POLITECNICO DI TORINO

Corso di Laurea in Computer engineering

Tesi di Laurea

FIDO2 Authentication for Embedded Systems

Relatore
prof. Antonio Lioy

Gianluca Moliteo

Anno accademico 2021-2022

To my dad, who gave me his strength.

To my mum, who gave me her love.

To my brother, who gave me his support.

To my grandmothers, who have hoped, until the last, to see me in time of success.

Summary

Nowadays, the number of cybercrimes performed against companies, associations and entities is increasing drastically: organisations, from the smallest to the biggest ones, must protect themselves from all kinds of attacks. In this risky context, information systems must rely on dependable security mechanisms which have to accompany all their functional processes.

Security is composed of different concepts and properties. The authentication is among the most important ones: having the proof the user is who or what it says to be is crucial, since the whole security architecture is based on this information. Therefore, a reliable security system needs a strong authentication phase.

The riskier the context is, the more reliable the security must be, and the more impenetrable the authentication phase should be. Within a company, employees differ by their role, their position, and their responsibilities. A cyber-attack could cause more damage whether it was performed against a labourer at the top of the organisation hierarchy. Therefore, important workers such as CEOs should provide a strong proof when doing crucial processes or authorizing critical transactions.

A solution based on an embedded system is provided by this thesis to address these situations; FIDO2 has been chosen as the authentication mechanism due to its reliability and functionalities. It is resistant thanks to the usage of asymmetric techniques and algorithms, which are implemented transparently for the user: he can access to the system just by remembering a simple password or by using his fingerprint. Consequently, the experience of this system is user-friendly, independently of the complexity of the utilised techniques.

Within FIDO2, the provision of the user's identity proof is performed with the usage of digital signatures, independently of the executed task (registration, authentication, or authorisation). Offering the proof in this way is particularly crucial in authorisation tasks: the newly approved transaction could be verified in the future to assert its integrity and to check that it has been authorised by that specific user, satisfying the non-repudiation property.

The architecture of this project is logically composed of:

- a FIDO2-enable Web Application: the Relying Party (RP), the entity having the resource the user wants to access;
- a Web Application Database: storing non-secret data about users and transactions;
- a FIDO2-aware Platform: running the client side of the Relying Party;
- a FIDO Certified FIDO2 Authenticator: producing the proof of user's identity on his behalf;
- a FIDO Certified FIDO2 Server: verifying the user's identity on behalf of the Relying Party;
- a FIDO2 Server Database: storing registered users' public keys and information.

RP client, RP server and FIDO2 server need to be published into the network, which, within the built prototype, is a private one.

This logical architecture has been physically implemented using the following components:

- a Server Machine: running the FIDO Certified FIDO2 Server and the FIDO2 Server Database;
- a Raspberry Pi4: the embedded system representing the RP, which executes FIDO2-enabled Web Application, the Web Application Database, and offers the client side of RP;
- a Client Machine: running the RP client on the FIDO2-aware Platform;

- an Authenticator: it can be implemented inside the Client Machine or it can be external to it.

This system can perform different kinds of tasks, such as:

- registration of new users,
- deletion of registered users,
- authentication of registered users,
- logout of authenticated users,
- authorisation of new transactions,
- view of executed transactions,
- confirmation of executed transactions in case the algorithm used for their computation is supported by the RP.

Even though the network is the same for every component, FIDO2 server would not consider requests directly made by the RP client due to its authentication system: only the RP server, which is associated to an authentication profile within the FIDO2 server, is allowed to communicate with it.

To satisfy the confidentiality property, all the traffic passing through the network must be enciphered. Therefore, TLS is used in communication between components: the second version of this protocol is used between RP server and FIDO2 server, whilst the third version is used between RP client and server. Moreover, TLS builds a secure context required by WebAuthn APIs.

The system has been tested in several ways. First, a log system has been implemented in the RP server: all the calls performed to FIDO2 server APIs are stored in a log file with details regarding data exchanged, as well as all requested pages and logouts. This system allows the testers to see parameters and details about input and output data of the Relying Party.

Secondly, all the traffic within the network has been sniffed during the execution of all the defined tasks. Even though every packet was enciphered due to the usage of TLS, this test was anyway useful to associate messages to their senders and receivers, so that to generate logical maps among components.

Finally, a testing tool verifying the signature of transactions has been implemented: data to be checked must be manually provided. The tester could obtain information about the transaction to be verified from the log file, so that to be sure to process the actual values exchanged between components of the system: this makes this test crucial.

Acknowledgements

Thanks to professor Cataldo Basile, who has supported me in time of need.

Contents

Chapter 1

Introduction

1.1 The role of Computer Science

Information technology is one of the most crucial sciences in Human history; its relevance has crossed the boundaries of the scientific scope and has started to get involved in general human mind processes. During its still initial evolution, computer science has let humanity to reach aims and targets that it could have never considered or imagined. It has not even been a century since 1936, year in which the mathematician and philosopher Alan Turing has built his machine, the first computer in history, opening the way to actual information technology; nevertheless, the mentality of generations living this phenomenon has grown swiftly and is now growing even more quickly.

1.1.1 Cybersecurity

This context of quick technological evolution has nowadays let computer science reach people in both professional and private spheres. In a world where technologies are accessible to everyone, the gates to cyber-crimes are opened to any computer fan with some knowledge and skills. In this dangerous environment, individuals, companies, organisations, and institutions must protect themselves against criminals. The presence of a stable and reliable security system is now crucial in every technology ambience since crimes are increasing in number, types, and ease of execution. If protection of private data is essential for individuals, preservation of safety, integrity, and confidentiality of crucial data is mandatory for companies, which must deal with a big amount of data which could be critical or secret.

Since a secure system should prevent all the possible attacks, which are a lot, building a protection system is not simple; at the same time, building an attack to a known system is quite easier. Therefore, over the years, the problem of cyber-crimes is increasing considerably: in a study conducted by Accenture Security [1], for instance, it is possible to note that in 2019 the number of breaches in IT systems is increased as well as the annual cost of cybercrime, respectively by 11% and 12% in one year and 67% and 72% in five years (figures 1.1 and 1.2).

1.1.2 Cybersecurity pyramid

The security of a system can be reached building a shield which must cover all the different aspects of protection. Some of the main properties of a security system are:

- **Authentication of users**: making sure that the user is who or what it has said to be;
- **Authentication of data**: making sure that specific data have been created by a specific user;
- **Authorisation**: defining what a specific user could do and which resources it could access to;
- **Integrity**: detecting if some data has been illegitimately modified;
- **Confidentiality**: maintaining data secret;

Figure 1.1. The increase in security breaches (source: Accenture Security [1])

Figure 1.2. The increase in the annual cost of cybercrime (source: Accenture Security [1])

- **Non repudiation**: binding an action to who has performed it, even in front of a court of justice.

Considering all these features while designing a security system is superb, but not enough: the designer should prioritize properties based on their attack surface, the space of contact with potential attackers. The most exposed property, the one the whole system is based on, is *authentication*; solution architects could plan the best possible project using the most sophisticated techniques and technologies, but if the authentication phase is neglected or not secure enough, attackers could violate the security system easily.

1.2 The aim of this project

The aim of this project is to build a strong authentication system to deal with high critical contexts in which security is the principal issue to address. It is easy to find this kind of situations in the ambience of companies, from smallest to biggest: critical data of various type, from private data of employers to secrets of companies or customers, is often exchanged between high level workers.

The ideal contexts where to adopt functionalities offered by this project are offices of managers or workers with serious responsibilities such as CEOs. The project uses an electronic embedded system as the only gate between the office and the external world. The authentication system protagonist of this work considers this board as the Relying Party, the entity which wants to have the authentication proof. This card is logically put between the authentication server, the entity performing authentication, which can be located inside or outside the private network, and the worker's personal computer, the client side which, with the help of an additional security

hardware, can provide an authentication proof thereby letting the user to be authenticated. Once the authentication process has been successfully executed, the user can access to the RP and to its resources and services: now, the user can even authorise critical transactions.

1.2.1 Kinds of solutions

If the design of a security system can be bearable for IT companies, it could not be for all other kinds of businesses, in which there could be less than a few employees specialized in computer systems. Moreover, building the own internal security system can be very difficult even for an IT company, both for the huge amount of effort needed and for the risk of making errors: internal employees working at designing security systems may be negligent, malicious or may just not have enough knowledge in cybersecurity field.

On the other side, a business can use third parties' systems, which could reduce risks in errors and effort of internal workers; nevertheless, big companies, in which security should have a crucial role, could not give trust to others.

One of the aims of this project is to address this kind of issues offering a system which, even though developed by a third party, shows its whole functioning thereby letting anyone search for vulnerabilities and threats; moreover, potential companies can verify the security level of this project by their own, checking its validity and deciding more consciously.

1.2.2 Techniques and mechanisms to use

This project has the goal to build a system in which a Relying Party can offer its services and resources to a high level worker delegating the authentication task to an external server thereby having a strong authentication mechanism. Many mechanisms have been evaluated and FIDO2 has been chosen, as it is possible to see in the next chapter, for its strength, reliability, and advantages.

Once authenticated through the help of the authentication server, the user can gain the access to the service offered by the Relying Party, which is an embedded system. This thesis has been developed so that it could be possible to have security-safe functionalities inside a simple but secure board to be put in a worker office: just by adding this card, the security level of the whole office system would increase, allowing the worker to perform different types of actions, after the authentication phase, which can be different from office to office.

In practise, finally, this project wants to offer a user-friendly strong authentication mechanism for boards and embedded systems which can be used in high-risk places.

Chapter 2

State of the art

Thereby choosing the most adapted authentication algorithm to use in this project, the first need is to analyse and confront different techniques. In this section, some of them are listed, explained and compared.

2.1 The authentication property

NIST (National Institute of Standards and Technology), in "Glossary of Key Information Security Terms" [2], gives various definitions of authentication. The principal is:

> "Verifying the identity of a user, process, or device, often as a prerequisite to allowing access to resources in an information system".

This definition does not refer only to people: the protagonist of the authentication process, said "actor", can also be a device or a process, software or hardware. However, this concept can be extended to something even more general. According to "Internet Security Glossary" [3], the authentication does not only concern the identity of someone or something, but it is:

> "The process of verifying a claim that a system entity or system resource has a certain attribute value".

The common point between these two definitions is the verification step: the authentication process must execute a verification action to a proof provided by the actor.

2.1.1 Authentication factors

The authentication proof, provided by the actor, can be of different types, called factors:

Knowledge it is supposed that each user, process, or system knowing a specific secret can be authenticated as the actor; an instance of this factor is the usage of a password, a combination of digits only known by the actor.

Ownership the actor provides something it owns; it can be a software element, such as a token, or a hardware component, such as a security card.

Inherence the proof is something the actor is; the most famous example of this factor is the fingerprint.

Through one or more of these factors, an actor can produce the proof attesting it is what it says to be; nevertheless, each one of these authentication elements has weaknesses. The usage of a password is surely comfortable for users, but it is rarely secure, since it can be forgotten, stolen or even guessed. A criminal can also steal a security card, if the ownership factor is used; moreover, an hardware component can be easily lost or cloned. Finally, the usage of a fingerprint or other inherence factors can be highly insecure, since the authentication process must approximate the measure thereby working properly; furthermore, a criminal succeeding in simulating this factor could be very dangerous, since the user cannot easily change its inheritance element.

2.1.2 Authentication model

Usually, different entities are involved in an authentication system. The general schema is composed of an actor and a RP, Relying Party: the first is willing to prove its identity to build an authenticated session with the second, which can fulfil actor's requests only if authenticated.

First, the actor and the CSP, Credential Service Provider, execute an enrolment protocol; in this phase, the actor is called *applicant*. When successfully ended, CSP stores actor's attributes, crucial properties to verify its identity, and associate them to it. Sometimes, an authenticator, such as a X.509 certificate, is provided to the applicant thereby having a formal identity proof: in this case, the presence of a reliable CSP binds the authenticator to that specific user. Once the credentials are given to the actor, it can perform an authentication protocol with the Verifier: this entity communicates with the CSP asking for user's attributes; in this phase, the actor is named *claimant*. If the authentication phase is executed with a positive result, the claimant becomes *subscriber*, an authenticated actor, and the Verifier exchanges with RP authentication assertions which assure that the subscriber has specific properties.

2.2 User authentication

The user authentication is a model of authentication in which an actor, in possession of an identity ID and the associated secret S, wants to be authenticated on a server, knowing the user's ID and an associated value obtainable from the user's secret S using the function g. When the actor starts the authentication phase, it sends its ID to the server; as a consequence, the server asks for a proof. Then, the user sends a value, result of a function f which takes the secret S as input: once this proof is received by the server, it compares this value with the one it has stored previously as result of $g(S)$, sometimes after having performed some algorithms taking the proof as input. If the comparison succeeds, the proof is valid; otherwise, the proof is rejected (figure 2.1).

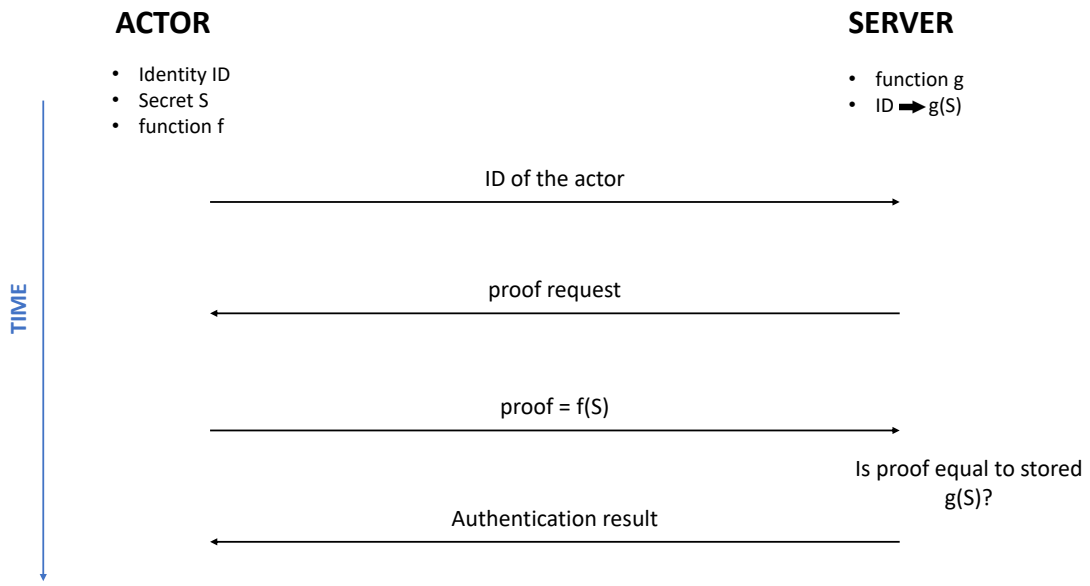


Figure 2.1. User authentication model

2.2.1 Password-based authentication

The most iconic way to perform the user authentication is the password-based authentication. In this technique, the secret S is a sequence of digits and it is sent by the actor to the server in clear; therefore, the function f used in the transmission phase is the identity function $I(x) = x$

On the server side, the secret is sometimes stored in clear, using the identity function I again as the store function g: once the proof arrives to the server, it compares the stored value with the received one (figure 2.2). Although it is simple and intuitive, this storage technique is insecure and offers vulnerabilities which can be exploited by criminals: a malicious hacker can access to the database and easily see the secret. To avoid this vulnerability, the server can store the information regarding the password as the result of a hash function which takes S as input, thereby allowing a stronger protection from malicious database accesses: even if this value is disclosed, it would be unfeasible to invert the hash function thereby extracting the secret. In this case, when the proof is received by the server, it first applies the hash function on the received value and then compares the result with the stored one (figure 2.3).

Even though the system is now more secure than before, the server can still be victim of hackers performing dictionary attacks, eventually using rainbow tables. The core of this kind of attacks is the pre-computation: the hacker performs the known hash function on foreseeable passwords thereby building a table that binds them to their associated digests. Once the table is built, the attacker can compare these digest values with the ones present in the attacked database: if it finds equal values, then the associated passwords are the actual used ones. The existence of this type of attacks makes the server vulnerable again: to avoid this threat, the server should store a salt, a sequence of random digits which is used to perform the hash function: the password is first concatenated to this salt and then put as input of the hash algorithm (figure 2.4). Now, the dictionary attack becomes unfeasible.

Although an issue is solved, the password-based authentication could have other weaknesses. First, the password is not only stored at the server side: the storage of the user should be secure and inaccessible by others; moreover, the transmission of the password in clear is highly insecure, since a hacker could sniff the channel of transmission accessing directly to the secret, or it can build a fake server and perform a MITM, Man In The Middle, attack. In case the password is simple, attackers can simply guess it and use it; finally, a password can be easily stolen with the usage of social engineering.

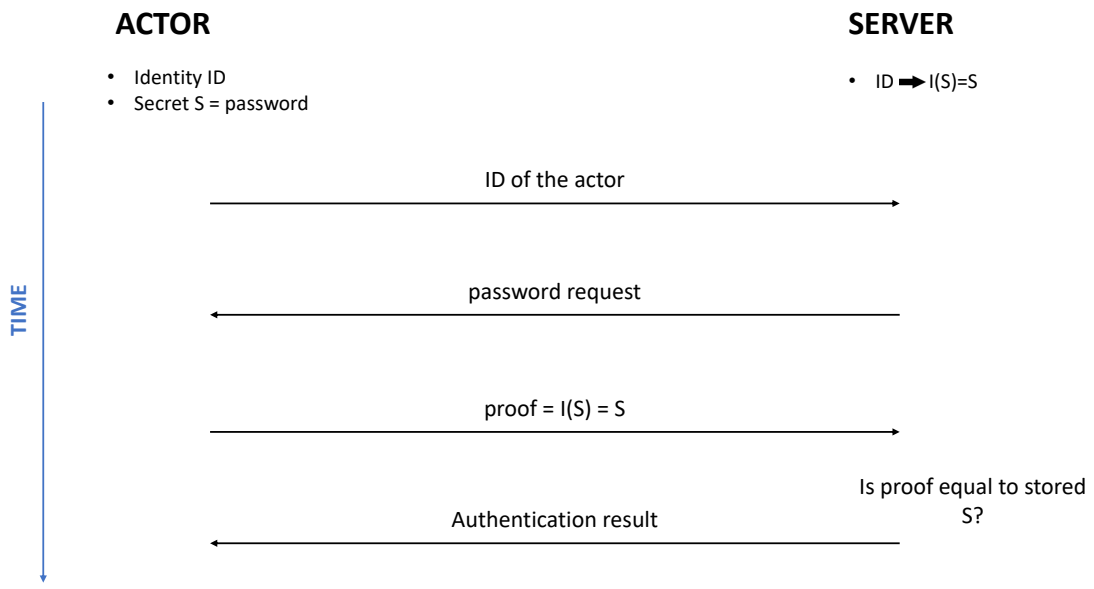


Figure 2.2. Password-based authentication model

2.3 Challenge-response authentication

The CRA, Challenge-Response Authentication, allows to limit attacks exploiting the transmission phase. In this system, the actor, in possession of a secret S, sends its identity ID to the server,

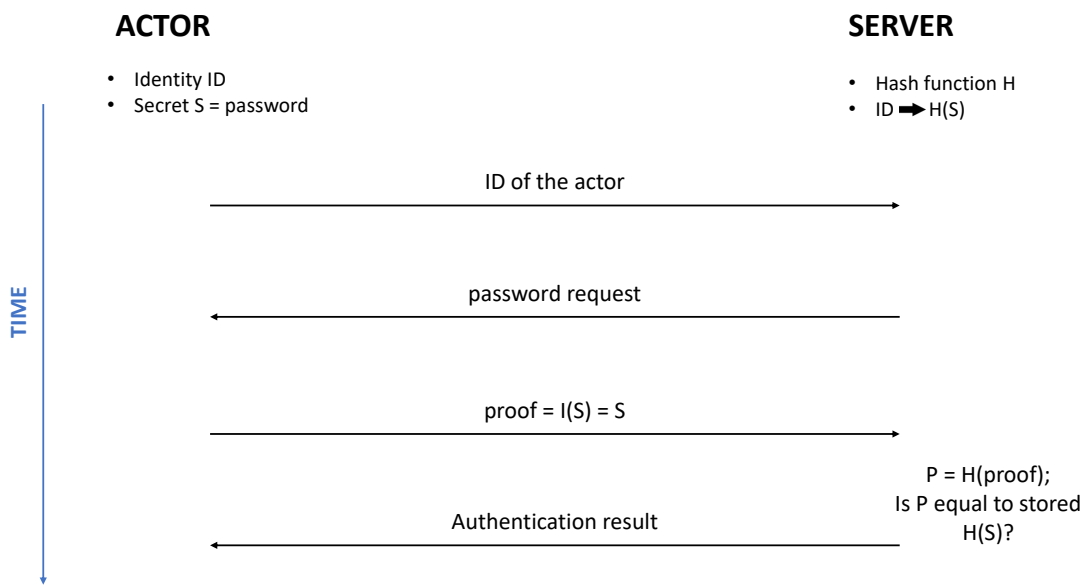


Figure 2.3. Password-based authentication model with the usage of a hash function on server side

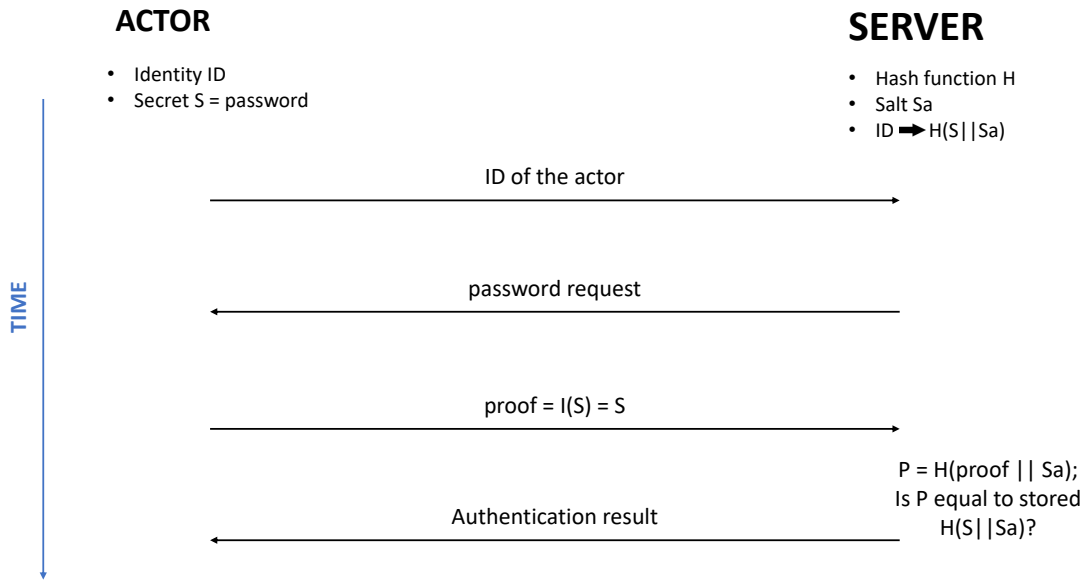


Figure 2.4. Password-based authentication model with the usage of a hash function and a salt value on server side

which knows a value K, secret or not, associated to that specific actor. The server now generates a challenge C, which must be non-repeatable to avoid replay attacks, stores it and sends it to the claimant; once this nonce (number used once) is received by the user, it computes a value R, result of a function f that takes as input the received challenge and the secret S; then, it sends R back to the server. Thereby resisting to sniff attacks, the algorithm f used here needs to be non-invertible; otherwise, an attacker sniffing the net and accessing to R and C can easily perform the inverse function obtaining the secret S. Afterwards, the server can perform another function g that takes as input the challenge C, previously stored, and the secret K associated to that actor;

the result P of this function is now compared to the received value R: the authentication ends successfully if they are equal; otherwise, the user is not accepted (figure 2.5).

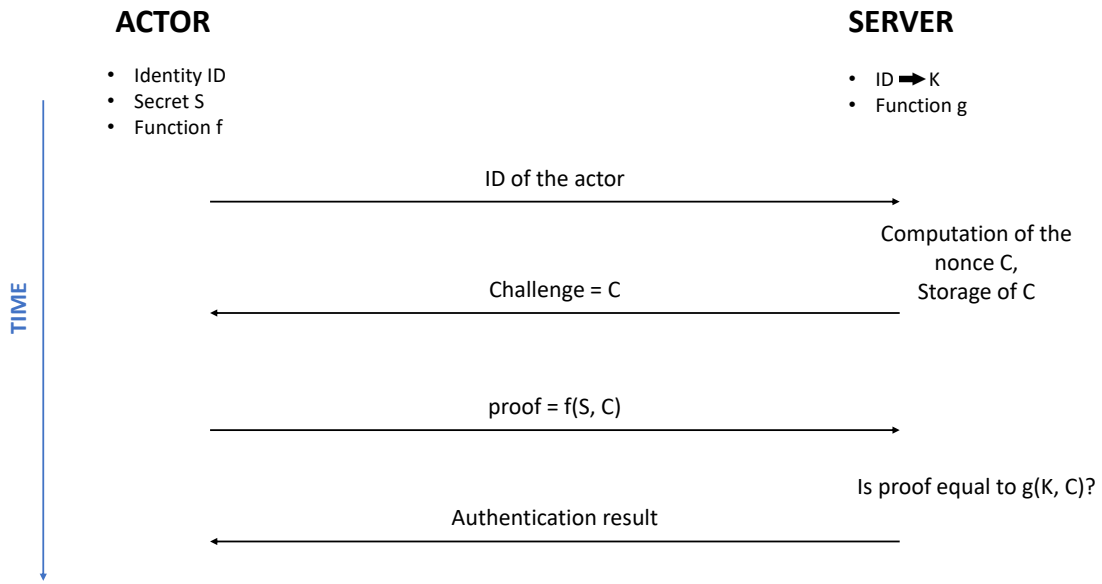


Figure 2.5. Challenge-response authentication model

2.3.1 Symmetric CRA

The first application of the challenge-response authentication model is the symmetric one. In this system, server and claimant know the same secret S: the actor performs a function f, for instance a hash function, taking as input S and the received challenge C; afterwards, it sends the result back to the server, which will perform the same function using the challenge previously sent to the client and the stored secret; finally, it will be able to compare the result with the received value (figure 2.6). This technique can be also used to implement mutual authentication, a system in which the client authenticates the server and vice versa: server and client share the same secret and, therefore, also the actor can verify that the interlocutor knows S. In this case, let's have two peers, A and B, willing to authenticate each other. A sends to B its ID and a challenge C_a; on response, B sends to A both the proof R_b, result of the execution of the function f using C_a and the secret S, and a new challenge C_b. Now, A computes its proof R_a using the same function and sends it to B. Finally, both actors can verify the received proof executing themselves the function again (figure 2.7).

Here, storing the secret S in clear on the server is mandatory for the system to work. However, this is a huge weakness: if an attacker obtains the access to server databases, then the secret could be stolen. SCRAM, Salted Challenge Response Authentication Mechanism, solves this vulnerability [4]: here, the server stores the digest of the client secret thereby being more resistant to hackers accessing to its database, as explained before.

Nevertheless, even symmetric SCRAM is not immune to risks, especially if using mutual authentication. The mutual authentication model, indeed, is vulnerable: an attacker could be authenticated as someone else, even without knowing the secret. This weakness is based on the fact that A and B are logically identical and share the same secret: the key of this attack is to make the victim to compute the requested proof on behalf of the attacker, which then will send it to the victim itself (figure 2.8). The hacker, first, starts a connection, sending a false ID and a challenge C_h; then, the victim will perform the function and send back the result as proof, with a challenge C_v. Now, the attacker could not compute its proof because it misses the secret S; nevertheless, it can induce the victim to compute it for it: the malicious user opens another different connection and sends to the interlocutor a false ID as before, but choosing as challenge

C_v, the same challenge previously received in the first connection. As answer, the victim will compute its proof using C_v and S as input and will send it back to the hacker, which now knows the result of the computation. It can now close this second connection, go back to the first one and send here the value just received. After this operation, the victim will be convinced to talk with the actor whose ID is the one received before.

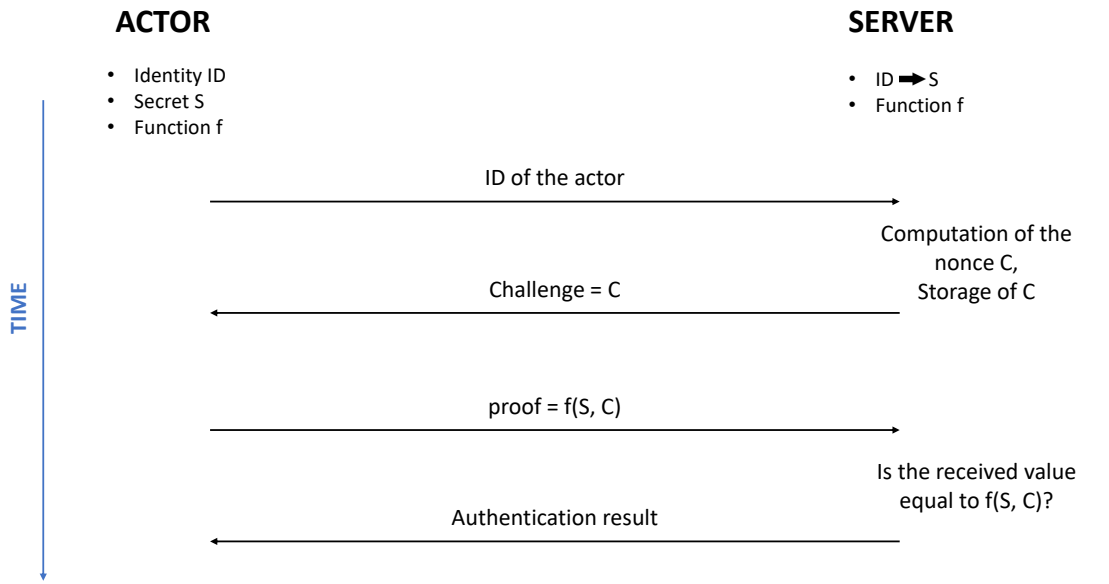


Figure 2.6. Symmetric challenge-response authentication model

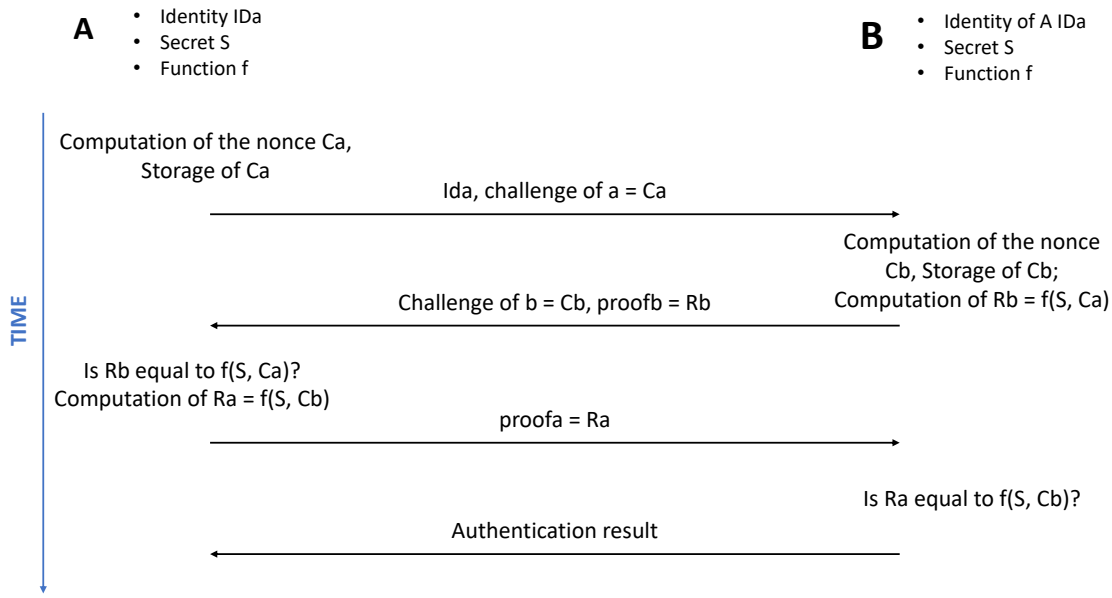


Figure 2.7. Mutual authentication between two peers using symmetric CRA

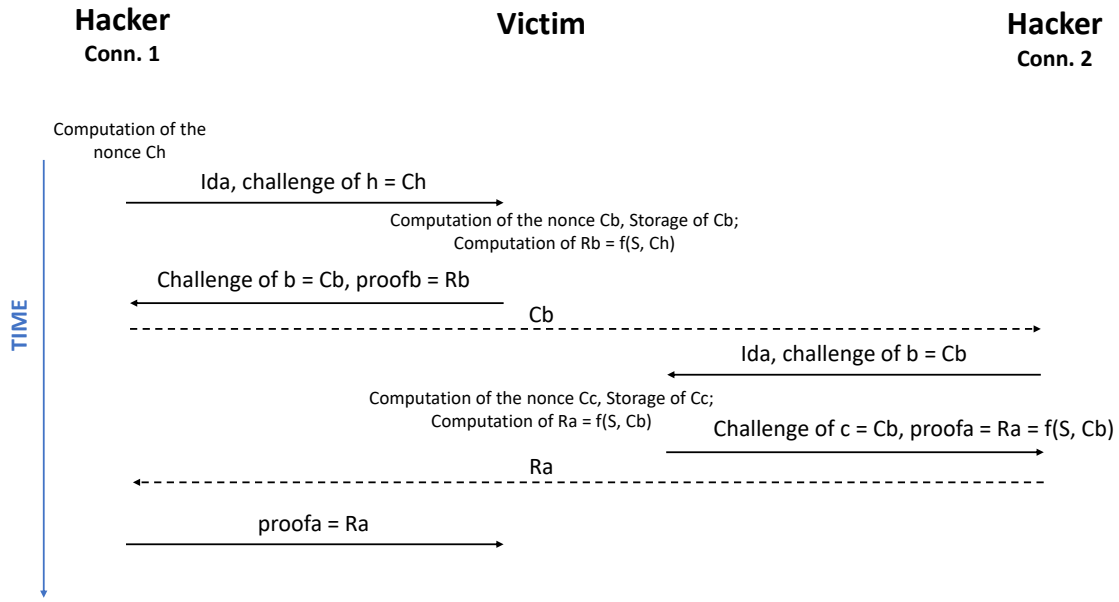


Figure 2.8. Attack against a mutual authentication architecture that uses a symmetric CRA model

2.3.2 Asymmetric CRA

The asymmetric challenge-response authentication mechanism employs the CRA technique using asymmetric algorithms: the actor has a private key K_{pr} and an associated public key K_{pu}, which can be shared with others as a certificate, usually with x.509 format; through certificates and PKI system, it is possible to bind identities and public keys.

The claimant sends to the server its certificate with the aim to prove the ownership of the private key associated to the public one inside the certificate it has sent. Thereby verifying the identity of the interlocutor, the server first checks whether the certificate received is valid downloading CRL, Certificate Revocation List, or using OCSP, Online Certificate Status Protocol. Once the verifier has checked the certificate validity, it computes a nonce N, stores it, and ciphers it using an asymmetric algorithm taking K_{pu} as input: the result will be sent to the actor as challenge C. The claimant can now decipher the arrived value using its private key K_{pr}, obtaining the result R: this value can only be computed by the actor whose id is the one put inside the certificate, as it is the only entity possessing K_{pr}. Afterwards, the claimant sends back R to the server, which, comparing this value with the nonce N previously stored, can verify if the actor is who or what it has said to be (figure 2.9).

This system has several advantages. First, nothing except the user ID is stored on server side, which makes the verifier immune to some kinds of attacks. Moreover, the transmission phase is resistant to sniff attacks, since challenges and responses are results of cipher algorithms which take as input nonces, so numbers that cannot be repeated.

The main weakness of this system is that, since it uses asymmetric algorithms, it is quite slower than other mechanisms. Furthermore, the claimant should pay attention on the format of the received challenge: in some cases, if the challenge is similar to a file instead of being composed of random digits, the actor, deciphering the file with its private key, following the process, could involuntary sign the file. Finally, the usage of a X.509 certificate brings all the vulnerabilities related to PKI, Public Key Infrastructure.

2.4 One-time password

OTP, or One-Time Password, is an authentication mechanism in which the authentication proof is a password, a sequence of digits, which is used only once and never again. This system must have an initial phase in which passwords are precomputed and given to the actor, which stores

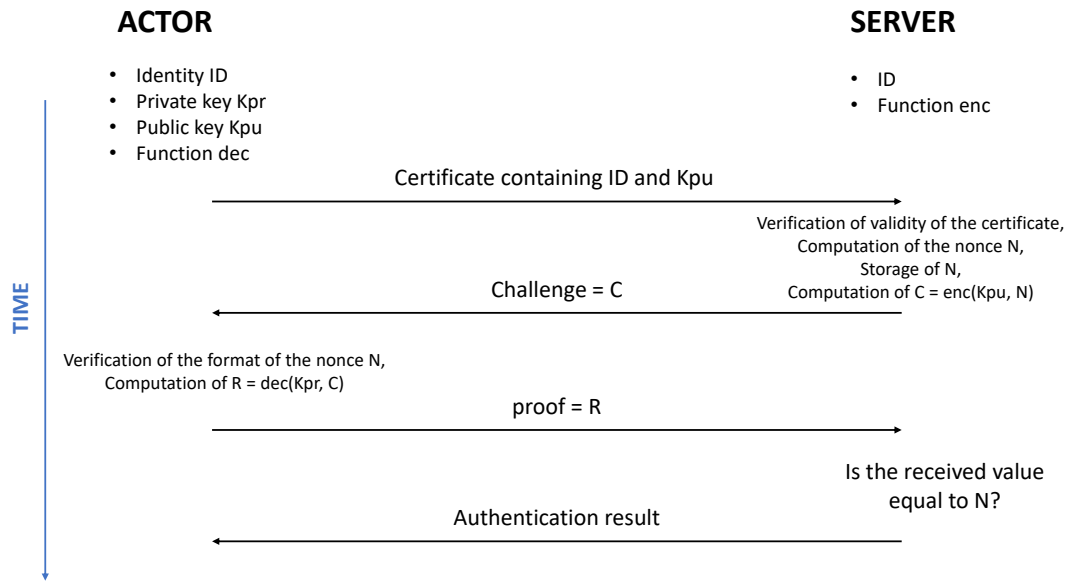


Figure 2.9. Asymmetric challenge-response authentication model

them in order. When the claimant ID is sent to the server, it sends back the request of a specific password in the list which was not requested yet. Then, the user can look at its passwords list, select the correct password and give it to the server, with the awareness that the specific password will never be used anymore. To work, these passwords cannot be random: there exists a root S which is used to perform a function f that takes S and the index of the specific password as input and gives as result the value of that password; this root S is the actual secret of this system and must be stored at server side: once the password requested, sent in clear, is received, it will be compared with the result of the function explained before (figure 2.10).

The more relevant characteristic of this method of authentication is the precomputation. When a root is chosen, a number n of passwords is computed; each time the actor wants to be authenticated, a password of these is used and then discarded. After n authentications, the number of remaining passwords will be equal to zero and another secret will be chosen and used to compute another passwords list.

Even though passwords are sent in clear, this technique is immune to sniffing, since an attacker accessing to one password cannot use it in any way. Nevertheless, a hacker can impersonate the server requesting the next password in the list and then using it with an actual connection with the real server, performing a MITM attack. For this reason, the authentication of the server is mandatory in this kind of systems. Moreover, another disadvantage of this method is the difficulty of the secret storage of the entire list of passwords in client side: they could be a lot and very difficult to remember, then it could be difficult to build OTP if the client has untrusted or obsolete devices without specific cryptographic hardware components that store the list in their behaviour.

2.4.1 HMAC-based OTP

HMAC-based one-time password authentication system, HOTP, is an application of the authentication mechanism explained as OTP [5]. In this method, the passwords list is not precomputed since all the passwords used are calculated in run-time as

$$P_C = Trunc(HMAC_H(S, C))$$

where

Trunc is a function truncating the result of the HMAC function,

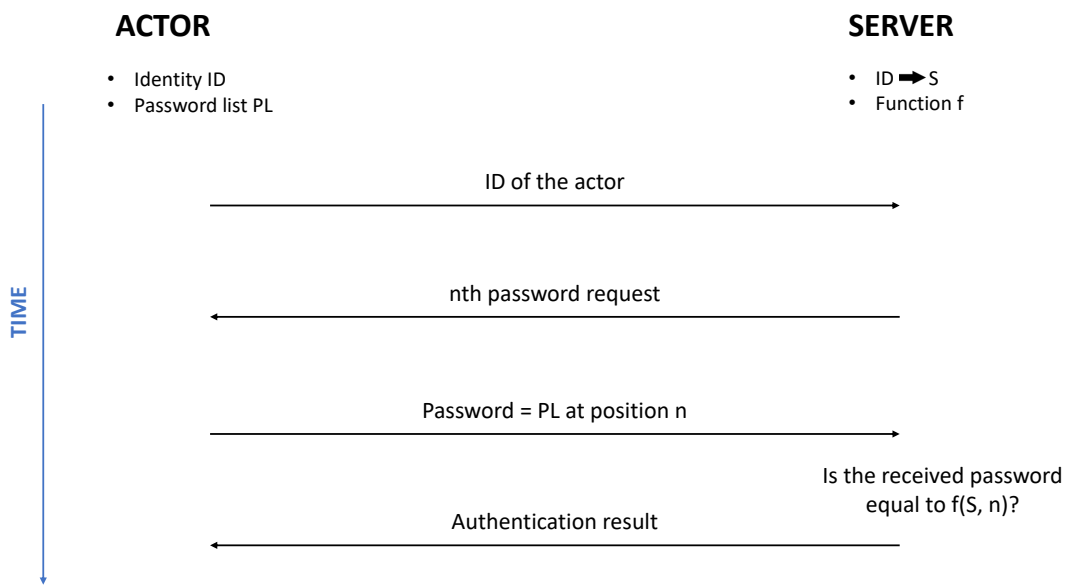


Figure 2.10. One-time password model

H is the hash function used to compute HMAC,

S is the secret shared between client and server,

C is a counter that starts from zero and is incremented each time the authentication succeeds.

Since server and client know the same secret S, it is possible to perform a mutual authentication using HOTP (figure 2.11).

One of the most common weaknesses of this authentication system is the desynchronization: since HMAC is computed on a counter, if the server and the client have different values of C, then the authentication process fails regardless of whether they share the same secret S. This method has, then, to offer a strategy to adopt if this happens.

Moreover, MITM is a possible attack that a hacker can perform. The attacker, impersonating the server, can send a malicious request of OTP obtaining the actual OTP value, which can then be used to have an authenticated session with the real server. This vulnerability is not patched even in the mutual authentication case.

2.4.2 Time-based OTP

The Time-based one-time password authentication system is a mechanism of authentication that uses the OTP model modifying it in such a way to use the actual time t and a shared secret S to compute, through a function f, the ephemeral password, valid only for brief slots of time [6]. TOTP is in practice a special case of HMAC-based OTP in which, rather than using a counter which can be problematic for synchronization problems, it uses the value

$$C_T = (T - T_0)/T_S$$

where

To is the *Unix epoch*, a fixed and standardised time equal to 1/1/1970;

T is the current time, expressed in the format of seconds passed since the *Unix epoch*;

Ts is the slot of time, a fixed interval of time expressed in seconds.

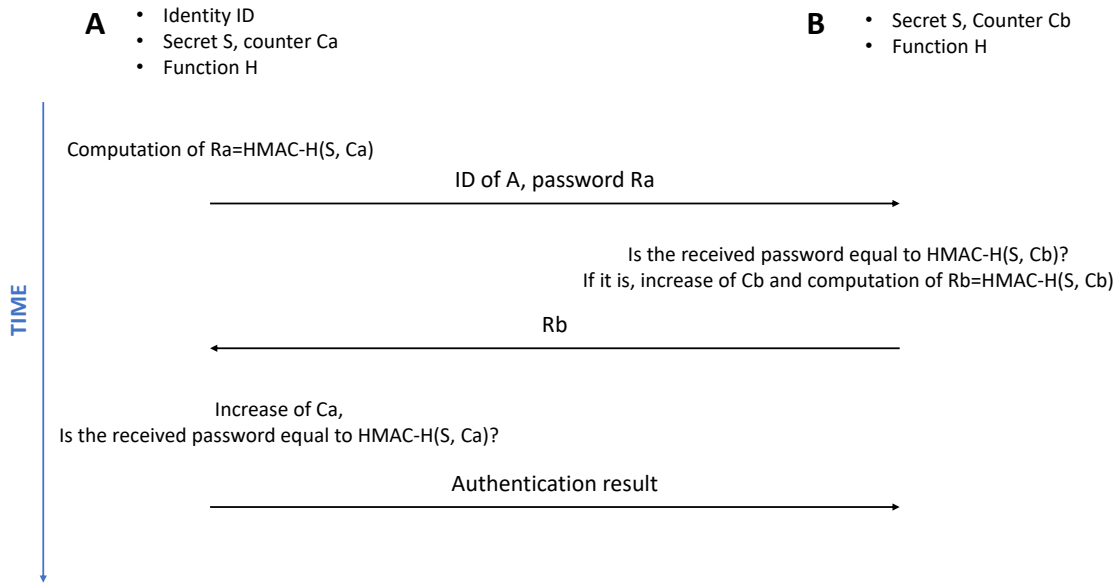


Figure 2.11. Mutual authentication using HMAC-based one-time password model

TOTP computes every password locally in run time and, therefore, it requires computational power in the actor's device; moreover, since password computation is based on the time, it is mandatory to have clock synchronization between client and server. Even in this case, however, a packet can be sent at a time close to the end of a slot and arrive at the beginning of the next slot, inducing a forced desynchronization. Thereby avoiding this issue, the verifier must compare the received password P to the ones computed in the actual slot $f(ID_u, T)$, in the previous one $f(ID_u, T-1)$ and in the next one $f(ID_u, T+1)$.

2.5 Fast Identity Online

FIDO, or Fast Identity Online, is one of the most sophisticated authentication mechanisms used nowadays. FIDO allows users to be registered to a service and to be authenticated gaining access to it; the actor can also authorise transactions.

Fast Identity Online was created to address common weaknesses associated to other authentication techniques. The usage of a password is dangerous, for instance, due to the great number of online accounts a single user owns: this leads actors to reuse their passwords many times and, when one is compromised for one service, also the others will be damaged. As a result, the percentage of data breaches caused by passwords, for this and other weaknesses, is over 80% [7].

Thereby addressing this vulnerability, FIDO could not use passwords to perform the authentication phase, as it can be *passwordless*.

2.5.1 Authenticators and multi-factor authentication

Fast Identity Online, to perform authentication, registration and authorisation phases, uses authenticators, objects giving some sort of evidence to the server: they can be software, such as the ones installed in many smartphone or computer operating systems, or hardware, like security keys and devices. Thereby gaining the access to these authenticators, it is possible to use different methods: the user can have a knowledge factor, like a password, or can configure an inherence factor, such as facial or voice recognition or fingerprint. This kind of factors are in general dangerous to use, since a password is guessable or easy to steal, while inherence factors are not precise and, if compromised, cannot be changed. Nevertheless, in FIDO these factors are used locally just to gain access to authenticators: actual protocols are performed with asymmetric techniques, which

are more secure. For this reason, the usage of this type of authentication elements is not bad and can contribute to build a mechanism easy to employ for final users, who are only supposed, in the worst case, to remember a simple password.

The usage of these authenticators makes possible the increasing of security performances of FIDO mechanism due to Multi-Factor Authentication, or MFA. MFA is an authentication technique in which more factors of different types are used at the same time to perform the same authentication process. Through the usage of multi-factor authentication, it is possible to implement a so said "strong" authentication technique. To properly work, all the factors chosen need to be mutually independent: the validity of one must be independent of the validity of the others; in this way, even though one element is compromised, the validity of the others, remaining undamaged, would not allow attackers to be authenticated and perform malicious actions. Moreover, thereby increasing the effectiveness of the security technique, it is mandatory to have at least one of the factors to be non-reusable and non-replicable.

FIDO can use two factors to perform authentication. U2F, Universal 2nd Factor, can be implemented due to the usage of authenticators: the user must first possess the authenticator and, in addition, must have the possession of the right factor used to gain access to it. In other words, let the user have a hardware key as authenticator and a password as access factor. In this case, an attacker can act in two ways.

- The attacker is a hacker and steals the password from the user: the criminal cannot be authenticated since it does not have the secret key in its hands;
- the attacker is a thief and steals the security key from the user: the criminal cannot be authenticated because it does not know the password.

As it is possible to ascertain, the only way the criminal has to properly attack its victim is to steal both the authenticator and the other factor used, which is very unlikely.

2.5.2 Asymmetric techniques

Authenticators are used to perform asymmetric functions which are mandatory to make the authentication process with the server. At registration phase, a new asymmetric key pair is computed locally and stored in the authenticator; then, the public key is shared with the server, which can use it later to perform authentication and authorisation phases. The usage of this technique makes possible the server to not store any secret in any way, which makes this mechanism immune to some kinds of attacks that have as victim the server database, such as the dictionary attack.

Every user can have more than one account and each of these can have more than one key pair. Moreover, every key pair is associated to one specific RP that uses FIDO mechanism and cannot be used for other RP or services. This makes the security stronger and the user experience easier, since it is not bound to one single key pair.

2.5.3 Protocols

FIDO mechanism allows users to be registered, authenticated, and let them to authorise and confirm transactions. Thereby offering all these features, Fast Identity Online uses protocols which have as protagonists:

- final user;
- user's device, divided in
  - FIDO authenticator, the authenticator which manages user's asymmetric keys;
  - FIDO Client, the client side of the FIDO mechanism;
  - User Agent, the client side of the web application which can be an application or a program, often run in a browser;
- relying party, divided in
  - FIDO server, the server side of the FIDO mechanism, which can also be located outside the RP;

– web application, the server side of the web application, the actual RP that manages the service or the resource the user wants to access.

Registration Protocol

To be registered to a relying party, it is mandatory to perform the registration protocol graphically described in figure 2.12. First, the user agent, on behalf of the user, sends the registration request to the web application, which will forward it to the FIDO server (step 1). This request only contains user's information such as first name, last name, username, display name. The server now generates a registration request and sends it to the client with a specific policy: this is what FIDO server accepts as valid in terms of, for instance, algorithm and key types used (step 2). Then, FIDO client, receiving this request due to the user agent, starts a protocol with the user through the authenticator; at the end of this protocol, the user will be enrolled and a new key pair, associated to the user and to the RP, will be generated and stored in the authenticator (step 3). Once this protocol is terminated, the registration attestation is computed and sent to the server with the registration response, while the public key calculated in the preceding step is shared (step 4). Finally, the FIDO server has now all the data needed to verify the response and the attestation received and, in case of success, it stores the public key (step 5).

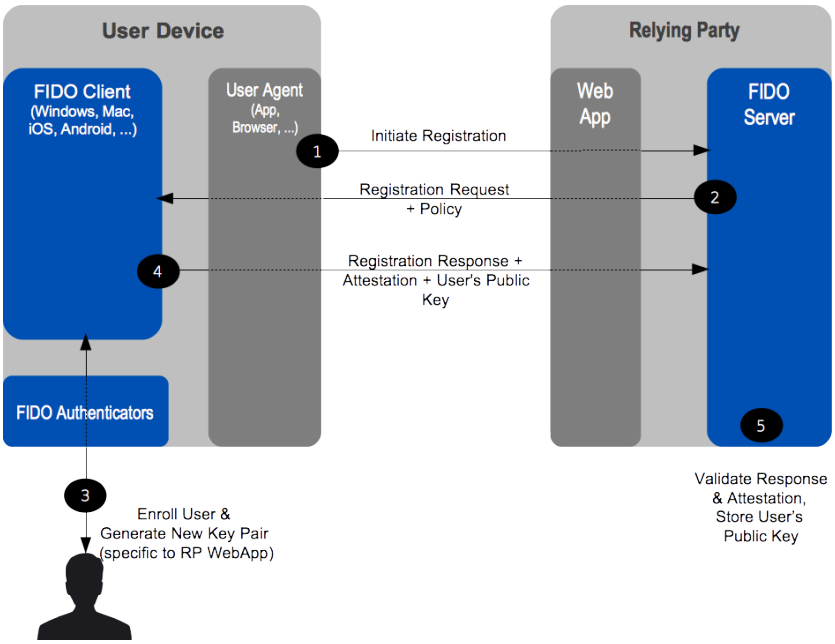


Figure 2.12. Graphical description of FIDO registration protocol (source: FIDO Alliance [7])

Authentication Protocol

Once the user is registered to the relying party, it can be authenticated thereby performing login and accessing to resources and services offered by RP (figure 2.13). In step 1, the authentication request is generated by the user agent using few user data like just username and user identifier. In the following step, the second, a request is created and sent to the client with a defined policy; in addition, a challenge is computed and sent to the user agent: this is some non-repeatable value which is temporally stored in the server and it is needed to verify the identity of the user. Now, the user offers an authentication proof to the authenticator thereby unlocking its secret key associated to the RP; after the computation of the authentication response, the private key is used to calculate the signature of the response to be sent to the FIDO server (step 3-4), which can now verify the signature received using the public key associated to the user that it has previously stored, in the registration phase, and the challenge stored just before, in step 2 (step 5).

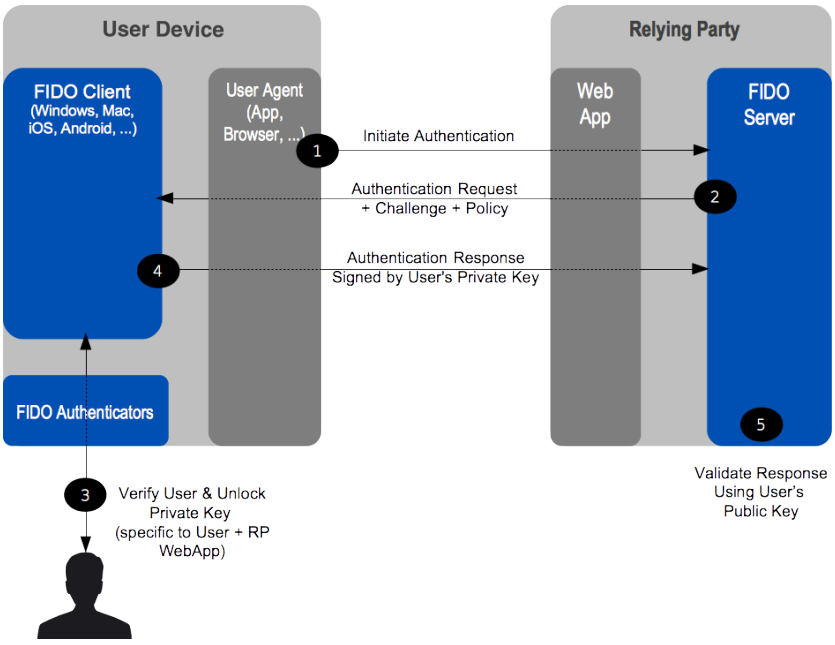


Figure 2.13. Graphical description of FIDO authentication protocol (source: FIDO Alliance [7])

Authorisation Protocol

As written above, FIDO allows users to confirm transactions in a way that permits the non-repudiation property. The authorisation protocol, described in figure 2.14, starts with a transaction confirmation request, which contains information about the user and about the transaction to be confirmed (step 1). As in the authentication protocol, the second step wants the server to send an authentication request; in addition, a transaction text, a description of the transaction to be confirmed, is sent in order to be visualized by the user agent to the user: once the user has checked the validity of the transaction text, it can confirm it after the authentication protocol with the authenticator. Now, the user can use its unlocked private key thereby signing the hash value of the transaction text viewed (step 3). This signature is the confirm that only that specific user can have allowed the transaction and, thanks to this, it is possible to have the non-repudiation property for this transaction, which means that the user cannot deny having confirmed the transaction in front of a court of justice. So, this signature is sent with the authentication response to the server (step 4), which can now validate both response and signature (step 5).

2.5.4 User view

So far, it is described how FIDO protocols work and what happens beyond user's eyes. Fast Identity Online, however, is not only efficacious, but it is also a great user-friendly mechanism.

Registration phase

Let the user have as authenticator its smartphone (figure 2.15). If user wants to access to a relying party which uses FIDO mechanism, it has first to register a new key pair associated to the specific RP. To perform the registration phase, the user has to contact the relying party and approve the registration using a password or, even better, a biometric factor such as fingerprint. Once its smartphone has authenticated it, the system generates the key pair, stores the private key in user's smartphone and the public key in the FIDO server database.

Authentication phase

As in the registration one, the authentication phase is very simple for the user (figure 2.16). Once it wants to perform the login to the relying party in which it is registered, the user contacts

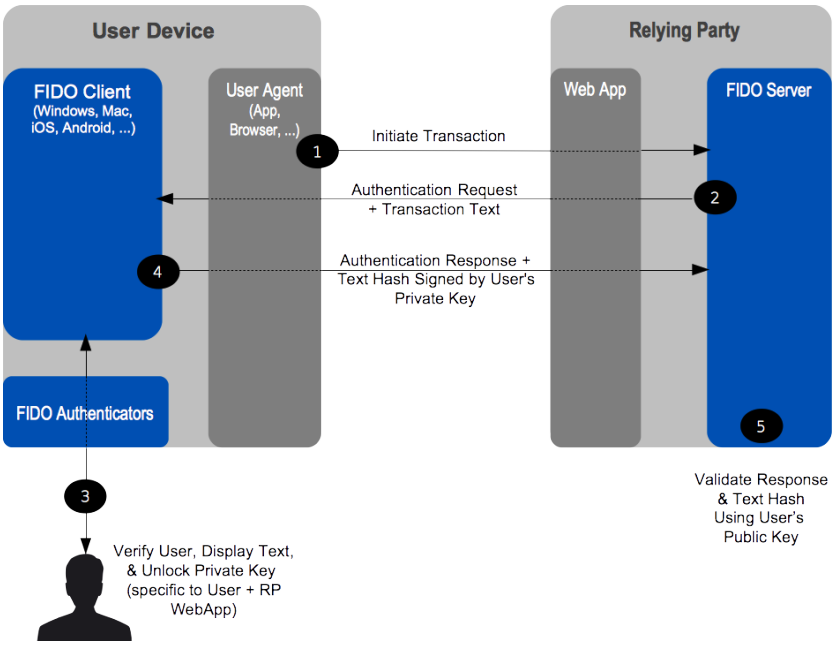


Figure 2.14. Graphical description of FIDO authorisation protocol (source: FIDO Alliance [7])

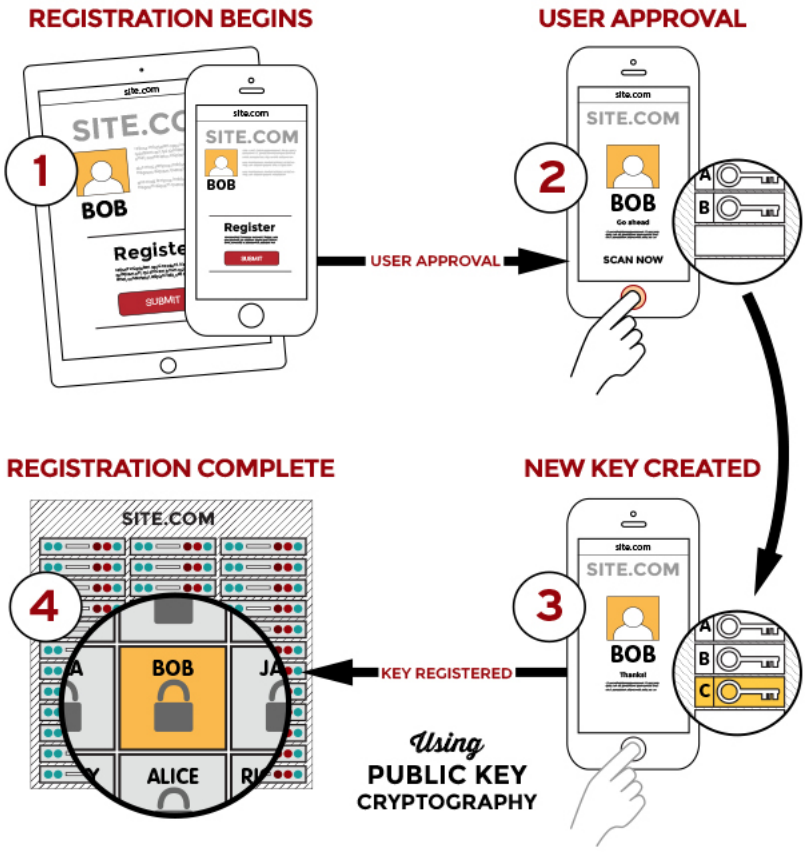


Figure 2.15. Registration phase from the user point of view (source: FIDO Alliance [7])

the RP and performs the login challenge simply approving it accessing to its authenticator with the fingerprint or the password used in registration phase. Once the approval phase is complete,

its smartphone chooses the right private key, the one associated to that specific RP, and use it thereby performing asymmetric functions to continue the authentication; if user wants to authorise a transaction, the steps are the same.

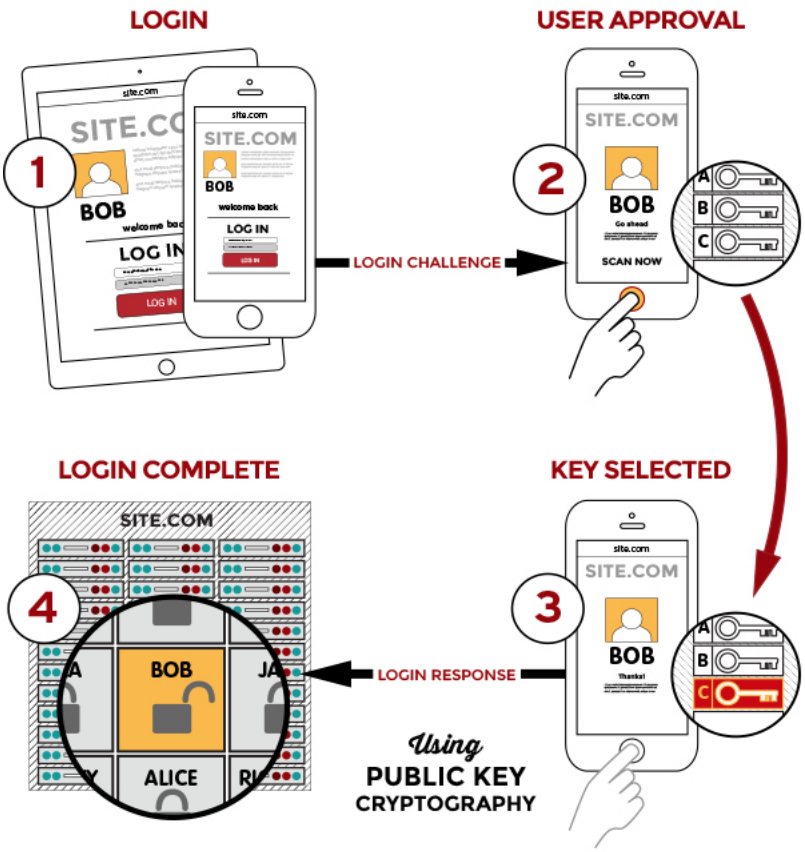


Figure 2.16. Authentication phase from the user point of view (source: FIDO Alliance [7])

User experience

For both the registration and authentication phases, such as for the authorisation phase, the user experience is put as priority: from its point of view, the entire FIDO mechanism is based on simple passwords or, even more user-friendly, fingerprint, which makes this technique very simple to use once implemented. In Fast Identity Online there is a complete separation between the actual secure mechanism, which uses strong asymmetric algorithms and multi-factor authentication, and the user perception of something very simple, usually related to weak authentication techniques.

2.5.5 FIDO2

The second version of FIDO, FIDO2, is the standardised one. It uses FIDO protocols and has the same characteristics of the first version, but with some additions. First, the specifications of web authentication had been standardised in W3C, World Wide Web Consortium's, WebAuthn specification. WebAuthn defines a specific set of web APIs needed for FIDO2: it is supported by most of web browsers and operating systems, as figure 2.17 shows.

Secondly, FIDO2 supports Client To Authenticator Protocol, or CTAP, a protocol needed to let the user to use not only authenticators embedded in the system used, such as the smartphone or personal computer, but also external hardware security keys. As FIDO Alliance specifies [7],

> "CTAP is complementary to the W3C's Web Authentication (WebAuthn) specification; together, they are known as FIDO2".

In general, then, a FIDO system sees a client device, which is running the web application client side in a browser, and the server, which is composed on the relying party application server and an internal or external FIDO server (figure 2.18). FIDO protocols are executed between FIDO client and FIDO server, through the RP application. Inside the user's device, the client side of the RP application calls W3C WebAuthn APIs using the browser, if it supports them. These APIs want to communicate with an authenticator and the user can choose between platform, embedded authenticators, such as Windows Hello, or external authenticators through CTAP, such as USB or NFC secret keys.

Figure 2.17. View of technologies which support WebAuthn (source: FIDO Alliance [7])

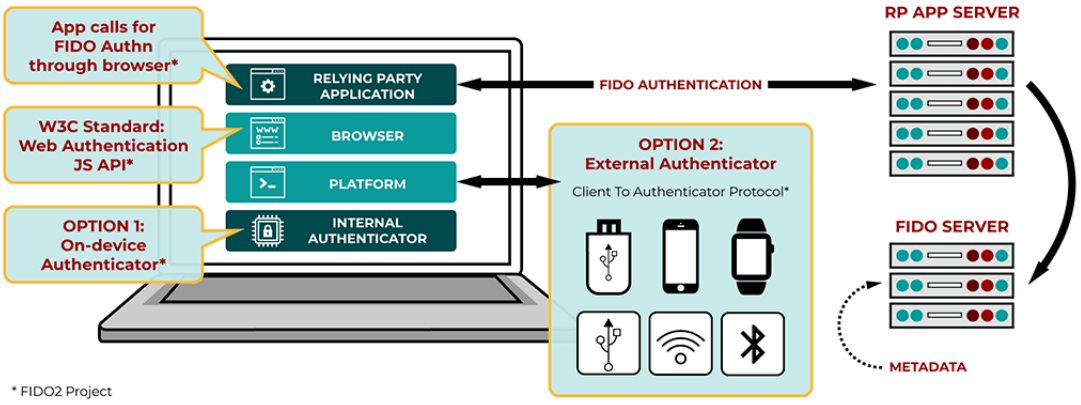


Figure 2.18. Architecture of FIDO and FIDO2 mechanism (source: FIDO Alliance [7])

2.5.6 Conclusions

FIDO2 is one of the most secure and sophisticated authentication mechanisms nowadays available. Since it uses protocols which employ asymmetric algorithms and non-repeatable challenges, Fast Identity Online is resistant to sniffing. FIDO2 supports external security keys through CTAP, it supports password-less authentication though UAF (figure 2.19), or Universal Authentication Framework, and it supports second-factor and multi-factor user experience through U2F (figure 2.20). FIDO2 has many security features and supports non-repudiation of transactions confirmation and, despite its security performances, it is very user-friendly: users can utilise FIDO protocols just with fingerprint or very simple passwords without making the mechanism weak, as they are used only locally to unlock their private keys. Moreover, Fast Identity Online is versatile, as the same user can have multiple accounts, each having multiple key pairs. FIDO does not use third party entities in protocols and the server does not store user's secrets and this, as said above, increases the security of the entire mechanism.

For all these reasons and pros, the usage of FIDO2 authentication technique is widely recommended in a critical system such as the one treated in this thesis.

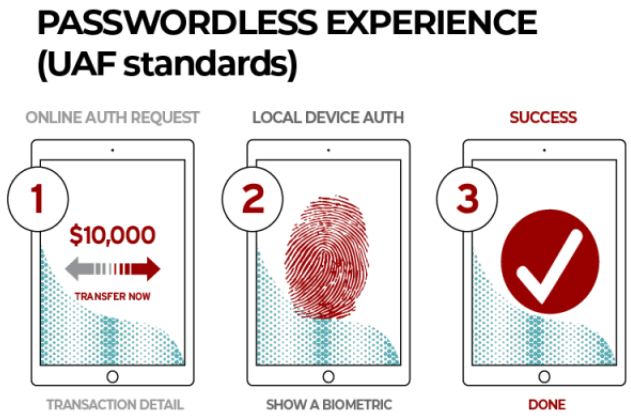


Figure 2.19. UAF standards scheme (source: FIDO Alliance [7])

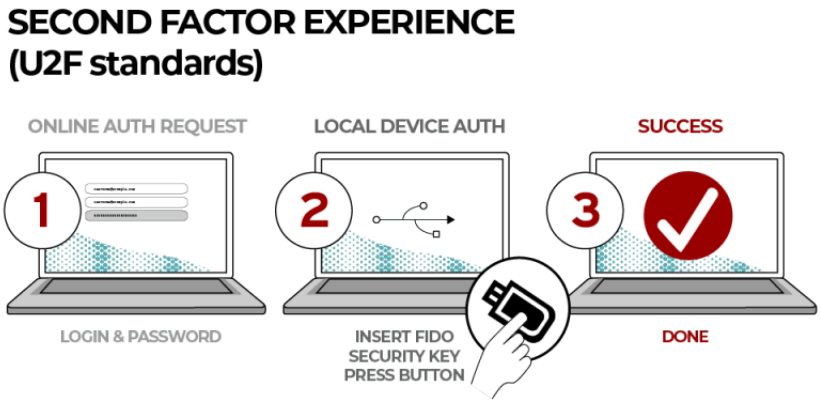


Figure 2.20. U2F standards scheme (source: FIDO Alliance [7])

Chapter 3

Project design

The aim of this project is to implement the FIDO2 authentication mechanism to be used in an embedded system, a system having a weak computational power and few resources. In this chapter, the system design and implementation are shown.

3.1 Network design

As specified before, the FIDO2 technique needs different components:

- a FIDO2 server,
- the web application server,
- the web application client.

Because of the small power and resources of the embedded system, it has been chosen to run the FIDO2 server externally; it stays, using a static IP address, in the same network of the embedded system, which is using another static IP address and running a service capable to offer the web application. Therefore, this board has the logical role of the Relying Party, something owning the service or resource the final client wants to access. Thereby executing the whole authentication mechanism, the client device must be put in communication with the embedded system; for the final client, the usage of an external FIDO2 server is transparent. The figure 3.1 shows this context.

Since the presence of an external FIDO2 server is not seen by the client device, it is possible to use only one network in which all three nodes are present. In this type of design, security is not compromised because FIDO2 server requires an authentication proof by FIDO2 client to allow requests: even if an attacker stands within the net, it cannot access to FIDO2 server (figure 3.2). Thereby making this project simpler, it has been chosen this kind of solution: the network is private and it is the same for all nodes.

For more complex situations, it is also possible to implement more than one FIDO2 server; in this case, all these components need to be synchronised. This type of solution needs the usage of a proxy which guides requests to one of the available servers. In this way, first, the downfall of one server does not interrupt the service; moreover, if many clients are connected at the same time, the proxy can have the role of load balancer, sending, for instance, half of the requests to one server and the other half to the other. Some proxies, furthermore, support also different authentication mechanisms to be used to add more security. Nevertheless, one of the principles of security is the simplicity of system design: a more complex system can show more vulnerabilities to an attacker. Figures 3.3 and 3.4 graphically describes this solution.

3.2 Implementation choices

The implementation choices have been made with awareness of working with an embedded system. In particular, the web application has been built using only PHP and JavaScript, with no framework. Since the final client willing to be authenticated will be a user with a device with

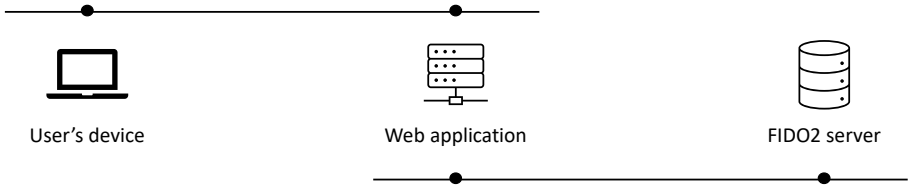


Figure 3.1. Design of a possible project network

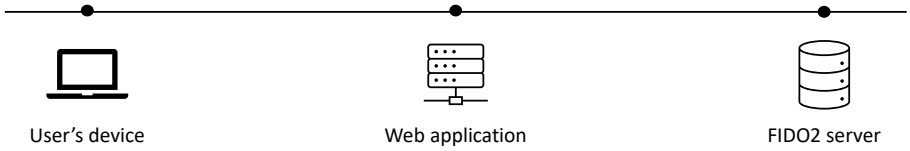


Figure 3.2. Design of the network chosen for this project

much more resources, such as a PC or a smartphone, it will not be a problem to let it execute instructions requiring a big amount of computational power. Then, as the usage of FIDO2 requires, the client side of the web application uses WebAuthn APIs thereby running cryptographic functions, like generating keys and signing challenges.

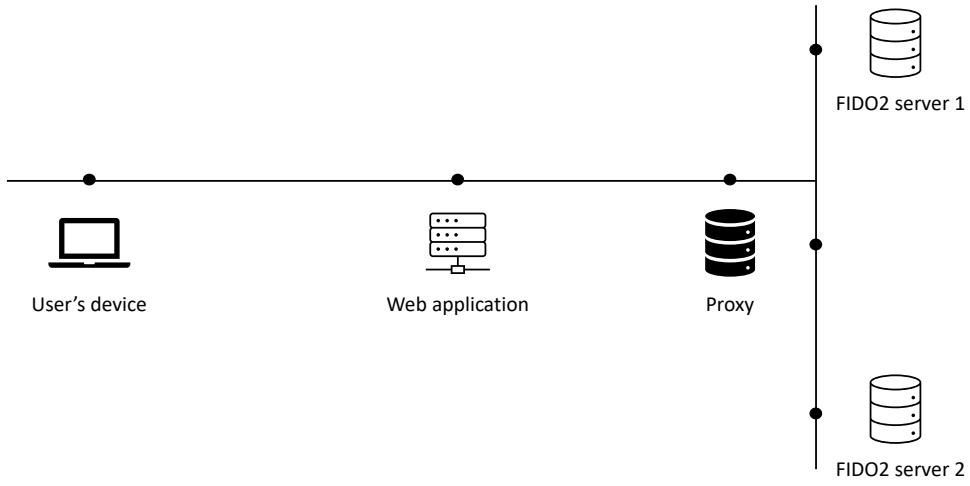


Figure 3.3. Design of a possible project network in which two FIDO2 servers are managed by one proxy. In this case, one network is used for all the components

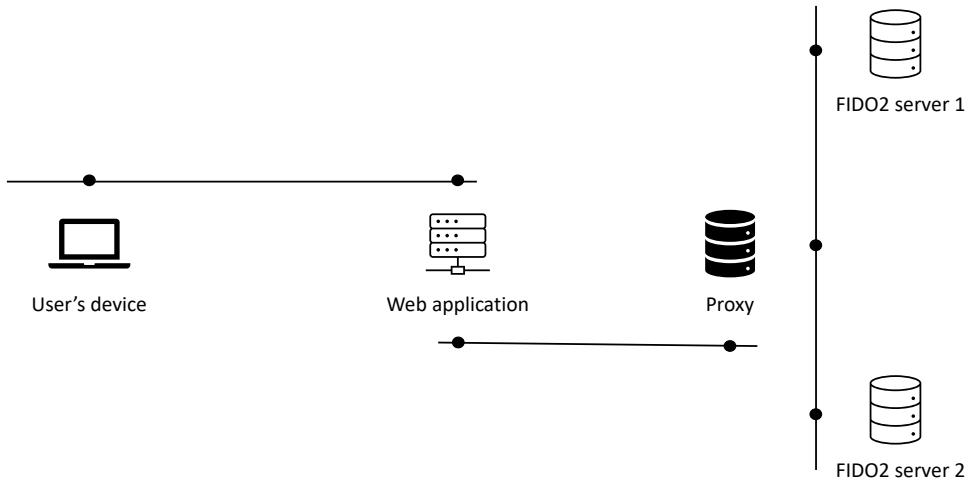


Figure 3.4. Design of a possible project network in which two FIDO2 servers are managed by one proxy. In this case, different networks are used

3.3 FIDO2 server

FIDO2 server runs in another device; for this project, it has been chosen to execute it in a virtual machine running in a personal computer with Windows11, eight gigabytes of RAM and a i5 10400F as processor. This server needs a big and stable amount of power since it has the role to offer different endpoints, manage many RPs at the same time, and execute cryptographic functions, such as verifying the challenges signed and sent by clients.

The virtual machine is equipped with four gigabytes of RAM and forty gigabytes of disk; it

can use one processor and a network adapter which is set in such a way to access to the external physical network. This virtual system runs CentOS 7.

There are many different solutions to implement a FIDO2 server: among the others, the one offered by StrongKey [8] has been chosen.

3.3.1 StrongKey

Strongkey is a company which offers security solutions. Besides other products, they developed SKFS, or StrongKey Fido Server, which is an open source FIDO2 server being constantly updated. This solution is the one chosen for this project because of the importance StrongKey has and the assurance it gives.

To install SKFS, official instructions have been followed [8]. In this project, this server runs in port 8181 and it has been chosen to let it stand directly inside the network: for this reason, it has been mandatory to open that port.

It is possible to implement more than one SKFS using a proxy as guide for requests. This solution has been evaluated for this project using one single SKFS and a proxy on the same machine, with the port 8181 closed and the 443 one opened and used by the proxy; as recommended by Strongkey, it has been chosen to use HAProxy Load Balancer [9]. In this case, after having installed the proxy software, client authentication has been added using TLS and installing the RP certificate.

StrongKey Fido Server implements an authentication system which can be based on username and password or HMAC. Thereby using a new RP, it is mandatory to register it and assign to it all the privileges it needs. Within the SKFS, RP credentials are named SCID, or Service Credentials Identifiers; in order to manage SCIDs, StrongKey has developed a tool: "manageSKFSCred.sh". Due to it, it is possible first to deal with the initial admin user, which has the same username and password for all the instances of the server when initially installed. This user is dangerous to keep unchanged, since it has every possible privilege; to deal with this SCID, three options are available:

- delete it,
- change its password,
- change the groups it joins, so that to limit its privileges.

For this project, the second option has been chosen.

After having patched this user, it is needed to create a SCID for the RP to be installed. Afterwards, SKFS requires that every API called by the web application carries also the credentials obtained after the SCID registration, so that none can access to these endpoints but the registered RPs.

RP privileges

After SKFS executes the authentication of a RP, it must know what kind of endpoints this web application is allowed to call. To enable and implement access control, SKFS makes use of "roles". Each role is a sort of group in which a RP can be enrolled; if a web application joins a role, then it acquires the permission to call all the APIs associated to that specific group. Roles are

FidoRegistrationService-AuthorizedServiceCredentials this role enables access to the preregister and register APIs.

FidoAuthenticationService-AuthorizedServiceCredentials this role enables access to the preauthenticate and authenticate endpoints.

FidoAuthorizationService-AuthorizedServiceCredentials this role enables access to the preauthorize and authorize APIs.

FidoAdministrationService-AuthorizedServiceCredentials this role enables applications to access the getKeys, updateKeys, deleteKeys, viewPolicy, addConfig, updateConfig, deleteConfig, viewConfig, ping, and updateUsername endpoints.

FidoCredentialService-AuthorizedServiceCredentials this role enables access to the getKeys, updateKeys, and the deleteKeys APIs.

FidoPolicyManagementService-AuthorizedServiceCredentials this role enables applications to access the addPolicy, updatePolicy, deletePolicy, and viewPolicy endpoints.

FidoMonitoringService-AuthorizedServiceCredentials this role enables applications to access the viewPolicy, viewConfig, and ping APIs.

These names of roles are the default ones; if willing, the designer of the system can change them or even add or remove roles. Figure 3.5 shows the matrix of roles with endpoints associated.

	FidoRegistrationService-AuthorizedServiceCredentials	FidoAuthenticationService-AuthorizedServiceCredentials	FidoAuthorizationService-AuthorizedServiceCredentials	FidoAdministrationService-AuthorizedServiceCredentials	FidoCredentialService-AuthorizedServiceCredentials	FidoPolicyManagementService-AuthorizedServiceCredentials	FidoMonitoringService-AuthorizedServiceCredentials
register	x						
preregister	x						
authenticate		x					
preauthenticate		x					
authorize			x				
preauthorize			x				
getKeys				x	x		
updateKeys				x	x		
deleteKeys				x	x		
addPolicy						x	
updatePolicy						x	
deletePolicy						x	
viewPolicy				x		x	x
addConfig				x			
updateConfig				x			
deleteConfig				x			
viewConfig				x			x
ping				x			x
updateUsername				x			

Figure 3.5. Matrix of StrongKey Fido Server roles in association with endpoints they unlock (Source: Strongkey [8])

SKFS does not make usage of this mechanism just to implement simple access control: due to roles, it is possible to implement a more complex system in which the web application is not just one and each application is delegated to offer a single service. For instance, it could be possible to dedicate a web application implementing the registration service and another for authentication and authorisation tasks.

More information about this can be found in StrongKey web site [8].

Endpoints

SKFS offers different endpoints to be used in protocols such as registration, authentication, authorisation and deletion of accounts. These APIs are protected with the server authentication so that only registered RPs can use them; the information regarding RP authentication is sent every time as input of every endpoint as value of "svcinfo" attribute. Endpoints are:

preregister it is the first endpoint component of the registration task and, as figure 3.6 shows, wants as input

username the name associated to the user account registering the FIDO credential,

displayName a label assigned to the unique authenticator used by the user,

options a JSON object containing an optional list of features the web application may request to SKFS to associate with this registration request,

extensions a JSON object containing an optional list of extensions the web application may request to SKFS to associate with this registration request;

while returns as output

rp a JSON object containing information about the RP sending the request,

user a JSON object containing information about the user account willing to be registered,

challenge a Base64Url encoded random nonce generated by the SKFS to challenge the Authenticator to sign it with the newly generated Private Key of the FIDO credential,

pubKeyCredParam array of JSON objects each of which describes the Public Key algorithm from the set of algorithms the SKFS will accept as valid for generated keys;

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe",
    "displayname": "johndoe_dn",
    "options": {
      "attestation": "direct"
    },
    "extensions": "{}"
  }
}
```

Figure 3.6. Input of StrongKey Preregister endpoint (source: StrongKey [8])

register it is the endpoint representing the second phase of the registration task and, as figure 3.7 shows, wants as input

strongkeyMetadata a JSON object describing metadata about the RP, such as the origin and the location,

publicKeyCredential a JSON object containing details of the response, composed as following

id FIDO credential identifier,

rawid raw byte sequence of the id, often equal to it,

response a JSON object containing the response to the challenge received, public key credential, and some details about the authenticator,

type the type of FIDO credential, always "public-key" if using FIDO;

while returns as output

response a string containing the result of the registration task;

preauthenticate as the preregister API, this one is the first part of a task, in this case the authentication task, and, as figure 3.8 shows, wants as input

username the name associated to the user account willing to perform authentication,

options a JSON object containing an optional list of features the web application may request to SKFS to associate with this authentication request;

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "strongkeyMetadata": {
      "version": "1.0",
      "create_location": "Sunnyvale, CA",
      "username": "johndoe",
      "origin": "https://demo4.strongkey.com"
    },
    "publicKeyCredential": {
      "id": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg",
      "rawId": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg",
      "response": {
        "attestationObject":
"o2NmbXRmcGFja2VkZ2F0dFN0bXSjY2FsZyZjc2lnWEcwRQIhAKh568CoVnRo3MIwVyLbYTiXuO7FTbsKfuqin4vhpu9YAiAEWQuISPN74PyBD_tpWm
jKix9gg_sQjf7xj0hO096XDGN4NWOBWQHkMIIB4DCCAYOgAwIBAgIEbCtY8jAMBggqhkjOPQQDAgUAMGQxCzAJBgNVBAYTAlVTMRcwFQYDVQQKEw5Td
HJvbmdBdXRoIEluYzEiMCAGA1UECxMZQXV0aGVudGljYXRvciBBdHRlc3RhdGlvbjEYMBYGA1UEAwwPQXR0ZXN0YXRpb25fS2V5MB4XDTE5MDcxODE3
MTEyN1oXDTI5MDcxNTE3MTEyN1owZDELMAkGA1UEBhMCVVMxFzAVBgNVBAoTDlN0cm9uZ0F1dGggSW5jMSIwIAYDVQQLExlBdXRoZW50aWNhdG9yIEF
0dGVzdGF0aW9uMRgwFgYDVQQDDA9BdHRlc3RhdGlvbl9LZXkwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAAQx9IY---
LDv1TjffHaHKRTJ_6HTapfASSnIs3Xk3rUyFgNOUzoC5aL6uas7uQDQt0_edIk1h53DoCoCHpFmJaMjKHlvWelbigwlFLWxgwlWRFs1WXAGlLL_Q5ba
lEou9VJGPHNPstewZX--KQgdI",
        "clientDataJSON":
"eyJ0eXBlIjoid2ViYXV0aG4uY3JlYXRlIiwiY2hhbGxlbmdlIjoiTENkbXlPQ2ZEUzltZDVJZkFYTzhtZyIsIm9yaWdpbiI6Imh0dHBzOi8vcWEtaW
5mb3N5cy1maWRvLTIuc3Ryb25na2V5LmNvbTo4MTgxIn0"
      },
      "type": "public-key"
    }
  }
}
```


Figure 3.7. Input of StrongKey Register endpoint (source: StrongKey [8])

while returns as output

challenge a Base64Url encoded random nonce generated by the SKFS to challenge the Authenticator to sign it with the Private Key of the FIDO credential,

rpId a string containing the origin representing RP's domain,

allowCredentials array of JSON objects each of which shows details about credentials allowed;


```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe",
    "options": {}
  }
}
```


Figure 3.8. Input of StrongKey Preauthenticate endpoint (source: StrongKey [8])

authenticate it is the final step for authentication task and, as figure 3.9 shows, wants as input

strongkeyMetadata a JSON object describing metadata about the RP, such as the origin and the location,

publicKeyCredential a JSON object containing details of the response, composed as following

id FIDO credential identifier,

rawid raw byte sequence of the id, often equal to it,

response a JSON object containing the response to the challenge received and some details about the authenticator,

type the type of FIDO credential, always "public-key" if using FIDO;

while returns as output

response a string containing the result of the authentication task;

jwt a JSON Web Token that provides an RP with the ability to establish SKFS as an Identity Provider to enable single sign-on for users with other web applications within their domain


```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "strongkeyMetadata": {
      "version": "1.0",
      "last_used_location": "Sunnyvale, CA",
      "username": "johndoe",
      "origin": "https://demo4.strongkey.com"
    },
    "publicKeyCredential": {
      "id": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg",
      "rawId": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg",
      "response": {
        "authenticatorData": "WnTBrV2dI2nYtpWAzOrzVHMkwfEC46dxHD4U1RP9KKMBAAAAFA",
        "signature": "MEUCIEB1evFffkyk1TwLRNtPWTv3G40DqABEuU8PJIdevt-lAiEAq9EiWwPicP3Ln2rQ17C1g--
OEYGxhp1Q1aHV3rUrE2c",
        "userHandle": "",
        "clientDataJSON":
"eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY2hhbGxlbmdlIjoiNWVWRGRGbVg0Y3JmNWJaVkE4WGhkZyIsIm9yaWdpbiI6Imh0dHBzOi8vZGVtbzQuc3
Ryb25na2V5LmNvbSJ9"
      },
      "type": "public-key"
    }
  }
}
```


Figure 3.9. Input of StrongKey Authenticate endpoint (source: StrongKey [8])

preauthorize this endpoint composes, with the authorize one, the task of authorisation of transactions and, as figure 3.10 shows, wants as input

username the name associated to the user account willing to perform authorisation of a transaction,

txid the unique transaction identifier,

txpayload the transaction payload describing details about the transaction authorised,

options a JSON object containing an optional list of features the web application may request to SKFS to associate with this authorisation request;

while returns as output

challenge a Base64Url encoded random nonce generated by the SKFS to challenge the Authenticator to sign it with the Private Key of the FIDO credential,

rpId a string containing the origin representing RP's domain,

txid the unique transaction identifier,

txpayload the transaction payload describing details about the transaction authorised,

allowCredentials array of JSON objects each of which shows details about credentials allowed;

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe",
    "txid": "ABC123",
    "txpayload": "ABBBBCCCDDD",
    "options": {}
  }
}
```

Figure 3.10. Input of StrongKey Preauthorize endpoint (source: StrongKey [8])

authorize it is the endpoint representing the second step for transactions authorisation and, as figure 3.11 shows, wants as input

txid the unique transaction identifier,

txpayload the transaction payload describing details about the transaction authorised,

strongkeyMetadata a JSON object describing metadata about the RP, such as the origin and the location,

publicKeyCredential a JSON object containing details of the response, composed as following

id FIDO credential identifier,

rawid raw byte sequence of the id, often equal to it,

response a JSON object containing the response to the challenge received and some details about the authenticator,

type the type of FIDO credential, always "public-key" if using FIDO;

while returns as output

response a string containing the result of the authorisation task;

txdetail a JSON object containing details of the transaction authorised such as txid, tx-payload, the nonce used and the challenge, the time of authorisation,

FIDOAuthenticatorReferences an array of JSON objects each of which gives details about the authenticator used: the most important data are

id FIDO credential identifier,

rawid raw byte sequence of the id, often equal to it,

rpId a string containing the origin representing RP's domain,

authenticatorData details about the authenticator used which, with clientDataJSON, can be used to implement transaction confirmation,

clientDataJSON a serialised representation of a JSON object containing platform data which, with authenticatorData, can be used to implement transaction confirmation,

signerPublicKey the base64-encoded public key associated to the private key used for the authorisation task,

signature the base64-encoded digital signature confirming this transaction,

signingKeyAlgorithm the algorithm used to confirm this transaction;


```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "txid": "ABC123",
    "txpayload": "ABBBBCCCDDD",
    "strongkeyMetadata": {
      "version": "1.0",
      "last_used_location": "Sunnyvale, CA",
      "username": "johndoe",
      "origin": "https://demo4.strongkey.com"
    },
    "publicKeyCredential": {
      "id": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg",
      "rawId": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg",
      "response": {
        "authenticatorData": "WnTBrV2dI2nYtpWAzOrzVHMkwfEC46dxHD4U1RP9KKMBAAAAFA",
        "signature": "MEUCIEB1evFffkyk1TwLRNtPWTv3G40DqABEuU8PJIdevt-lAiEAq9EiWwPicP3Ln2rQ17C1g--
OEYGxhp1Q1aHV3rUrE2c",
        "userHandle": "",
        "clientDataJSON":
"eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY2hhbGxlbmdlIjoiNWVWRGRGbVg0Y3JmNWJaVkE4WGhkZyIsIm9yaWdpbiI6Imh0dHBzOi8vZGVtbzQuc3
Ryb25na2V5LmNvbSJ9"
      },
      "type": "public-key"
    }
  }
}
```


Figure 3.11. Input of StrongKey Authorize endpoint (source: StrongKey [8])

getkeysinfo this API is used to gain information about all the registered keys belonging to a specific user and, as figure 3.12 shows, wants as input

username the name associated to the user account;

while returns as output

keys an array of JSON objects each of which gives details about one key, the most important data it shows are

keyid a string identifying a unique key,

fidoProtocol the protocol the key uses,

credentialId FIDO credential identifier,

status it represents the actual status of the key, whether the key is active or inactive;

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe"
  }
}
```


Figure 3.12. Input of StrongKey Getkeysinfo endpoint (source: StrongKey [8])

deregister endpoint which allows to delete a user key and, as figure 3.13 shows, it wants as input

keyid a string identifying a unique key;

while returns as output

response a string describing the result of this operation;


```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "keyid": "79U433x2hykUyf-h02qXwEkpyLN15N61MhYDTlM6AuWi-
rmrO7kA0LdP3nSJNYedw6AqAh6RZiWjIyh5b1npW4oMJRS1sYMJVkRbNVlwBpSy_0OW2pRKLvVSRjxzT7LXsGV_i4r7KRE83ItVOS_cDKbYn3axDcYi
UNaRXAR1DfHC5UP3hpystaKsOKvfCop2oA0rfrymTsUmF7RGKP-MNCiMP_Z5EnO8hHntAs41kTg"
  }
}
```


Figure 3.13. Input of StrongKey Deregister endpoint (source: StrongKey [8])

Details of the information described are available on the website of StrongKey [8].

3.4 Web application

As already specified, the web application must be executed inside a small embedded system and, for this reason, it needs to be the lightest possible: for this project, PHP and JavaScript are the programming languages chosen. The web application is logically divided in what the embedded system executes, the web application server, and what is run by the browser standing in the client device, the web application client.

3.4.1 Implementation of the service

First, the system needs to execute a service offering the web application to clients: for this project, Apache has had this role. WebAuthn APIs require the usage of TLS and the protocol https: to use them, first Apache must be configured to support TLS and to open the port 443; then, a certificate should be requested to a Certificate Authority. For this project, a self-signed certificate has been generated using openssl software.

To use the application, the system must have PHP installed. Moreover, the application will need a database: for this project MariaDB has been used.

Finally, since the embedded system must execute a server, this service must have a static IP address in such a way that every client willing to access the server can easily contact it.

In case the FIDO2 server or the proxy server needs TLS client authentication, if the certificate is self-signed, as in this project, it is mandatory to add this RP certificate in the server list of allowed certificates.

3.5 Client device

A client willing to be authenticated with the mechanism built in this project must be connected in the same network of the embedded system. It is possible to have a DNS server in the network which can guide the client to the actual web application; however, for this prototype it has been chosen to configure the device thereby automatically associating the web application server domain name to its static IP address.

3.6 How this project works

In this section is shown how the tasks are implemented: registration, authentication, authorisation, and deletion of accounts. The project design with all the logical actors is graphically explained in figure 3.14.

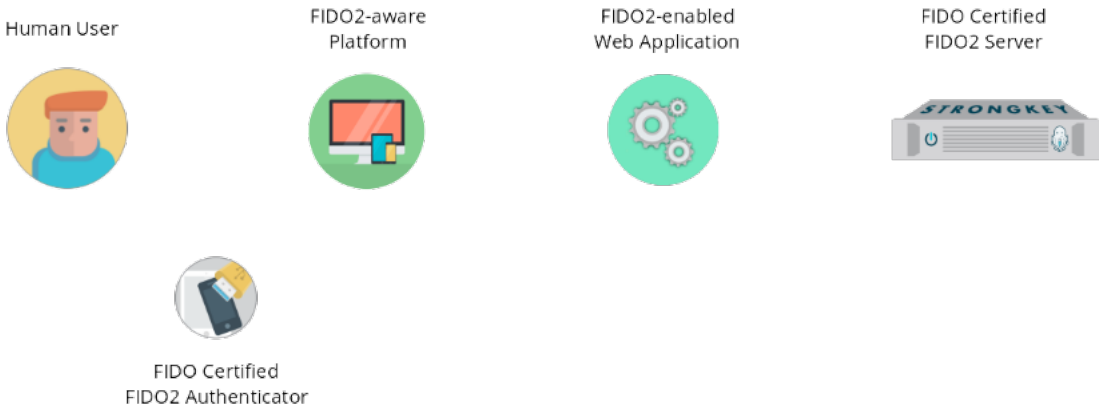


Figure 3.14. Scheme of all the actors of the project design (source: StrongKey [8])

3.6.1 Registration

The registration task can be schematised in seventeen steps. First, figure 3.15, the user chooses his username which must be unique; it is also possible to let him insert other personal data needed by the RP. Then, step 2, the platform will send to the web application this information and it will check if that username can be used, step 3. After having checked this, the preregister endpoint is called, step 4.

FIDO2 server can now elaborate the request and compute the challenge to be sent to the RP, step 5 (figure 3.16); then, this challenge is immediately sent to the platform, step 6, which can now use the authenticator to solve the challenge, step 7. If FIDO2 is used, the authenticator can also be external and CTAP can be used. During this phase, it is possible the mechanism to require the presence of the user, step 8.

As figure 3.17 shows, at the end of the verification phase, the authenticator can answer back to the platform with the challenge response, step 9, which is forwarded to the web application server, step 10. It now possesses all the needed data to call the register API of the FIDO2 server, step 11, which elaborates the request: if this challenge response is valid, the user's key counter is increased, step 12, and the new key is store, as figure 3.18 describes graphically as step 13.

FIDO2 server gives, then, the register response to the web application, step 14, which, in case of success, will update its databases thereby adding the new user, step 15. Then, it will forward the register response to the platform which can finally show to the human user the result of the task, step 16 and 17.

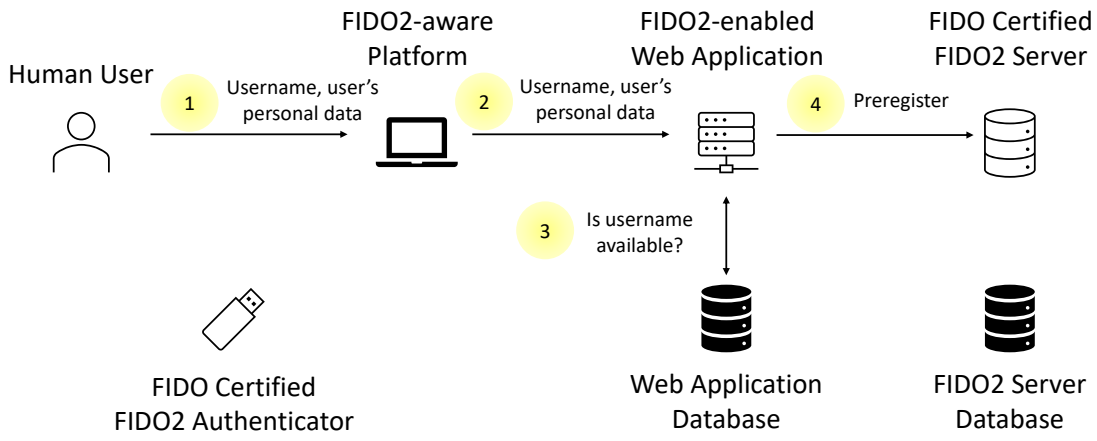


Figure 3.15. Scheme of the first part of registration task

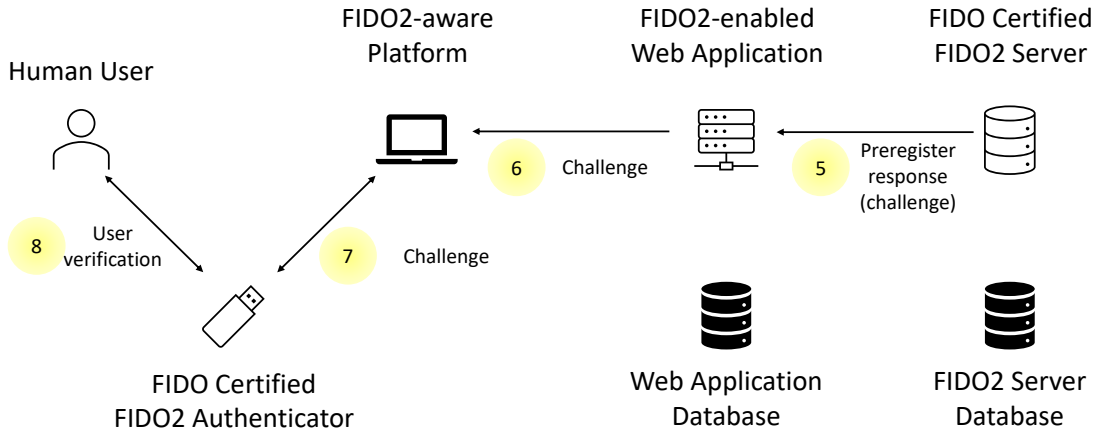


Figure 3.16. Scheme of the second part of registration task

3.6.2 Authentication

Once the user has been registered, he can be authenticated thereby accessing to the willing resource or service. To do this, first, as figure 3.19 shows, the user has just to provide his username to the platform, step 1. It is now possible to note that neither in the registration task nor in the authentication one the user needs to use passwords. Then, the username is forwarded to the web application server, step 2, which checks whether the username exists or not in its databases, step 3. In case of success, the preauthenticate endpoint is called, step 4.

The FIDO2 server computes and sends a challenge to the web application, step 5, to be

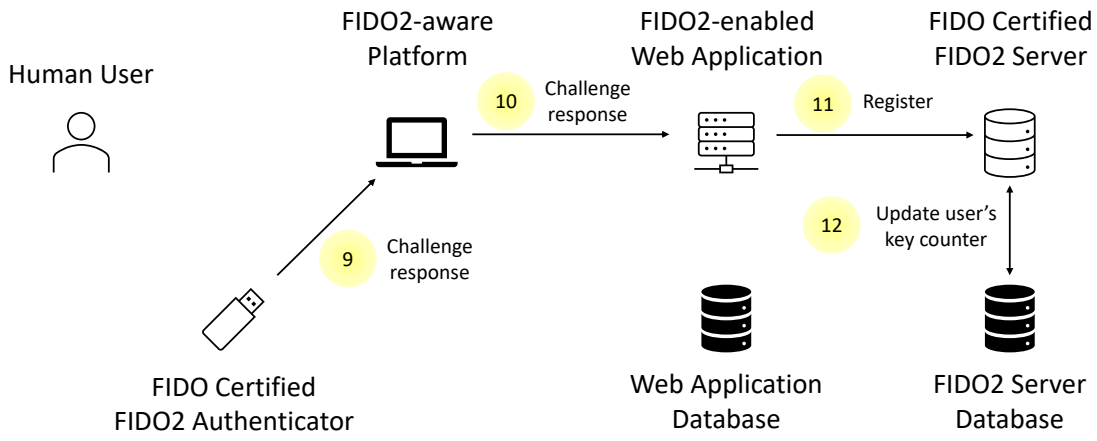


Figure 3.17. Scheme of the third part of registration task

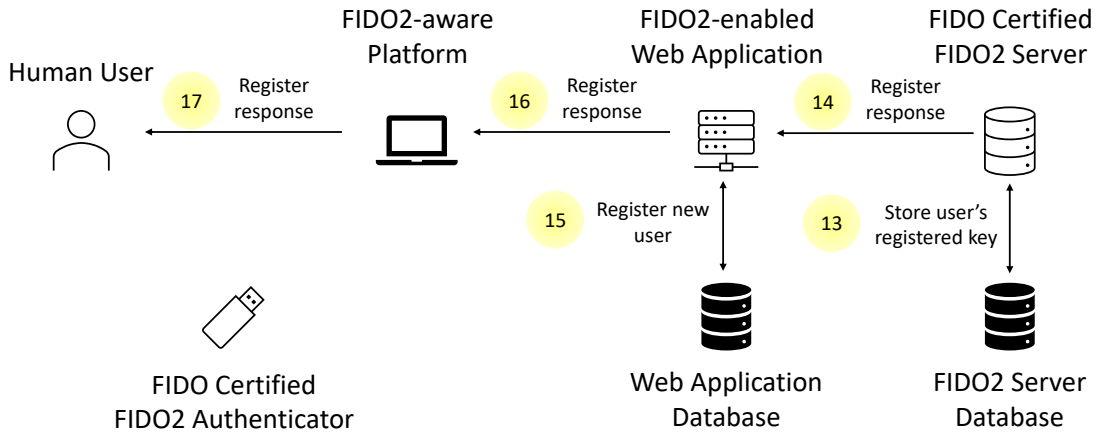


Figure 3.18. Scheme of the fourth part of registration task

forwarded to the platform, step 6; it is possible to follow these phases in figure 3.20. As for the registration task, the platform can now execute a protocol to solve the challenge with the authenticator, step 7, which can also require user verification, step 8.

Figure 3.21 explains that, at the end of the resolution of the challenge, the response obtained is given to the platform, step 9, and then sent directly to the web application server, step 10. With the information received, the web application builds data needed for the authenticate endpoint and calls it, step 11. FIDO2 server uses data arrived to check and verify the challenge response, step 12.

As figure 3.22 shows, the authenticate response is then generated and sent by the FIDO2

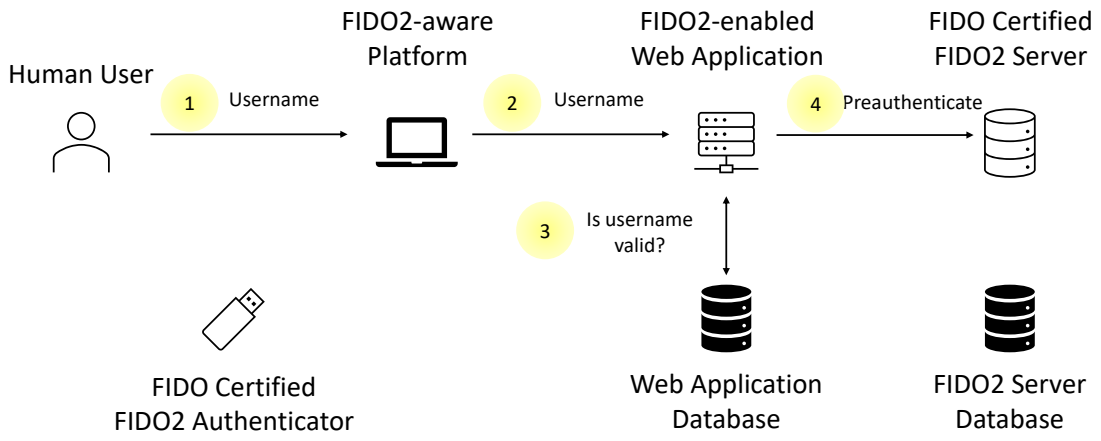


Figure 3.19. Scheme of the first part of authentication task

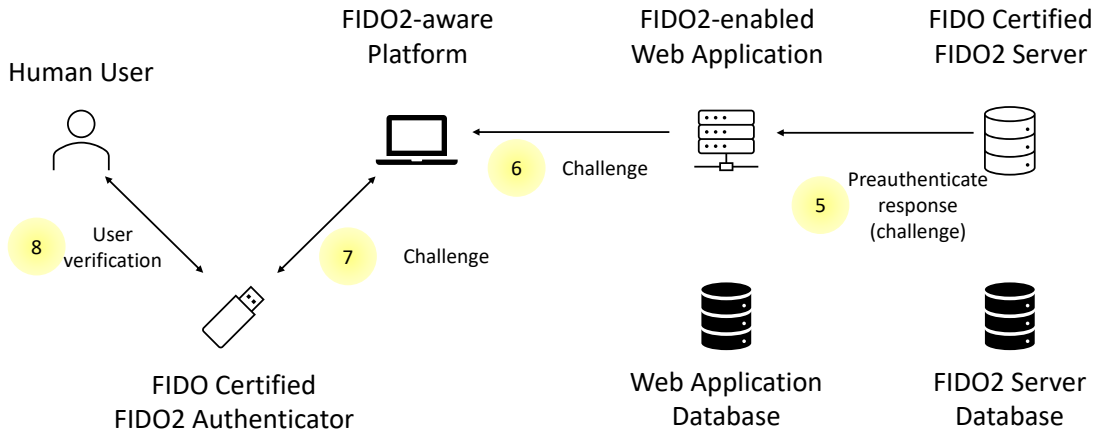


Figure 3.20. Scheme of the second part of authentication task

server to the web application server, step 13, which can use it to perform additional processing tasks, such as establishing sessions or following access control rules, step 14. Afterwards, the authenticate response is forwarded to the platform, step 15, which will then show it to the user, step 16.

3.6.3 Authorisation

After the authentication, a user obtains access to the resource or service wanted. Let the resource be the possibility of starting a transaction such as, for instance, buying an object. FIDO2 offers the

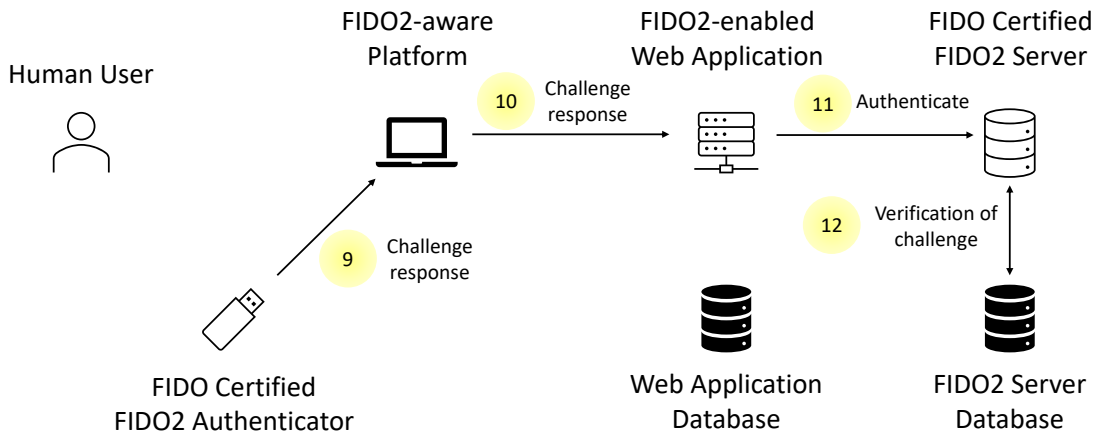


Figure 3.21. Scheme of the third part of authentication task

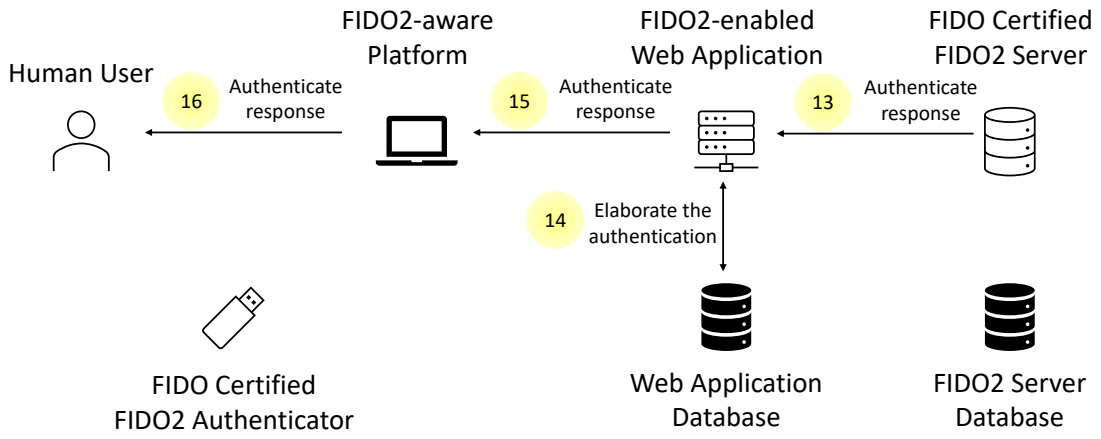


Figure 3.22. Scheme of the fourth part of authentication task

possibility of making the authorisation task, which is an authentication task with the transaction details associated.

First, the user tells the platform that it is willing to start a transaction (figure 3.23), step 1; then, the client of the web application sends to the server transaction details, such as transaction identifier and a transaction summary, step 2. It is mandatory that these details are the exact copy of what the user has seen when starting the transaction. When the server receives this information, it can use it to build data in the format expected by the FIDO2 server, step 3, and it calls the preauthorize API, step 4. The web application server does not need here to receive the username from the platform because the user has been authenticated before and, therefore, a

session has been established and it is now carrying all the most important information about the user such as its username. This is very important since, as consequence, different things happen. First, the user is the only one that can have rights of making transactions in his behalf; moreover, he cannot choose to send a different username and to start a transaction for another user. Finally, since the authentication phase has been executed, the web application server has the assurance that the user exists and it can avoid making checks in this sense.

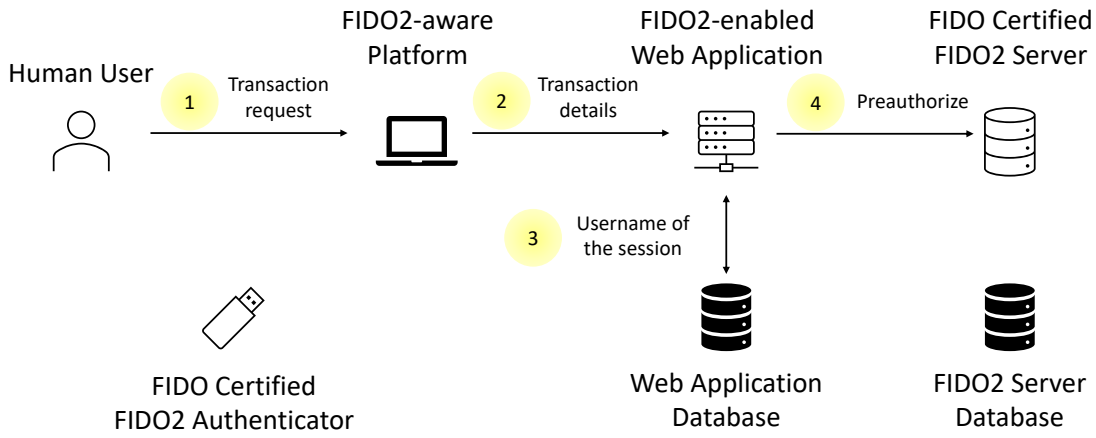


Figure 3.23. Scheme of the first part of authorisation task

FIDO2 server has to make itself sure the user is who he is saying to be; for this reason, as described in figure 3.24, it computes a challenge which is first sent to the web application server, step 5, and then forwarded to the web application client, step 6, which, as in the authentication phase, executes WebAuthn APIs to solve the challenge with the authenticator, step 7. Even in this case, depending on how the user has been registered, it is possible that this phase requires his verification, step 8.

Figure 3.25 shows that, in step 9, the challenge response is sent to the platform which will provide it to the web application server in step 10; the server now calls the authorize API of the FIDO2 server, step 11, which can finally verify the signature made by the user, step 12.

Once the result of the verification is computed, the authorize response is built and sent back to the web application, step 13 of figure 3.26. So far, this task is very similar to the authentication phase, but the authorize response is one of the main differences between the two tasks, since it contains all the details about the transaction authorised, the user public key used to make the authorisation and the signature made by the authenticator. This is crucial because, in this way, it will be later possible to have the confirm that that specific user has authorised that specific transaction with those details in that specific time, even in front of a court of justice.

When the response arrives to the web application, it can now process it, step 14: in the example done, the object is now officially bought by the user. Thereby showing the result to the end user, this authorize response is forwarded to the platform and to the user in steps 15 and 16.

3.6.4 Deletion

In case the user wants to delete his account or the RP wants to expel a user, the deletion task is executed. This task can be executed also to delete a single user key instead of the user account; this project prototype, nevertheless, does not support multiple keys for the same user account.

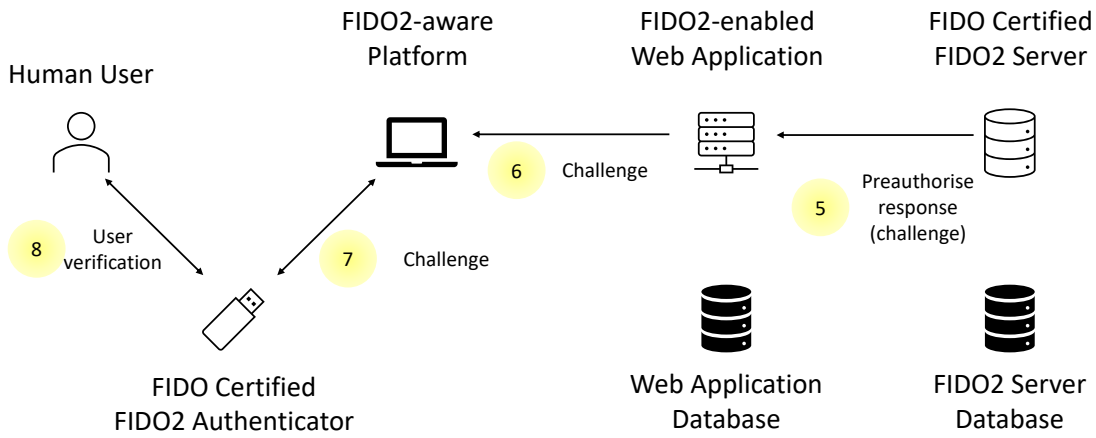


Figure 3.24. Scheme of the second part of authorisation task

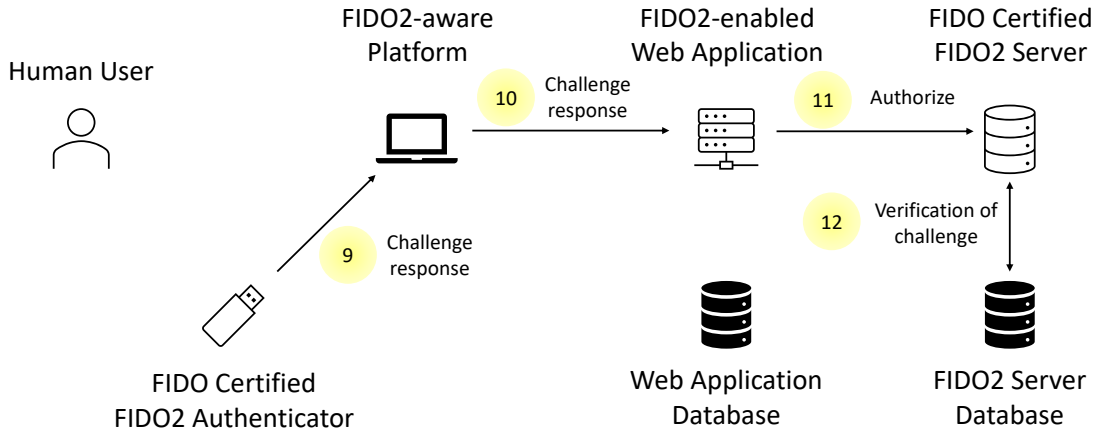


Figure 3.25. Scheme of the third part of authorisation task

As said, this task can be started on behalf of a user, as it is possible to see in figure 3.27, or directly by the web application server, as figure 3.28 shows. In the first case, the authenticated user, who has an active session, tells to the platform that he is willing to delete his account, step 1; the web application client sends this decision to the web application server, step 2, which now takes the actual username used from the active session, step 3. It is important to notice that the user has the authority to cancel its account because, since it has an active session, it has been authenticated first; moreover, neither the user nor the platform chooses the user to delete: the only account a user can delete is his own. In case the entity willing to delete a user account is the web application server, it can choose the username to delete, step 1 of the figure 3.28. For both

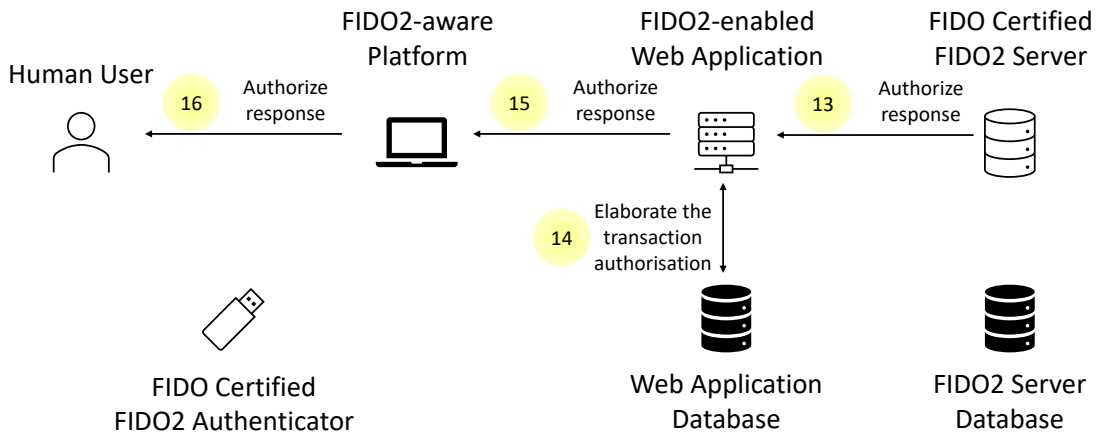


Figure 3.26. Scheme of the fourth part of authorisation task

the scenarios, the step 4 shows the web application calling the getkeysinfo API.

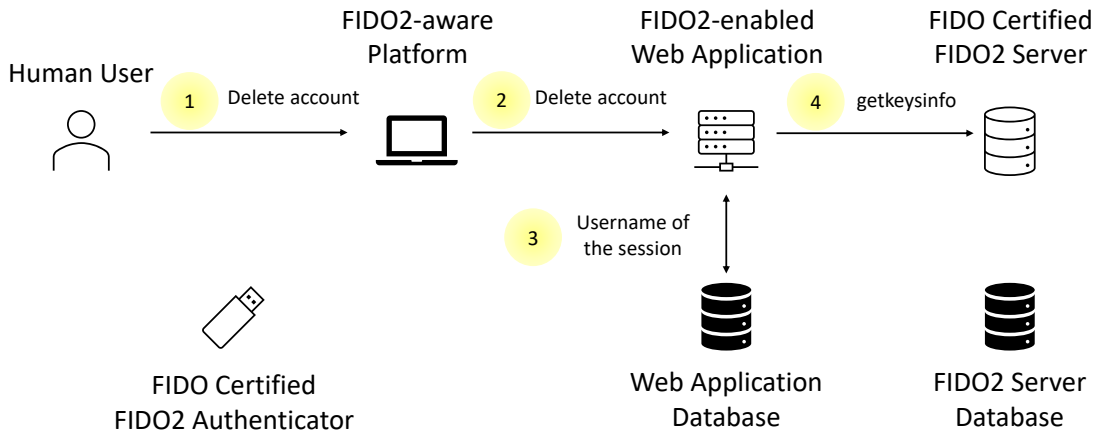


Figure 3.27. Scheme of the first part of deletion task

In this project, only the first case is supported so that not to give web application server managers the rights to delete users.

Figure 3.29 describes graphically that FIDO2 server answers, in step 5, with a getkeysinfo response, which contains an array of all the keys associated to that specific user. Now, the web application server can choose the identifier of the user's key, step 6, and calls the deregister endpoint sending the keyid chosen, step 7. As already said, this project does not support multiple keys and, therefore, the list contained in the getkeysinfo response will only have one key and the

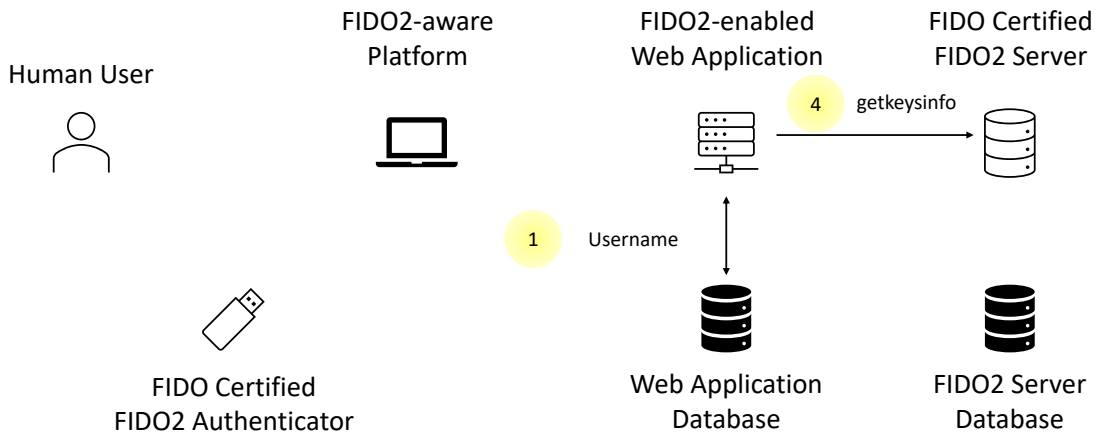


Figure 3.28. Scheme of the second version of the first part of deletion task

web application will choose every time that specific key.

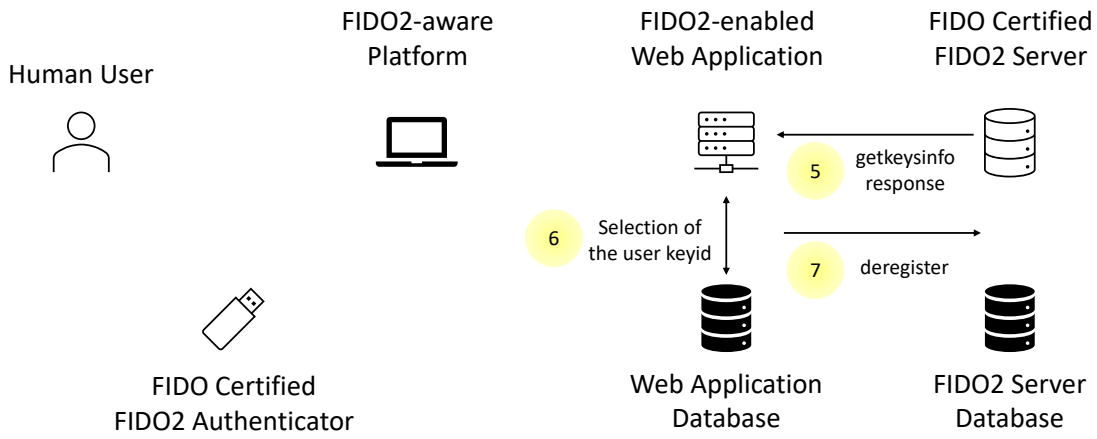


Figure 3.29. Scheme of the second part of deletion task

The third phase of the deletion task is shown in figure 3.30. After the reception of a request in the deregister endpoint, FIDO2 server will decrease the key counter and delete the chosen key, step 8. Then, the deregister response is built and sent to the web application server, step 9, which can now delete from its databases the key and, in case, the user account, step 10; afterwards, it will send the deregister response to the platform, step 11, which will show it to the user, step 12.

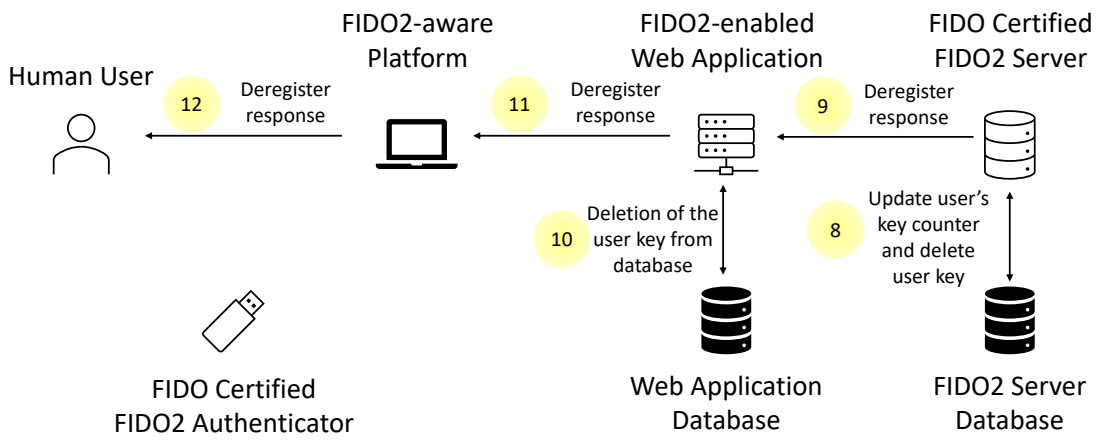


Figure 3.30. Scheme of the third part of deletion task

Chapter 4

Web application

The web application has been built to be executed by the embedded system, the security card. It is a crucial component of the system since it has the actual resource the user wants to access and uses FIDO2 server thereby implementing a secure authentication system. This RP stands, as already described, between the FIDO2 server and the authenticator and, therefore, the human user.

The web application is composed of two different parts: the client and the server. The first one is requested and executed by a FIDO-aware platform such as the most updated versions of the most used browsers, while the other is run directly by the embedded system.

4.1 Application design

The client and the server sides of the web application communicate each other whenever a task has to be performed. The server side is the part the user does not see and it has the role of communicating with the FIDO2 server using the endpoints described before; the client side, on the contrary, shows options and results to the user and executes protocols with the FIDO2 authenticator, independently of the fact that it is internal or external. The web application client makes use of WebAuthn APIs, as FIDO2 requires, to manage keys and the communication with the authenticator.

4.1.1 Interactions with users

As FIDO2 is a password-less authentication technique, the interaction with users is simple and easy to learn. Basically, every time the user wants to execute a task, it must interact with the client, sometimes using its authenticator. For this project, the authenticator tested are the Windows Hello, which is the internal embedded authenticator of Windows Operating System, and the security authenticator key developed by SoloKeys [10], which is an external authenticator.

Registration

When the user wants to be registered to the service, he contacts the web application and communicates his willing; then, he provides:

- first name,
- last name,
- username,
- display name.

After the web application elaborates the request, it is needed the user to choose the kind of authenticator to use. During the elaboration, the web application server and the FIDO2 server work together but in a transparent way, so that the user does not see them or their effects. Now, the user uses Windows Hello or the hardware authenticator to solve the challenge just by choosing the favourite authenticator; in this phase, user does not have to do anything else or give other

information. The authenticator used here will be needed to perform the other tasks. Once this phase ends, the web application elaborates the response sent by the platform and the task is completed.

Authentication

After the end of the registration task, the user is automatically redirected to the login interface, which can be used to gain access to the service. Thereby doing that, the input the web application client expects is:

- username.

The only information the user must remember, therefore, is his username. Moreover, since no other data is needed, the login phase is easy and fast from the user's point of view. Afterwards, the web application will ask the user to utilise the exact authenticator he used in the registration phase to prove his identity; once this is done, the whole login phase ends and the user is redirected to the resource page.

Authorisation

The web application allows authenticated users to perform transactions. In this project, the application pretends to offer a personal computer the user can buy as an example of transaction. To do that, the final user, if willing, can simply click to one button to start the transaction without inserting any information: indeed, the username data is unnecessary, since it is stored within the session, and dangerous, since a user can pretend to be another one which, possibly, can have different privileges.

Then, the web application requires a proof of identity which can be provided using the same authenticator chosen in the registration phase and used in the authentication one. Afterwards, the web application client will show the result of the authorisation phase.

Deregister

As the authorisation task, neither the deregister task requires the user to provide some particular data. For this task, the only interaction between the user and the platform is a button to be clicked. This is possible, obviously, only after the user has been authenticated: due to the session created, the user's identifier is already known by the web application. So, after brief time spent by the server to elaborate the request, the platform shows the results of this task.

Transaction confirmation

It is possible within the web application to see all the transactions done by the user and their details; in *Transactions* interface there is the history of transactions authorised. Here, the user can see details about transactions and he can check their validity just by clicking a button. After a while, the result of the verification is shown and even more details are shown.

A particular user, the admin, has an additional privilege: he can see the list of users and, for each one, the list of transactions authorised. In this other interface, this user can have the confirm of the validity of transactions in the same way explained before, just by pressing buttons.

4.1.2 Interactions with authenticators

The authenticator is needed whenever the web application must make itself sure about the identity of the user; therefore, the authenticator is required for registration, authentication and authorisation steps. The interactions with authenticators are based on WebAuthn APIs, as described in the official documentation ([7] and [11]).

Registration

In registration phase, the authenticator and the platform are needed to create credentials. In this task, a new secret key and the associated public key need to be generated; the public key and

the proof of the association with the secret one are then given before to the web application and then, through it, to the FIDO2 server.

For this purpose, *createCredentials* WebAuthn API has been used. As input, this function expects

challenge computed by FIDO2 server,

rp details,

user details,

pubKeyCredParams which describes parameters and details about the key to generate, such as the algorithm to use and the type of key,

authenticatorSelection which is an optional object which describes the type of authenticators allowed,

timeout the amount of time the user can spend to respond to the prompt for registration,

attestation which describes how the web application will track the user.

The output this API sends back after the computation of the key pair is composed as following

id FIDO credential identifier,

rawid raw byte sequence of the id, often equal to it,

response a JSON object containing the response to the challenge received, public key credential, and some details about the authenticator,

type the type of FIDO credential, always "public-key" if using FIDO.

In this task, the user chooses which authenticator to use: once chosen, the authenticator needs to be the same for all the other tasks which require the authentication proof.

Authentication

In the authentication task, the authenticator is obviously needed to generate the proof of the user's identity. In this phase, the communication with the web application client is required to gain access to the secret key stored inside the authenticator: without disclosing its key, the authenticator uses it to compute the signature on the challenge sent by the web application server and then answers with its proof. Using the *getCredentials* WebAuthn API, the web application provides to the authenticator

challenge computed by FIDO2 server,

allowCredentials which is an array of objects each of which describes a type of credential allowed,

timeout the amount of time the user can spend to respond to the prompt.

After the user responds to the prompt popped up to communicate with the authenticator, the API will return

id FIDO credential identifier,

rawid raw byte sequence of the id, often equal to it,

response a JSON object containing the response to the challenge received, public key credential, signature computed, and some details about the authenticator,

type the type of FIDO credential, always "public-key" if using FIDO.

The difference between the output of this API and the output of the *createCredentials* one stands inside the response, which is now carrying directly the signature used as identification proof and optionally a parameter called *userHandle*, which can be used by the server to associate this assertion to a specific registered user.

Authorisation

Since the authorisation task is an authentication task with a different purpose, the interaction with the authenticator remains the same as the authentication one. Basically, the design wants the user to authenticate again when checking details of a specific transaction in such a way to give his permission to start the transaction.

4.1.3 Interactions with web application database

The web application has its own database needed for authentication. For this project, it has been used a MySQL database with two tables: *users* and *transactions*. The first is mandatory to implement the authentication and, if the system needs it, authorisation; the second, on the contrary, depends on the actual RP and on resources and services it provides.

The *users* table has

id an autoincrement integer which has the role of identifier of the user and, therefore, it is the primary key,

username text representing the name associated to the user, chosen at registration time,

firstName text representing the user's first name,

lastName text representing the user's last name,

displayName text representing the name which will be displayed within the application referring to the user.

This table contains user's first name and last name as example of user's personal information. The table *transactions* contains information about transactions:

txid integer primary key, which is the transaction identifier,

txpayload integer representing the transaction payload describing details about this specific transaction,

username text representing the username of the user who has authorised this transaction,

signature text representing the digital signature computed at authorisation phase,

signerPublicKey text representing the public key associated to the user's secret key used to compute the signature,

signingKeyAlgorithm text representing the algorithm used to compute signature,

signingKeyType text representing the type of the key used to compute signature,

authenticatorData text representing data about the authenticator,

clientDataJson text representing data about the platform.

A schematized representation of this database is the following:

```
Users
(
                    id,
                    username,
                    first_name,
                    last_name,
                    display_name
);
       PRIMARY_KEY(id);

Transactions
(
```

```
                    txid,
                    txpayload,
                    username,
                    signature,
                    signerPublicKey,
                    signingKeyAlgorithm,
                    signingKeyType,
                    authenticatorData,
                    clientDataJson
);
       PRIMARY_KEY(txid);
       FK(username) -> Users(username);
```

The web application interacts with its database whenever a task is executed.

Registration

At registration phase, the database is consulted twice. First, before calling any endpoint, the web application server needs to be sure that the username the platform has sent to it is not used by other users. Once it has this assurance, the web application calls preregister and register endpoints. If this phase successfully ends, the server adds to the *users* table the new entry with the new information.

Authentication

When an authentication request arrives to the web application, it first checks whether the username is registered or not looking up at its database. If an entry is found, then the whole authentication task can start.

Authorisation

After the user has been authenticated, it can authorise transactions. For this task, the database is not used for checking the presence of the username since a session has been established and, therefore, an instance of the authentication phase has been successfully performed. After the authorisation endpoints are called, the FIDO2 server will return the response: if it is valid, a new transaction must be added in the *transactions* table: for this purpose, the output of the authorise endpoint is enough to provide all the needed information.

Deregister

When a user requests to be deregistered, the database is needed. As the authorisation task, neither for this task it is needed to check the validity of the request. If the whole deregister task successfully ends, the web application has the assurance that the user has been deregistered from the FIDO2 server and, therefore, it can continue the task deleting the entry corresponding to the user from the *users* table.

Transactions and admin

When a user goes in the transactions interface, it can see all the transactions it has authorised within the web application. Thereby doing this, the application needs to look up into the *transactions* table searching for all the transactions having as username parameter the one standing in the session.

The same happens when the admin user accesses to the admin interface, so that to see the list of users with their list of transactions. To do that, the web application searches for all the entries of both the *users* and the *transactions* tables.

4.1.4 Interactions with FIDO2 server

On the other side of the system, the web application server must communicate with FIDO2 server. This external server has the role to assure the web application, the RP, that the user is who he says to be; therefore, every time a task is called, RP has the need to interact with it.

Registration

When the web application server receives a registration request from a new user, it needs first to elaborate this request building the right data and calling the preregister API. As it will happen every time, with the actual task data, RP must send svcinfo data, needed by FIDO2 server to recognize the RP and to let it use its endpoints. After the call, web application server waits for the API output and sends it to the user, so that it can answer to the challenge.

After the user has responded to the registration prompt using its authenticator, the web application obtains all the necessary data to compute input for the register endpoint. Then, the RP uses this information and, appending it with its SCID credentials to be authenticated to FIDO2 server, calls the API and waits for the answer. Depending on the result, then, the web application may register the user and ends the task.

Authentication

The authentication task needs FIDO2 server to grant for the user. In this phase, RP receives the name of the user to authenticate and sends it to the FIDO2 server through the preauthenticate endpoint. As in the registration phase, the response of this API contains a challenge to be solved by the user: it needs to be sent to the platform.

This is the moment in which the authenticator is called. After the protocol, the web application server will have the response of the challenge previously sent and it is now ready to call authenticate API. If the result of this call is positive, then the user is authenticated and redirected to the service or resource wanted; if not, he is blocked.

Authorisation

As said above, the authorisation task is a more particular and less general type of authentication: an authentication associated to the willing of starting a transaction. For this reason, even the interaction with FIDO2 server, in the authorisation task, is quite similar to the one described before, in the authentication task. First, RP sends username and some details about the transaction using preauthorise API and waits for the response, a challenge to send to the user. Once the authenticator has solved the challenge, the information is sent to FIDO2 server thereby letting it authenticate the user and authorise the specific transaction: this phase is done calling the authorise endpoint. As before, the result of this function is used by the web application to let the transaction start or block it.

Deregister

When a user wants its account to be deleted, he needs to be authenticated first. For this reason, the web application already knows his credentials and can use them to make the entire deregister task without interacting with the user's authenticator.

Nevertheless, even though the RP knows the username, it is not enough to deregister directly the user. Since the entire authentication and registration engine is managed by the FIDO2 server, RP does not use user's keys and cannot have the knowledge about their identifiers, needed by the FIDO2 server to deregister the user. Therefore, an intermediate step is required for this task: the web application server has first to call getkeysinfo endpoint using the username already known to obtain the list of keys owned by the user. With this information, the RP can now use the key identifier to call the deregister API in order to delete that specific key. If the whole task successfully ends, also the RP deletes the user from its database.

4.1.5 Interaction between web client and web server

The role of the client side of the web application is mainly to gather information about the user and his willing and to send them to the server side; the web application server, instead, is in

charge of using the information received by the platform thereby building data in the format requested by the FIDO2 server endpoints and calling the right API. Moreover, when an answer is sent by the FIDO2 server, the web application server has the role of forwarding the result to the platform in such a way it can understand it and show it to the user. In practice, the interaction between client and server is mainly of conversion of data, both from users to the FIDO2 server and from the FIDO2 server to users.

4.2 Developer manual

The web application has been built using PHP and Javascript. The application folder contains different subfolders in which files are classified based on their type.

In the root folder there are

fido2service.sql the database file, a file containing a summary of tables and data structures,

index.php the starting file representing the homepage of the web application, containing only some information about the author of the project,

constants.php a file containing all important constant values needed for many things, such as instantiate calls for APIs, authenticate the RP, identify all endpoints, and enable or disable logs,

server.log a log file needed to test the system.

In following sections, all the subfolders are described.

4.2.1 Certificate

This subfolder, initially empty, is needed to store and install the FIDO2 server certificate. The FIDO2 server chosen, StrongKey, accepts only requests done with TLS; nevertheless, in this prototype, the server certificate is auto-signed and not requested to a Certificate Authority: for this reason, the installation of its certificate is needed to avoid issues in communication with it.

If client authentication is somehow enabled in FIDO2 server side, as happens for instance in the case in which a proxy stands between RP and StrongKey Fido Server, the RP certificate needs to be installed at server side since it is, in this case, self-signed.

4.2.2 Css

This subfolder contains all the .css files. These files are needed to personalize the style of the application and there is one for each interface plus one, layout.css, that is common for every page. They are

- admin.css,
- index.css,
- layout.css,
- login.css,
- registration.css,
- resource.css,
- transactions.css.

4.2.3 Images

Inside this folder, the web application stores all the images used within its client side. In particular, among background and application icon images, there are icons which are used in transaction confirmation phases to give a user-friendly output of the result of the computation.

4.2.4 JS

This subfolder contains all the Javascript files used in the web application. This folder is one of the most important ones since in these files are built all the algorithms at the client side. Among utilities files, here is one file per interface but the homepage:

constants.js this is a file containing constants about the web application server endpoints and locations and utilities functions,

login.js in this file the submission of the login form is handled as previously described,

registration.js this file contains algorithms used in the registration phase,

resource.js in this file there are functions which handle authorisation of new transactions and deletion of users,

transactions.js this file handles transaction confirmation phases building keys and performing verification of signatures.

The *transaction.js* file is particularly important since it performs the confirmation of transactions step: in here, the computation of data, the import of public keys and the verification steps are performed. These actions are executed differently basing on the algorithm used in the computation of the transaction signature; therefore, a manager willing to add support for other algorithms must modify this file in particular. The project, at present, supports only RSA.

Every file which communicates with the web application server side has at least two functions, used to handle the two steps of each task. For instance, in registration.js file there is *handleSubmit*, a function used to send values inserted in the registration form to server, and *callFIDO2RegistrationToken*, a function used after the reception of the challenge to communicate with the authenticator.

In these algorithms, it is usual to find asynchronous functions calls and promises; the communication with web application server is performed using the asynchronous *fetch* function.

4.2.5 PHP

This folder contains all PHP files and, therefore, all the interfaces and the web application server. Besides the homepage, which is described by the file *index.php* standing in the root folder, in this folder there are

- admin.php,
- login.php,
- registration.php,
- resource.php,
- transactions.php.

All these files build a particular interface.

API

This folder, standing inside the *PHP* one, contains all the files used to describe all the endpoints offered by the web application server to the web application platform. Then, here are

preauthenticate.php here first it is checked if the username is present within the database and then calls the preauthenticate endpoint of the FIDO2 server,

authenticate.php here the authenticate endpoint of the FIDO2 server is called and, in case of success, the session is created,

preauthorize.php here the preauthorize endpoint of the FIDO2 server is called,

authorize.php here the authorize endpoint of the FIDO2 server is called and, in case of success, the transaction details are stored in the database,

preregister.php here first the server makes sure the username chosen does not exist in its database yet and then it calls the preregister endpoint of the FIDO2 server,

register.php here the register endpoint of the FIDO2 server is called and, in case of success, the user information is memorized in the database,

logout.php here the session is destroyed,

deregister.php here getkeysinfo and deregister endpoints of the FIDO2 server are called using information stored in the session and, in case of success, the entry corresponding to the user is deleted from the database and the current session is destroyed.

In all the endpoints communicating with FIDO2 server, the communication has always been built using *curl* functions.

4.2.6 Utils

In this directory, two subfolders containing utility tools are present: *database_utils* and *verifysignature*.

The first contains useful files of a quick database installation:

configuration.sql defining how the database and the user utilising it within the web application are created,

create.txt summary file pointing to the other ones thereby configuring the database with only one single command,

transactions.sql defining the *transactions* table,

users.sql defining the *users* table.

The second subfolder contains a tool used to test the system by manually verifying transactions signatures using parameters the tester can find in the log file. This tool is composed of:

verifySignature.css defining the style of the interface of the tool,

verifySignature.html defining the structure of the interface of the tool,

verifySignature.js defining the behaviour of the tool.

4.3 Installation

Thereby using this system, the manager should first install all the components needed. First, a network should be created and all components should be attached to it. As already explained, different designs are here possible, but in this project a single network has been used for all components.

To install the whole system, the manager should acquire a Raspberry Pi4 [14], the embedded system used in this thesis, and another machine, which can be a server or a simple personal computer, as happens in this prototype; to communicate with the server part of the system, the manager needs a machine to have the role of client.

It is now possible to map the logical components to the physical ones. FIDO2 server is running in the physical server, as well as FIDO2 server database. The embedded system will execute the web application database and server, while offering the RP client to the client machine, which will run it through a FIDO2-aware platform such as the most updated and famous browsers. This client machine, moreover, will communicate with the user and the authenticator, which, depending on its being internal or external, will be run inside or outside the machine. Figure 4.1 describes graphically the association between logical and physical components.

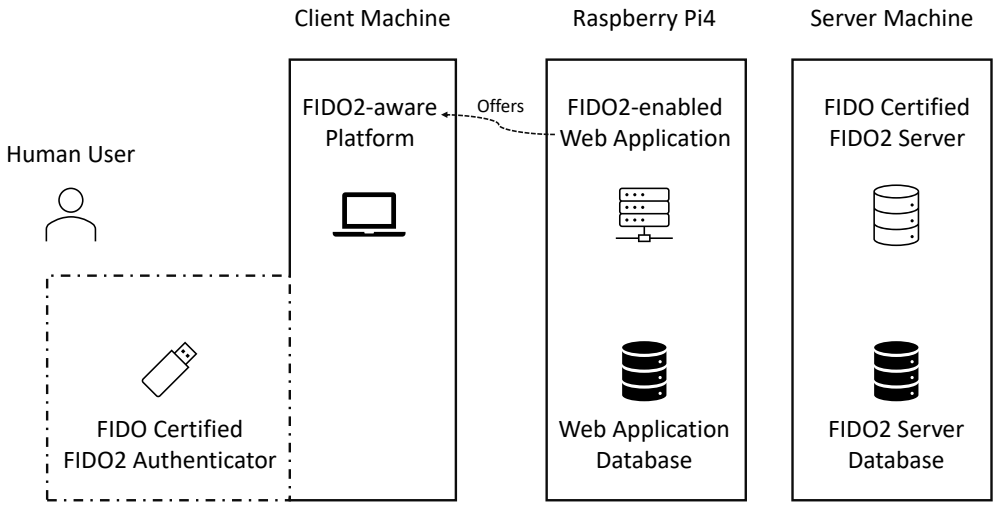


Figure 4.1. Association between logical and physical components. The authenticator component can be part of the client machine or can be external. The RP client is offered by the Raspberry board but it is executed by the client device.

4.3.1 SKFS

SKFS should be the first system to be installed in the server machine. To do that, it is possible to follow official StrongKey instructions [8]; these instructions can be followed whether the manager wants to build this system using virtualisation, as in this project, using dockerisation or even as a standalone server. In any case, this server must have the access to this network thereby communicating with the embedded system.

All the steps executed in this project to install SKFS are shown below.

Creation of the virtual machine

First, it is mandatory to download CentOS 7 iso file [12]: CentOS 7 is one of the operating systems used in testing StrongKey Fido Server by StrongKey and it has been chosen to be the OS of the virtual machine which will run the FIDO2 server. For the creation of the VM, VMWare Workstation 16 Player has been used [13]; to create the VM, it is possible to use the wizard offered by the software (figure 4.2).

The next step consists in the generation of a username, as figure 4.3 shows. For this virtual machine, I used *username* as username and *password* as password; as it is obvious, someone who wants to create his own server must change these parameters. After this screen, it is possible to choose the virtual machine name and the location where to store it.

In the next window the manages chooses the disk size, which is 20GB according to the instructions provided officially by StrongKey. Moreover, it has been chosen to store the virtual disk as a single file (figure 4.4). Afterwards, since a few changes must be done in the virtual hardware, the "Customize Hardware" button must be clicked. Here it is needed to change the network adapter, which is set in NAT by default: since the server must be connected directly to the network, its adapter must be set as bridged, as it is possible to see in figure 4.5. Furthermore, at least 4GB of RAM are needed (figure 4.6).

Once the virtual machine has been correctly created, it must be powered on.

Installation of SKFS

For the installation of the actual FIDO2 server, it is first needed the machine to have the right hostname. To set the hostname, the manager should open a terminal window and run:

```
$ su
```

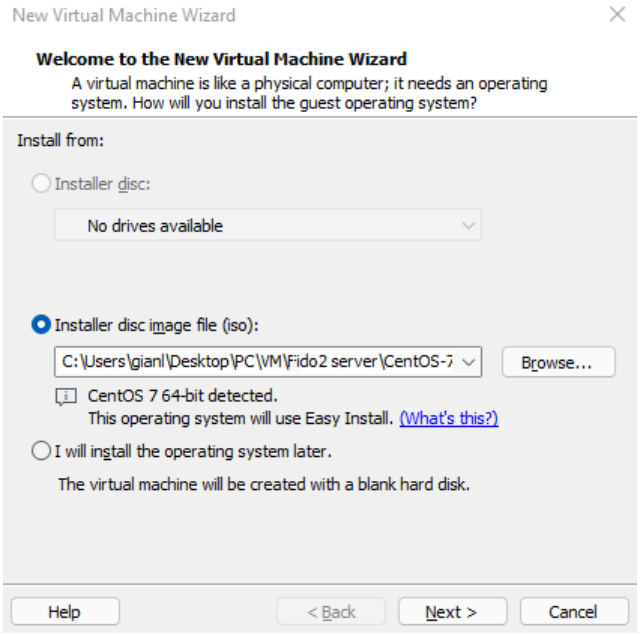


Figure 4.2. The first step of the installation of SKFS: setting the iso file

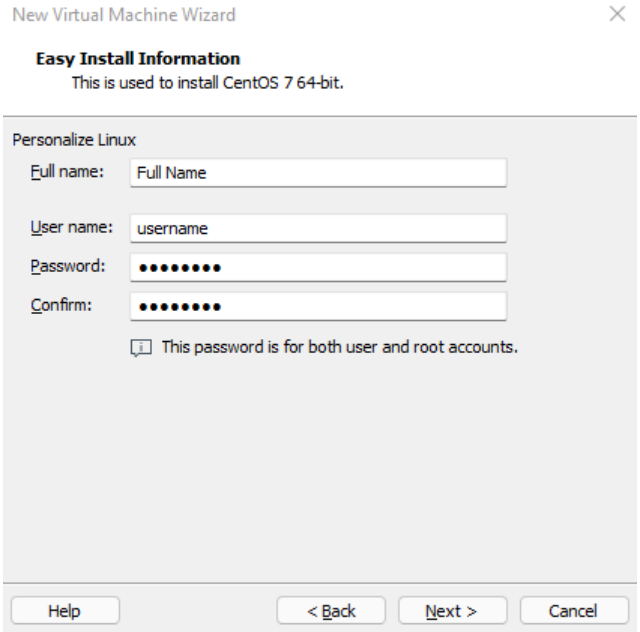


Figure 4.3. The second step of the installation of SKFS: creating the user

```
# hostnamectl set-hostname fido2server.strongkey.com
# exit
```

This hostname has to be the same as in the x.509 certification; as already explained, since this is a prototype, the certification is self-signed and generated by StrongKey software: for this reason, *fido2server.strongkey.com* has been chosen to be the hostname of the machine. In case the manager of the system wants to request a valid certificate, it can choose the hostname it wants.

Now, the archive containing the whole SKFS software must be downloaded. To do this, the manager can simply go to the StrongKey website [8] and go to the fido2 GitHub page by clicking on *Developer* menu item and, then, on the link pointing to the GitHub website. From here, it

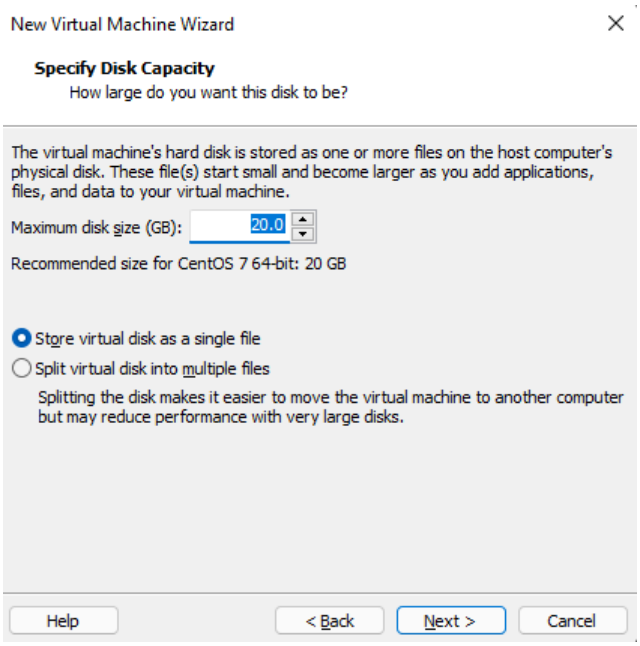


Figure 4.4. The third step of the installation of SKFS: setting disk parameters

is possible to download the zip file containing the project by clicking on *Code* button and on *Download ZIP* in the menu which appears (figure 4.7). Afterwards, the manager can go to the terminal and run:

```
$ cd ~
$ mkdir ./temporary
$ mkdir ./fido2server
$ unzip ./Downloads/fido2-master.zip -d ./temporary/
$ mv ./temporary/fido2-master/fido2server-v4.4.3-dist.tgz ./fido2server/
$ rm -r ./temporary/
$ cd ./fido2server/
$ tar xvzf ./fido2server-v4.4.3-dist.tgz
$ su
# sudo ./install-skfs.sh
# exit
```

The aim of these instruction is to download and extract the .tgz file present in the fido2 folder of the GitHub page of StrongKey. This file, which is of the version 4.4.3 in the case of this project, can change and be updated and contains all the files needed for the installation of SKFS. At the end of the extraction, it is possible to run the installer script; afterwards, the FIDO2 server is completely installed and working.

General settings

Once the server has been correctly set up, the manager must check whether the network adapter is properly working. To do that, it is possible to go to settings, then search for network and click on the gear icon in the *wired* section. Here, the manager can see all the network parameters of the server, such as the IP address, the DNS server, and the default route.

Since this virtual machine must be a server, it needs to have a static IP address. To set it, it is possible to go to IPv4 settings and select the *manual* method of assignment of addresses. Now, the manager can choose the address, the netmask, the default gateway, and the DNS server of the machine; in the case of this project, they are respectively 192.168.1.17, 255.255.255.0, 192.168.1.1, and 192.168.1.1, as figure 4.8 shows. Afterwards, the manager must apply these changes, turn off the network adapter and turn it on again.

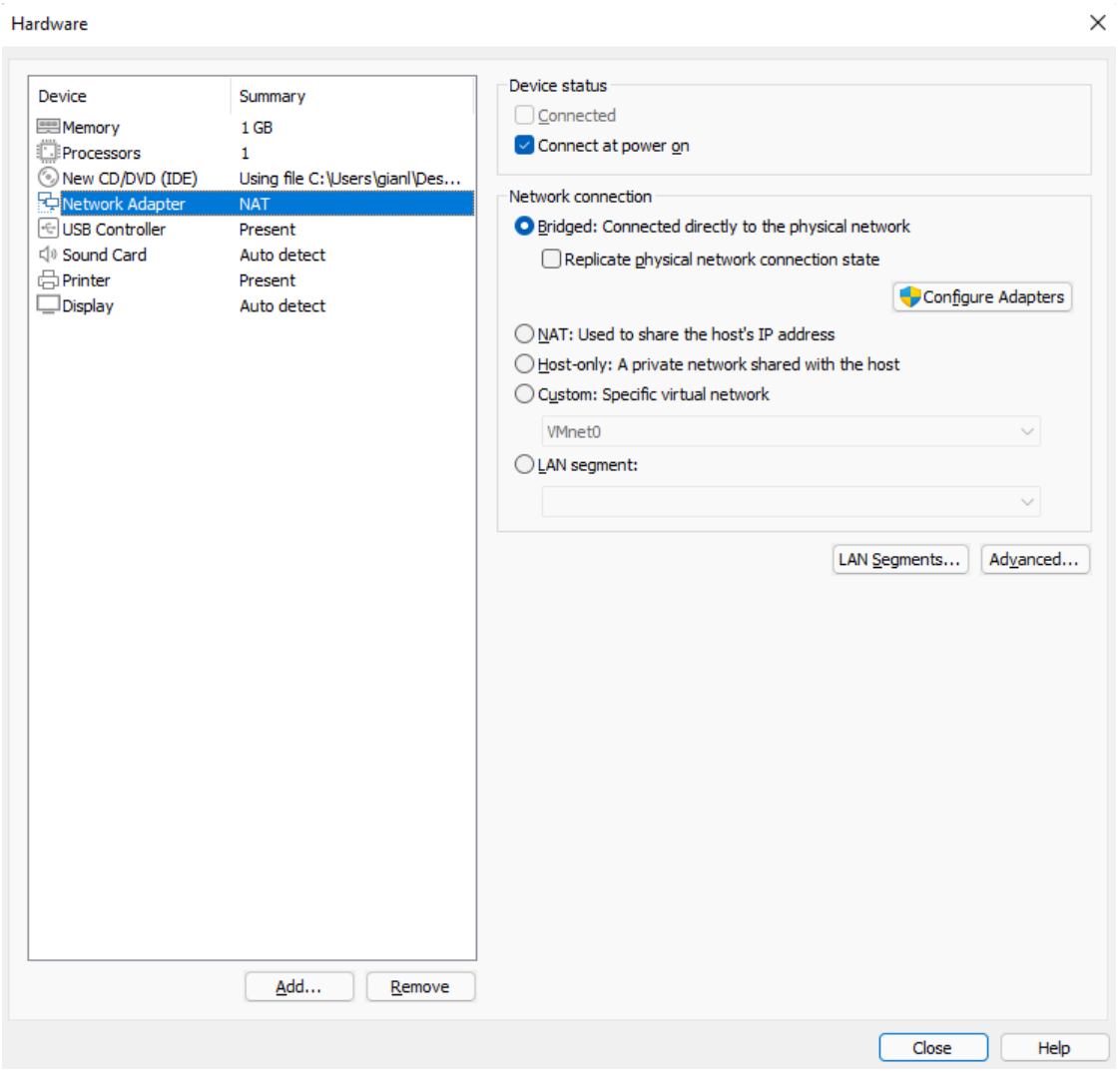


Figure 4.5. The fourth step of the installation of SKFS: setting network adapter parameters

Furthermore, to let the virtual machine reachable from the net in the port 8181, the port used by SKFS, the manager must open it using firewalld command. To do that, first the software needs to be updated: the manager must open the terminal and run the instructions

```
$ su
# yum update -y
```

Afterwards, the manager must insert the port 8181 in the public zone of the firewall and reload it:

```
# firewall-cmd --zone=public --add-port=8181/tcp --permanent
# firewall-cmd --complete-reload
# exit
```

Management of SCIDs

To let the RP communicate with SKFS, it is possible to use the default SCID present initially in the instance of the server; nevertheless, this solution is not secure. In this project, another SCID has been configured using the tool offered by SKFS:

```
$ su
# sudo ./manageSKFSCreds.sh addUser 1 fido2service
```

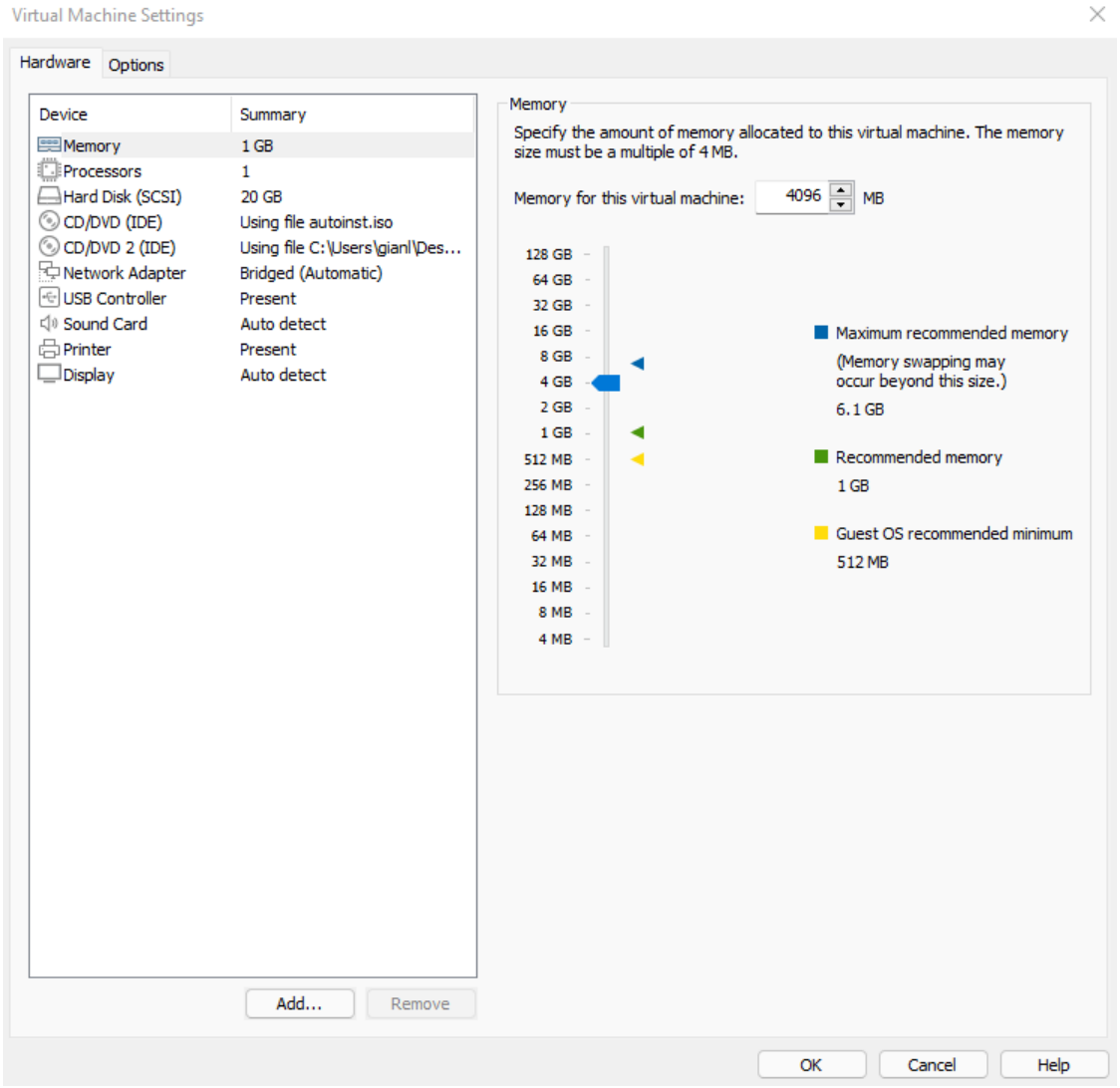


Figure 4.6. The fifth step of the installation of SKFS: setting RAM parameters

```
# sudo ./manageSKFSCreds.sh addUserToGroup 1 fido2service
FidoRegistrationService-AuthorizedServiceCredentials,FidoAuthenticationService-
AuthorizedServiceCredentials,FidoAuthorizationService-
AuthorizedServiceCredentials,FidoAdministrationService-
AuthorizedServiceCredentials
```

The first instruction allows the creation of the user and associates it to a password, which in the case of this project is *F1d02s3rv1c3_p4ssw0rd!*; the manager needs here to pay attention: the password it chooses here must be, then, statically added in the software executed as RP. The second instruction, instead, is needed to add the user to roles thereby letting it to access to different SKFS endpoints.

Once the SCID which will be used by the RP is created, it is mandatory to deal with the initial default user present by default in the FIDO2 server. In this project, as already explained, it has been chosen to change its password:

```
# sudo ./manageSKFSCreds.sh changeUserPassword 1 svcfidouser
# sudo service slapd restart
```

The password chosen for this superuser is, for this prototype, *Adm1np4ssw0rd!*.

Once all these things have been done, the FIDO2 server can finally be considered as successfully installed.

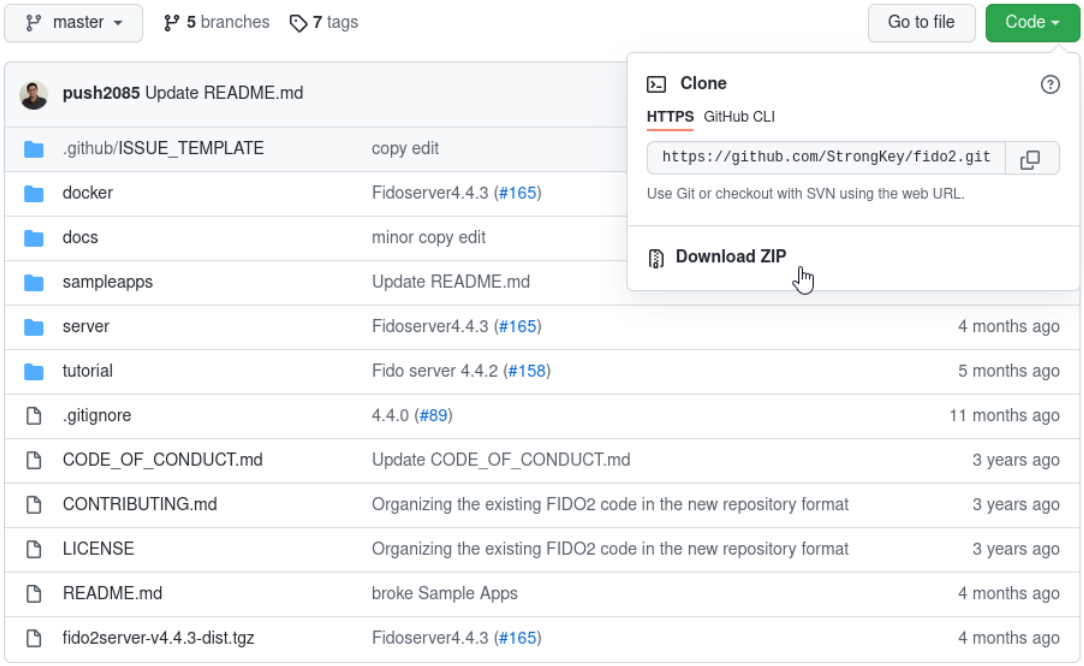


Figure 4.7. GitHub page of StrongKey [8], from which it is possible to download SKFS project

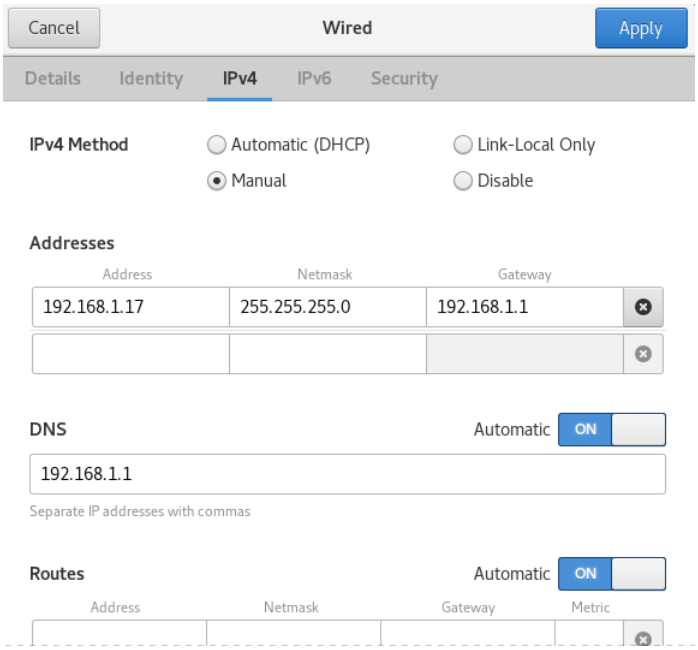


Figure 4.8. Settings of the static IP address

```
# exit
$ exit
```

4.3.2 Web application

To build the system, the manager should first obtain the hardware; in this project it has been used Raspberry Pi 4 (figure 4.9). Besides of this card, the manager should have a keyboard, mandatory to correctly configure the hardware, a monitor, and a cable used to connect the card

to it, and optionally an ethernet cable. This hardware supports also Wi-Fi mechanism, so it is also possible to use it in substitution to the connection cable. Raspberry Pi 4 is provided with two micro-HDMI ports, which allow it to be connected to monitors through some special cables. In this project, the monitor was provided with an HDMI port and, therefore, a HDMI - micro-HDMI cable has been purchased.

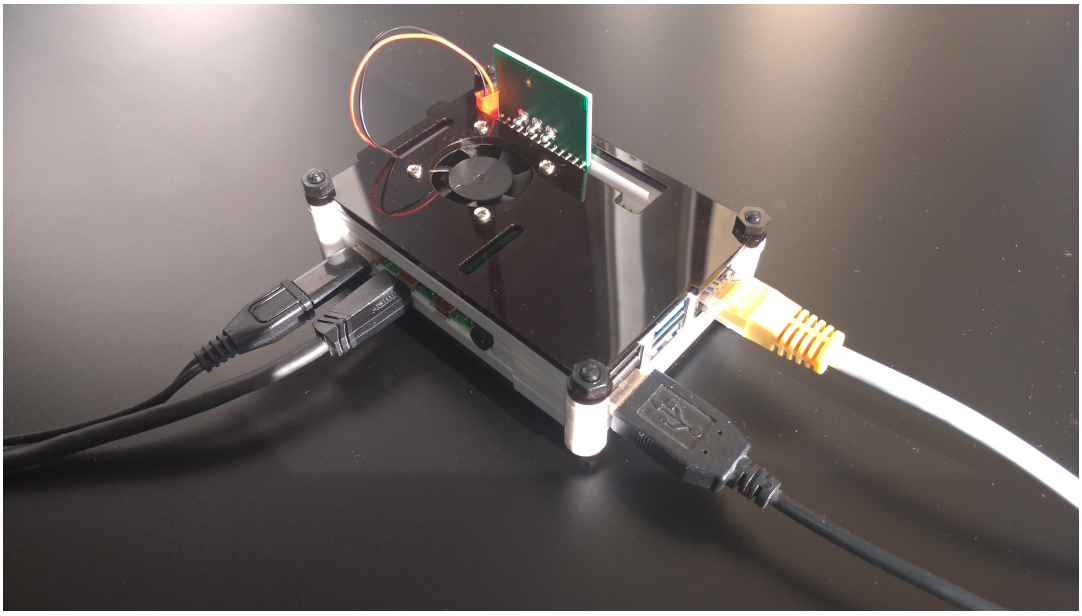


Figure 4.9. Raspberry Pi 4

This hardware uses a SD card as disk; in this project, Raspbian [15] has been flashed in that card: this OS is a particular version of Debian Linux properly built for Raspberry Pi cards.

Once the SD card is inserted and the embedded system is turned on, it is possible to be logged in and enter in the system by using the default username and password, which are respectively *pi* and *raspberry*.

Preliminary actions

There are few things to do before starting the installation of the web application: the software must be updated, the initial authentication must be changed in order not to let attackers to enter in the system, the hostname of the system must be changed, and the IP address of the card must be put static. The first three actions can be executed just running these commands and following their instructions:

```
$ sudo apt-get update -y
$ passwd
$ sudo hostnamectl set-hostname fido2service.strongkey.com
```

To avoid issues regarding the modification of the hostname, the manager should check the file */etc/hosts* to be sure that the new hostname is put near *raspberrypi*; if it is not, then it should add it manually:

```
$ sudo nano /etc/hosts
```

Thereby setting a static IP address in the card, it is possible to follow these instructions:

```
$ ip -c link show
$ ip -c addr show eth0
$ cat /etc/network/interfaces
$ sudo nano /etc/network/interfaces.d/fido2service
```

First it is mandatory to be aware about the card network interfaces; once the right one has been found by running the first command, it is possible to view details of the interface using the second one: for this prototype, the interface used was eth0, but it may be different for other projects. Afterwards, a new network configuration file can be created and set; thereby doing this, the manager should first check that */etc/network/interfaces.d/* folder is included as source-directory in */etc/network/interfaces* file, through the third instruction. Then, a new file can be created in the configuration folder: for this project, the name of the file is *fido2service*; its content must be the following:

```
auto eth0
iface eth0 inet static
 address 192.168.1.16
 netmask 255.255.255.0
 gateway 192.168.1.1
 dns-nameservers 192.168.1.1
```

The parameters viewable in the previous example have been chosen for this prototype, but might be modified basing on the needs of the manager and the parameters of the network. After the execution of these instructions, the network service must be restarted and its status must be checked:

```
$ sudo systemctl restart networking.service
$ sudo systemctl status networking.service
```

Installation of the environment

Once the preliminary actions are executed, the first thing to install is Apache2, the system which will offer the web application service [16].

```
$ sudo apt-get install apache2 -y
```

Now, the manager can choose whether to offer web application services using another new virtual host or to use the one initially set; anyway, the configuration file of the web application must be created or modified. For this project, the default space has been used:

```
$ sudo nano /etc/apache2/sites-available/default-ssl.conf
```

Inside this configuration file, manager should insert information about the RP such as the server admin mail, if one exists, the server name, the document root of the application. For this prototype, the only lines which have been modified, added, or which are needed to be checked are the following:

```
ServerName fido2service.strongkey.com
ServerAlias www.fido2service.strongkey.com
DocumentRoot /var/www/html
ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combined

SSLEngine on
SSLCertificateFile /etc/apache2/ssl/apache.pem
SSLCertificateKeyFile /etc/apache2/ssl/apache.key
```

Even in this case, the manager should set his own parameters.

After this phase, it is mandatory to add TLS to the system. Thereby doing this, the manager should first enable ssl modality for apache; then, it should generate RP certificate or request it to a certificate authority. In this project, a self-signed certificate has been created using *openssl* tool:

```
$ sudo a2enmod ssl
$ sudo mkdir /etc/apache2/ssl
$ sudo openssl req -new -x509 -days 365 -nodes -out /etc/apache2/ssl/apache.pem
-keyout /etc/apache2/ssl/apache.key
$ sudo chmod 644 /etc/apache2/ssl/apache.key
$ sudo chmod 644 /etc/apache2/ssl/apache.pem
```

Here are parameters used for this prototype in the creation of the certificate:

```
Country Name: IT
State of Province Name: Catania
Locality Name: Catania
Organization Name: fido2service
Organization Unit Name: Development
Common Name: fido2service.strongkey.com
```

The final step to configure Apache2 is to set the right server name in the Apache configuration file:

```
$ sudo nano /etc/apache2/apache2.conf
```

In the file just opened, the manager should append the following line, choosing as parameter the fully qualified domain name chosen for the server:

```
ServerName fido2service.strongkey.com
```

Now that configuration of Apache is set, the manager should first check if it is correct and, in case of positive result, it should enable the right site, disable the default one, and restart the service to let all the changes available:

```
$ apachectl configtest
$ sudo a2ensite default-ssl.conf
$ sudo a2dissite 000-default.conf
$ sudo systemctl restart apache2
```

After the complete installation of Apache, the manager must install a database in which RP will store information about users, transactions and other data needed. In this project, the database chosen has been an oracle one: since the operating system is a version of Debian, then MySQL is available only through the installation of MariaDB.

```
$ sudo apt-get install mariadb-server -y
$ sudo mysql_secure_installation
```

Afterwards, a user named *fido2service* should be created within the database using *fido* as password; then, the manager must create a database with *fido2service* name and he should grant to the newly created user all privileges on the newly created database. To automatically execute this entire process, the manager could run the *configuration.sql* file present in the software folder:

```
$ sudo su
# mysql
> source PATH_TO_SOFTWARE_FOLDER/utils/database_utils/configuration.sql
> exit
# exit
```

where *PATH_TO_SOFTWARE_FOLDER* is the path he has chosen where to mount the software directory.

As an alternative or in error cases, the manager can manually execute the commands:

```
$ sudo su
# mysql
> CREATE USER 'fido2service'@'localhost' IDENTIFIED BY 'fido';
> CREATE DATABASE fido2service;
> GRANT ALL PRIVILEGES ON fido2service.* TO 'fido2service'@'localhost';
> exit
# exit
```

All the parameters chosen here can be modified, but the manager should pay attention in this since these parameters are also used statically in the software of the web application. Once the database and its user have been created, the manager should create all the tables needed. As before, he can simply let the machine execute a configuration file:

```
$ cd PATH_TO_SOFTWARE_FOLDER/utils/database_utils
$ sudo mysql -u fido2service -p -h localhost fido2service < ./create.txt
```

If the manager wants to configure the database manually, he can do it following the instructions below:

```
$ sudo mysql -u fido2service -p -h localhost fido2service
> CREATE TABLE users(
> id int PRIMARY KEY,
> username text,
> first_name text,
> last_name text,
> display_name text);
> CREATE TABLE transactions(
> txid int PRIMARY KEY,
> txpayload text,
> username text,
> signature text,
> signerPublicKey text,
> signingKeyAlgorithm text,
> signingKeyType text,
> authenticatorData text,
> clientDataJson text);
> exit
```

Finally, PHP must be installed to let the application work, since it requires this software with all the needed modules, such as php-curl and php-mysql.

```
$ sudo apt-get install php libapache2-mod-php php-mcrypt php-mysql php-curl -y
```

Afterwards, the application files can be put in the document root folder so that the service can start to work, in the path:

```
/var/www/html/
```

DNS, certificates, and prototype issues solving

Now that all the components have been installed, there is the need to let them communicate each other. In this project, SKFS, web application and the user's device are put in the same private network and should obtain the IP addresses of the others. To let the system work, therefore, the implementation of a DNS server is recommended; nevertheless, since this project is a prototype, the DNS server has not been built. In substitution, the IP addresses of the FIDO2 server, which is static, as already described, is statically written in the *hosts* file of the operating system of Raspberry Pi card, so that to allow the communication between web application and SKFS.

```
$ sudo nano /etc/hosts
```

In the file just opened, the manager should append the following line, modifying the parameters basing on the actual IP address and the name of SKFS:

```
192.168.1.17 fido2server.strongkey.com
```

Since also the RP must be reachable, it has been needed to write its IP static address in the *hosts* file inside the operating system of the user's device used as test. In Windows 10 and Windows 11, this file can be found in the path

```
C:\Windows\System32\drivers\etc\
```

Another issue to be solved is the problem related to self-signed certificates. Since a TLS session is required by SKFS to work, it is mandatory for the web application to explicit the certificate it expects. To do this, the FIDO2 server certificate must be downloaded and inserted in the *certificate* folder of the web application as *fidoserver.pem*. After having done it, curl tool will

know that, even though the certificate is self-signed, it must consider it a valid one. To download the server certificate, the manager can simply start a browser instance and go to the SKFS. For this process, Mozilla Firefox has been used: once within FIDO2 server, it is possible to click in the padlock icon near the URL and select to view additional information about the certificate. In the window which will appear, it is possible to click the button *view certificate* and a new tab will open; here, in the *Download* section, the manage can download the certificate in the PEM format. Once this procedure has been completed, this file must be put as *fidoserver.pem* in the card, in the path:

```
/var/www/html/certificate/
```

Moreover, the same must be done on the contrary in case the FIDO2 server needs TLS client authentication: in this case, which is not the case of this project, the manager should obtain the certificate generated before and put it in the configuration folder of the FIDO2 server.

4.4 Interfaces and how to use the application

But the first homepage interface, all the interfaces expect the user to interact with them. The web application built in this project is not complex from the user's point of view, so this interaction is intuitive and simple.

4.4.1 Homepage

The Homepage interface (figure 4.10) shows some details about this project, such as the author's and supervisor's name and the title of the project.

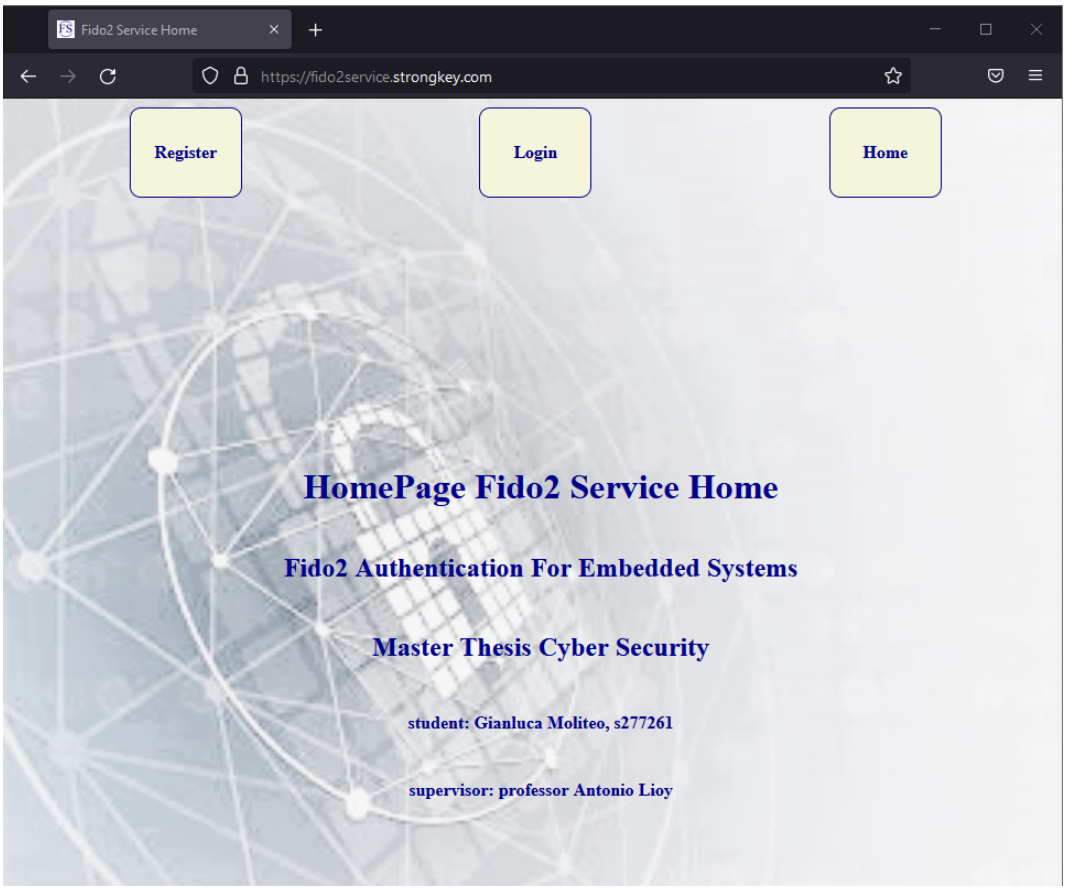


Figure 4.10. Interface of the Homepage of the web application, showing information about this project

In this first page, it is possible to note many things. First, the browser used to access to the web application has been Mozilla Firefox. Since the certificate is self-signed, a security warning should appear; thereby avoiding this, it has been necessary to change some settings of the browser. To do it, the manager should first write in the address bar

```
about:config
```

Once in the settings page, the manager should set the following flags:

```
security.insecure_field_warning.contextual.enabled = false
security.certerrors.permanentOverride = false
network.stricttransportsecurity.preloadlist = false
security.enterprise_roots.enabled = true
```

Afterwards, a restart of the browser is required. An important note is that this settings change is dangerous and has only been performed to show the prototype interfaces.

In case the certificate was requested to a Certificate Authority, this procedure is not needed.

The page shows three buttons in the top of the screen: Register, Login, and Home. They compose menu, links to other interfaces of the application, but the Home one that points to itself. Register button links to the registration interface, the page in which a user can register a new account. Login button points to the authentication interface, where a registered user can gain access to the web application.

4.4.2 Registration

In the Registration interface, figure 4.11, four input texts appear: a user willing to create his own account can fulfil them, entering his first name, last name, username, and display name, and then press the *Sign Up* button.

After the button has been clicked, the user has to wait a while for a prompt to appear, as figure 4.12 shows; this prompt is popped up thanks to WebAuthn APIs calls: it is the standardised way to communicate with authenticators. In the figure 4.12, the prompt asks for the embedded authenticator, in this case Windows Hello. If the user wants to be authenticated with a hardware key, he can click on the x top right of the prompt or press the *ESC* key in the keyboard so that to close the prompt; now, another window will appear asking to insert the key and to configure it (figure 4.13). Notice that this kinds of prompt, since they are the way the browser has to talk with authenticators, appear also in the Login interface and the Resource one; nevertheless, in those cases, the user cannot change between types of authenticators: the prompt will only be the one used in the registration phase.

4.4.3 Login

The Login page is shown in figure 4.14. Thanks to this page, registered users can gain the access to web application just by entering the username chosen in the registration phase and clicking on *Login*. Afterwards, a prompt appears and the user can utilise his authenticator thereby proving his identity. Then, if the right authenticator is used and the authentication task ends successfully, the user is automatically redirected to the Resource page.

4.4.4 Resource

As figure 4.15 shows, the Resource interface describes an example of resource the web application can provide to authenticated users. In this example, the unlocked service is the permission to buy a personal computer: this transaction can be started due to the *Buy* button. After having clicked it, the prompt used in the communication with the authenticator will appear and the transaction authentication can be performed.

In this interface, the user can also perform the deregister task, if willing to delete his own account. Thereby doing this, the user can click on *Deregister* button and nothing more: for this task, the authentication with the prompt does not happen. At the end, the session is destroyed, the account deleted and the user redirected to the Login page.

It is important to notice that here the menu items are changed. This happens because the user is now authenticated and can access to other pages. With this menu, the user can perform the logout, simply destroying the session, or go to Transactions interface.

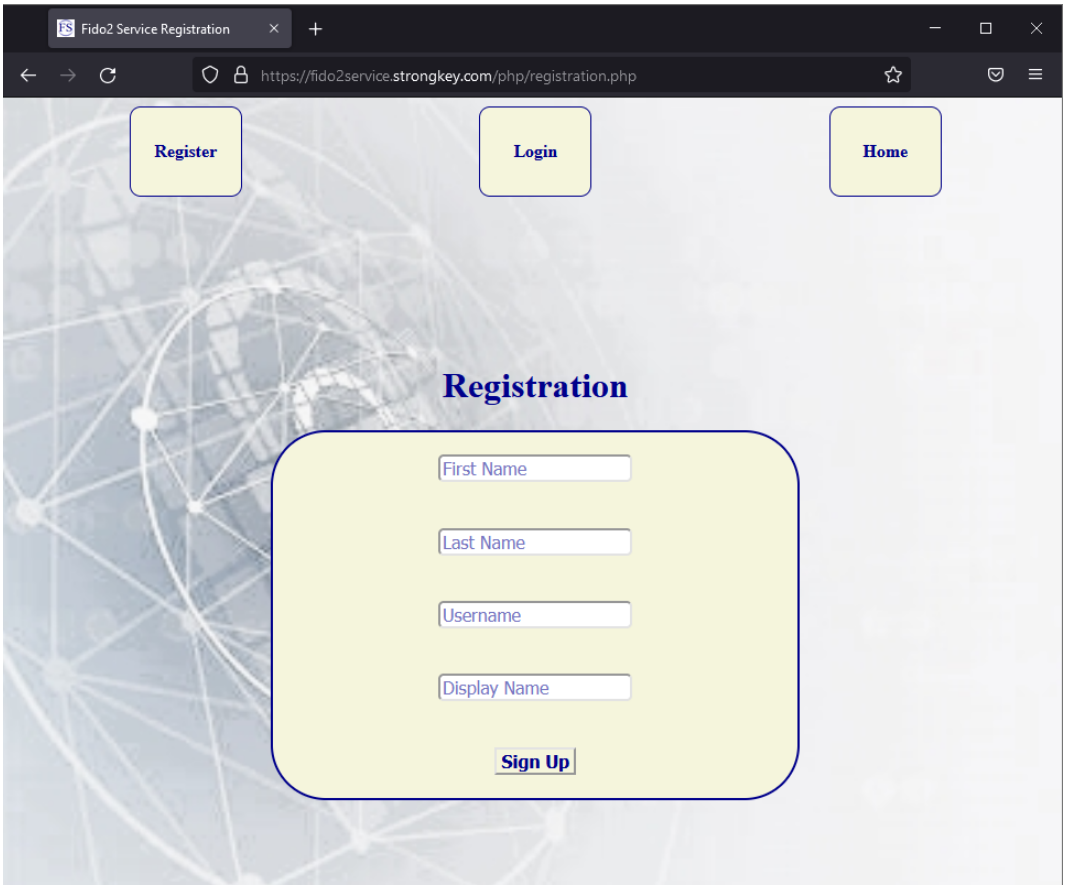


Figure 4.11. Interface of the Registration page of the web application, which can be used to create new accounts

4.4.5 Transactions

In the Transactions interface, an authenticated user can see the list of all the transactions he has approved (figure 4.16) or a simple string, in case no transaction has been authorised yet (figure 4.17). Every transaction is shown with some details:

txid the transaction identifier,

txpayload the summary of the transaction details,

key type the type of key and algorithm used to perform all the cryptographic functions needed for the transaction.

Near this information, the *View details* button is present: if the user presses it, a modal page will appear (figure 4.18) and the web application will show further details about the transaction, such as:

Signature the base64-encoded signature done at transaction authorisation phase,

Public key the base64-encoded public key associated to the secret one used to compute the signature,

Algorithm the algorithm used for the computation of the signature,

AuthenticatorData some base64-encoded data about the authenticator,

ClientDataJson some base64-encoded data about the platform.

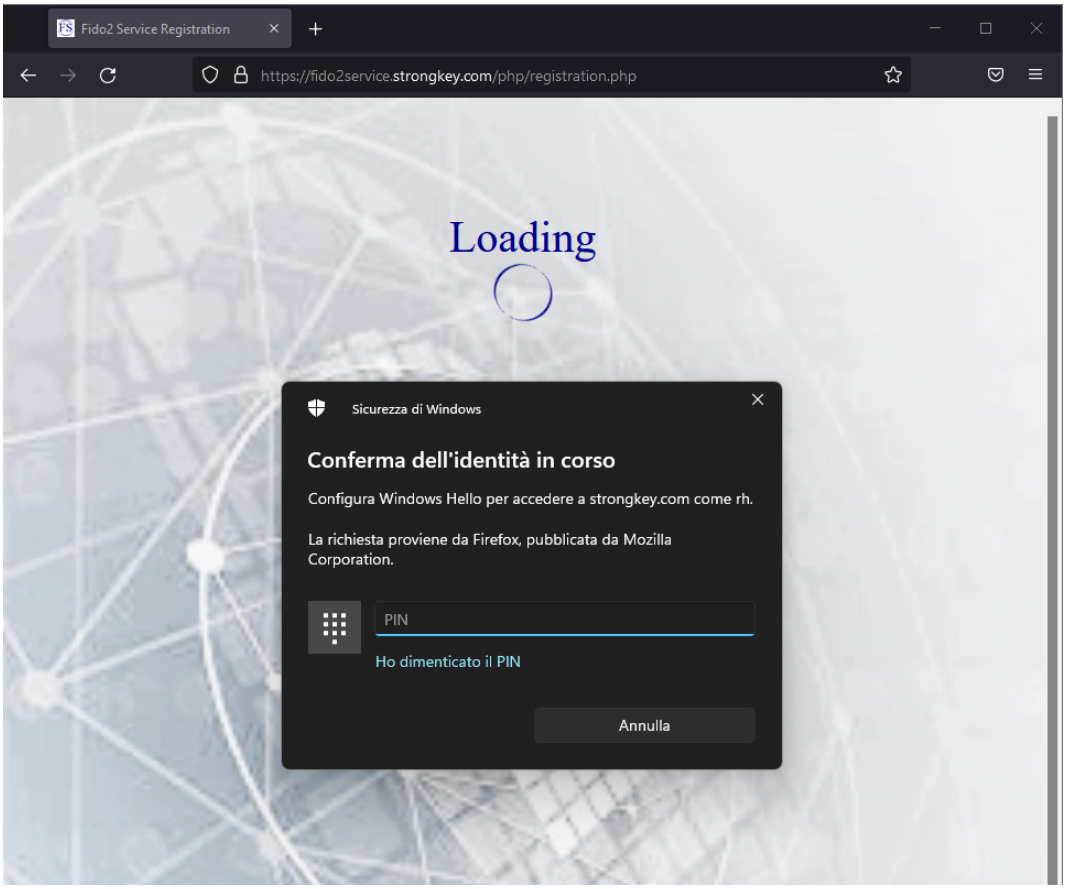


Figure 4.12. Figure showing the prompt popped up to permit the communication with Windows Hello

The user can go back to the list of transactions just by clicking anywhere. Moreover, if the transaction confirmation feature of the web application supports the algorithm used in the computation of the signature of this transaction, then a *Verify signature"* button will be present in this page: the user can press it thereby verifying the signature and, therefore, the validity of that transaction. Once the button has been clicked, an icon will appear: it will be a green check mark if the result is positive; it will be a red x if the result is negative, instead. In addition, as it is possible to see in figure 4.19, additional information about the validation will be shown. In particular,

data base64-encoded data used to compute the signature at authorisation phase,

challenge the base64-encoded challenge used by FIDO2 server to authenticate the user at authorisation phase,

crossOrigin boolean value representing if the platform has enabled cross origin authentication for this transaction authorisation,

origin the RP origin associated to the transaction,

type the WebAuthn API used to create "clientDataJSON" value, which contains some of these attributes and which is used to compute "data" attribute.

4.4.6 Admin

If the user is the admin, he will also have the authorisation to access to the Admin interface, figure 4.20. In this page, the admin can see the whole list of users and all the lists of transactions they have authorised. For each user, the whole information is reported:

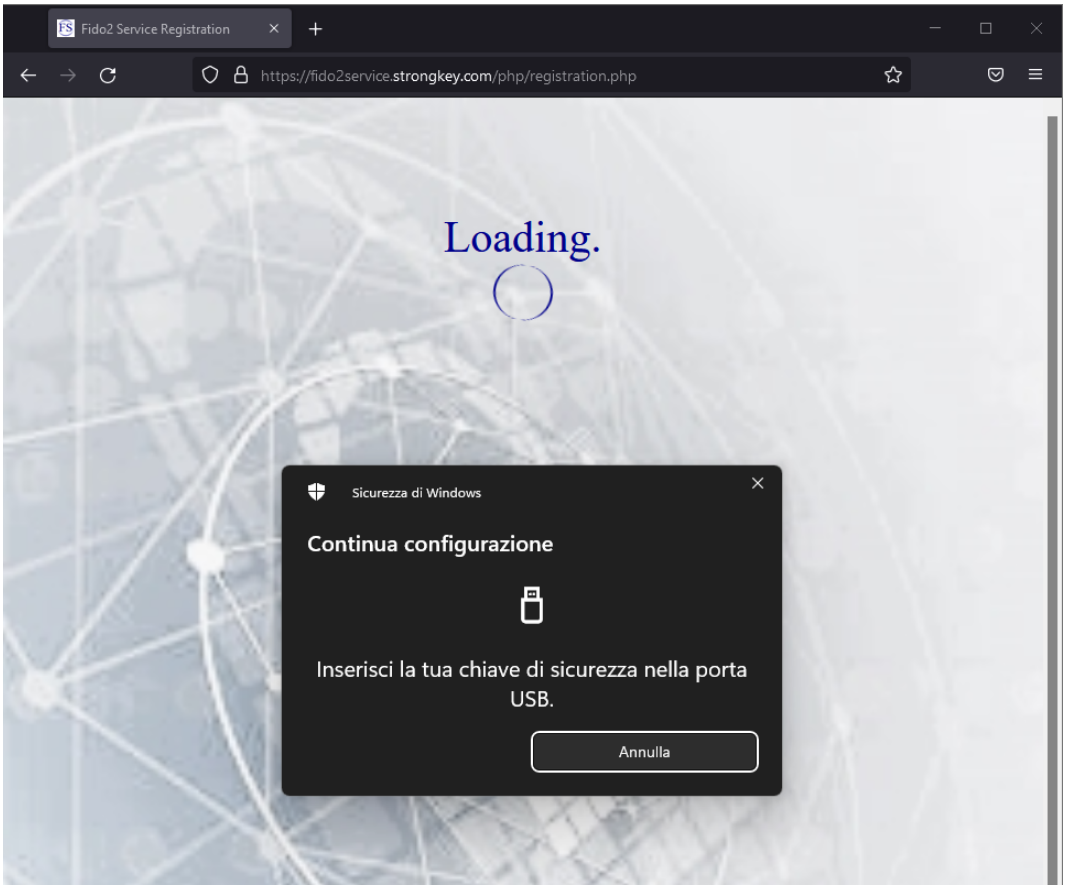


Figure 4.13. Figure showing the prompt popped up to permit the communication with hardware security keys

- id,
- username,
- first name,
- last name,
- display name.

As well as the Transactions interface, this page allows the user, that in this case is the admin, to access to transactions details by clicking *View details* buttons, as figure 4.21 shows; here, if the algorithm is supported, the admin can check the validity of the transaction even though it has been authorised by other users (figure 4.22).

Notice that if the user is the admin, menu changes again thereby adding the *Admin* button linking to this interface.

4.4.7 session, redirect and loading

As it is possible to notice, two types of users can navigate in the web application: authenticated users and non-authenticated ones. The difference, from the web application point of view, is the presence of the authentication session: a user who has performed the login phase obtains this session and, thanks to it, he can be recognized within the application server.

These two different kinds of users can access to two different groups of pages, which have in common only the Homepage, viewable by both authenticated and non-authenticated users. In practice, a user who does not have an active session can only access to the Registration and Login pages, which are needed to obtain the session it does not have yet. On the contrary, a user who

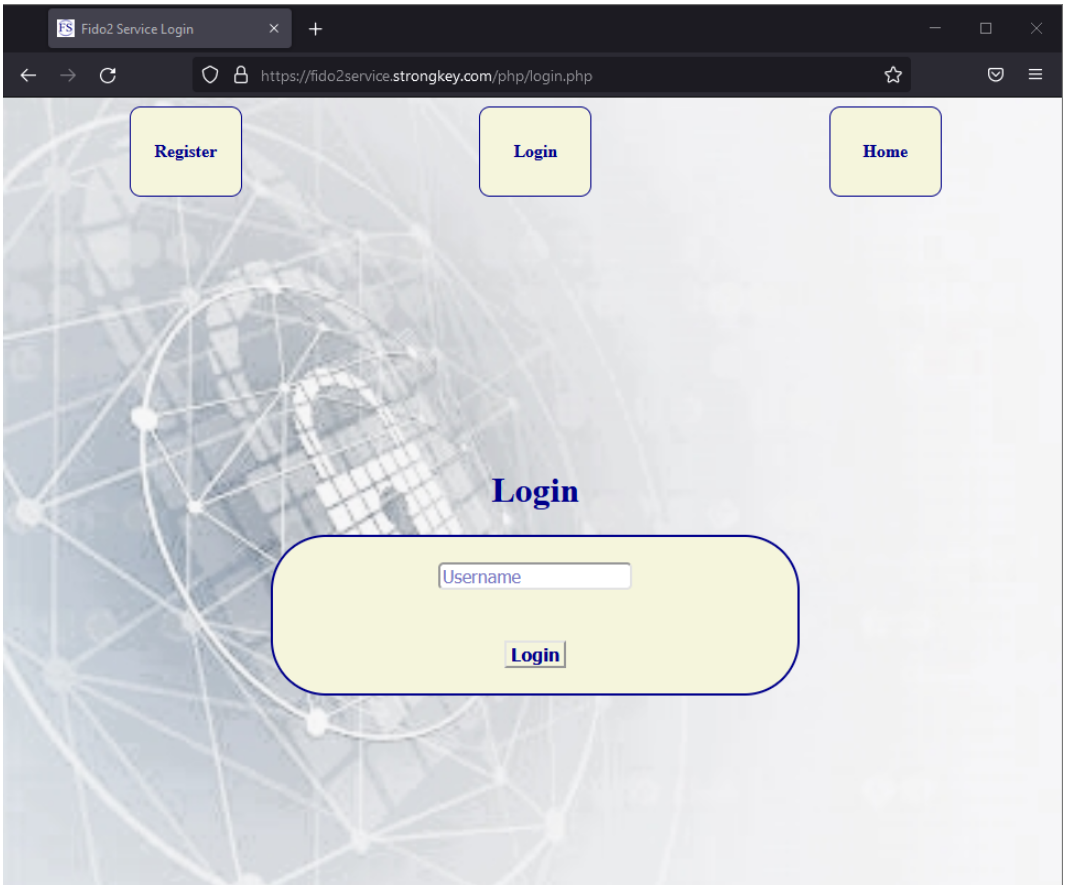


Figure 4.14. Interface of the Login page of the web application, which can be used to gain access to the resource of the web application

has been provided with a session can access to the Resource and the Transactions pages and, if he is the admin, also to the Admin interface.

If a kind of user tries to access to a page of the other group, the request is rejected. This happens even with an authenticated user: if he tries to access to the Login page, for instance, the request is not valid since he cannot be authenticated twice and cannot have two sessions. As a result, in these cases, the user is redirected to the Resource page if he is an authenticated user or to the Login interface if he is a non-authenticated one.

It is also possible to notice that every time the web application calls an asymmetric function and, therefore, can make the user wait for the answer, the interface is hidden and a small and moving loading icon appears.

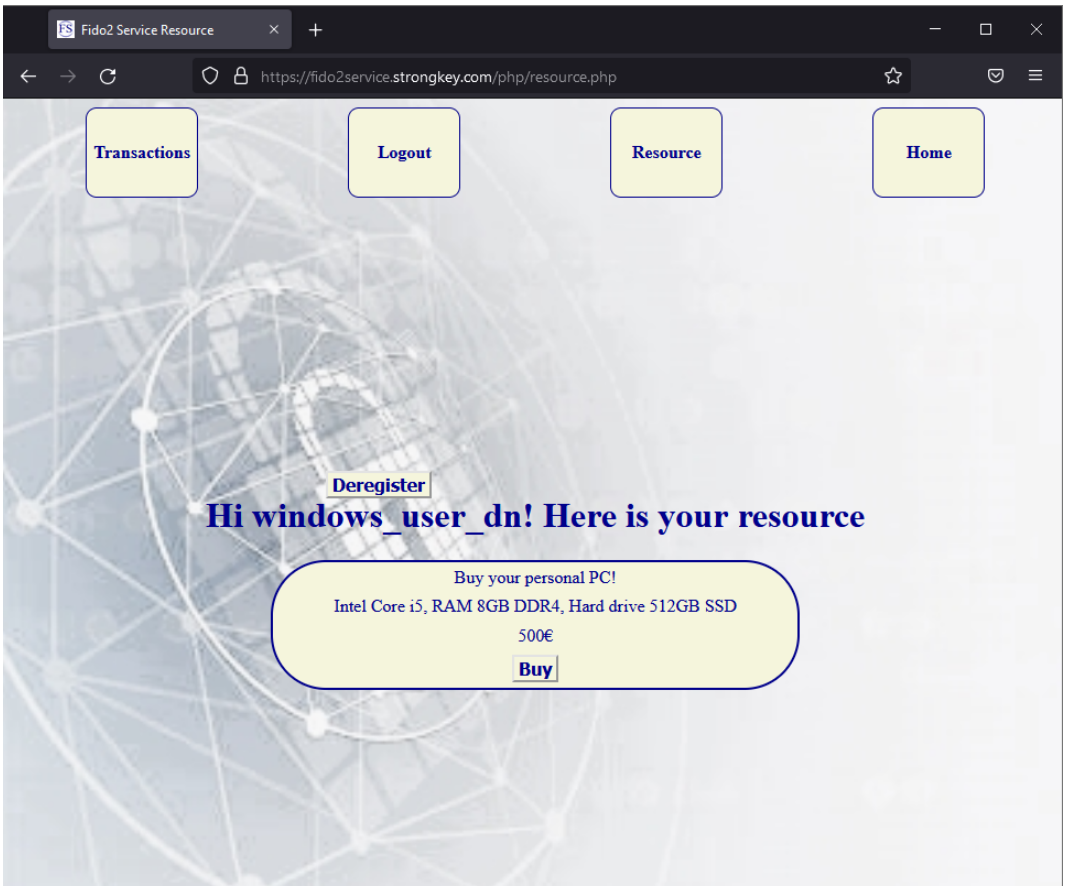


Figure 4.15. Interface of the Resource page of the web application, which can be used to start transactions and delete the account

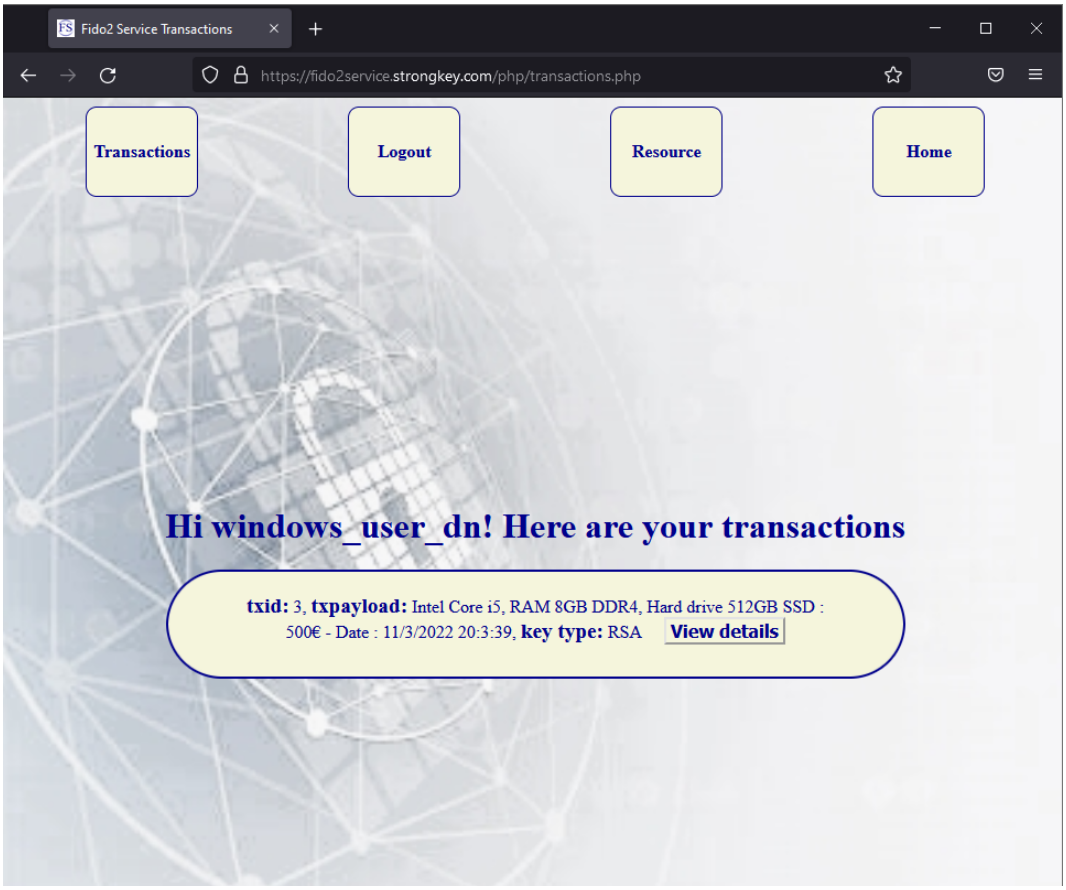


Figure 4.16. Interface of the Transactions page of the web application, which shows the list of the transactions approved

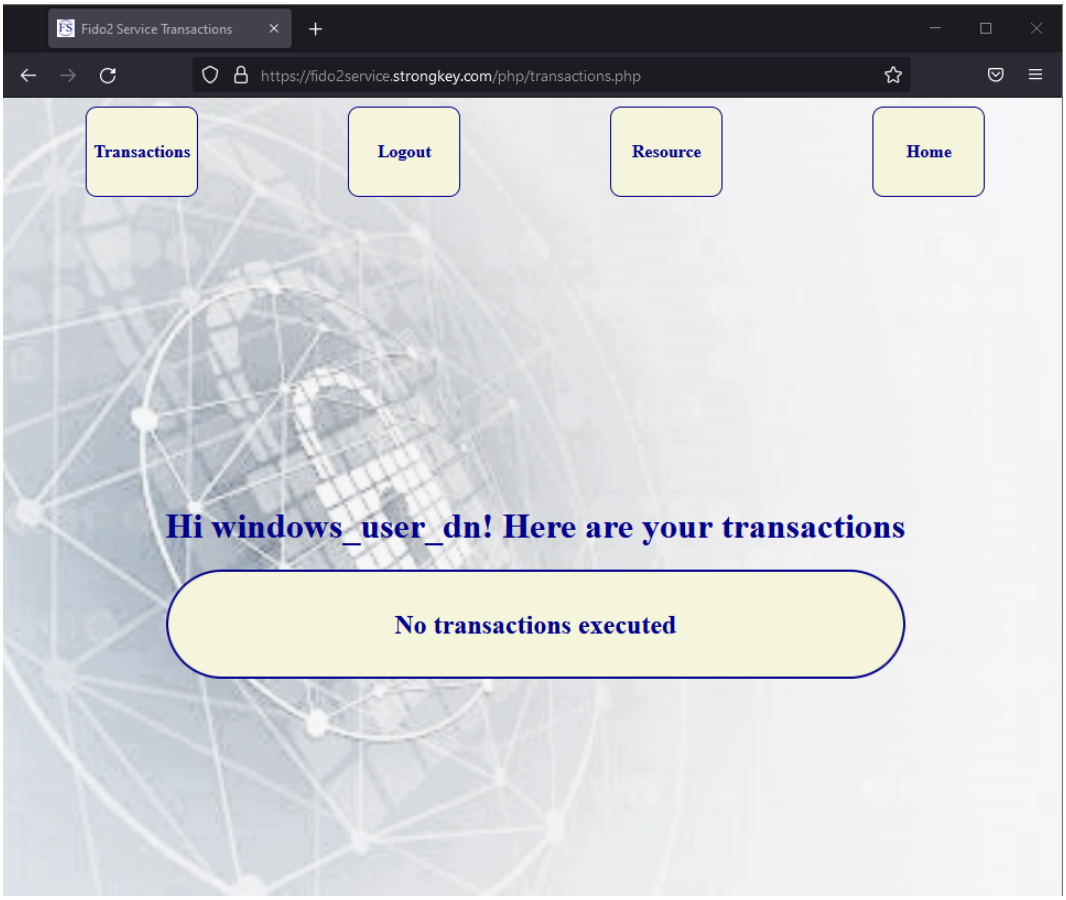


Figure 4.17. Interface of the Transactions page of the web application whether no transaction has been approved

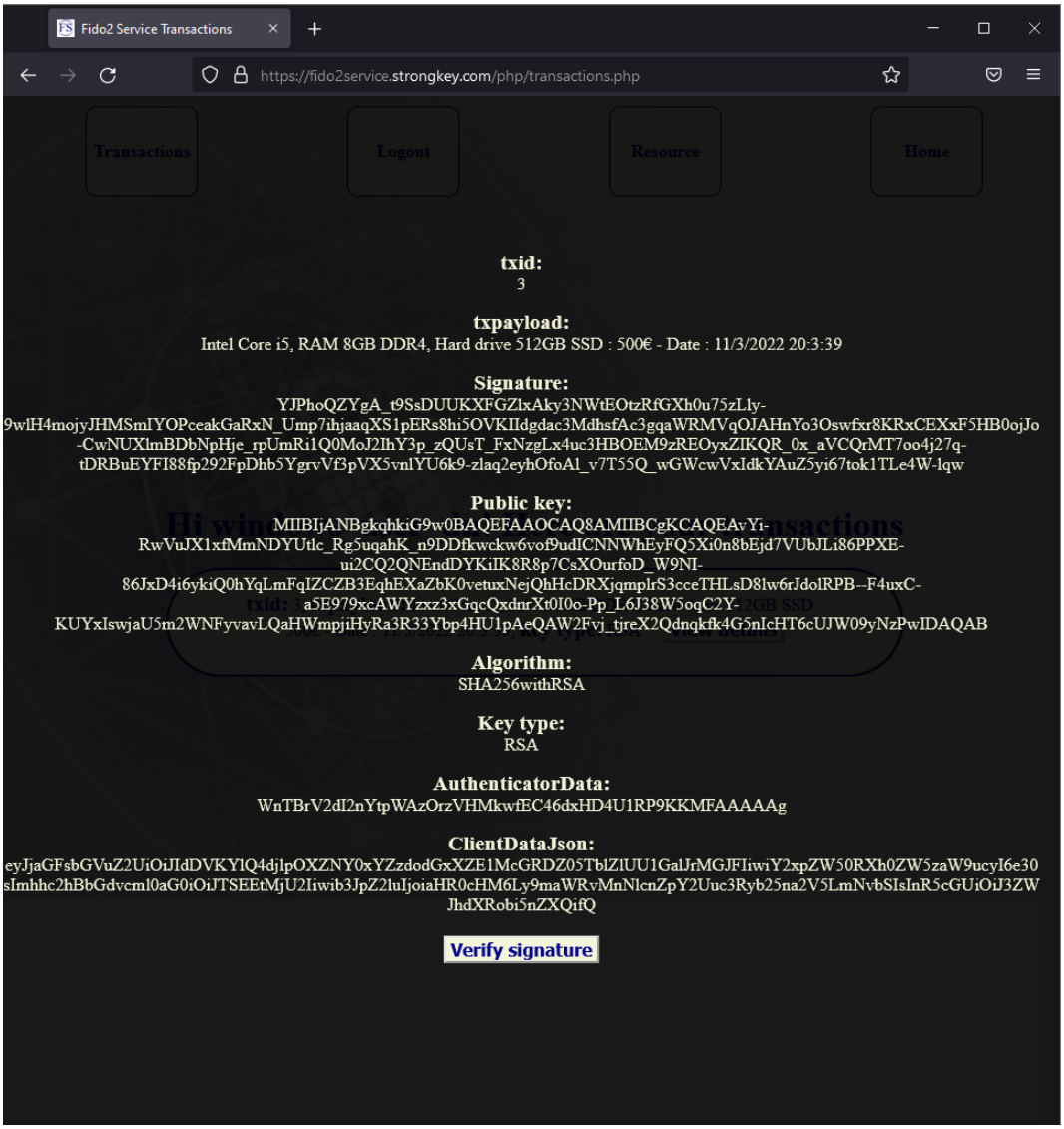


Figure 4.18. The modal view of details of a transaction in the Transactions page

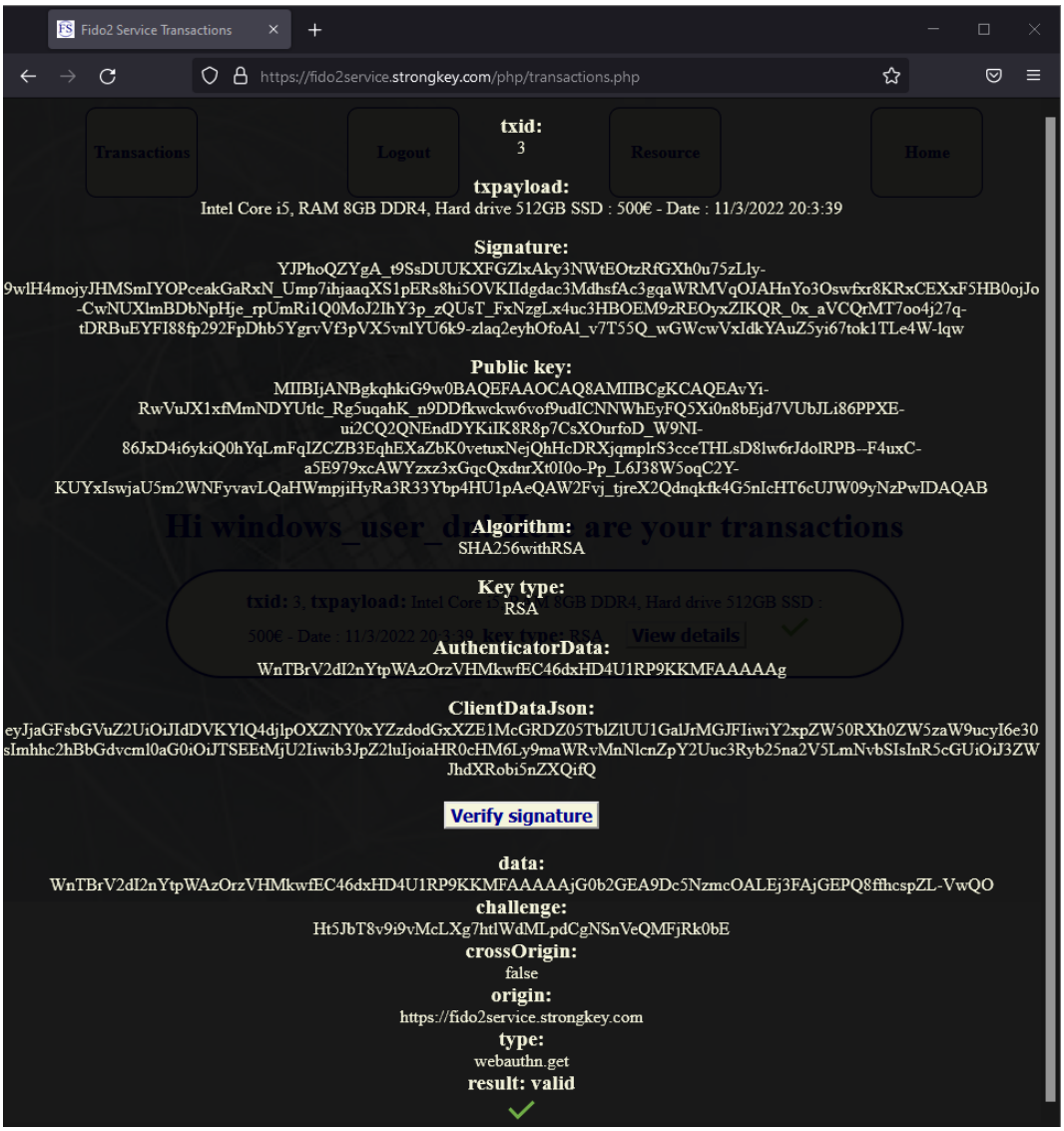


Figure 4.19. The modal view of details of a transaction in the Transactions page once the validity has been checked

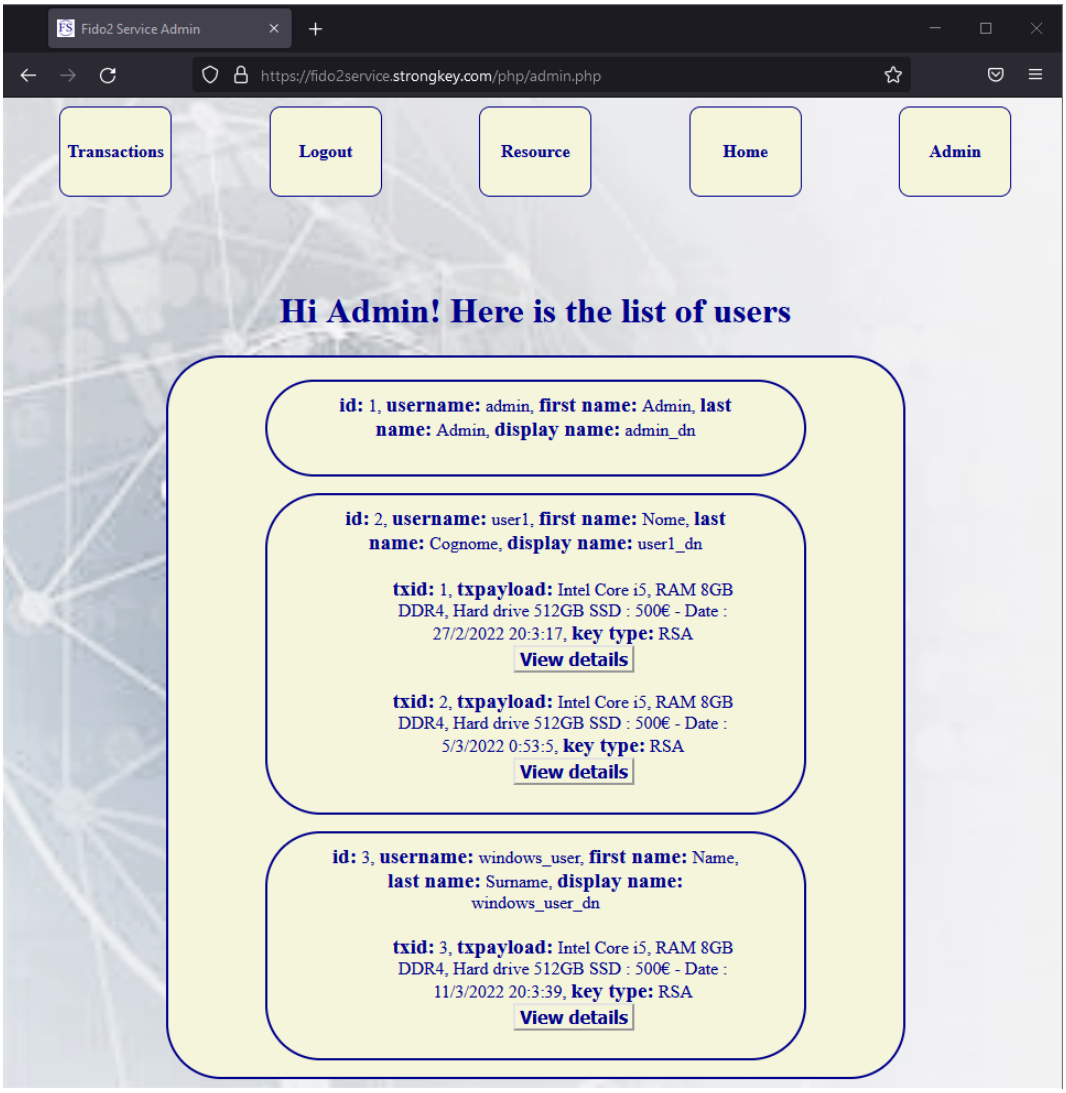


Figure 4.20. Interface of the Admin page of the web application, which is accessible only by the admin and shows the list of users and their transactions

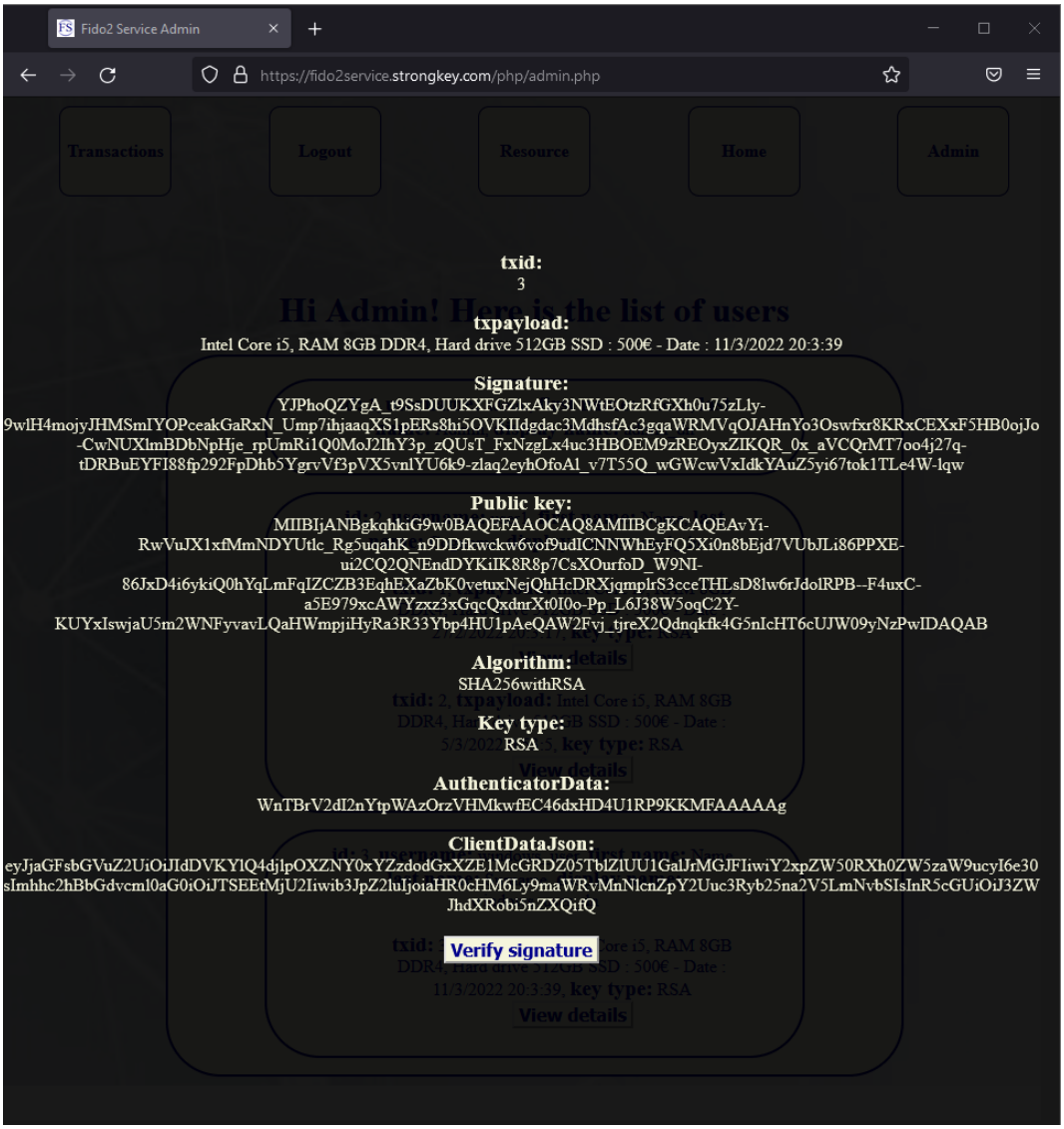


Figure 4.21. The modal view of details of a transaction in the Admin page

Figure 4.22. The modal view of details of a transaction in the Admin page once the validity has been checked

Chapter 5

Tests

Thereby testing this project, it is possible to see and analyse packets sent and received by the web application. To do this, Wireshark software [17] has been used. To clean the test, the IP address filter has been applied, as well as the one regarding the packets protocol, put as TLS.

Moreover, a simple log system has been implemented in the software of the web application: every FIDO2 Server API call is stored in a log file, such as every other action like requesting the access to a page or logging out. This system is useful since it easily tracks the process flow as well as the data flow.

5.1 Connection to the web application

Initially, the user connects to the web application using a browser. In this step, it is possible to assure the usage of TLS protocol. In this case, as figure 5.1 shows, the version 1.3 of TLS has been used by apache. In this figure, TLS messages like Client Hello, Server hello, Change Cipher Spec are shown: these messages are used to perform TLS handshake.

No.	Time	Source	Destination	Protocol	Length	Info
528	36.597704	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
539	36.617107	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
540	36.617107	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
543	36.617573	192.168.1.44	192.168.1.16	TLSv1.3	84	Change Cipher Spec, Application Data

Figure 5.1. Messages in initial connection phase

The static IP address owned by the web application is 192.168.1.16, while the IP address 192.168.1.44 is used by the user's device; finally, 192.168.1.17 is the address of SKFS.

It is possible to have the assurance the Homepage has been requested through the log file, which is showing:

```
Page viewed at 05/03/2022 15:09:48, path: /var/www/html/index.php,
   page: Homepage, username: Not authenticated
```

5.2 Registration

In this phase, it is possible to see web application and FIDO2 server communicating. First, platform sends to the web application server messages containing information entered by the user in the registration form, as figure 5.2 shows; then the preregister endpoint is called. The TLS version used between the two servers is 1.2, since SKFS does not support TLS 1.3 (figure 5.3). After the handshake has been performed, different Application Data messages are exchanged between the two actors to perform the preregister task. At the end of this step, some messages are then sent from the web application to the user's device, representing the response of the endpoint, as it is possible to see in figure 5.4.

In the platform, now, the browser will perform the createCredentials WebAuthn API, asking the user to use an authenticator. At the end of this step, data is sent as response from the

No.	Time	Source	Destination	Protocol	Length	Info
82	7.183731	192.168.1.44	192.168.1.16	TLSv1.3	685	Client Hello
85	7.188213	192.168.1.16	192.168.1.44	TLSv1.3	294	Server Hello, Change Cipher Spec, Application Data, Application Data
86	7.188457	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
88	7.188576	192.168.1.44	192.168.1.16	TLSv1.3	1448	Application Data
90	7.188603	192.168.1.44	192.168.1.16	TLSv1.3	161	Application Data
93	7.189737	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

Figure 5.2. Messages in first step of registration phase

No.	Time	Source	Destination	Protocol	Length	Info
98	7.202196	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
101	7.206822	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
104	7.211860	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
105	7.226956	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
107	7.228663	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
108	7.228663	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
109	7.228663	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
110	7.228663	192.168.1.16	192.168.1.17	TLSv1.2	171	Application Data
111	7.229859	192.168.1.16	192.168.1.17	TLSv1.2	345	Application Data
114	7.232298	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
116	7.232512	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
118	7.233475	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
123	7.280613	192.168.1.17	192.168.1.16	TLSv1.2	125	Application Data
125	7.280828	192.168.1.17	192.168.1.16	TLSv1.2	724	Application Data
128	7.283479	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert

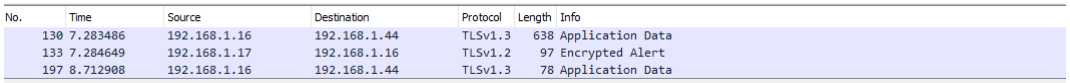
Figure 5.3. Messages in second step of registration phase

browser to the web application server (figure 5.5). It can now use this information to call the register SKFS endpoint, as figure 5.6 shows. Here, many messages are exchanged between the two servers. When the web application server receives the complete response, it sends the result to the browser, figure 5.7.

Even though this task ends here, the messages exchanged between client and server continue, since the user is automatically redirected to the Login page: all the data needed to load this page is here requested.

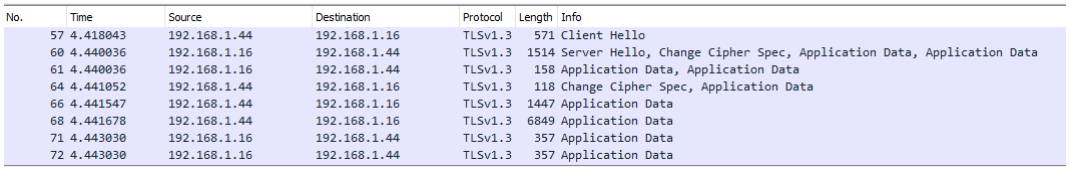
Opening the log file now, it is possible to see the appended actions performed by the web application system:

```
Page viewed at 05/03/2022 15:10:02, path: /var/www/html/php/registration.php,
   page: Registration, username: Not authenticated
Curl executed at 05/03/2022 15:10:10,
   path: https://fido2server.strongkey.com/skfs/rest/preregister
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{
         "username":"user2","displayname":"user2_dn",
         "options":{"attestation":"direct"},"extensions":"{}"}}
   Response: {
      "Response":{
         "rp":{"name":"FIDOServer","id":"strongkey.com"},
         "user":{"name":"user2",
            "id":"vZPP6ho6FFgm5O43URA2BB8pc4AjQcEsl-my6LSQUfM",
            "displayName":"user2_dn"},
         "challenge":"JIsp0m8uc-All_Y4UsjqeQ",
         "pubKeyCredParams":[
            {"type":"public-key","alg":-7},{"type":"public-key","alg":-35},
            {"type":"public-key","alg":-36},{"type":"public-key","alg":-8},
            {"type":"public-key","alg":-47},{"type":"public-key","alg":-257},
            {"type":"public-key","alg":-258},{"type":"public-key","alg":-259},
            {"type":"public-key","alg":-37},{"type":"public-key","alg":-38},
            {"type":"public-key","alg":-38}],
          "excludeCredentials":[],"attestation":"direct"}}
Curl executed at 05/03/2022 15:10:15,
```



No.	Time	Source	Destination	Protocol	Length	Info
130	7.283486	192.168.1.16	192.168.1.44	TLSv1.3	638	Application Data
133	7.284649	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert
197	8.712908	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.4. Messages in third step of registration phase



No.	Time	Source	Destination	Protocol	Length	Info
57	4.418043	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
60	4.440036	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
61	4.440036	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
64	4.441052	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
66	4.441547	192.168.1.44	192.168.1.16	TLSv1.3	1447	Application Data
68	4.441678	192.168.1.44	192.168.1.16	TLSv1.3	6849	Application Data
71	4.443030	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
72	4.443030	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

Figure 5.5. Messages in fourth step of registration phase

```
   path: https://fido2server.strongkey.com/skfs/rest/register
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{
         "strongkeyMetadata":{
            "version":"1.0","create_location":"Catania, CT","username":"user2",
            "origin":"https://fido2service.strongkey.com"},
      "publicKeyCredential":{
         "id":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
         "rawId":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
         "response":{
            "attestationObject":"o2NmbXRjdHBtZ2F0dFNOYDVROTAQH...",
            "clientDataJSON":"eyJ0eXBlIjoid2ViYXV0aG4uY3JlYXRlIiwiY...",
            "type":"public-key"}}}
   Response: {"Response":"Successfully processed registration response"}
Page viewed at 05/03/2022 15:10:15, path: /var/www/html/php/login.php,
   page: Login, username: Not authenticated
```

First, the Registration page is requested; so far, the user is not recognized yet. Now, curl is performed twice, first calling the preregister API and then calling the register API. Here, request data and response data are shown. If the response of the last endpoint is positive, the user is redirected to the login page, so that he can request the authentication task.

5.3 Login

When the user wants to be authenticated, he goes to the login interface. Here, as already explained, he enters his username and the web application server is called. As it is possible to see in figure 5.8, the first kind of messages exchanged is of TLS 1.3 type between the browser and the Relying Party. In particular, the client sends the information about the username to the server, which can now proceed, as figure 5.9 shows, to call the preauthenticate endpoint of SKFS using the 1.2 version of TLS. When the RP receives the answer by the FIDO2 server, containing the challenge to be solved, it forwards this result to the platform (figure 5.10).

After the exchange of these messages, the user is called to provide the solution of the challenge using his authenticator: this step is performed thanks to WebAuthn APIs. Afterwards, the response of the authenticator is sent by the client to the web application server (figure 5.11). Then, the authenticate API of SKFS is called and some messages are exchanged between FIDO2 and web application servers, as viewable in figure 5.12. Afterwards, the result computed by the StrongKey Fido Server is sent from the web application server to the client: in case of success, the user is redirected to the Resource interface and, due to this, some messages are exchanged to allow the display and the operation of this page (figure 5.13).

No.	Time	Source	Destination	Protocol	Length	Info
79	4.451384	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
82	4.470355	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
85	4.475447	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
86	4.486981	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
88	4.488807	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
89	4.488807	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
90	4.488807	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
91	4.488807	192.168.1.16	192.168.1.17	TLSv1.2	170	Application Data
94	4.489856	192.168.1.16	192.168.1.17	TCP	1514	48086 → 8181 [ACK] Seq=899 Ack=1307 Win=64128 Len=1448 TSval=2181329888
98	4.489856	192.168.1.16	192.168.1.17	TLSv1.2	858	Application Data
101	4.508623	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
103	4.508961	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
105	4.509887	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
108	4.575313	192.168.1.17	192.168.1.16	TLSv1.2	125	Application Data
110	4.575554	192.168.1.17	192.168.1.16	TLSv1.2	163	Application Data
113	4.584964	192.168.1.16	192.168.1.44	TLSv1.3	428	Application Data
114	4.584970	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert
118	4.586306	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert

Figure 5.6. Messages in fifth step of registration phase

No.	Time	Source	Destination	Protocol	Length	Info
167	4.608187	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
172	4.609191	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
173	4.609519	192.168.1.44	192.168.1.16	TLSv1.3	1379	Application Data
178	4.609901	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
181	4.612547	192.168.1.16	192.168.1.44	TLSv1.3	1514	Application Data
182	4.612547	192.168.1.16	192.168.1.44	TLSv1.3	1079	Application Data, Application Data
185	4.625958	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
186	4.625958	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
189	4.626361	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
191	4.627021	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
192	4.627021	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
195	4.627265	192.168.1.44	192.168.1.16	TLSv1.3	84	Change Cipher Spec, Application Data
199	4.627547	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
200	4.627547	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
229	9.617727	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data
264	11.247416	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.7. Messages in sixth step of registration phase

The log file can offer a different vision of what happens here:

```
Page viewed at 05/03/2022 15:10:15, path: /var/www/html/php/login.php,
   page: Login, username: Not authenticated
Curl executed at 05/03/2022 15:10:20,
   path: https://fido2server.strongkey.com/skfs/rest/preauthenticate
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{"username":"user2","options":{}}}
   Response: {
      "Response":{
         "challenge":"wgsMaONwuUvjPsm_qoqg6A",
         "allowCredentials":[{"type":"public-key",
            "id":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew","alg":-257}],
         "rpId":"strongkey.com"}}
Curl executed at 05/03/2022 15:10:24,
   path: https://fido2server.strongkey.com/skfs/rest/authenticate
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{
         "strongkeyMetadata":{
            "version":"1.0","last_used_location":"Catania, CT","username":"user2",
            "origin":"https://fido2service.strongkey.com",
            "clientUserAgent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
               AppleWebKit/537.36 (KHTML, like Gecko)
               Chrome/98.0.4758.102 Safari/537.36"},
```

No.	Time	Source	Destination	Protocol	Length	Info
26	1.938391	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
29	1.960572	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
30	1.960572	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
33	1.960941	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
35	1.961110	192.168.1.44	192.168.1.16	TLSv1.3	1465	Application Data
38	1.963021	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
39	1.963021	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

Figure 5.8. Messages in first step of authentication phase

No.	Time	Source	Destination	Protocol	Length	Info
46	1.971119	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
49	1.975441	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
52	1.981293	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
53	1.987713	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
55	1.989338	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
56	1.989338	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
57	1.989338	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
58	1.989338	192.168.1.16	192.168.1.17	TLSv1.2	174	Application Data
59	1.989338	192.168.1.16	192.168.1.17	TLSv1.2	280	Application Data
62	2.000191	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
64	2.000627	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
66	2.001532	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
67	2.041085	192.168.1.17	192.168.1.16	TLSv1.2	126	Application Data
69	2.041343	192.168.1.17	192.168.1.16	TLSv1.2	283	Application Data
72	2.042467	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert
73	2.043168	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert

Figure 5.9. Messages in second step of authentication phase


```
            "publicKeyCredential":{
               "id":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
               "rawId":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
               "response":{
                  "authenticatorData":"WnTBrV2dI2nYtpWAzOrzVHMkwfEC46d...",
                  "clientDataJSON":"eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY2hhbG..."},
               "type":"public-key"}}}
   Response: {
      "Response":"Successfully processed sign response",
      "jwt":"eyJhbGciOiJFUzI1NiIsIng1YyI6Ii0tLS0tQkVHSU4gQ0VSVElGSUN..."}
Page viewed at 05/03/2022 15:10:24, path: /var/www/html/php/resource.php,
   page: Resource, username: user2
```

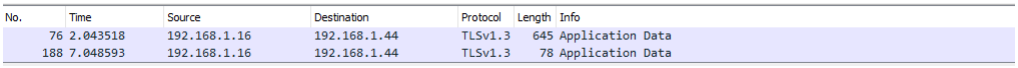

After having requested the access to the Login page, a first curl action is performed to the FIDO2 server preauthenticate endpoint and a second one is executed to the FIDO2 server authenticate API. In this case, the user willing to be authenticated is "user2". At the end of the task, if successful, the user is redirected to Resource page: here, it is possible to see that the session is established since the username is recognized.

5.4 Authorisation

In the authorisation phase, the user accepts to start a transaction. In this case, the transaction is just an example: the purchase of a personal computer. As figure 5.14 shows, the task starts with messages exchanged between client and server: the content of these messages is about transaction details, but does not regard username, since the web application server will obtain it using the active session. With this information, the Relying Party contacts SKFS using the preauthorize endpoint, like described in figure 5.15. After a while, the RP receives the challenge that it expects the user to solve. Therefore, this answer is sent to the platform, like figure 5.16 shows.

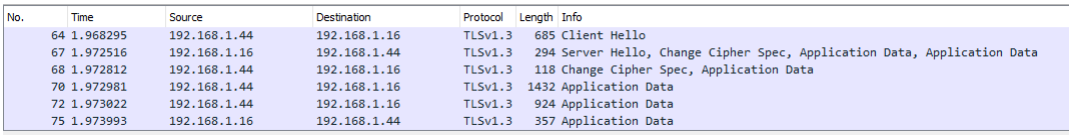
After the user has solved the challenge, the result of the computation is sent by the web application client to the server (figure 5.17). Now, RP, using data received in addition to the username provided due to the session, calls the authorize API, as schematized in figure 5.18. After these messages have been exchanged between servers, the web application one sends the result to the platform, figure 5.19.

Even for the authorisation task, watching the log file can offer a different and more detailed view of the action performed:



No.	Time	Source	Destination	Protocol	Length	Info
76	2.043518	192.168.1.16	192.168.1.44	TLSv1.3	645	Application Data
188	7.048593	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

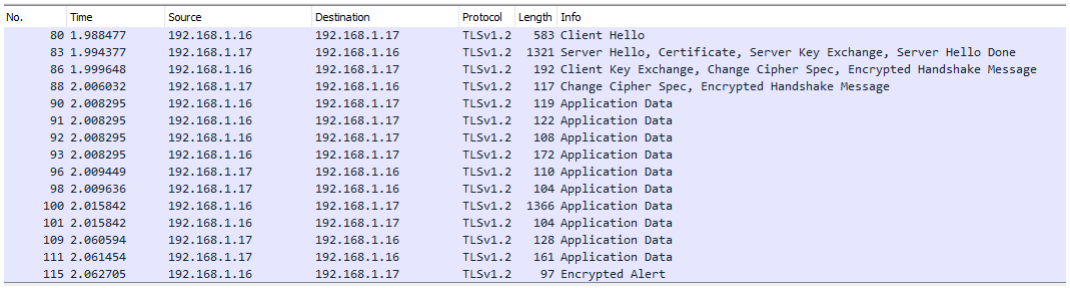
Figure 5.10. Messages in third step of authentication phase



No.	Time	Source	Destination	Protocol	Length	Info
64	1.968295	192.168.1.44	192.168.1.16	TLSv1.3	685	Client Hello
67	1.972516	192.168.1.16	192.168.1.44	TLSv1.3	294	Server Hello, Change Cipher Spec, Application Data, Application Data
68	1.972812	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
70	1.972981	192.168.1.44	192.168.1.16	TLSv1.3	1432	Application Data
72	1.973022	192.168.1.44	192.168.1.16	TLSv1.3	924	Application Data
75	1.973993	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

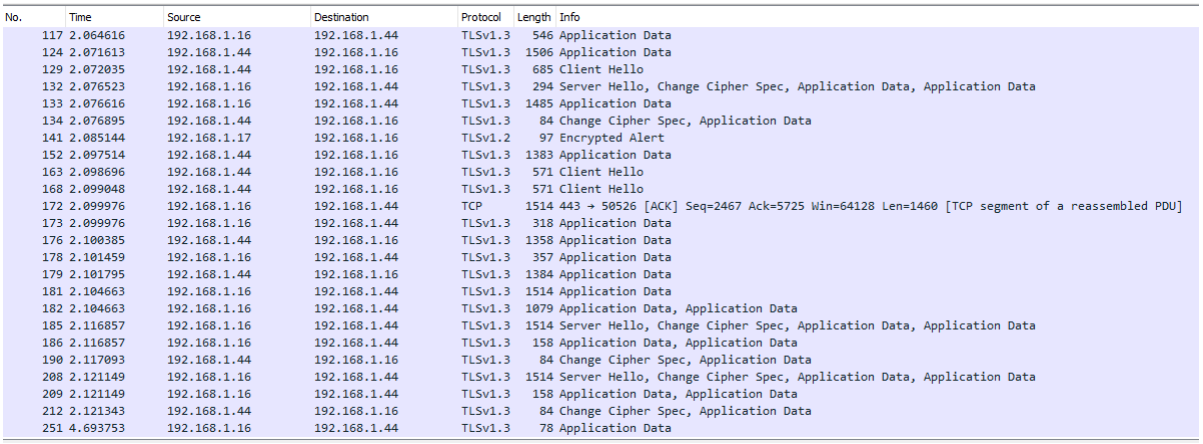
Figure 5.11. Messages in fourth step of authentication phase

```
Page viewed at 05/03/2022 15:10:24, path: /var/www/html/php/resource.php,
   page: Resource, username: user2
Curl executed at 05/03/2022 15:10:31,
   path: https://fido2server.strongkey.com/skfs/rest/preauthorize
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{
         "username":"user2","txid":"4",
         "txpayload":"Intel Core i5, RAM 8GB DDR4, Hard drive 512GB SSD : 500eur
            - Date : 5/3/2022 15:10:31",
         "options":{}}}
   Response: {
      "Response":{
         "challenge":"fKrJJ5AGJBELF6qeEvVKMTISKA29HfCkL53zSUbPnBo",
         "allowCredentials":[{
            "type":"public-key",
            "id":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew","alg":-257}],
         "txid":"4",
         "txpayload":"Intel Core i5, RAM 8GB DDR4, Hard drive 512GB SSD : 500eur
         - Date : 5/3/2022 15:10:31","rpId":"strongkey.com"}}
Curl executed at 05/03/2022 15:10:34,
   path: https://fido2server.strongkey.com/skfs/rest/authorize
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{
         "txid":"4",
         "txpayload":"Intel Core i5, RAM 8GB DDR4, Hard drive 512GB SSD : 500eur
            - Date : 5/3/2022 15:10:31",
         "strongkeyMetadata":{
            "version":"1.0","last_used_location":"Catania, CT","username":"user2",
            "origin":"https://fido2service.strongkey.com",
            "clientUserAgent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
               AppleWebKit/537.36 (KHTML, like Gecko)
               Chrome/98.0.4758.102 Safari/537.36"},
         "publicKeyCredential":{
            "id":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
            "rawId":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
            "response":{
               "authenticatorData":"WnTBrV2dI2nYtpWAzOrzVHMkwfEC46dxH...",
```



No.	Time	Source	Destination	Protocol	Length	Info
80	1.988477	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
83	1.994377	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
86	1.999648	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
88	2.006032	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
90	2.008295	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
91	2.008295	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
92	2.008295	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
93	2.008295	192.168.1.16	192.168.1.17	TLSv1.2	172	Application Data
96	2.009449	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
98	2.009636	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
100	2.015842	192.168.1.16	192.168.1.17	TLSv1.2	1366	Application Data
101	2.015842	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
109	2.060594	192.168.1.17	192.168.1.16	TLSv1.2	128	Application Data
111	2.061454	192.168.1.17	192.168.1.16	TLSv1.2	161	Application Data
115	2.062705	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert

Figure 5.12. Messages in fifth step of authentication phase



No.	Time	Source	Destination	Protocol	Length	Info
117	2.064616	192.168.1.16	192.168.1.44	TLSv1.3	546	Application Data
124	2.071613	192.168.1.44	192.168.1.16	TLSv1.3	1506	Application Data
129	2.072035	192.168.1.44	192.168.1.16	TLSv1.3	685	Client Hello
132	2.076523	192.168.1.16	192.168.1.44	TLSv1.3	294	Server Hello, Change Cipher Spec, Application Data, Application Data
133	2.076616	192.168.1.16	192.168.1.44	TLSv1.3	1485	Application Data
134	2.076895	192.168.1.44	192.168.1.16	TLSv1.3	84	Change Cipher Spec, Application Data
141	2.085144	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert
152	2.097514	192.168.1.44	192.168.1.16	TLSv1.3	1383	Application Data
163	2.098696	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
168	2.099048	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
172	2.099976	192.168.1.16	192.168.1.44	TCP	1514	443 → 50526 [ACK] Seq=2467 Ack=5725 Win=64128 Len=1460 [TCP segment of a reassembled PDU]
173	2.099976	192.168.1.16	192.168.1.44	TLSv1.3	318	Application Data
176	2.100385	192.168.1.44	192.168.1.16	TLSv1.3	1358	Application Data
178	2.101459	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
179	2.101795	192.168.1.44	192.168.1.16	TLSv1.3	1384	Application Data
181	2.104663	192.168.1.16	192.168.1.44	TLSv1.3	1514	Application Data
182	2.104663	192.168.1.16	192.168.1.44	TLSv1.3	1079	Application Data, Application Data
185	2.116857	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
186	2.116857	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
190	2.117093	192.168.1.44	192.168.1.16	TLSv1.3	84	Change Cipher Spec, Application Data
208	2.121149	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
209	2.121149	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
212	2.121343	192.168.1.44	192.168.1.16	TLSv1.3	84	Change Cipher Spec, Application Data
251	4.693753	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.13. Messages in sixth step of authentication phase


```
            "clientDataJSON":"eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY2hhbGxlb..."},
            "type":"public-key"}}}
Response: {
   "Response":"Successfully processed authorization response",
   "txdetail":{
      "txid":"4",
      "txpayload":"Intel Core i5, RAM 8GB DDR4, Hard drive 512GB SSD : 500eur
         - Date : 5/3/2022 15:10:31",
      "nonce":"-z1TsRQSKF4P-lAQdLJSBA","txtime":1646489431061,
      "challenge":"fKrJJ5AGJBELF6qeEvVKMTISKA29HfCkL53zSUbPnBo"},
   "FIDOAuthenticatorReferences":[{
      "protocol":"FIDO2_0",
      "id":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
      "rawId":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
      "userHandle":"vZPP6ho6FFgm5O43URA2BB8pc4AjQcEsl-my6LSQUfM",
      "rpId":"strongkey.com",
      "authenticatorData":"WnTBrV2dI2nYtpWAzOrzVHMkwfEC46dxHD4U1R...",
      "clientDataJSON":"eyJ0eXBlIjoid2ViYXV0aG4uZ2V0IiwiY2hhbGxlbmdlIjoi...",
      "aaguid":"08987058-cadc-4b81-b6e1-30de50dcbe96",
      "authorizationTime":1646489434755,"uv":true,"up":true,
      "signerPublicKey":"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCA...",
      "signature":"oSCDOp2vABWKa1ItD9u63hqv2fGzrHNSEp9hyFfb5P0Tz7...",
      "usedForThisTransaction":true,"signingKeyType":"RSA",
      "signingKeyAlgorithm":"SHA256withRSA"}]}
```


First, the Resource interface is requested and shown. To execute the authorisation task, a first call to preauthorize API of FIDO2 server is needed: the request data will contain information

No.	Time	Source	Destination	Protocol	Length	Info
105	4.485563	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
108	4.508960	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
109	4.508960	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
112	4.509952	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
114	4.510372	192.168.1.44	192.168.1.16	TLSv1.3	1445	Application Data
116	4.510458	192.168.1.44	192.168.1.16	TLSv1.3	150	Application Data
118	4.510904	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
119	4.510904	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

Figure 5.14. Messages in first step of authorisation phase

No.	Time	Source	Destination	Protocol	Length	Info
55	1.812753	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
58	1.819632	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
61	1.824902	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
62	1.829307	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
64	1.831071	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
65	1.831071	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
66	1.831071	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
67	1.831071	192.168.1.16	192.168.1.17	TLSv1.2	172	Application Data
70	1.832114	192.168.1.16	192.168.1.17	TLSv1.2	366	Application Data
71	1.833245	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
73	1.833461	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
75	1.834599	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
76	1.854824	192.168.1.17	192.168.1.16	TLSv1.2	126	Application Data
78	1.855006	192.168.1.17	192.168.1.16	TLSv1.2	388	Application Data
81	1.856143	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert

Figure 5.15. Messages in second step of authorisation phase

about the newly transaction to authorise, such as the identifier and the payload. Then, another curl is executed to the authorize endpoint: besides of the result of the task and the information regarding the transaction, the response of this final call offers data about the authenticator and the process of generation of the proof; these data are needed to perform transaction confirmation tasks.

5.5 Transactions and Admin

When visiting the Transactions or Admin interfaces, the user wants to access to data contained in the web application server database. Since here no verification is required by FIDO2 Server, there is not exchange of messages with it; nevertheless, the size of information requested can be considerable and therefore the number of packets sent by the Relying Party may be quite big, especially in Admin page, where it has to be loaded information about all transactions of all users registered in the web application.

In support of this, in the log file it is possible to find simply the request of access to Transactions or Admin pages:

```
Page viewed at 05/03/2022 15:10:25, path: /var/www/html/php/transactions.php,
   page: Transactions, username: user2
Page viewed at 05/03/2022 15:10:53, path: /var/www/html/php/admin.php,
   page: Admin, username: admin
```

To test the validity of a transaction, a tester can use information contained in the log file, under the response of a curl executed to the *authorize* endpoint.

To perform this verification, many tools can be used, such as *openssl*. It is crucial to notice that, thereby executing this type of test, the tester should manually compose data as FIDO2 specifications impose [7].

In this project, a verification tool has been implemented. To use it, the tester should connect to the web application in the path */utils/verifysignature/verifySignature.html*.

As figure 5.20 shows, this tool needs as input

authenticatorData information about the authenticator used, needed to generate data the signature is computed on;

clientDataJson information about the client, needed to generate data the signature is computed on;

No.	Time	Source	Destination	Protocol	Length	Info
82	1.857275	192.168.1.16	192.168.1.44	TLSv1.3	726	Application Data
84	1.869723	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert
135	3.784739	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.16. Messages in third step of authorisation phase

No.	Time	Source	Destination	Protocol	Length	Info
85	6.362368	192.168.1.44	192.168.1.16	TLSv1.3	685	Client Hello
88	6.366520	192.168.1.16	192.168.1.44	TLSv1.3	294	Server Hello, Change Cipher Spec, Application Data, Application Data
89	6.366837	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
91	6.366991	192.168.1.44	192.168.1.16	TLSv1.3	1443	Application Data
93	6.367021	192.168.1.44	192.168.1.16	TLSv1.3	907	Application Data
96	6.368550	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

Figure 5.17. Messages in fourth step of authorisation phase

signerPublicKey public key associated to the private one used to compute the signature at authorisation step;

signature signature computed at authorisation step;

signingKeyAlgorithm algorithm used in the computation of the signature.

Once the tester has provided the required information, he can press the *Verify* button thereby triggering the execution of the algorithm. At the end, the response is presented as a string and an icon (figure 5.21).

5.6 Logout

When user performs the logout task, it wants to exit from its account destroying the session. In this case, the messages exchanged, as figure 5.22 shows, are sent and received only by and to web application server and client: since neither user verification nor change in user's account must be performed, FIDO2 server is not needed here. First, messages about the user's willing are sent by the platform to the server; it can now destroy the active session initialized before and redirect the user to the Login page.

When the logout action is requested and performed, the web application logs the action:

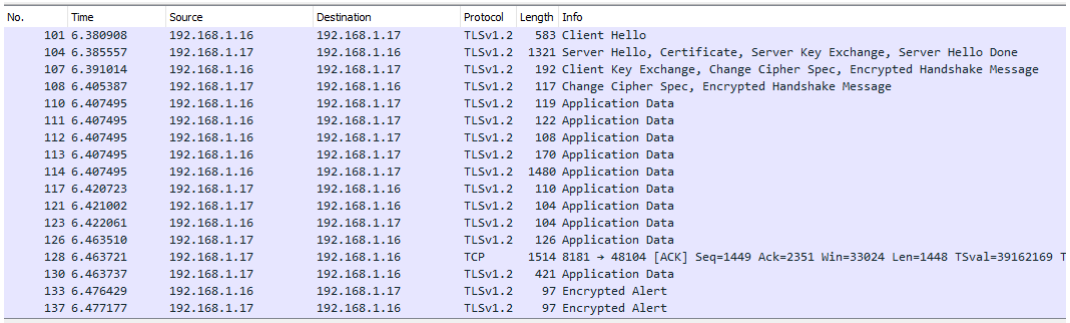
```
Logout at 05/03/2022 15:10:44, path: /var/www/html/php/api/logout.php,
   username: user2
Page viewed at 05/03/2022 15:10:44, path: /var/www/html/php/login.php,
   page: Login, username: Not authenticated
```

5.7 Deregistration

In this phase, the user chooses to delete his account from both the Relying Party and the FIDO2 Server. When this task starts, the client tells the web application server the user's willing as described in figure 5.23. Afterwards, this server takes the username from the session it has established before and, using it, proceeds to call SKFS endpoints. Figure 5.24 shows a long sequence of messages: this is because two FIDO2 Server APIs are called in a raw. First, getkeysinfo is called to let RP obtain the identifier of user's key within SKFS; then, the web application server calls deregister endpoint using the newly information as input to delete user's account. It is possible to see in figure 5.24, indeed, two different TLS sessions starting with two different handshakes.

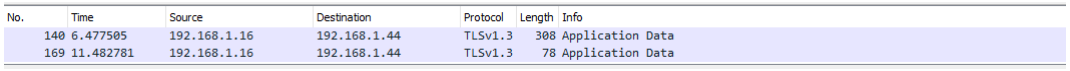
Afterwards, the result of the second endpoint is forwarded to the client, as figure 5.25 shows. Since the user's account does not exist anymore, the session created before has to be destroyed and, therefore, the authenticated user turns to a non-authenticated user; for this reason, it is redirected to the Login interface, which requires some data from the web application server to be loaded. Figure 5.25 shows this behaviour, too.

It is possible to use the logging system to see details about the process of deregistration task:



No.	Time	Source	Destination	Protocol	Length	Info
101	6.380908	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
104	6.385557	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
107	6.391014	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
108	6.405387	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
110	6.407495	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
111	6.407495	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
112	6.407495	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
113	6.407495	192.168.1.16	192.168.1.17	TLSv1.2	170	Application Data
114	6.407495	192.168.1.16	192.168.1.17	TLSv1.2	1480	Application Data
117	6.420723	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
121	6.421002	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
123	6.422061	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
126	6.463510	192.168.1.17	192.168.1.16	TLSv1.2	126	Application Data
128	6.463721	192.168.1.17	192.168.1.16	TCP	1514	8181 → 48104 [ACK] Seq=1449 Ack=2351 Win=33024 Len=1448 TSval=39162169 T
130	6.463737	192.168.1.17	192.168.1.16	TLSv1.2	421	Application Data
133	6.476429	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert
137	6.477177	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert

Figure 5.18. Messages in fifth step of authorisation phase



No.	Time	Source	Destination	Protocol	Length	Info
140	6.477505	192.168.1.16	192.168.1.44	TLSv1.3	308	Application Data
169	11.482781	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.19. Messages in sixth step of authorisation phase

```
Curl executed at 05/03/2022 15:11:15,
   path: https://fido2server.strongkey.com/skfs/rest/getkeysinfo
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{"username":"user2"}}
   Response: {
      "Response":{
         "keys":[{
            "keyid":"1-1-8","fidoProtocol":"FIDO2_0",
            "credentialId":"v_doEGtTFZbxOuVnBmImxcCYx6erzcqsA_DyG6SLUew",
            "createLocation":"Catania, CT","createDate":1646489415000,
            "lastusedLocation":"Catania, CT","modifyDate":1646489474000,
            "status":"Active","displayName":"user2_dn","attestationFormat":"tpm"}]}}
Curl executed at 05/03/2022 15:11:15,
   path: https://fido2server.strongkey.com/skfs/rest/deregister
   Request: {
      "svcinfo":{
         "did":1,"protocol":"FIDO2_0","authtype":"PASSWORD",
         "svcusername":"fido2service","svcpassword":"F1d02s3rv1c3_p4ssw0rd!"},
      "payload":{"keyid":"1-1-8"}}
   Response: {"Response":"Successfully deleted"}
Page viewed at 05/03/2022 15:11:15,
   path: /var/www/html/php/login.php, page: Login, username: Not authenticated
```

This sequence of logs shows a first curl action executed to FIDO2 server getkeysinfo endpoint: this is needed to obtain information about the key used by the user. Once this information has been received, a second FIDO2 server API is called, the deregister one; in this call, data obtained in the first curl action are used. If the entire task ends successfully, the session is destroyed, the used becomes a non-authenticated user and it is redirected to the Login page.

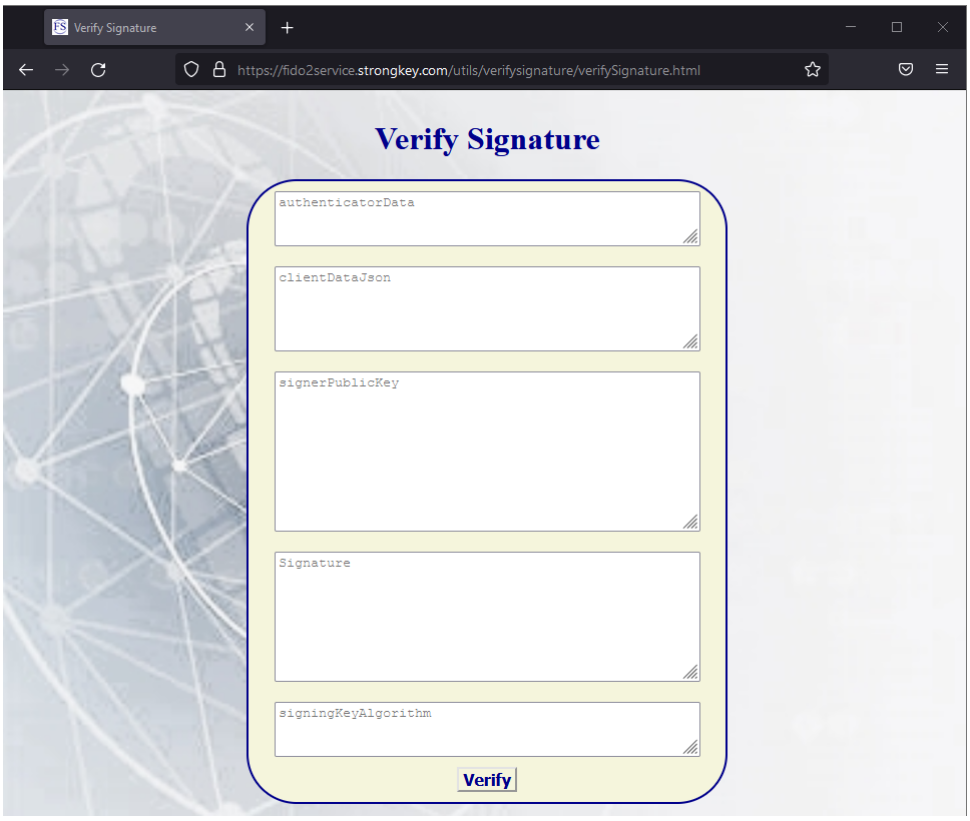


Figure 5.20. Interface of the tool verifying the validity of transactions signatures

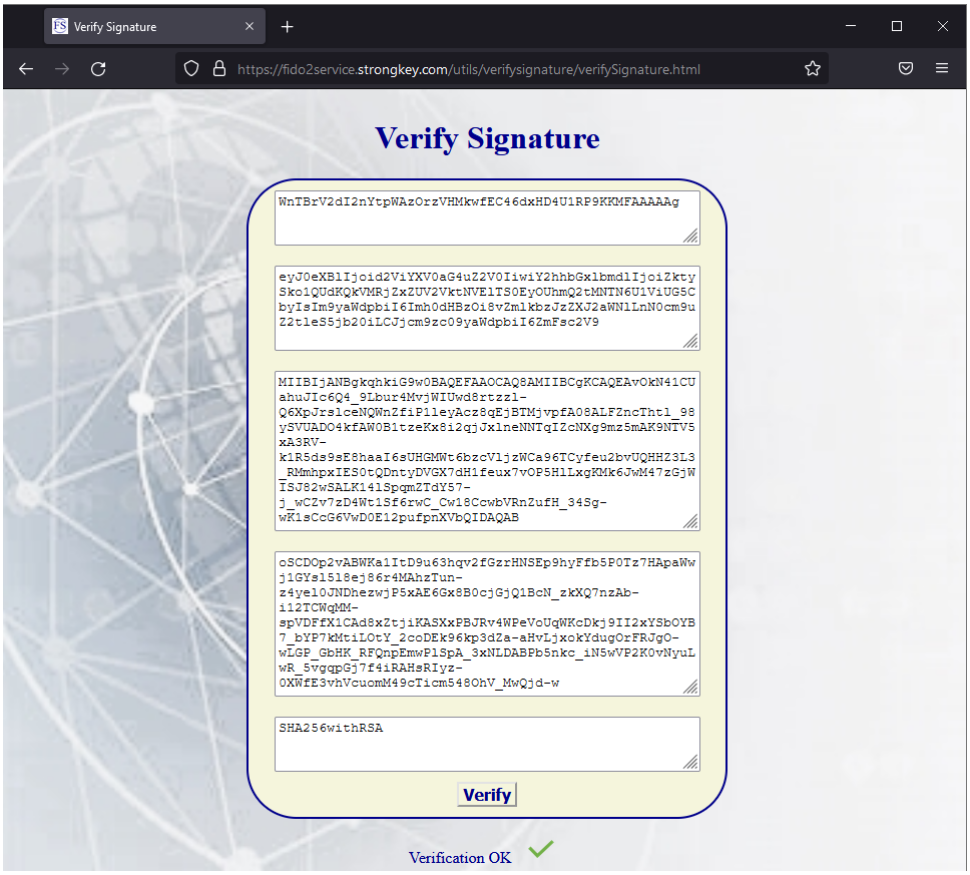


Figure 5.21. Interface of the tool after the verification of a transaction signature

No.	Time	Source	Destination	Protocol	Length	Info
50	2.129045	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
59	2.150685	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
60	2.150685	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
63	2.150986	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
65	2.151138	192.168.1.44	192.168.1.16	TLSv1.3	1531	Application Data
67	2.152593	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
68	2.152593	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
71	2.162143	192.168.1.16	192.168.1.44	TLSv1.3	418	Application Data
72	2.163897	192.168.1.44	192.168.1.16	TLSv1.3	1526	Application Data
81	2.169346	192.168.1.16	192.168.1.44	TLSv1.3	1060	Application Data
84	2.182553	192.168.1.44	192.168.1.16	TLSv1.3	1377	Application Data
92	2.183839	192.168.1.44	192.168.1.16	TLSv1.3	685	Client Hello
95	2.185470	192.168.1.16	192.168.1.44	TLSv1.3	1461	Application Data
96	2.186343	192.168.1.44	192.168.1.16	TLSv1.3	1352	Application Data
98	2.186768	192.168.1.16	192.168.1.44	TLSv1.3	294	Server Hello, Change Cipher Spec, Application Data, Application Data
99	2.187026	192.168.1.16	192.168.1.44	TLSv1.3	84	Change Cipher Spec, Application Data
103	2.187431	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
107	2.191318	192.168.1.44	192.168.1.16	TLSv1.3	1379	Application Data
109	2.194325	192.168.1.16	192.168.1.44	TLSv1.3	1514	Application Data
110	2.194325	192.168.1.16	192.168.1.44	TLSv1.3	1079	Application Data, Application Data
162	6.517163	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.22. Messages in logout phase

No.	Time	Source	Destination	Protocol	Length	Info
74	11.455047	192.168.1.44	192.168.1.16	TLSv1.3	685	Client Hello
77	11.459346	192.168.1.16	192.168.1.44	TLSv1.3	294	Server Hello, Change Cipher Spec, Application Data, Application Data
78	11.459600	192.168.1.44	192.168.1.16	TLSv1.3	118	Change Cipher Spec, Application Data
80	11.459724	192.168.1.44	192.168.1.16	TLSv1.3	1410	Application Data
83	11.461284	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data

Figure 5.23. Messages in first step of deregistration phase

No.	Time	Source	Destination	Protocol	Length	Info
88	11.482572	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
91	11.486835	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
94	11.492283	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
95	11.496333	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
97	11.498296	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
98	11.498296	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
99	11.498296	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
100	11.498296	192.168.1.16	192.168.1.17	TLSv1.2	172	Application Data
101	11.499499	192.168.1.16	192.168.1.17	TLSv1.2	267	Application Data
106	11.506583	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
108	11.506760	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
110	11.507741	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
113	11.568742	192.168.1.17	192.168.1.16	TLSv1.2	126	Application Data
115	11.568939	192.168.1.17	192.168.1.16	TLSv1.2	418	Application Data
118	11.570234	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert
122	11.570846	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert
132	11.574362	192.168.1.16	192.168.1.17	TLSv1.2	583	Client Hello
135	11.577444	192.168.1.17	192.168.1.16	TLSv1.2	1321	Server Hello, Certificate, Server Key Exchange, Server Hello Done
138	11.582222	192.168.1.16	192.168.1.17	TLSv1.2	192	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
139	11.585459	192.168.1.17	192.168.1.16	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
141	11.586992	192.168.1.16	192.168.1.17	TLSv1.2	119	Application Data
142	11.586992	192.168.1.16	192.168.1.17	TLSv1.2	122	Application Data
143	11.586992	192.168.1.16	192.168.1.17	TLSv1.2	108	Application Data
144	11.586992	192.168.1.16	192.168.1.17	TLSv1.2	170	Application Data
145	11.586992	192.168.1.16	192.168.1.17	TLSv1.2	264	Application Data
148	11.588173	192.168.1.17	192.168.1.16	TLSv1.2	110	Application Data
150	11.588314	192.168.1.17	192.168.1.16	TLSv1.2	104	Application Data
152	11.589063	192.168.1.16	192.168.1.17	TLSv1.2	104	Application Data
153	11.619595	192.168.1.17	192.168.1.16	TLSv1.2	125	Application Data
155	11.619755	192.168.1.17	192.168.1.16	TLSv1.2	139	Application Data
158	11.630633	192.168.1.16	192.168.1.17	TLSv1.2	97	Encrypted Alert
160	11.630991	192.168.1.17	192.168.1.16	TLSv1.2	97	Encrypted Alert

Figure 5.24. Messages in second step of deregistration phase

No.	Time	Source	Destination	Protocol	Length	Info
190	11.642951	192.168.1.16	192.168.1.44	TLSv1.3	1060	Application Data
193	11.652483	192.168.1.44	192.168.1.16	TLSv1.3	1377	Application Data
202	11.653698	192.168.1.44	192.168.1.16	TLSv1.3	571	Client Hello
211	11.669759	192.168.1.16	192.168.1.44	TLSv1.3	1461	Application Data
212	11.670328	192.168.1.44	192.168.1.16	TLSv1.3	1352	Application Data
214	11.671982	192.168.1.16	192.168.1.44	TLSv1.3	357	Application Data
215	11.672497	192.168.1.44	192.168.1.16	TLSv1.3	1379	Application Data
217	11.673138	192.168.1.16	192.168.1.44	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data, Application Data
218	11.673138	192.168.1.16	192.168.1.44	TLSv1.3	158	Application Data, Application Data
221	11.673330	192.168.1.44	192.168.1.16	TLSv1.3	84	Change Cipher Spec, Application Data
225	11.675955	192.168.1.16	192.168.1.44	TLSv1.3	1514	Application Data
226	11.675955	192.168.1.16	192.168.1.44	TLSv1.3	1079	Application Data, Application Data
303	14.664559	192.168.1.16	192.168.1.44	TLSv1.3	78	Application Data

Figure 5.25. Messages in third step of deregistration phase

Chapter 6

Conclusions

This project was born to increase the security of high risk workplaces in which a breach would be very dangerous and could do huge damage. To do that, the design of the system wants the hardware board to be put as the only point of contact with the external world; this electronic card should be the gateway in input and output. So, the aim of this project is to have a strong authentication system in which the worker can be authenticated thereby accessing to the service which, in this case, could be meaning to be able to talk with external net.

6.1 Production

Thereby being load in production, this project needs to be turned from a prototype to a real system. The security level of a mechanism is defined by its weakest point. This can be a section of the software, a hardware component, a mistake in design or configuration, or the final user.

In order to be put in production, this system needs first a secure DNS server; without this kind of server, there would be the need to statically write all the required IP addresses: the RP one in the user's device and the FIDO2 server one in the web application. Doing this can raise different vulnerabilities exploitable by attackers. For instance, a hacker could access to the device of the user and change the static IP address with another: this action brings, at least, to DoS attack, Denial of Service, if the static IP is just wrong.

Another crucial action to perform to make the system secure is to put the hardware board in a protected place: cybersecurity is not just about virtual world, but also about physical security. In this case, for instance, if an attacker manages to reach the board, it could easily access to all the data inside it: it is true that the card is protected with a username and a password, but there exist many methods hackers have and can use to discover, guess, or steal them. Moreover, even though the hacker failed in accessing to the board, it would anyway be able, for instance, to destroy it, so that to perform a DoS attack.

As this kind of systems sees the hardware card as the only gateway of the work office, designing this system as in this project could be dangerous, as the board represents the only point of failure of the mechanism. This problem is not just about attackers, but also about incidents in general: if an issue raises into the embedded system causing it to be unable to properly work, the whole mechanism would downfall.

Since this system is just a prototype, it is not a problem to have self-signed auto-generated certificates; in production, however, it is mandatory the web application and the FIDO2 server to have valid certificates provided by trusted Certificate Authorities.

6.2 Improvements and other functionalities

Many improvements may be done for this project: the first, which has already been discussed, is the addition of a DNS server which can allow to avoid some vulnerabilities. In this case, nevertheless, the DNS server can raise other treats, if implemented in a non-secure way.

Another functionality this web application may have is the management of more than one key per user. FIDO2 implements this possibility and SKFS, the StrongKey Fido Server [8], offers different mechanisms and APIs which can be used to let users manage different key pairs. This

functionality is not so important to build in this prototype since the example done for the service is just the permission to purchase a personal computer; nevertheless, in a more complex web application in which the user can access to more than one resource or service, it could be useful the user to have two or three key pairs. In this kind of servers, a more sophisticated access control could be implemented.

Finally, an improvement which can be implemented in the web application might be the addition of other types of authentication algorithms supported in confirmation transaction tasks.

6.3 Connection with other projects

This project regards only the registration, authentication, and authorisation mechanism of an embedded system to be used within a high-risk work office. To acquire sense, this system could be associated to other similar projects implemented in the same hardware to improve the security level. An example of this can be the implementation of a TPM mechanism which can grant the absence of intrusions or the implementation of an embedded system-oriented software which can offer VPN services.

Bibliography

[1] K. Bissell, L. Ponemon, "Accenture/Ponemon Institute: The Cost of Cybercrime", Network Security, Volume 2019, Issue 3, March 2019, pp. 4-4, DOI 10.1016/S1353-4858(19)30032-7

[2] C. Paulsen, R. Byers, "Glossary of Key Information Security Terms", NIST Interagency or Internal Report (NISTIR) 7298 July 2019, DOI 10.6028/NIST.IR.7298r3

[3] R. W. Shirey, "Internet Security Glossary, Version 2", RFC-4949, August 2007, DOI 10.17487/RFC4949

[4] T. Hansen, "SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms", RFC-7677, November 2015, DOI 10.17487/RFC7677

[5] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, O. Ranen, "HOTP: An HMAC-Based One-Time Password Algorithm", RFC-4226, December 2005, DOI 10.17487/RFC4226

[6] D. M'Raihi, S. Machani, M. Pei, J. Rydell, "TOTP: Time-Based One-Time Password Algorithm", RFC-6238, May 2011, DOI 10.17487/RFC6238

[7] FIDO Alliance, https://fidoalliance.org/

[8] StrongKey, https://www.strongkey.com/

[9] HAProxy, https://www.haproxy.org/

[10] SoloKeys, https://solokeys.com/

[11] WebAuthn, https://webauthn.io/

[12] CentOS, https://www.centos.org/

[13] VMWare, https://www.vmware.com/

[14] Raspberry Pi, https://www.raspberrypi.com/

[15] Raspbian, https://www.raspbian.org/

[16] Apache, https://httpd.apache.org/

[17] Wireshark, https://www.wireshark.org/