. 4.5.: Use cases of the generic scenario: WebAuthn web application. The main actors *administrator* and *user* are associated with the RBAC roles.

Final scenario: Captive portal with WebAuthn

The second scenario describes the complete scenario the contribution of the Master's Thesis is based on. Specifically, this scenario includes the captive portal technology integrated with the WebAuthn web application of the generic scenario. Therefore, it will be used to authorise network traffic after a successful authentication.

The captive portal is a new actor that interacts with the authentication server. Using the same web application of the generic scenario, the implementation should add a new use case:

1. **Authorisation of all authenticated user sessions by the captive portal.** Once users were authenticated in the web application, the captive portal should authorise them. This use case will provide a list of active user sessions associated with the captive portal.

Figure 4.6 shows the use case diagram of the final scenario involving the Captive Portal actor. This actor represents the captive portal technology that controls the access to network resources. The new system integrates the corresponding new use case.

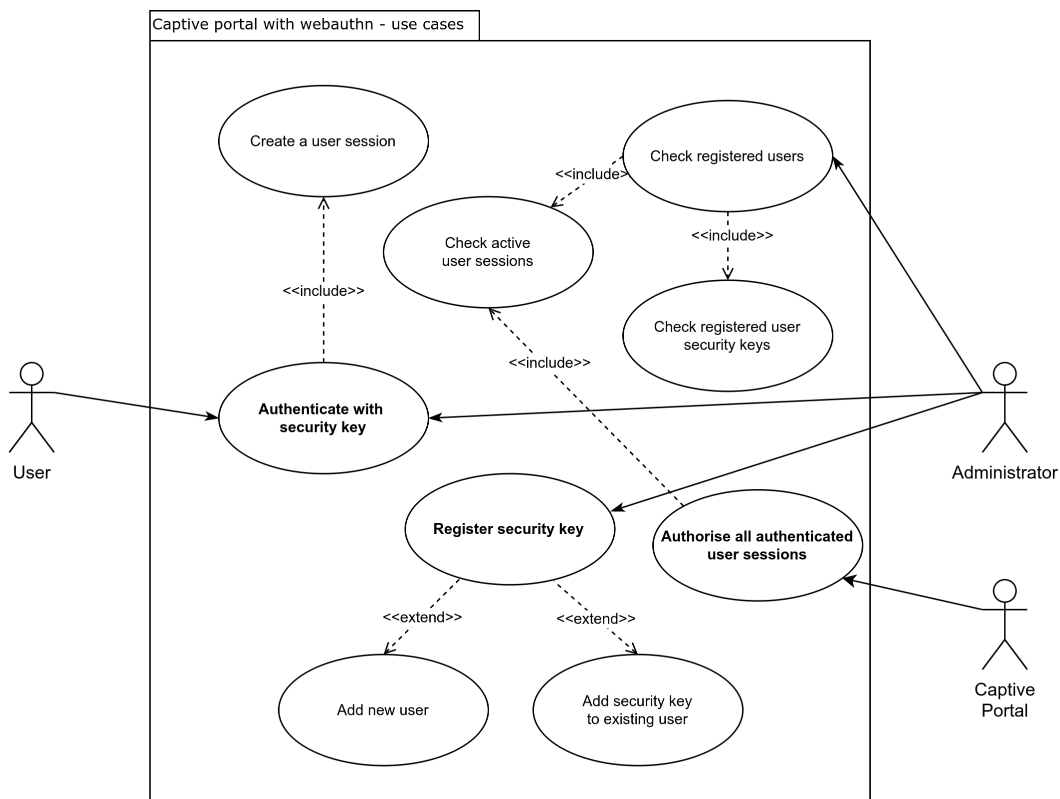

Fig. 4.6.: Use cases of the final scenario: Captive portal with WebAuthn. The Captive Portal actor represents the captive portal external system.

4.2 Environment setup

Although it is not a project phase itself, this section depicts the environment setup for the development of the aforementioned two scenarios: the web application development and the integration with the captive portal. Regardless of their independent nature, the environment here described is based in the final scenario, as it is more complex.

The final scenario, as described in section 4.4.1, includes the captive portal technology. The selected technology is OpenNDS installed in the OpenWRT router firmware, which is included in this topology (see section 4.2.1). Moreover, OpenNDS can be configured with a remote server, using the *Forwarding Authentication Service* (FAS) option. In this topology, the remote server is also known as *FAS server*.

In conclusion, the environment should implement a simplified version of the final scenario (see figure 4.3). Specifically, with three main components: (1) a client device; (2) a router with captive portal technology; (3) the web application server.

4.2.1 Virtual networking topology

Considering the needs specified above, the environment has been based on a virtual environment. For this purpose, the author created several Virtual Machines (VM) and networking interfaces in VirtualBox that enabled connectivity between using *Host-only* adapters.

The designed topology divides the corporate networking into four different LANs attached to the OpenWRT router (see figure 4.7):

1. **Internet**. Configured with NAT in VirtualBox, represents the WAN interface to Internet.

2. **Demilitarized Zone (DMZ)**. It is also known as the server's LAN, where the corporate servers are placed.

3. **Management (MNG)**. This LAN represents the network for administration purposes. In the virtual networking topology, it represents the host machine.

4. **Clients Network**. The client VMs are attached to this interface. It represents the captive portal clients.

For this purpose, the virtual environment can be summarised in three Virtual Machines:

1. **Client VM**. The client machine can run one of the most used desktop environments. At the beginning of the project, a Kali Linux OS was installed. Later, in the captive portal testing, Ubuntu Desktop, Windows 10 and Manjaro were installed.

2. **FAS Server VM**. Ubuntu Server OS was used for this VM.

3. **OpenWRT VM**. The router firmware is a Linux-based OS for routers.

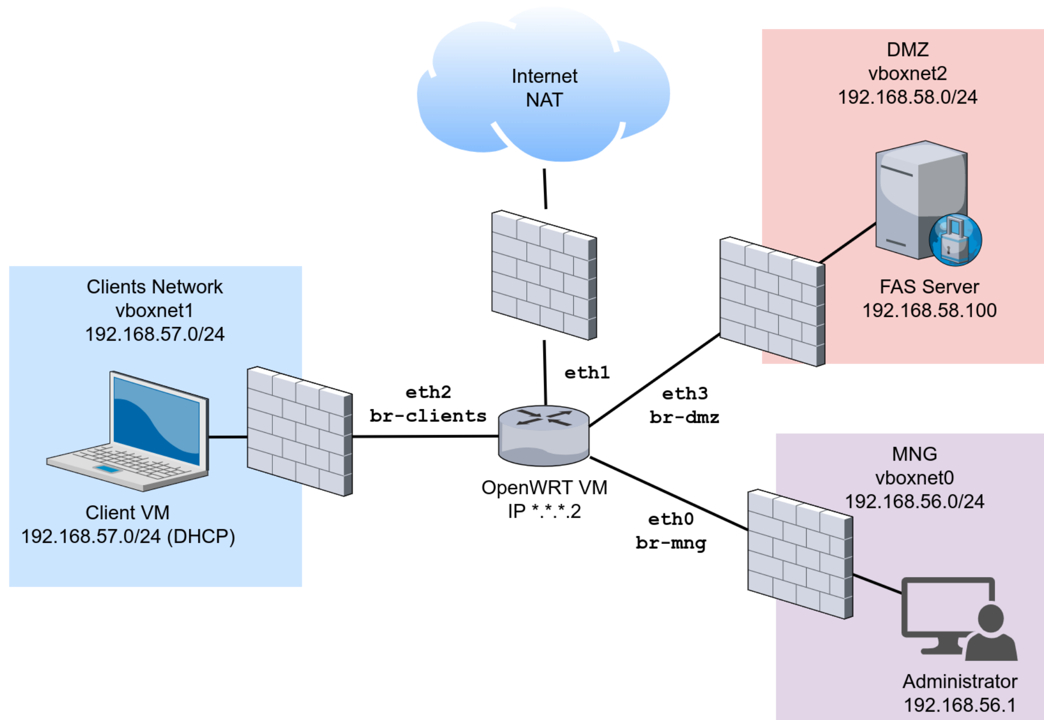

Fig. 4.7.: Virtual networking topology used in the environment. Setup in VirtualBox. The router manages the four defined networks. The OpenWRT interfaces are represented as `eth0-3`. The logical OpenWRT bridges are represented by `br-<tag>`.

4.2.2 OpenWRT router installation and configuration

Following the OpenWRT guide for virtualisation in VirtualBox [@Bur16], in order to install the firmware on a VM, the downloaded image should be converted. Therefore, after downloading the version `21.02.2 x86 64 bits`, it can be converted to VDI using the CLI client of Virtualbox: `VBoxManage convertfromraw <img>`.

During the first boot, two network interfaces are configured: (1) a LAN interface, with a *Host-only* adapter; (2) an Internet interface, with a *NAT* adapter. Once the router is started, the basic configuration can be done connecting through SSH to the assigned LAN IP. From there, we can install the LUCI web interface that can ease the OpenWRT configuration: `opkg install luci`.

Network configuration

There are three interfaces in the OpenWRT router, with a corresponding *bridge device* and a *wired port*. These virtual ports represent wired connections that provide connectivity to the VirtualBox network. Table 4.1 includes are all interfaces with their corresponding *bridge devices*, wired ports, virtual networks and IPv4 addressing.

VirtualBox network	Addressing	VM wired port	OpenWRT bridge dev.	OpenWRT interface
vboxnet0	192.168.56.0/24	eth0	br-mng	MNG
vboxnet1	192.168.57.0/24	eth2	br-clients	CLIENTS
vboxnet2	168.168.58.0/24	eth3	br-dmz	DMZ
NAT - Internet	-	eth1	-	WAN

Tab. 4.1.: Network topology and OpenWRT interfaces: three LANs configured as VirtualBox networks, and one NAT interface that enables access to Internet.

As described in figure 4.7, there are three LAN networks, that will be configured as *Host-only* VirtualBox networks. These are represented in figure 4.7 as `vboxnet0-2`, together with the corresponding IPv4 addressing: `192.168.56-58.0/24`.

For each of these VirtualBox networks, the host machine sets one logical interface. The corresponding IPv4 address for the host machine will always be `x.x.x.1`. The OpenWRT router will be assigned with the `x.x.x.2` IPv4 address in each of the LANs. Addressing is configured manually, so the VirtualBox DHCP server for these networks should be disabled.

Finally, each of the attached LANs are configured in OpenWRT as a *bridge device*, specifying the corresponding wired port to which the corresponding LAN is attached via the VirtualBox adapter. Figure 4.8 shows a screenshot of the configuration of a *bridge device*. Then, each of these *bridge devices* is related with a *logical interface* (see figure 4.9).

Zone-based firewall configuration

OpenWRT operates with a zone-based firewall, configured by default. Following the diagram on figure 4.7, we can define four different zones: (1) WAN, representing the Internet; (2) DMZ; (3) LAN, representing the clients network; (4) MNG, the management network.

The zone-based firewall can be configured from the LUCI admin panel in OpenWRT. For the environment needs, traffic from the LAN zone will be forwarded to the WAN

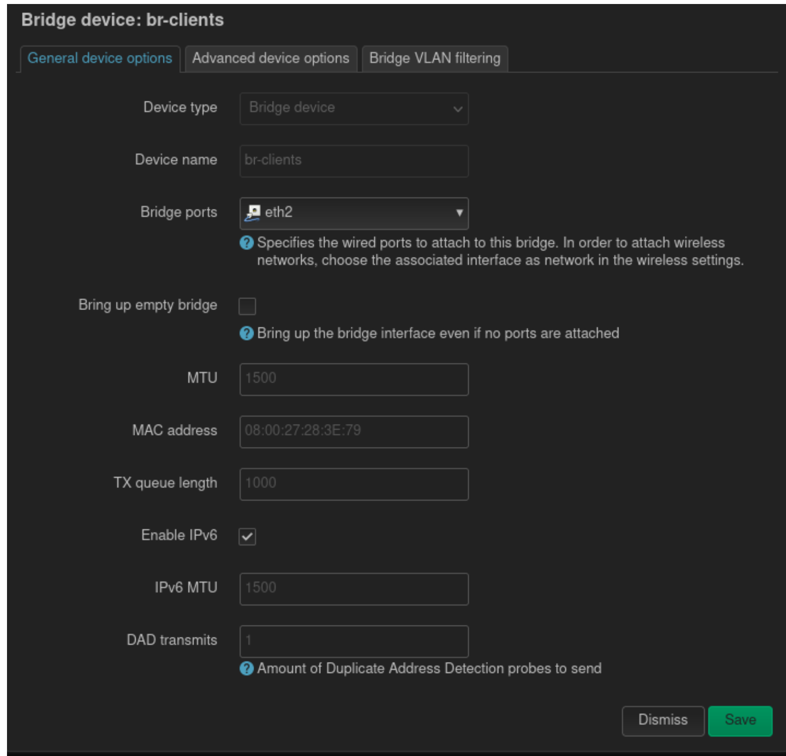

Fig. 4.8.: Configuration of a *bridge device* in OpenWRT through the LUCI web admin panel.

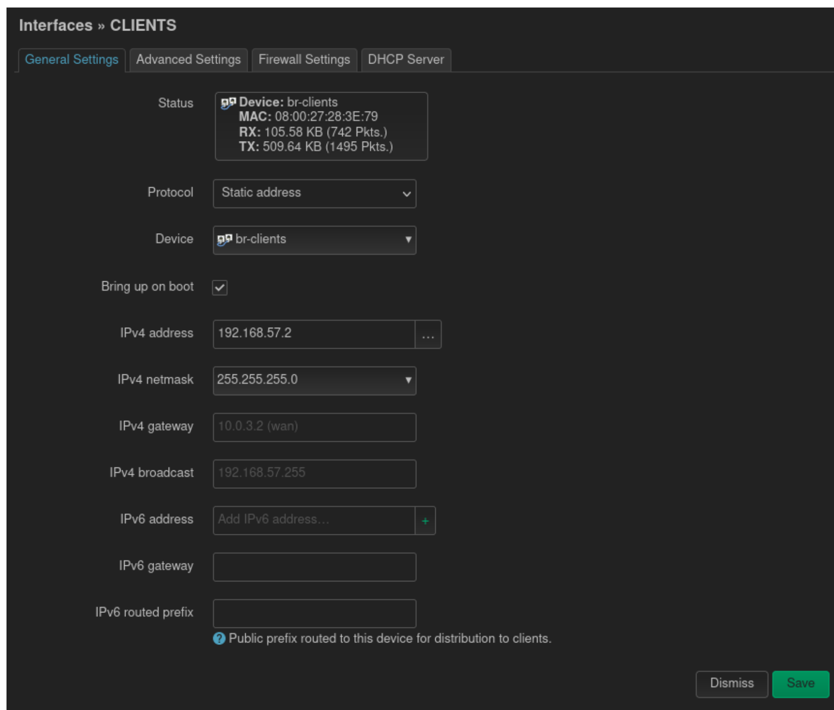

Fig. 4.9.: Configuration of a *interface* in OpenWRT through the LUCI web admin panel.

zone. Then, input traffic from the WAN zone will be rejected, while the output traffic is accepted. Finally, traffic to the DMZ is rejected by default.

However, this configuration would not allow clients to access servers in the DMZ. In order to configure exceptions to these general rules, other stateful *Access Control Lists* (ACLs) can be configured from the *traffic rules* tab. Here, a rule was added to allow HTTP traffic from the LAN to the DMZ. Therefore, the client will be able to communicate with the captive portal.

OpenNDS installation: a captive portal package

As it will be detailed in section 4.4, OpenNDS is the Captive Portal technology used in the scenario for the integration with WebAuthn authentication. This technology is also released in form of a OpenWRT package, so it can be installed with `opkg install opennds`.

After the installation, the configuration file can be found in `/etc/config/opennds`. For testing its basic functionality, we can use the default configuration adapting the `gatewayinterface` to `br-clients`, the clients *bridge device* interface in OpenWRT.

The basic functionality tested can be summarised in three steps:

1. Access a HTTP web page from the client, once plugged in to the network.
2. OpenNDS will redirect the request to the default Captive Portal page. This page asks the user to accept the *Terms of Service*.
3. After the user accepts, the OpenNDS shows an information page and allows the redirection to the original requested page.

4.2.3 Development environment

During development, the captive portal server was running at the host machine. In order to add it to the virtual network topology, the FAS server ran a HTTP Nginx reverse proxy, forwarding the requests to the host machine.

As explained in more detail in section 4.3, the development of the captive portal will be done in the *Visual Studio* IDE in the *Typescript* programming language, powered by *NodeJS* and the *NPM* dependency manager.

This environment is used by the author in the host machine for development. During the development testing, the *NodeJS* server will reply to clients requests in the VirtualBox network. These requests are routed from the network by the OpenWRT

VM and finally proxied by the FAS server with Nginx reverse proxy. Notice that this traffic should be encrypted with TLS, a requirement of the OpenNDS configuration. More details can be found in Appendix A.

After the development phase, the final captive portal software can be directly installed in the FAS server within the virtual topology. In a real physical topology, the setup will be more similar to that final setup.

4.3 Authentication server development

As explained in section 4.1.3, the generic scenario is the WebAuthn web application that allows (1) registration of a WebAuthn security key by the administrator and (2) authentication with a WebAuthn security key by the user. During this section, the development process of the generic WebAuthn web application is described.

Following an iterative methodology, as described in section 3.3.3, the development sprints can be summarised in the following sub-sections. They are ordered chronologically.

Used libraries and technology

For this server development, the *Typescript* programming language was used, based on *NodeJS*. For the REST web application interface, the *ExpressJS* library was used. Regarding the WebAuthn support, we relied on the updated library *SimpleWebAuthn* [@Mil22]. The database used later in the development is stored with *MongoDB* database management system.

At the front end, the development included the use of *Almond.CSS* styles library [@Mon]. The execution powered with client-side *JavaScript* included the front end module from the *SimpleWebAuthn* library [@Mil22].

Integrated development environment and package manager

During the development, the author used *VisualStudio* IDE together with `git` for version control. The dependencies of the project were handled by `npm` (*Node Package Manager*) together with some setup scripts for building.

In addition, the *MongoDB* database is deployed in a *Docker* container and deployed through a simple *Docker Compose* file for a quick start during the development.

Finally, it is worth mentioning that the web application uses some configuration environment variables for adding flexibility when running the final server software.

For this reason, these variables are grouped in an `.env` file at the root of the code repository.

4.3.1 WebAuthn with discoverable credentials

The development of this feature is the main characteristic of the WebAuthn authentication server. It allows using WebAuthn security keys compatible with discoverable credentials. Specifically, the devices can be registered associated to a user, and then used during authentication, that is, during the verification of the user identity.

There are two types of WebAuthn credentials related to security keys: discoverable and non-discoverable. Discoverable credentials are stored physically within the security key. On the contrary, non-discoverable credentials are stored encrypted in the server. More detail can be found in section 2.6.

As a first development, the server was designed to support discoverable credentials. During user authentication, the WebAuthn server sends a challenge to the security key without previously identifying the user. When receiving the response from a security key, the server can identify the corresponding user with the `userHandle` parameter.

Figure 4.10 shows a cross-functional diagram with the data flow of the registration operation. Firstly, when registering, a `username` is specified. If the user does not exist, a new user is created. Then, the device (security key) is registered for that user.

When authenticating with a security key, no `username` is required. As shown in the cross-functional diagram of figure 4.11, the `userHandle` included in the authenticator response is used as the `userId` to find the registered device for verifying the authenticator response. Hence, using discoverable WebAuthn credentials implies that the user is identified *after* the authentication of the credential. Therefore, the user does not need to identify themselves with a username during the process.

4.3.2 User database and application architecture

The WebAuthn authentication web application follows the classical software design pattern *Model-View-Controller* (MVC). This section details the stored data necessary for the application: the *model* of the application architecture.

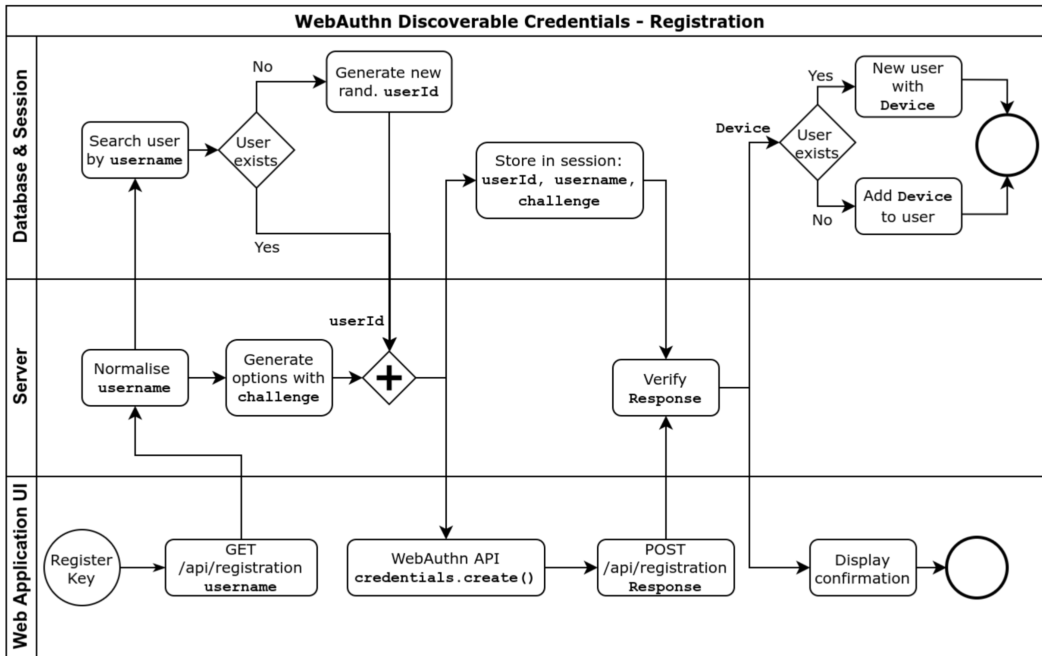

Fig. 4.10.: WebAuthn registration with discoverable credentials: cross-functional diagram. The device is added to a user if it exists. Otherwise, the user is created.

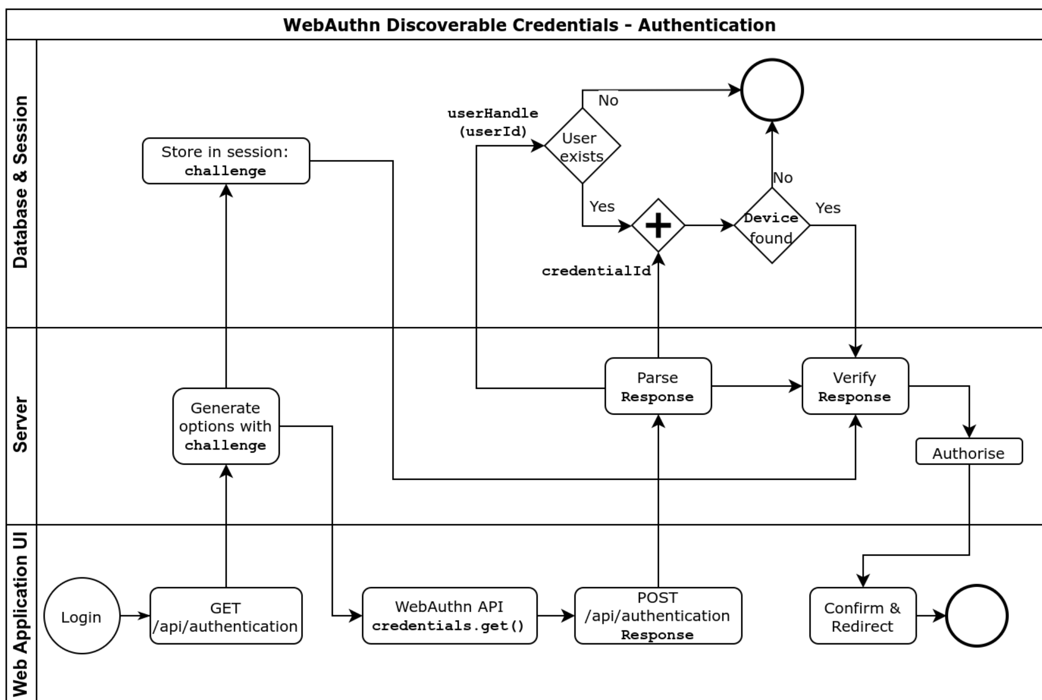

Fig. 4.11.: WebAuthn authentication with discoverable credentials: cross-functional diagram. The userHandle is used for identifying the user to authenticate.

The data hold by this web application directly corresponds to the minimal user information and their authentication credentials. Considering the implementation of the authentication as explained in section 4.3.1, the authentication credentials are WebAuthn public keys.

Specifically, there are two main types of data handled by the web application:

1. **Web session details**. After user authentication, a web session is initiated. In a complete web application, it allows authorisation of web operations to that authenticated user. By definition, these details are tight to the web session, so they are *not* considered persistent.

2. **User and their corresponding credentials**. For the user management and authentication, the users and their corresponding credentials are stored in a *persistent* database.

The web session details are initially only functional identifiers for the *controllers* (see section 4.3.5) to manage web sessions. In section 4.3.6, a more advanced concept of sessions was added to the application.

Regarding the user details, figure 4.12 shows the class diagram with the attributes of a specific user, together with the WebAuthn credential data model for storing the devices registered for the authentication of that user.

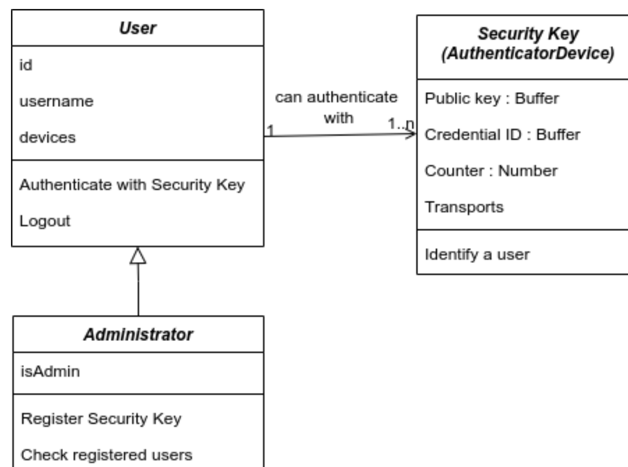

Fig. 4.12.: User and credentials class diagram. The user has at least one Security Key associated.

As explained in section 4.1.3, some users are assigned to the *administrator* role, to authorise the registration of WebAuthn security keys. This role is represented with the `isAdmin` user attribute represented in the aforementioned figure.

The used database technology is *MongoDB*, a NoSQL database management system. This technology facilitates the storage of objects through the serialisation and deserialisation powered by *Mongoose* schemas. These define the aforementioned models for the database storage.

4.3.3 Administration interface: registration and information

As explained in the generic scenario in section 4.1.3, the registration of the security keys are restricted to administrators. Using a RBAC, whether the users are authorised to register new devices or see restricted information depends on their role.

In this development *sprint*, an administrator interface was created. That interface is restricted to users with the *admin* role. As detailed in the use cases (see figure 4.5), the *admin* role authorises a user to:

1. **Check registered users, their active sessions** (see section 4.3.6) **and registered security keys**.

2. **Register a WebAuthn security key**, as designed in the implementation explained in section 4.3.1 or section 4.3.7.

The administrator interface is designed as a web page within the web application, only authorised to *admin* users. That interface requests information about the registered users and their active sessions displaying them in a table (see figure 4.13). Besides, this web page is the only interface where the registration form is placed.

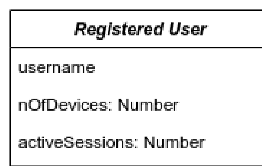

Fig. 4.13.: Class diagram representing the registered user database model. The user class is simplified for showing only the aggregated user details.

On the other hand, when the application is started and no users are registered, the application needs to allow the registration of an administrator. For this purpose, when no admin is found within the user database, the *admin* interface and the RBAC system is disabled, enabling usability of the complete web application.

4.3.4 Securing the web application

Cross-Site Scripting (XSS) attacks are based on the injection of malicious *JavaScript* code into web applications. There are some ways to mitigate these attacks. One of them is user input validation, that prevents XSS payload to be inserted and stored in the database for later execution when the payload is inserted into a web page.

In this web application, there is only one identified user input. That is, a single point where a user can insert a malicious XSS payload: the `username`. When specified during the registration security keys, an attacker could inject XSS payload into the web application via the registration form. Also, when authenticating with non-discoverable credentials (see section 4.3.7), the user should identify themselves *before* the authentication.

For preventing such injection, the input is sanitised by using the *DOMPurify* library available at the NPM package `isomorphic-dompurify`. This library has a profile that removes HTML from any input and, therefore, prevents the injection of malicious XSS injections embedded in HTML tags like `<script>` or `<img src:>`.

The other web application security countermeasure regarding the REST interface are explained in section 4.3.5.

4.3.5 ExpressJS and API management

Regarding the MVC architecture of the web application (see section 4.3.2), the *controller* of this application is managed by a HTTP REST interface powered by the *ExpressJS* framework. This section explains the details of this controller interface.

This *ExpressJS* project starts a *NodeJS* HTTP(s) server that listens for requests. Upon receiving a request, the server triggers the specific registered controller, also known as *middleware*. These work at the programming level as callback functions that asynchronously handle the requests. For this purpose, the static web resources, as the HTML web pages, CSS style sheets and JS scripts are served using the `express.static` middleware. Then, other routes conforming the REST *Application Programming Interface* (API) are registered as middleware handlers.

HTTP REST API endpoints

For allowing the user interface web page to display the user details, the registered users or allowing the registration and authentication WebAuthn operations, some HTTP REST API endpoints are configured.

All these routes are linked with asynchronous functions known as controllers, that are specified in separate modules. The REST interface at the end of the development of the WebAuthn authentication server can be found in table 4.2.

REST endpoint	Method	RBAC	Query parameters	Request	Response
/api/registration	GET	*admin*	username nonDiscoverable	-	Registration Options
/api/registration	POST	*admin*	-	Registration	Confirmation
/api/authentication	GET	-	username nonDiscoverable	-	Authentication Options
/api/authentication	POST	-	-	Authentication	Confirmation
/api/user-details	GET	*user*	-	-	Logged in user details: username and RBAC role
/api/registered-users	GET	*admin*	-	-	List of Registered Users
/api/logout	GET	*user*	-	-	-

Tab. 4.2.: HTTP REST API for the WebAuthn authentication server before the integration with the captive portal. The RBAC column represents the role that can access the API. All logged in users have the role *user*, but not all of them have the role *admin*, which is restricted to administrators.

RBAC authorisation middleware

Configured as a middleware controller, the authorisation middleware `authorizeOnlyAdmin` can be configured in any route. Its working verifies that a user is logged in and if the logged in user has the *admin* role. This works thanks to the session details managed by *ExpressJS* web sessions. The protected routes are also detailed in table 4.2.

If the user is not logged in, the middleware aborts the request sending a HTTP 401 error code (*Unauthorized*). In case the user does not have the *admin* RBAC role, the middleware aborts the reqeust sending a HTTP 403 error code (*Forbidden*).

4.3.6 Session management: expiration and multiple sessions

Web sessions represent an authenticated user and links their HTTP requests with that identity, so they can be authorised. In this web server, *ExpressJS* uses cookies to maintain that session and expires them within a configured time, specified in an environment variable.

For administrators to enumerate active sessions for the users, the sessions have also been stored in the database. That way, from the administrator interface (see section 4.3.3), the administrator can list the active user sessions per registered user. That is also useful in the future integration with the captive portal to authorise authenticated users.

For achieving this, a new *MongoDB* collection is created to store sessions. The *MongoDB* schema for the session is configured to expire the data within the same time configured to expire web sessions. That way, when web sessions are expired, these details are removed from the database. For doing that, the `expireAt` attribute is configured in the *MongoDB* session schema.

Finally, each time the user is authenticated in the server, a new session is stored linked with the user identifier. With this approach, a user can have multiple concurrent sessions, which are represented in the administration interface. When integrating the server with other systems like the captive portal, users are able to authenticate more than once without requiring them to logout first from other sessions.

4.3.7 WebAuthn with non-discoverable credentials

The last *sprint* of the development of the WebAuthn authentication server covered the support of non-discoverable credentials. As explained in the first *sprint* in section 4.3.1, non-discoverable credentials do not reside in the WebAuthn security key, but they are stored in the server.

The main advantage of supporting non-discoverable credentials, as explained in section 2.1.3, is that more security keys, browsers and operating systems will be compatible. The reason behind this is that the first standards required identifying the user *before* authenticating them, which did not require security keys to store any information to respond to the authentication request.

With discoverable credentials, the user was identified *after* the authentication. On the contrary, the authentication response from the security key will not contain a `userHandle` that identifies the user. Therefore, the `username` should be specified at the authentication form.

When generating the WebAuthn authentication options, a list of the registered credentials for the corresponding user is provided. This list contains credential identifiers that are actually encrypted private keys that only the registered security key can decrypt to correctly authenticate the user (see section 2.1.3).

Implementing this change also implies modifying the generation of WebAuthn options, according to the standard. During WebAuthn registration, the `authenticator-Selection` should be modified to include (1) `requireResidentKey` set to *false*, (2) `userVerification` to *discouraged* and (3) `residentKey` to *discouraged*. During WebAuthn authentication, (1) the attribute `userVerification` is set to *discouraged*; and (2) a list of allowed credential identifiers is provided in `allowCredentials`.

These options instruct the security key to use non-discoverable credentials. Also, the user verification through a device PIN is not required, as the credentials are not resident. These options are retro-compatible with old WebAuthn and FIDO implementations that support non-discoverable credentials.

Figure 4.14 shows the corresponding cross-functional diagram with the data flow of the authentication operation. This diagram slightly differs from the one presented in figure 4.11, with the aforementioned differences.

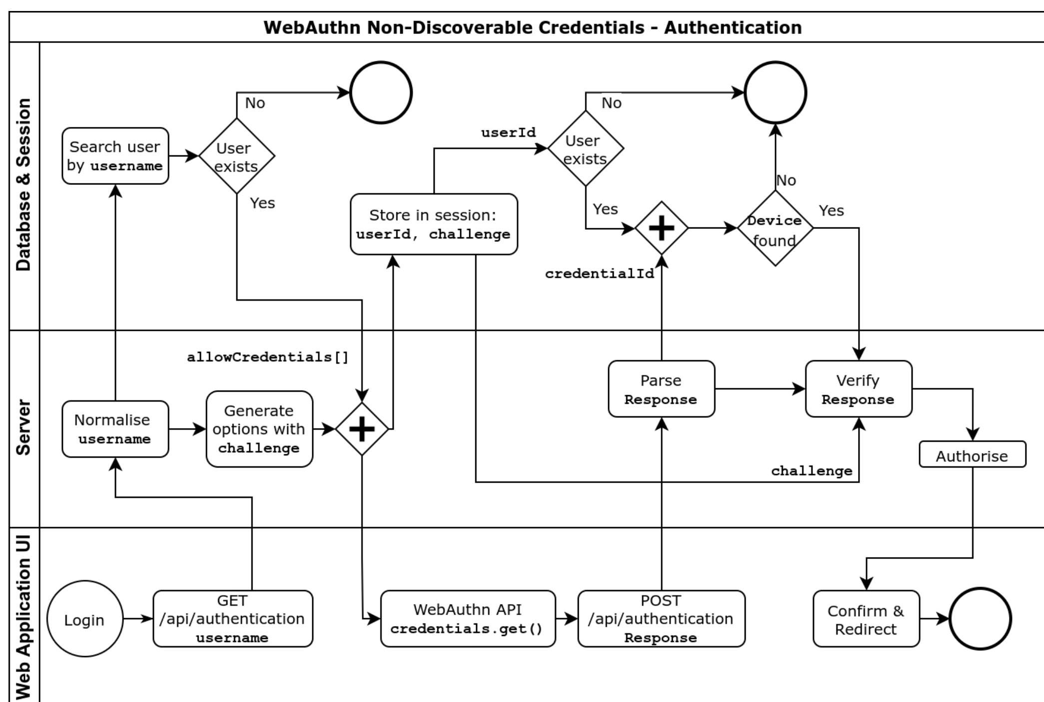

Fig. 4.14.: WebAuthn authentication with non-discoverable credentials: cross-functional diagram. The list of allowed credentials is sent in the authentication options, as the user is identified *before* authentication verification.

Finally, it is worth mentioning that this *sprint* implied a couple of further changes: (1) the API was updated to the one shown in table 4.2 to configure non-discoverable registration and authentication; and (2) the registration and authentication forms were updated with an extra option that enables the implemented feature. During registration, the new option makes the server to create the corresponding non-discoverable credential registration options. Then, during authentication, the option shows the user a new field to specify the `username`, which is sent to the server when requesting the authentication options.

4.4 System integration

Once the generic WebAuthn authentication server was developed, it was integrated with a captive portal solution: *OpenNDS*. Together with the previous section, the system integration explains another core element of the work of this Master's Thesis. The environment taken as reference is described in section 4.2.

The following sub-sections represent a chronological view of the incremental work that achieved a final integration of the developed WebAuthn server with selected captive portal solution. Before the integration development tasks were performed, a reverse engineering study was required to document and gather the requirements for a successful system integration.

During this section, the captive portal is commonly referenced with the chosen solution name: *OpenNDS*. On the other hand, the developed WebAuthn authentication server is commonly referenced as *FAS server*, nomenclature used in the OpenNDS configuration.

4.4.1 Selection of a captive portal solution

Section 2.4 shows the existing captive portal technology nowadays. In this project phase, the integration was sucessfully achieved with OpenNDS installed on a OpenWRT firmware. However, the first viability tests were performed with pfSense.

Both pfSense [@pfS] and OpenNDS [@Whi22] are captive portal solutions that are suitable for this integration for being open-source. Therefore, they could be modified to integrate the WebAuthn authentication server to achieve a final WebAuthn captive portal. However, the design of both solutions is different. While pfSense is a complete router firmware that implements captive portal, OpenWRT [@Pro16] firmware relies on the OpenNDS independent package to implement the captive portal functionality. This makes OpenDNS a more self-contained solution whose project code is easier to approach.

According to the architecture presented in section 2.3.2, a captive portal includes two key elements: an enforcement device and a web server. One specific advantage of OpenNDS that determines the final selection of the solution, is that it allows externalising the web server module of the captive portal. This way, the developed web server can be modified to be integrated with OpenNDS.

OpenNDS feature for externalising the web server is called *Forwarding Authentication Service* (FAS) [@Ope], which forwards the redirected requests to a captive portal to the selected external web server. Finally, OpeNDS enforcement device should be integrated with the external web server to identify the authenticated client and authorise its access to the network. Therefore, during this section, the developed WebAuthn authentication server was modified to implement an external FAS server compatible with OpenNDS.

4.4.2 Reverse engineering of OpenNDS

Although OpenNDS has some documentation and, specifically, on the configuration of a FAS external server, during this integration a reverse engineering of the inner functioning of this system was required. All this process caused the delay of this project phase.

At the end of this process, the findings allowed to redesign the requirements for the successful integration with the developed WebAuthn authentication server, detailed in the next sub-sections.

Available FAS configurations in OpenNDS

There are four main alterantives to configure FAS in OpenNDS. According to their documentation [@Ope] their characteristics are different:

- **FAS security option 0**: Uses HTTP. The token is sent in cleartext along with the query string. It is not secure, as it is easy to bypass. This is the default configuration, when the security is disabled.
- **FAS security option 1**: Uses HTTP. A hashed id is sent encoded in base64, together with some client parameters.
- **FAS security option 2**: Uses HTTP. The information sent is encrypted with AES-256-CBC with a randomly generated initialisation vector. The symmetric key is known by the FAS server and OpenNDS.
- **FAS security option 3**: Uses HTTPS. Identical with option 2 but using HTTPs protocol. Additionally, a module named *Authmon* is loaded, which is in charge to pull a list of authenticated clients from the FAS server.

Considering the specific needs of our case, the selected option is 3, as it is the only alternative that uses TLS-based HTTP communication for the captive portal, which is required by the WebAuthn standard.

Configuration of the environment to analyse

The base of the reverse engineering process was the presented environment (see section 4.2) configured with a demo FAS server at the DMZ zone. The demo server in PHP is available within the OpenNDS package. A Docker deployment with PHP-FPM and the Nginx web server was done to serve the demo PHP file (see Appendix B).

From the available security options listed before, in this reverse engineering the option 3 was analysed. However, the basic process described here was performed to verify differences with the security option 2, which is much simpler.

In order to successfully deploy option 3, TLS certificates are required at the FAS server. In this configuration, the TLS certificates are self-signed, so apart from the client to trust them, OpenNDS should also accept such certificates to send the periodic requests that form part of the integration. That means that signing Certificate Authority (CA) certificate should be installed in OpenWRT at `/etc/ssl/cert.pem`. For more details, see Appendix A.

Additionally, a domain name for the external server is required. During this phase, `fas.localhost.pri` was used when generating the certificates added to Nginx and added a hostname resolution in OpenWRT, pointing to the address of the FAS server in the DMZ zone (see the environment topology in figure 4.7).

Sniffing the TLS-encrypted traffic in HTTP communication

Now traffic is encrypted with TLS as required by security option 3 of OpenNDS, we need access to the plain HTTP to analyse it. For this purpose, the aforementioned Docker deployment was modified to add an additional Nginx reverse proxy. This way, the first Nginx server (the reverse proxy) is in charge to handle TLS-encrypted HTTP requests and responses, while it forwards the traffic to the second Nginx which is serving the PHP demo server.

With this solution, we can sniff the forwarded decrypted traffic between the two Nginx servers (see Appendix C). Taking into account the virtual environment (see section 4.2.1), the host machine runs the *Wireshark* GUI. Then, using `tcpdump` in the FAS server VM, the traffic of the Docker interface can be dumped to the host machine through SSH. In the host machine, the command would be something like listing 4.1.

List. 4.1: Command pipe to dump traffic from remote server to analyse it locally with Wireshark GUI.

```
1 ssh user@192.168.58.100 \
    sudo tcpdump -i br-<tag> -U -s0 -w - 'not port 22' \
    | wireshark -k -i -
```

Traffic, log and code analysis

Once the traffic between the two Nginx Docker containers is captured, the resultant traffic represents the communication between OpenNDS (Authmon) or the client browser and the FAS Server. Apart from this traffic, OpenNDS logs are set to debug mode for the usage in the analysis (see Appendix C). Finally, all this analysis will also rely on a white-box code review.

4.4.3 Cryptographic details of FAS in OpenNDS

The first request the FAS server receives is from the client, originated by a redirection performed by OpenNDS. This redirection URL points to the FAS Server and adds some parameters necessary for the later client authorisation. With the security option 3 configured, this information is encoded in base64 and encrypted with AES-256-CBC. One of the biggest challenges of this integration is to correctly decode and decrypt this payload once it arrives to the developed WebAuthn server in Typescript. Notice that the demo is written in PHP and the encryption and codification is done within the OpenNDS code. In practice, the cryptographic libraries have some implementation differences which make the inverse process not trivial.

The AES block cipher in *Cipher-Block-Chaining* (CBC) mode used here has a 256 bit block length. In Typescript, the corresponding required key length is of 32 bytes. The reference implementation of PHP, however, uses the `openssl_decrypt()` function, which accepts a key of an arbitrary length. After some tests, one of the solutions that work with both implementations is to use a shared key of length 32 bytes.

Another important cryptographic aspect is the calculation of the authenticated hash performed by OpenNDS. This authenticated hash has to be used by the FAS server to send an authorisation token that allows an client to be authorised by OpenNDS. The hash used by OpenNDS is SHA256. However, in order to authenticate the hash, OpenNDS developers have chosen to include the symmetric key at the end of the payload to hash. In this case, the FAS server should return a re-hashed version of the `hid` parameter when the client is authenticated. Letting k be the 32 byte shared key, the operation is:

$$rhid = sha256(hid||k)$$

All these operations are reflected in the resulting final code in Typescript that manages these operations, as shown in listing 4.2.

List. 4.2: FAS Server adapted code with the required cryptographic operations.

```
// Decode fas from base64 and get iv query parameter
let fas : string = req.query.fas as string;
let iv : string = req.query.iv as string;
fas = Buffer.from(fas, 'base64').toString('utf-8');

// Decrypt the fas query parameter
let decipher = crypto.createDecipheriv(
        'aes-256-cbc',
        FAS_SHARED_KEY as string,
        iv);
fas = Buffer.concat([
        decipher.update(Buffer.from(fas, 'base64')),
        decipher.final()
    ]).toString('utf-8');

// [...]

// Calculate and store rhid
let rhid : string = crypto.createHash('sha256')
                    .update(hid+FAS_SHARED_KEY, 'utf8')
                    .digest().toString('hex');
```

4.4.4 Extending the API for Authmon integration

For OpenNDS authorising a client the access to the network resources, the system needs to know the successfully authenticated clients in the FAS server. For this purpose, the *Authmon* OpenNDS module sends periodic requests to the FAS server.

Handling these periodic requests and integrate their operation in the developed WebAuthn authentication server is the core integration with OpenNDS. This implementation makes the developed server a compatible OpenNDS FAS Server.

Authmon polling periodic requests

The Authmon module of OpenNDS sends periodic requests to the FAS server to poll for new authenticated clients. According to the statechart presented in the design phase in section 4.1.2, after the client authentication, the router will authorise access to the network resources.

Figure 4.15 shows a flow diagram that represents the integration of the Authmon operation to authorise clients with the developed FAS server. This integration was possible after the reverse engineering process presented in section 4.4.2.

The diagram of figure 4.15 represents the following steps. The captive portal run by OpenNDS will redirect unauthorised clients to the FAS server URL (step 3). When

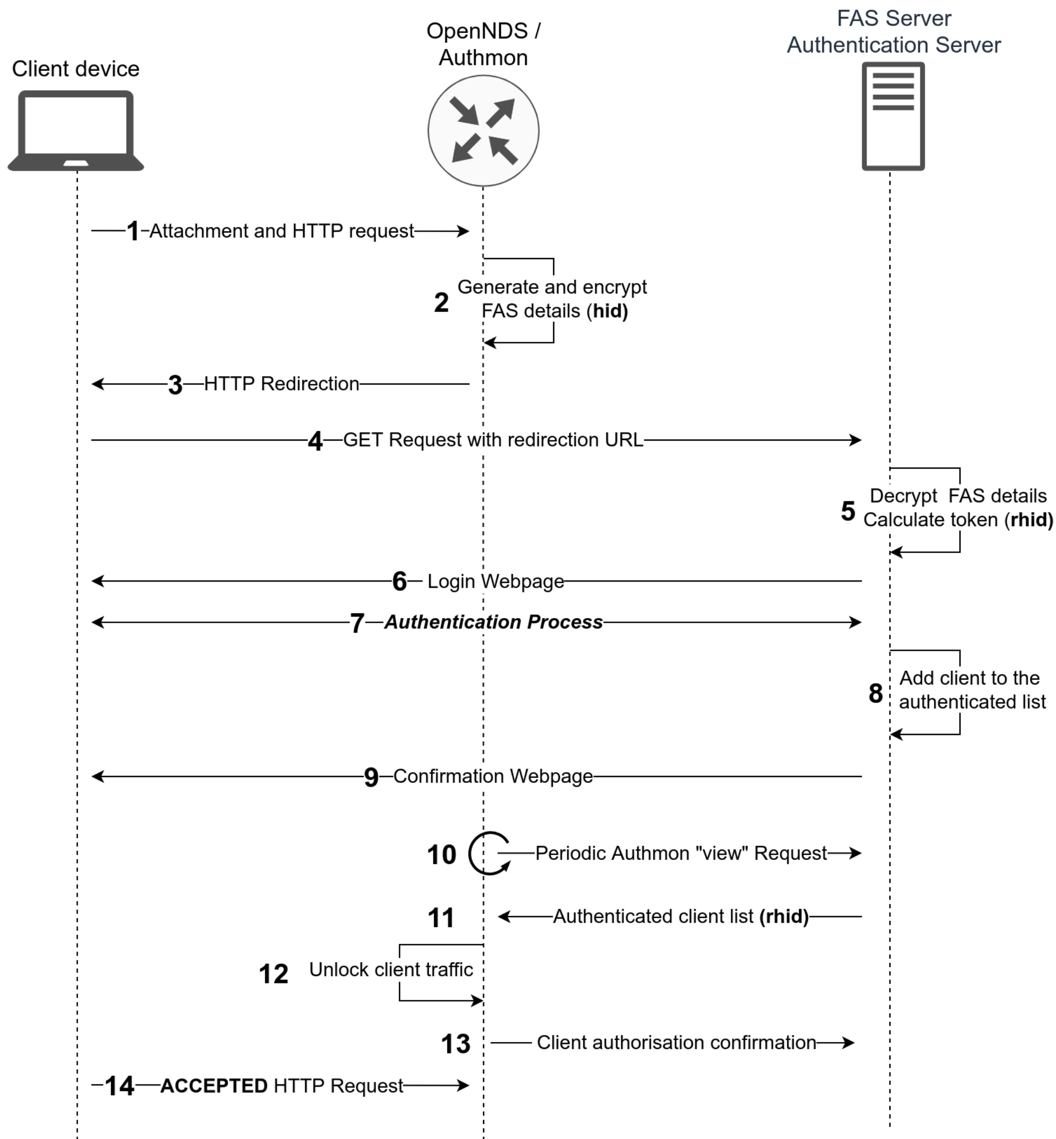

Fig. 4.15.: Communication between OpenNDS Authmon module and the developed FAS server. The client connects, gets redirected to the captive portal hosted at the FAS server and, once authenticated, it is authorised by OpenNDS.

doing that, it will add some parameters. The most important is the hashed id (`hid`). These parameters are encoded and encrypted as explained in section 2.1.2 (step 2). When handling client requests, the FAS server will decrypt these details and calculate the authorisation token `rhid`, as explained in section 4.4.3 (step 5). When the client authentication (step 7) is successful, the FAS server marks the client as authenticated (step 8).

All authorisation codes (the `rhid`) are requested periodically to the FAS server (step 10). Once the Authmon periodic request is handled by the server, a list of authorisation tokens is sent to the OpenNDS router. Using this token, OpenNDS can authorise the corresponding client for their access to network resources. Then, this authorisation is confirmed to the FAS server (step 13).

New API endpoints for Authmon

Thanks to the reverse engineering, the following endpoints were designed to handle the *Authmon* periodic requests and the new client requests, as shown in table 4.3.

REST endpoint	Method	Query parameters	Request	Response
ALL	GET	fas iv	-	Root login webpage
/	POST	-	Authmon Request	Response to Authmon

Tab. 4.3.: Root endpoints of the API that are extended in functionality for handling requests once integrated with OpenNDS.

This implementation required the usage of a middleware module that processes all incoming GET requests from the client in the search of encrypted FAS details. When received, they are stored in the session details.

According to the reverse engineering process, OpenNDS Authmon module can issue three different requests to the FAS server. These are distinguished by the `auth_get` parameter of the request body.

- "**clear**": All authenticated clients should be clear from the list. That is, the list is reset. This is used by OpenNDS when booting up.
- "**list**": The list should be sent and the cleared. This type of request is not frequent and is kept for backwards compatibility.
- "**view**": The most often request. This type of request depends on its payload:
  - * or **none**: The complete list of authenticated clients is required. The corresponding `rhid` tokens should be sent in a list, according to the compatible format: `* <rhid>`.

– * **<rhid>**: Authmon is confirming the authorisation of a client.

To implement the list of authenticated clients, the session management feature developed in the WebAuthn authentication server can be used. As explained in section 4.3.6, the server incorporates a separate database collection for storing authenticated user sessions. Then, once OpenNDS confirms the client authorisation, it is marked as authorised. Finally, the session exipiration time is syncrhonised with OpenNDS expiration time.

4.4.5 Adding support for multiple gateways

A gateway is a router to which the clients connect, where an OpenNDS captive portal is installed. As OpenNDS uses the developed server as a remote (FAS) server, other OpenNDS instance could also use the same remote server. That is, more than one gateway could use the same remote authentication server.

This is ideal if the WebAuthn authentication server can be used as an identity provider for more than one network. This way, each network can have one or more OpenNDS captive portals configured to work with the central remote WebAuthn authentication server. Although the developed server supports the same account to have multiple active sessions, the integration with the captive portal should mark a client authorised for each network.

This feature was developed by using the *gateway hash* incorporated by OpenNDS in the FAS details (see step 3 in figure 4.15). Using this parameter, the FAS server can distinguish which gateway is associated with the client request. Also, when receiving the "view" request from Authmon, the *gateway hash* is also included. By using the session management database, the *gateway hash* can be used to uniquely identify the gateway and, therefore, it can be used as a filter for the creation of the client authorisation lists.

4.4.6 Webpage automatic redirection

Once the client is connected to the captive portal network, there are two main ways to initiate the captive portal authentication. Firstly, it can happen automatically via captive portal detection, for which there are some techniques that browsers and operating systems use, including modern approaches like a DHCP code (see section 2.3.2).

The other option is that the client starts web browsing and, therefore, generating HTTP traffic. When this is done, this traffic will get redirected by OpenNDS (see step 3 in figure 4.15), so now the client will be directed to the remote WebAuthn authentication server. After the authentication, however, the client will stay in the authentication server.

Once the client is authenticated by the WebAuthn server, and authorised by OpenNDS, the HTTP traffic will be allowed see step 14 in figure 4.15). After authentication, the client expects to be redirected to the original website that originated the initiation of the captive portal authentication see step 1 in figure 4.15).

This feature required an additional implementation, based on the `originurl` parameter included in the FAS details during redirection. This way, the developed FAS server decodes and stores this URL in the session information. After the authentication, a loading GIF is shown to the client and, when the authorisation confirmation is received, the client is redirected to the original URL at the browser side.

4.5 Captive portal testing

The final WebAuthn Authentication Server integrated with OpenNDS was validated during the last phase of the project. For this purpose, the tests were performed in the mentioned environment (see section 4.2). The virtual environment permitted to modify the operating system, the browser and the security key used during the authentication operation:

- **Operating Systems**: Windows 10 Enterprise v1909, Ubuntu 20.04.4 LTS, Kali Linux 2021.4 and Manjaro KDE 21.3.2, running on Virtual Machines plugged to the VirtualBox network attached to the clients network.
- **Web browsers**: Firefox and Chrome/Chromium were tested in all Operating Systems.
- **Security keys**: The Solokey, the Yubikey 5 NFC and the Titan Security Key USB were used with the Operating Systems and web browsers, via the VirtualBox USB forwarding that connects host USBs to the virtual machines.

The resulting environment configurations served to test both the WebAuthn authentication and registration but also the integration with the captive portal at the same time. In the first case, the support of discoverable credentials was tested, by using the developed configuration option in the WebAuthn Authentication Server.

During the integration validation, End-to-end (E2E) testing was performed with the complete environment, with the following steps:

1. The enviroment is reset: the WebAuthn Authentication Server and the OpenNDS service are restarted.
2. The virtual machine with the corresponding Operating System is plugged to the virtual clients network.
3. If the operating system detects the captive portal, the default web browser will connect to the captive portal.
4. A manual request to `http://gnome.org` is performed in the selected browser, which will be redirected to the captive portal.
5. According to the discoverable credentials support results, the authentication operation is configured. If it supports discoverable credentials, the default configuration is used. Otherwise, the non-discoverable credentials are configured.
6. The security key is plugged in the virtual machine and the authentication operation is initiated.
7. Once authenticated, the captive portal should redirect to `http://gnome.org`, the original request.

Both the WebAuthn support and the OpenNDS integration validation results can be found in the next chapter, sections 5.1 and 5.3.

5 Results

The work accomplished in this Master's Thesis resulted in a complete system that implements a web server compatible with WebAuthn security keys which can be integrated in OpenNDS captive portal for network authentication. In this chapter, the concrete results are described: (1) the WebAuthn Authentication Server; (2) the analysis of OpenNDS derived from the reverse engineering process performed prior to the integration; and (3) the integration of the server with OpenNDS captive portal. Finally, in section 5.4 the results are discussed.

The developed system software that implements webauthn authentication and is compatible with OpenNDS has been published as open source in Github [@Riv22]. Additionally, it provides documentation of the configuration options and some instructions for its deployment.

5.1 WebAuthn Authentication Server

The WebAuthn Authentication Server is a web server that supports a generic authentication schema, as described in the generic scenario use cases in section 4.1.3. All the designed use cases, which fulfill the objectives, were successfully implemented. The server allows a user to authenticate using security keys compatible with the WebAuthn standard, using a web browser in a compatible environment. Registration of users, and their corresponding security keys, are done by an administrator, which is a privileged user implemented using RBAC roles.

The web server back-end is implemented in *Typescript*, which can be executed in a NodeJS environment. The server can be cloned, installed and executed with simple commands, as all dependencies are managed by the *Node Package Manager*. The same code delivers the front-end implementation, composed of several web pages with custom JavaScript code that manages the registration and authentication client-side logic.

The developed server is compatible with the last W3C Recommendation of the WebAuthn standard [@W3C21], implementing both discoverable and non-discoverable

WebAuthn credentials. This feature makes all types of WebAuthn security keys compatible with the server registration and authentication procedures. Table 5.1 shows the validation results of the compatibility of discoverable credentials, and demonstrates that the majority of cases do not support them yet (see figure 5.1). If they are not compatible, the developed server will now support older non-discoverable credentials through the "Use key without internal storage" option (see figure 5.2).

Browser	Security key	Windows 10	Ubuntu 22.04	Kali Linux 2021.4	Manjaro KDE 21
Firefox	Solokey	No	No	No	No
	Yubikey	No	No	No	No
	Google Titan	No	No	No	No
Google Chrome	Solokey	No	Yes	Yes	Yes
	Yubikey	No	Yes	Yes	Yes
	Google Titan	No	No	No	No

Tab. 5.1.: Compatibility with discoverable credentials. Older non-discoverable credentials are compatible in all cases.

Registration and authentication operations make WebAuthn the first and single-factor authentication mechanism of the server, storing the registered security keys associated with each user in a persistent database in *MongoDB*. All this implementation relies on the popular *SimpleWebAuthn* software library [@Mil22] for both back and front-ends, which ensures future maintenance.

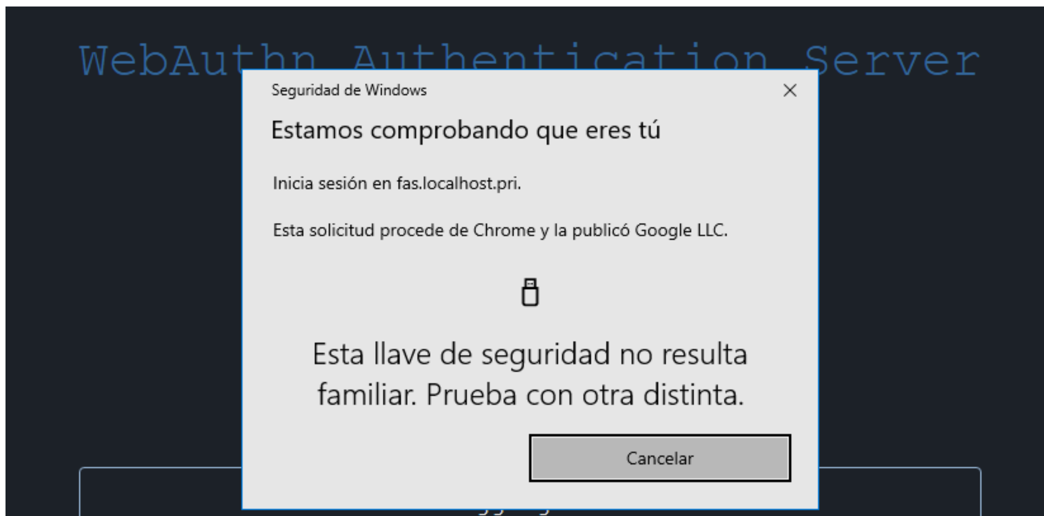

Fig. 5.1.: Using discoverable credentials in Windows has not yet been supported. The dialog shows an error: "This security key does not look familiar. Try with another one.". This screenshot is part of the performed tests to create table 5.1.

Additionally, an administrator user in the server can list authenticated users in real time (see figure 5.3). This is useful as the primary purpose of the server is user authentication. The session database allows users to open more than one authenticated web session, which are listed in the administration panel. Although

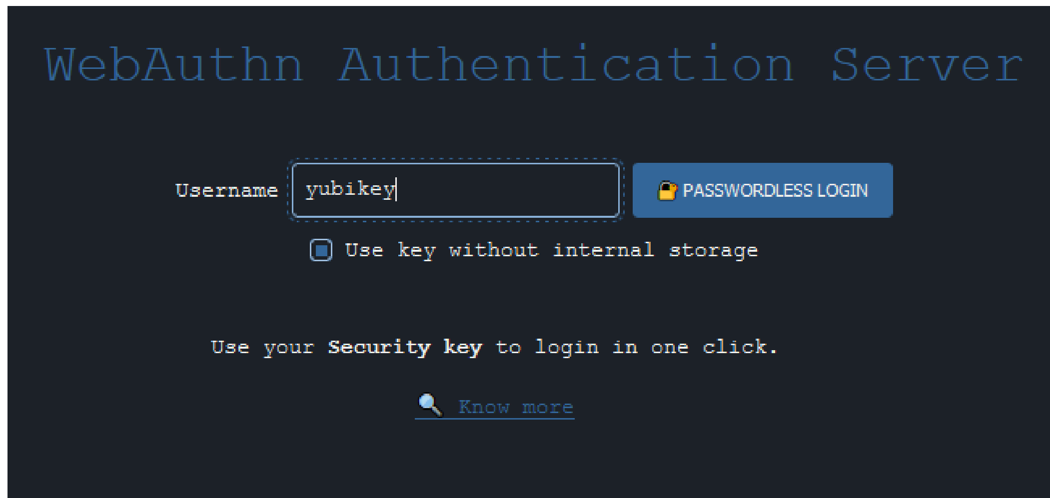

Fig. 5.2.: Developed Captive Portal UI. The WebAuthn login form supports non-discoverable credentials through the "Use key without internal storage" option.

the user can logout at any time during the session, the expiration time will force the end of the web session automatically.

Finally, an administrator can register security keys (see figure 5.3). These can be associated with an existing user by specifying the username, or can be registered and associated to a new user. Therefore, an user can have multiple registered security keys. This feature allows users to have a backup security key, which can be used in case of device loss to securely gain access to the network.

5.2 Analysis of OpenNDS FAS

During this project, one of the main works was the integration of the aforementioned developed server with a captive portal software, specifically, with OpenNDS. In order to achieve such integration, OpenNDS was analysed following a reverse engineering methodology, whose tasks are documented in section 4.4.2. Hereafter are presented the main results of this work: a complete analysis of OpenNDS.

OpenNDS is a captive portal system that can be installed on OpenWRT router firmware. It is designed to be configured with a default captive portal that can be modified with different parameters. However, the most relevant feature is the *Forwarding Authentication Service* (FAS) configuration option. This allows using an external server as authentication service and integrate it with the *Authmon* OpenNDS module. For this integration, the reverse engineering helped to detail how the authorisation operation is performed (see figure 5.4).

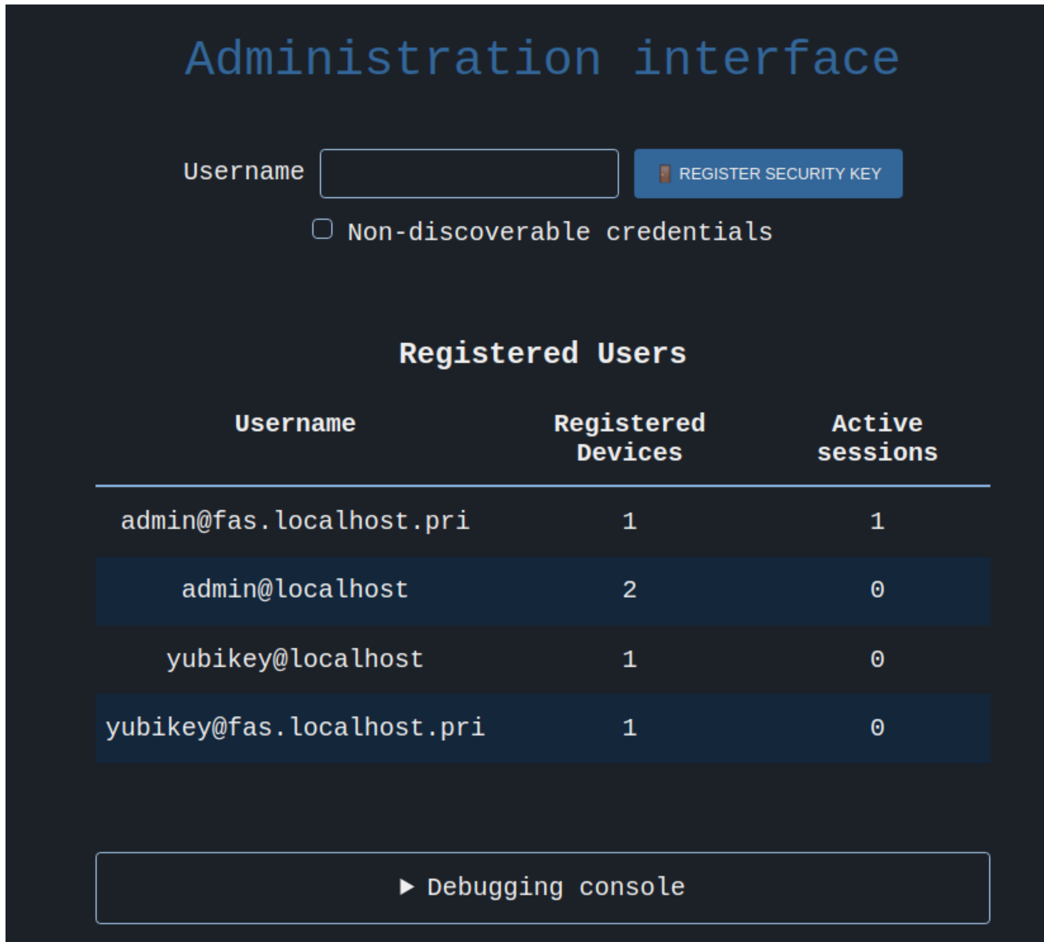

Fig. 5.3.: The administration interface of the developed authentication server. It allows registration of security keys. The table of registered users lists the registered device of each user and the active sessions in real time.

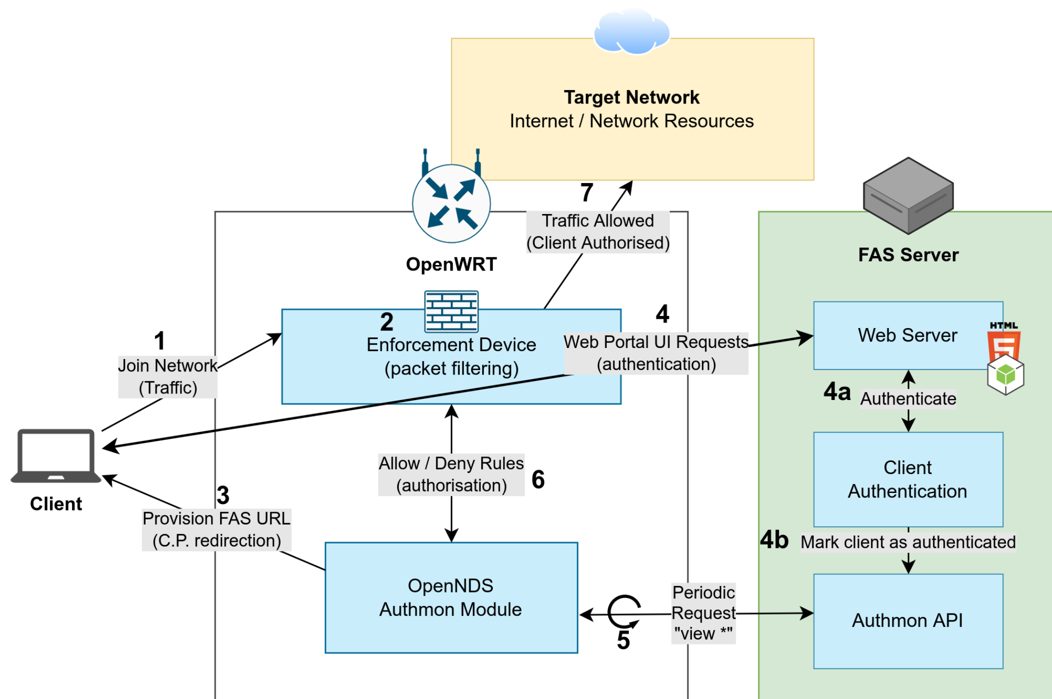

Fig. 5.4.: Diagram result of the analysis of OpenNDS working with a remote authentication server, in FAS mode. The remote server of this project after authentication works with WebAuthn. OpenWRT is the router firmware where OpenNDS is installed.

The Authmon module follows a polling message methodology. This means that it sends periodic HTTP REST requests to the external authentication server to retrieve a list of authorised clients (see step 5 of figure 5.4). In order to identify the clients, a hashed identifier is provided in the original request to the authentication server, originated from a redirection performed by the Authmon module. Once the client is authenticated, this identifier is included in the list and returned when the periodic Authmon request is received. Finally, Authmon notifies the external server the clients that have been successfully authorised.

All these operations work using encryption with a symmetric shared key. This key is configured in both the OpenNDS software and the external server. The client identifier, together with the rest of the details, is encrypted using AES-256-CBC. When answering the periodic requests, this identifier is hashed appending the symmetric key, authenticating the client hashed identifier. This cryptographic result serves as a token for Authmon to authorise the final client.

Finally, the periodic HTTP requests are required to be encrypted by TLS, which were decrypted in the analysis process. These requests use three types of different commands, as documented in section 4.4.2. The most important is the "view" command, which is used to retrieve the complete list of authenticated clients to be

authorised. The same "view" command, together with the authorisation token, is issued by Authmon to notify the confirmation of the authorisation of a client.

This analysis made possible the integration of the developed WebAuthn Authentication Server with OpenNDS.

5.3 WebAuthn authentication integrated in OpenNDS

The most relevant result of this Master's Thesis is the successful integration of the developed WebAuthn authentication server with the OpenNDS captive portal network authentication system. This integration work modified the developed server to work within a network authentication environment that uses a web-based captive portal when restricting access to network resources. Once a client is authenticated using security keys, the captive portal authorises access to the network.

In addition, this integration joins the compatibility benefits of OpenNDS and the developed WebAuthn Authentication Server. Therefore, the integrated solution was validated with two major Operating Systems and a compatible WebAuthn environment, as explained in section 5.1. Table 5.2 shows the captive portal validation of the correct network authorisation of the client traffic after a successful WebAuthn authentication (see figure 5.5). Besides, the last column represents if there is any method of Captive Portal Detection (CPD) in the Operating System that triggered the display of the captive portal once connected to the network. Figure 5.6 shows the Firefox browser opened after CPD in Windows 10, while figure 5.7 shows the default web browser displayed after CPD in Ubuntu 20.04 with the gnome shell, which does not support WebAuthn.

```
[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node index.ts`
🧱 CAPTIVE PORTAL mode.
🧱 If openNDS is restarted, wait at least 60 seconds to authenticate.
🚀 Server ready at https://fas.localhost.pri:4443 (192.168.58.1:4443)
✅ Database is connected
✅ Admin is registered
🔓 Confirmation of openNDS client authentication: dd787ad2a8eaea4b053a684aff9a19afa1aa8444b1a406221bf94e1b30f3e317
🟢 OpenNDS authlist cleared
🔓 Confirmation of openNDS client authentication: 8eb4073166638397782d9892c57612698692d9c593235278b1d76fb5a17e8aaf
🔓 Confirmation of openNDS client authentication: ef66b9d35a265c0236b127113206b5356cb381b917532aefde50730af00b81ce
```

Fig. 5.5.: Server logs of the developed authentication server, once integrated with OpenNDS captive portal in FAS mode.

Finally, the integration modifications enable two extra features. First, the web sessions of the WebAuthn Authentication Server have been synchronised with the

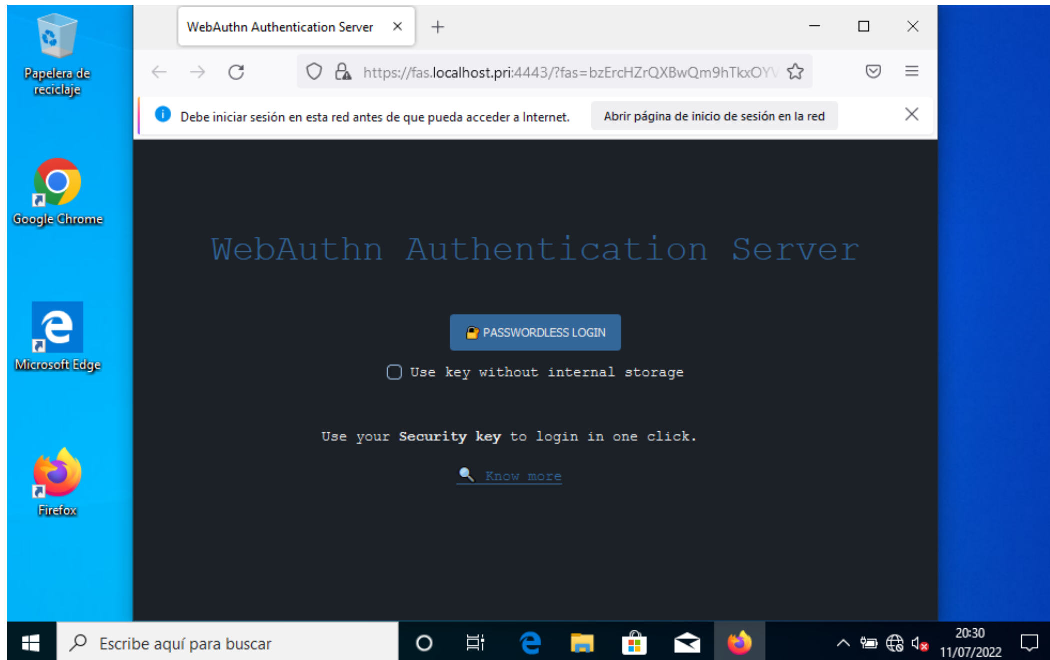

Fig. 5.6.: Firefox browser opened after CPD in Windows 10 Enterprise. Firefox also suggests to open the Captive Portal UI to login before using the network.

Operating System	Network authorisation	Captive Portal Detection
Windows 10 Enterprise	Correct, redirected	Supported. Connectivity test initiated by Windows active probing of Internet connection. The default browser can be configured the first time the captive portal is shown.
Ubuntu 20.04.4 LTS	Correct, redirected	Supported. Connectivity test initiated by gnome shell "hotspot login". Browser not compatible with WebAuthn.
Kali Linux 2021.4	Correct, redirected	Not supported
Manjaro KDE 21.3.2	Correct, redirected	Supported. Connectivity test initiated by Network Management. Firefox as default browser.

Tab. 5.2.: Validation results of the captive portal network traffic authorisation after successful WebAuthn authentication. The last column gives details of the automatic Captive Portal Detection (CPD) mechanisms of each operating system.

captive portal sessions. That means that the expire time of the network traffic authorisation can be configured to be synchronised with the web session. On the other hand, the resulting integrated server provides a central point of authentication compatible with a multi-gateway scenario. That is, the server can be deployed to be used as an WebAuthn authentication server used in captive portals managed by different OpenWRT routers in different networks. This way, the same user can be authenticated in different networks, creating different sessions that can be viewed from the administration panel, while all gateway requests are managed independently.

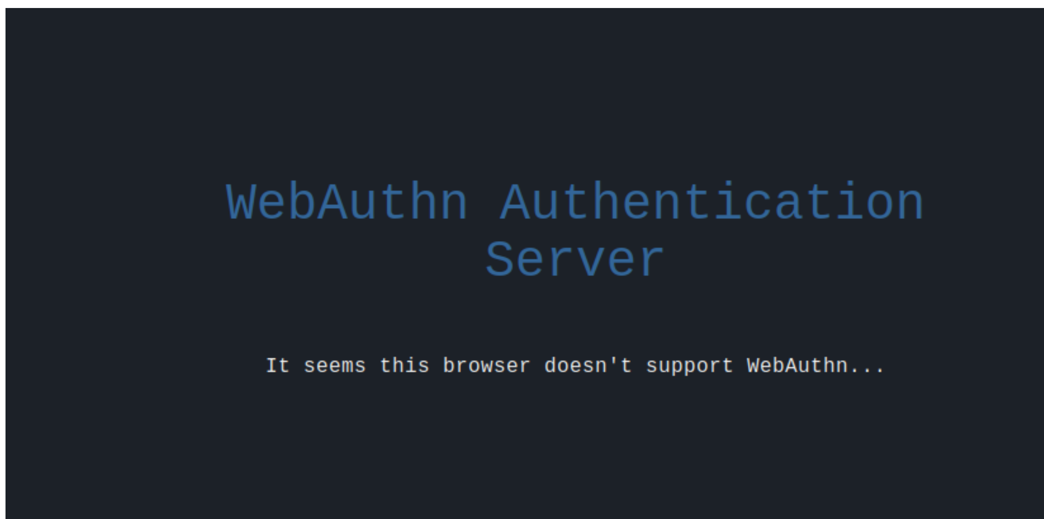

Fig. 5.7.: Ubuntu 20.04 Hotspot Login after CPD. The default opened web browser does not support WebAuthn.

Thanks to the integration with OpenNDS, the solution is compatible with a real deployment scenario. OpenNDS can be installed on OpenWRT router firmware, which can form part of a final production environment easy to deliver with multiple hardware routers. These devices can run the OpenNDS captive portal software, configured to be used with the integrated developed WebAuthn server. This way, when a client connects to the network via OpenWRT, it gets redirected to the WebAuthn Authentication Server, who manages the request accordingly. After the client authentication, the authentication server awaits for the periodic request (see figure 5.8) and confirms the client authorisation, redirecting them to the original webpage.

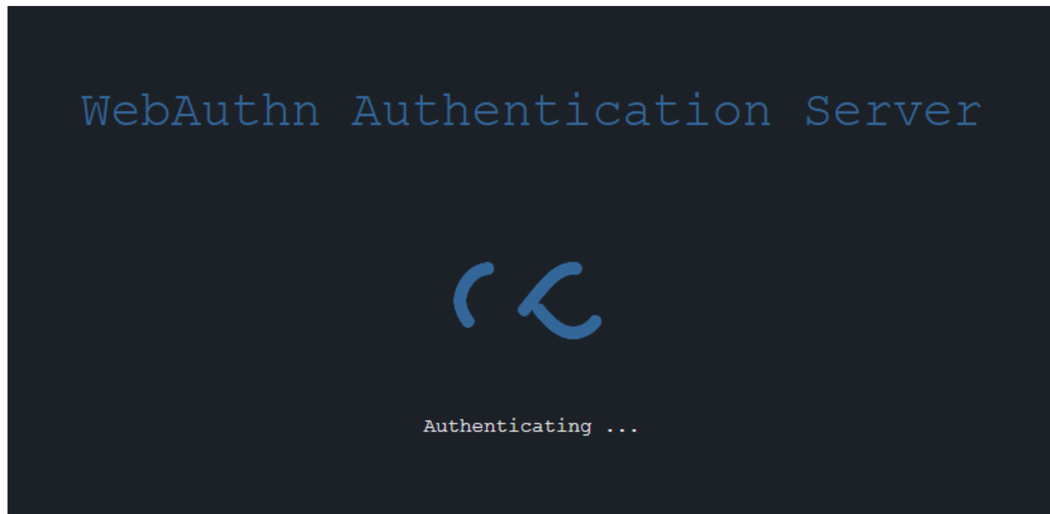

Fig. 5.8.: The captive portal loading GIF after authentication, awaiting for the confirmation of the authorisation of the client traffic. Once confirmed, the client gets redirected to the original URL.

5.4 Discussion

The Master's Thesis hypothesis has been proved right. The main objective and the secondary objectives have been fulfilled. Specifically, the development of a new authentication server based on WebAuthn and its integration with a captive portal. The final integrated system can be installed on a network to authenticate users with WebAuthn security keys.

An integration of this kind is not trivial. The work accomplished included a reverse engineering process of OpenNDS to detail the specific requirements of the working of this software integrated with an external authentication service. In order to integrate WebAuthn in the captive portal, first the WebAuthn Authentication Server was developed. This development required a study of the WebAuthn standard and the registration and authentication operations. Also, for creating an useful authentication server, several web technologies were applied, like REST interfaces and web sessions.

In conclusion, the project resulted on an useful software project that can be deployed in real environments. It serves as a *Proof of Concept* (PoC) of how WebAuthn authentication can be integrated with other systems, and serve as a replacement of passwords and their vulnerabilities. Therefore, security keys can now be used with captive portals that authenticate users in real network authentication environments.

6 Conclusions and future work

This chapter summarises the main author conclusions product of the Master's Thesis work during these months. All the documented process reflects specific design, analysis and development work in the cybersecurity context and, specifically, in the authentication mechanisms.

6.1 Conclusions

Passwords have formed part of the authentication paradigm for decades. They are simple for a user to understand, and their security for the verification of the user identity is based on the secrecy of the password itself. Threats like *phishing* have become a widespread attack among the different authentication systems.

Having multiple passwords for several services make their management difficult. Some identity systems present in business-like solutions try to centralise the authentication for their different services. This solution makes users to remember a single password but, if the password is compromised, all the business systems are left unprotected. In this context, WebAuthn security keys represent an opportunity to improve security by migrating the existing authentication services to a new authentication paradigm. Taking into account that central authentication involves different systems, the migration to WebAuthn security keys should be done in all authentication scenarios of a business. They include web authentication, but also other systems like network access control.

WebAuthn is a recent standard that is still under development. Although it has already been implemented in web authentication in different operating systems, browsers and devices, their applicability to further scenarios has not yet been studied. The research work of this Master's Thesis proves that there are other applications, like network authentication. Although the underlining technology is web-based, the integration with a real captive portal enforcement device demonstrates the potential of security keys in network authentication.

This work also demonstrates that integrating WebAuthn security keys with existing systems is a tedious task. Redesigning authentication and authorisation procedures have to take into account different factors and requirements, dependent on the different environments. For integrating security keys in other systems, the low-level FIDO standards can serve as a base to create new authentication protocols, like WebAuthn did for web authentication.

The main result of this work was a WebAuthn server that is compatible with OpenNDS captive portal system. To the best of the author knowledge, there is no previous work that integrates WebAuthn authentication in a captive portal for network authentication. Therefore, the results of this work directly contribute to the scientific knowledge of the applicability of the new authentication standard based on security keys to other environments.

The design and methodology used during this work can also help in the integration of security keys with different authentication systems. Besides, the analysis of OpenNDS can also serve as reference for developers that also plan to integrate external authentication servers with captive portal systems, as well as to identify possible issues with this network authentication technology.

6.2 Future work

This section includes the limitations of the work presented in this Master's Thesis and the future work and research lines. These are some of the most relevant ideas that are considered by the author to be executed in his following research projects.

Deployment of the developed system in real hardware

Although during this work the development and validation environment was virtual, the developed system can be deployed in real network hardware. There are many market routers compatible with OpenWRT and, therefore, with the developed system. Some examples are the ASUS RT-AC51U, the Netgear Nighthawk R7000 or the TP-Link AC1750 Archer A7. Deployment in this scenario can further validate the developed system with a real scenario.

Validation of the system with Android as a security key

Apart from hardware dedicated security keys, WebAuthn allows software developments of authenticator devices. One of the most relevant WebAuthn authenticator are present in new Android versions (like Android 11 or 12) which are compatible with the *Trusted Execution Environment* (TEE) module. That means that an Android

phone can be used as a security key. Therefore, a future research task can validate the compatibility of the developed system using an Android phone as a security key.

Integration of the developed system with central identity directories

The developed captive portal authentication system stores user details in a dedicated database. However, a real scenario may include identity directories like the Microsoft Active Directory. These identity directory often implement the *Lightweight Directory Access Protocol* (LDAP), which can be encapsulated with popular protocols like *Remote Authentication Dial-In User Service* (RADIUS). This integration would extend compatibility with real network authentication and user management environments.

Security key network authentication with link-level protocols

The authentication of a captive portal is performed at the application level. The enforcement policies are managed by the networking equipment, which forms part of the authorisation process that restricts or allows traffic to clients. That is, the connectivity and network access control system is independent of the authentication mechanism implemented in the captive portal. Other network authentication systems that rely on the link-level protocols like WPA2/3 Enterprise or MACsec do not form part of the developed system.

The future work that follows the developed system can continue with integrating the security keys in network authentication systems, further than web-based captive portals. This work can explore the limitations of the solution presented here, with protocols like the *Extensible Authentication Protocol* (EAP), implemented in access-layer networking devices to restrict connectivity of clients. This includes networks with WPA2/3 Enterprise or MACSec, which form part of the 802.1X protocols that can use EAP for authentication.

Bibliography

[Ali+19] Suzan Ali, Tousif Osman, Mohammad Mannan, and Amr Youssef. “On Privacy Risks of Public WiFi Captive Portals.” en. In: 11737 (2019). Ed. by Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro. Series Title: Lecture Notes in Computer Science, pp. 80–98 (cit. on p. 18).

[AT10] Pierre Anderberg and Erik Thorselius. “How to Circumvent a Captive Portal.” en. In: *Pennsylvania State University* (2010), p. 5 (cit. on p. 18).

[Gup+17] B. B. Gupta, Aakanksha Tewari, Ankit Kumar Jain, and Dharma P. Agrawal. “Fighting against phishing attacks: state of the art and future challenges.” en. In: *Neural Computing and Applications* 28.12 (Dec. 2017), pp. 3629–3654 (cit. on p. 1).

[KK20] Warren "Ace" Kumari and Erik Kline. *RFC 8910: Captive-Portal Identification in DHCP and Router Advertisements (RAs)*. Request for Comments RFC 8910. Num Pages: 11. Internet Engineering Task Force, Sept. 2020 (cit. on p. 20).

[LDL20] K. Larose, D. Dolson, and H. Liu. *RFC 8952: Captive Portal Architecture*. en. Request for Comments RFC8952. RFC Editor, Nov. 2020, RFC8952 (cit. on pp. 18, 19).

[MZB20] Nuno Marques, André Zúquete, and João Paulo Barraca. “EAP-SH: An EAP Authentication Protocol to Integrate Captive Portals in the 802.1X Security Architecture.” en. In: *Wireless Personal Communications* 113.4 (Aug. 2020), pp. 1891–1915 (cit. on p. 20).

[Riv20] Martiño Rivera-Dourado. “DebAuthn: a Relying Party Implementation as a WebAuthn Authenticator Debugging Tool.” gl. In: *RUC, Universidade da Coruña* (2020), p. 128 (cit. on pp. 15, 16, 27).

[Riv+21] Martiño Rivera-Dourado, Marcos Gestal, Alejandro Pazos, and José M. Vázquez-Naya. “An Analysis of the Current Implementations Based on the WebAuthn and FIDO Authentication Standards.” en. In: *Engineering Proceedings* 7.1 (2021). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 56 (cit. on pp. 13, 15).

[RGV20] Martiño Rivera-Dourado, Marcos Gestal, and José M. Vázquez-Naya. “Implementing a Web Application for W3C WebAuthn Protocol Testing.” en. In: *Proceedings* 54.1 (2020). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, p. 5 (cit. on p. 28).

[SC09] Seref Sagiroglu and Gurol Canbek. “Keyloggers: Increasing threats to computer security and privacy.” In: *IEEE Technology and Society Magazine* 28.3 (2009). Conference Name: IEEE Technology and Society Magazine, pp. 10–17 (cit. on p. 2).

[Tir+18] Emanuel Tirado, Brendan Turpin, Cody Beltz, et al. “A New Distributed Brute-Force Password Cracking Technique.” en. In: *Future Network Systems and Security*. Ed. by Robin Doss, Selwyn Piramuthu, and Wei Zhou. Communications in Computer and Information Science. Cham: Springer International Publishing, 2018, pp. 117–127 (cit. on p. 2).

Webpages

[@Alla] Wi-Fi Alliance. *Security | Wi-Fi Alliance*. URL: `https://www.wi-fi.org/discover-wi-fi/security#Wi-FiEnhancedOpen` (visited on July 6, 2022) (cit. on p. 20).

[@All18] FIDO Alliance. *Client to Authenticator Protocol (CTAP)*. 2018. URL: `https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-client-to-authenticator-protocol-v2.0-id-20180227.html` (visited on July 6, 2022) (cit. on pp. 2, 14).

[@Allb] FIDO Alliance. *FIDO2*. en. URL: `https://fidoalliance.org/fido2/` (visited on July 6, 2022) (cit. on p. 3).

[@All17] FIDO Alliance. *Universal 2nd Factor (U2F) Overview*. 2017. URL: `https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html` (visited on July 6, 2022) (cit. on pp. 2, 14).

[@ASU21] ASUS. *[Guest Network] How to set up Captive Portal?* en. 2021. URL: `https://www.asus.com/support/FAQ/1034977/` (visited on June 29, 2022) (cit. on p. 21).

[@Bur16] Alberto Bursi. *OpenWrt on VirtualBox HowTo*. en. Dec. 2016. URL: `https://openwrt.org/docs/guide-user/virtualization/virtualbox-vm` (visited on May 25, 2022) (cit. on p. 40).

[@Cis17] Cisco. *Enable a Captive Portal on your Cisco Wireless Network*. en. 2017. URL: `https://www.cisco.com/c/en/us/support/docs/smb/wireless/cisco-small-business-300-series-wireless-access-points/smb4937-enable-a-captive-portal-on-your-cisco-wireless-network.html` (visited on June 29, 2022) (cit. on p. 21).

[@Goo] Google. *Titan Security Key*. en. URL: `https://cloud.google.com/titan-security-key` (visited on July 6, 2022) (cit. on pp. 2, 15).

[@Lab22a] Duo Labs. *py_webauthn*. original-date: 2017-11-10T18:02:28Z. July 2022. URL: https://github.com/duo-labs/py_webauthn (visited on July 6, 2022) (cit. on p. 16).

[@Lab22b] Duo Labs. *WebAuthn Library*. original-date: 2017-10-26T16:15:55Z. July 2022. URL: https://github.com/duo-labs/webauthn (visited on July 6, 2022) (cit. on p. 16).

[@Lib22] Passwordless Library. *FIDO2 .NET Library (WebAuthn)*. original-date: 2018-05-31T07:51:51Z. July 2022. URL: https://github.com/passwordless-lib/fido2-net-lib (visited on July 6, 2022) (cit. on p. 16).

[@Lin] Linksys. *What is a Captive Portal?* en-US. URL: http://www.linksys.com/us/r/resource-center/captive-portal/ (visited on June 29, 2022) (cit. on p. 21).

[@Mil22] Matthew Miller. *SimpleWebAuthn Project*. original-date: 2020-05-22T16:21:42Z. July 2022. URL: https://github.com/MasterKale/SimpleWebAuthn (visited on July 6, 2022) (cit. on pp. 16, 44, 64).

[@Mon] Álvaro Montoro. *Almond.CSS*. en-US. URL: https://alvaromontoro.github.io/almond.css/ (visited on July 6, 2022) (cit. on p. 44).

[@Moz] Developer Mozilla. *Web Authentication API - Web APIs | MDN*. en-US. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API#browser_compatibility (visited on July 6, 2022) (cit. on p. 15).

[@NET] NETGEAR. *Captive Portal Guest WiFi | Insight*. en. URL: https://www.netgear.com/business/services/insight/captive-portal-guest-wifi/ (visited on June 29, 2022) (cit. on p. 21).

[@Onl] Onlykey. *OnlyKey Hardware Password Manager | One PIN to remember*. URL: https://onlykey.io/ (visited on July 6, 2022) (cit. on p. 14).

[@Ope] OpenNDS. *Forwarding Authentication Service (FAS) openNDS v9.7.0*. URL: https://opennds.readthedocs.io/en/stable/fas.html (visited on June 29, 2022) (cit. on p. 54).

[@pfS] pfSense. *Captive Portal | pfSense Documentation*. URL: https://docs.netgate.com/pfsense/en/latest/captiveportal/index.html (visited on June 29, 2022) (cit. on pp. 21, 53).

[@Pro16] OpenWrt Project. *Welcome to the OpenWrt Project*. en. Sept. 2016. URL: https://openwrt.org/start (visited on June 29, 2022) (cit. on pp. 22, 53).

[@Riv22] Martiño Rivera-Dourado. *WebAuthn Authentication Server*. July 2022. URL: https://github.com/martinord/webauthn-server (visited on July 6, 2022) (cit. on p. 63).

[@Sol] Solokeys. *Get your Solo*. URL: https://solokeys.com/collections/all (visited on July 6, 2022) (cit. on pp. 2, 14).

[@W3C21] W3C. *Web Authentication: An API for accessing Public Key Credentials - Level 2*. 2021. URL: https://www.w3.org/TR/webauthn-2/ (visited on July 6, 2022) (cit. on pp. 3, 8, 14, 63).

[@Wav] Waver. *WAVER - Guest WiFi, Captive Portal & Marketing without fees*. en-US. URL: https://www.wavertech.eu/ (visited on June 29, 2022) (cit. on p. 21).

[@web22] webauthn4j. *WebAuthn4J*. original-date: 2018-05-20T12:14:36Z. July 2022. URL: https://github.com/webauthn4j/webauthn4j (visited on July 6, 2022) (cit. on p. 16).

[@Whi22] Rob White. *openNDS/openNDS*. original-date: 2020-03-23T11:53:26Z. June 2022. URL: https://github.com/openNDS/openNDS (visited on June 29, 2022) (cit. on pp. 22, 53).

[@Yub] Yubico. *Yubico Products : Yubikey*. en-US. URL: https://www.yubico.com/products/ (visited on July 6, 2022) (cit. on pp. 2, 14).

List of Figures

2.1. Evolution of the FIDO and WebAuthn standards. FIDO and WebAuthn standards are wrapped arround the FIDO2 Project, which focuses in the new authentication method based on security keys. 7
2.2. General communication protocol of a WebAuthn operation. 9
2.3. Communication protocol of a WebAuthn registration. The operation is also known as attestation ceremony. 10
2.4. Communication protocol of a WebAuthn authentication. The operation is also known as assertion ceremony. 11
2.5. Discoverable credentials (resident credentials): the private key is saved at the authenticator memory. The public key is sent and stored in the server. 12
2.6. Non-discoverable credentials (non-resident credentials): the private key is sent encrypted to the server with a symmetric key. This symmetric key is unique to the device. The credential identifier is the result of the encryption, so it is sent during authentication back to the authenticator for decryption. The public key is sent and stored in the server. 13
2.7. Captive portal architecture as proposed in RFC 8952 [LDL20]. The captive portal architecture includes the user equipment, working as a client, and the elements of the captive portal implementations. 19

3.1. Gantt diagram of the Master's Thesis project. In green, the main project phases and in blue the corresponding subphases. Yellow boxes represent the unexpected final delays of the project, with respect to the planned timing of each phase and subphase. 25
3.2. Security keys used in this project. From left to right in the same order than in table 3.3. 27

4.1. Overview of the general WebAuthn registration and authentication scenario. 34
4.2. Statechart of WebAuthn authentication and the basic session management. 34
4.3. Overview of the captive portal integration with WebAuthn. 35
4.4. Statechart of the session management integrated with the router serving a captive portal that authorises the client. 35

4.5. Use cases of the generic scenario: WebAuthn web application. The main actors administrator and user are associated with the RBAC roles. . . . 37
4.6. Use cases of the final scenario: Captive portal with WebAuthn. The Captive Portal actor represents the captive portal external system. . . . 38
4.7. Virtual networking topology used in the environment. Setup in VirtualBox. The router manages the four defined networks. The OpenWRT interfaces are represented as `eth0-3`. The logical OpenWRT bridges are represented by `br-<tag>`. 40
4.8. Configuration of a bridge device in OpenWRT through the LUCI web admin panel. 42
4.9. Configuration of a interface in OpenWRT through the LUCI web admin panel. 42
4.10. WebAuthn registration with discoverable credentials: cross-functional diagram. The device is added to a user if it exists. Otherwise, the user is created. 46
4.11. WebAuthn authentication with discoverable credentials: cross-functional diagram. The `userHandle` is used for identifying the user to authenticate. 46
4.12. User and credentials class diagram. The user has at least one Security Key associated. 47
4.13. Class diagram representing the registered user database model. The user class is simplified for showing only the aggregated user details. . . 48
4.14. WebAuthn authentication with non-discoverable credentials: cross-functional diagram. The list of allowed credentials is sent in the authentication options, as the user is identified before authentication verification. 52
4.15. Communication between OpenNDS Authmon module and the developed FAS server. The client connects, gets redirected to the captive portal hosted at the FAS server and, once authenticated, it is authorised by OpenNDS. 58

5.1. Using discoverable credentials in Windows has not yet been supported. The dialog shows an error: ”This security key does not look familiar. Try with another one.”. This screenshot is part of the performed tests to create table 5.1. 64
5.2. Developed Captive Portal UI. The WebAuthn login form supports non-discoverable credentials through the ”Use key without internal storage” option. 65
5.3. The administration interface of the developed authentication server. It allows registration of security keys. The table of registered users lists the registered device of each user and the active sessions in real time. . 66

5.4. Diagram result of the analysis of OpenNDS working with a remote authentication server, in FAS mode. The remote server of this project after authentication works with WebAuthn. OpenWRT is the router firmware where OpenNDS is installed. 67
5.5. Server logs of the developed authentication server, once integrated with OpenNDS captive portal in FAS mode. 68
5.6. Firefox browser opened after CPD in Windows 10 Enterprise. Firefox also suggests to open the Captive Portal UI to login before using the network. 69
5.7. Ubuntu 20.04 Hotspot Login after CPD. The default opened web browser does not support WebAuthn. 70
5.8. The captive portal loading GIF after authentication, awaiting for the confirmation of the authorisation of the client traffic. Once confirmed, the client gets redirected to the original URL. 71

C.1. Wireshark Flow Graph of network traffic arriving to the remote authentication server. In purple, the Authmon periodic requests. In green, the client requests. 92

List of Tables

3.1. Human resources estimated and monitored cost in euros. 24
3.2. Total estimated and monitored cost in euros, including human resources and materials 26
3.3. Security keys used during this project. They are classified according to their support with discoverable credentials. 27

4.1. Network topology and OpenWRT interfaces: three LANs configured as VirtualBox networks, and one NAT interface that enables access to Internet. 41
4.2. HTTP REST API for the WebAuthn authentication server before the integration with the captive portal. The RBAC column represents the role that can access the API. All logged in users have the role user, but not all of them have the role admin, which is restricted to administrators. 50
4.3. Root endpoints of the API that are extended in functionality for handling requests once integrated with OpenNDS. 59

5.1. Compatibility with discoverable credentials. Older non-discoverable credentials are compatible in all cases. 64
5.2. Validation results of the captive portal network traffic authorisation after successful WebAuthn authentication. The last column gives details of the automatic Captive Portal Detection (CPD) mechanisms of each operating system. 69

A Deployment and installation of TLS certificates

The environment explained in section 4.2 of this document has been used during all project phases. The configuration of OpenNDS selected for the integration phase of a remote web server requires the usage of TLS encrypted traffic for the HTTP communication.

Although clients can manually accept exceptions of unknown *Certificate Authorities* (CAs) for self-signed TLS certificates when web browsing, the OpenNDS HTTPS requests need to have valid certificates. Therefore, for them to be valid, they have to be manually installed in the trusted CA certificate storage.

OpenNDS works on OpenWRT, a Linux-based minimal distribution. The process included here may be compatible with other Linux-based distributions similar to OpenWRT. Commands of the listing A.1 generate the required CA and TLS certificates.

List. A.1: Self-signed TLS certificates.

```
#CA
openssl genrsa -out ca.key 2048
openssl req -x509 -new -nodes \
     -key ca.key -sha256 \
     -days 1825 -out ca.crt
#TLS certificates
openssl genrsa -out server.key 2048
cat > csr.conf <<EOF
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
req_extensions = req_ext
distinguished_name = dn

[ dn ]
C = ES
ST = Coruna
L = Coruna
O = Development
OU = Development
CN = fas.localhost.pri
```

```
[ req_ext ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = fas.localhost.pri
IP.1 = 192.168.58.100

EOF

openssl req -new -key server.key -out server.csr -config csr.conf
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 10000 \
-extfile csr.conf
```

Finally, to install the custom CA certificate in OpenWRT, the following command should be used: `cat ca.crt >> /etc/ssl/cert.pem`. Now all HTTPS requests originated in OpenWRT will trust the generated TLS self-signed certificates.

B Reverse proxy deployment

For analysing how OpenNDS works, a reverse engineering process was followed. Although the environment for this process is essentially the same than the one exposed in section 4.2, there is one modification that allows the decryption of TLS web traffic that arrives to the remote authentication server.

In this process, the demonstration server was installed, based on PHP-FPM. For decrypting the traffic, an Nginx reverse proxy was setup. This proxy serves the web service with HTTPS (encrypted HTTP traffic with TLS), and forwards all requests to a second web server, who manages the PHP-FPM traffic.

This way, a sniffer between both servers can gather all requests in plain text.

The deployment in the remote authentication server was managed by Docker containers. The following are the deployment configurations: the *Docker Compose* file (listing B.1) and both Nginx configurations (listings B.2 and B.3).

List. B.1: *Docker Compose* configuration file, which deploys two different Nginx containers.

```
version: "3.3"

services:
    web-https:
        image: nginx:latest
        ports:
            - "4443:443"
        volumes:
            - ./proxy/default.conf:/etc/nginx/conf.d/default.conf
            - ./tls/certs:/etc/nginx/certs
        links:
            - web-http
    web-http:
        image: nginx:latest
        volumes:
            - ./src:/var/www/html:rw
            - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
        links:
            - php-fpm
    php-fpm:
        image: php:8-fpm
        volumes:
            - ./src:/var/www/html:rw
        user: "1000"
```

List. B.2: Nginx configuration of the proxy that decrypts traffic, forwarding the traffic to the second server.

```
server {
    server_name fas.localhost.pri;
    listen 443 ssl;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;

    ssl_certificate /etc/nginx/certs/fas.localhost.pri.crt;
    ssl_certificate_key /etc/nginx/certs/fas.localhost.pri.key;

    location / {
        proxy_pass http://web-http;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP  $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location ~ \.php$ {
        proxy_pass http://web-http;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP  $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

List. B.3: Nginx configuration of the PHP-FPM server which receives the final requests.

```
server {
    server_name fas.localhost.pri;
    index index.php index.html;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /var/www/html;

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass php-fpm:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME
                      $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
}
```

C Reverse engineering sniffing and logging

The reverse engineering process explained in section 4.4.2 required the usage of different techniques. This appendix includes some details regarding the network traffic sniffing with *Wireshark* and some relevant logs of OpenNDS.

C.1 Traffic sniffing with Wireshark

According to the environment and the technique explained in Appendix B, the decrypted HTTP traffic should be sniffed between the two Docker containers. Figure C.1 shows the *Flow Graph* of the HTTP traffic, filtered in Wireshark taking into account the private IP of the Docker container: `http && ip.src == 172.28.0.3`.

Additionally, other filters were used during this process, to analyse specific Authmon and client requests: `(urlencoded-form.value == "clear" || urlencoded-form.value == "view" || http.request.uri.query.parameter contains "fas=") || data-text-lines`.

Then, for decrypting the FAS details sent in the client requests, the PHP code extracted from the white-box analysis was executed (see listing C.1).

List. C.1: PHP code that decrypts the details sent by the client in the request to the remote server.

```php
<?php
$string="<base64payload>";
$iv="<randomIVvector>";
$key="1234567890";
$cipher="AES-256-CBC";
echo openssl_decrypt(base64_decode($string), $cipher, $key, 0, $iv);
?>
```

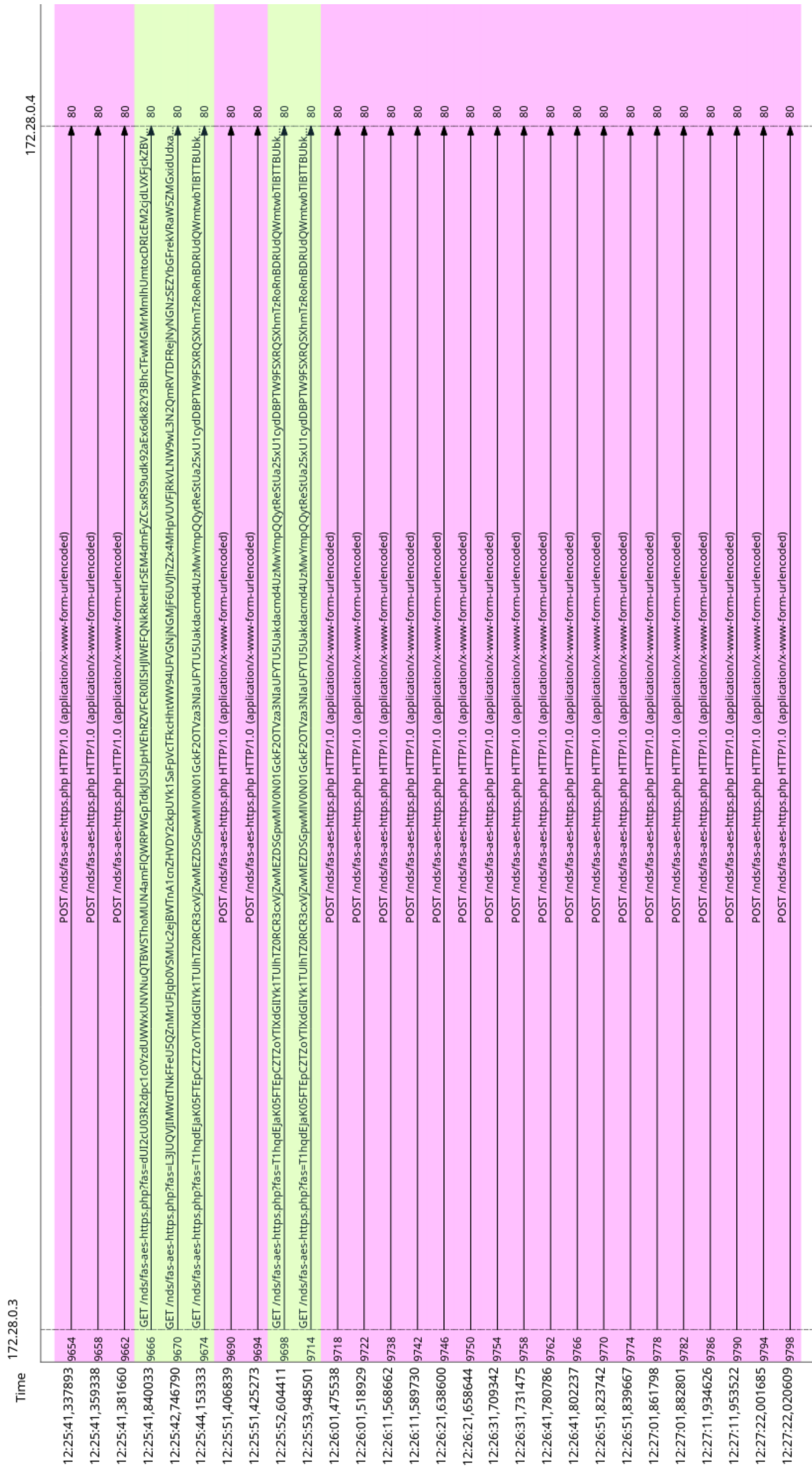

Fig. C.1.: Wireshark Flow Graph of network traffic arriving to the remote authentication server. In purple, the Authmon periodic requests. In green, the client requests.

C.2 OpenNDS verbose logging

The traffic sniffing was combined with a log analysis. Thanks to the verbose mode of OpenNDS, the requests can be identified with a specific behaviour of the captive portal system. Listing C.2 includes some of the logs produced by OpenNDS.

List. C.2: OpenNDS logs gathered during the analysis process. They have been modified to include the relevant information.

```
Locking client list
Client list locked
Unlocking client list
Client list unlocked
ndsctl request processed: [status]
Exiting thread_ndsctl_handler....
authmon - authlist *
authmon - remote FAS response [ack]
client access: GET /success.txt
client ip address is [ 192.168.57.131 ]
Executing command: /usr/lib/opennds/libopennds.sh get_interface_by_ip "
        192.168.57.131"
Setting default SIGCHLD handler SIG_DFL
Reading command output
command output: [br-clients]
Interface used to route ip [192.168.57.131] is [br-clients]
Gateway Interface is [br-clients]
Client ip address [192.168.57.131] is on our subnet using interface [br
        -clients]

Adding 192.168.57.131 08:00:27:50:4c:14 token c4b8e115 to client list
Unlocking client list
Client list unlocked
preauthenticated: host [detectportal.firefox.com] url [/success.txt]
url is [ /success.txt ], checking for [ opennds_preauth ]
url is [ /success.txt ], checking for [ opennds_deny ]
preauthenticated: Requested Host is [ detectportal.firefox.com ], url
        is [/success.txt]
Our host: 192.168.57.2:2050 Requested host: detectportal.firefox.com
preauthenticated: foreign host [detectportal.firefox.com] detected
Getting query, separator is [&].
Query string is [ ?ipv4 ]
URL encoded string: http%3a%2f%2fdetectportal.firefox.com%2fsuccess.txt
        %3fipv4, length: 58
originurl: http%3a%2f%2fdetectportal.firefox.com%2fsuccess.txt%3fipv4
URL encoded string: http%3a%2f%2fstatus.client, length: 26
gw_url: http%3a%2f%2fstatus.client
hid=2f11b49ba013fb85df96baea58d [...]
Executing command: /usr/lib/opennds/get_client_interface.sh 08:00:27:50
        :4c:14
Setting default SIGCHLD handler SIG_DFL
Reading command output
command output: [br-clients]
Client Mac Address: 08:00:27:50:4c:14
```

```
Client Connection(s) [localif] [remotemeshnodemac] [localmeshif]: br-
                                                clients
clientif: [br-clients]
phpcmd: echo '<?php $key="1234567890"; $string="hid=<>, clientip= [...]
                                                 | php-cli
Executing command: echo '<?php $key="1234567890"; [...]
Setting default SIGCHLD handler SIG_DFL
Reading command output
command output: [?fas=dUI2cU03R2dpc1c0YzdUWWxUN [...]
Constructed Query String [?fas=dUI2cU03R2dpc1c0YzdUWW [...]
splashpageurl: https://fas.localhost.pri:4443/nds/fas-aes-https.php?fas
                                                =dUI2cU03R2dpc1c [...]
send_redirect_temp: MHD_create_response_from_buffer. url [https://fas.
                                                localhost.pri:4443/nds/
                                                fas [...]
send_redirect_temp: Redirect body [<html><head></head><body><a href='%s
                                                '>Click here to continue
                                                 [...]
send_redirect_temp: Response created
send_redirect_temp: Location header added to redirection page
send_redirect_temp: Connection header added to redirection page
```