Thesis Dissertation

INTEGRATING FIDO2 AUTHENTICATION WITH AUTH.JS

Marios Papadiomedous

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2021

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Integrating FIDO2 Authentication with auth.js

Marios Papadiomedous

Supervisor

Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2021

Acknowledgments

Seizing the opportunity, I would like to express my sincere gratitude to my thesis supervisor, Dr Elia Athanasopoulo, whose expertise was valuable in guiding me throughout my thesis.

Finally, I would like to thank my friends and family, and especially Gregoris Christodoulou, for their physical and mental support that they generously provided and their continuous encouragement throughout the thesis process.

Abstact

Simple authentication schemas exist since the beginning of the internet. Over the years, these schemas upgraded themselves from plain-text to a relatively more secure version by hashing the password before saving. The security that these old-fashioned schemas provide is relatively low, considering the modern authentication schemas that are based on public-private key encryption.

The reasoning behind the usage of these traditional authentication schemas relies on the simple implementation that they require. This paper series introduces a simple, easy-to-use framework, auth.js, based on advanced authentication schemas using public-private key encryption. *auth.js* can be easily set up and used as the primary authentication schema. This specific paper is a walkthrough on how auth.js allows advance authentication using the FIDO2 Authentication schema that uses the *WebAuthn* standard. Finally, the *FIDO2 Authentication* and *auth.js* integration is tested and evaluated using an open-source, real web application, *WordPress*.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

One of the most critical parts of a website, from a security perspective, is the way that allows the user to prove their identity, formally known as the Authentication Procedure. This proof of identity procedure in web development may be, if vulnerable, a single point of failure in the system as a whole.

To be able to deliver personalized content to the end-user or restrict access, any web system must have a way to distinguish different kinds of users. Some widely used schemes for authentication are SSO, SAP, SAML. In most websites today, the Username/Password authentication is used as a default. In some cases, the user can enable second-factor authentication within the account settings. This verification follows the routine procedure by comparing the saved hashed version of the password against the given password. The authentication is successful only if the two hashes match.

If extra layers of security are not added, the adoption of Username/Password as the primary authentication method makes the system vulnerable. When choosing a new password, users tend to enter a password related to them(e.g. Birthday, pet name, anniversary), recycle common passwords or slightly modify a recycled password. The aftermath is to increase vulnerabilities. With today's computers compute speeds and algorithms, these hashes can easily be matched. Using rainbow, brute force, dictionary, and social engineering attacks, password cracking became easier. Strict password policies, password managers, expiring passwords and SFA are some safeguards used for eliminating vulnerability exploitation.

Despite all the vulnerabilities that username/password authentication has, web development still uses it as its primary authentication method. This method's straightforward implementation makes it preferable compared to other, more secure methods that need more effort to integrate into a new or preexisting system.

To eliminate the exploitable vulnerabilities associated with passwords as much as possible, username/password authentication based schemes need to add some extra steps to the authentication procedure, such as second step verification. One alternative to Username/Password authentication is to use Public/Private key pairs.

Auth.js is a framework that allows developers to use advanced authentication schemas effortlessly. By this time, three authentication methods are available(plain,scrypt_seed_ed25519_keypair, webauthn).This paper focused on the FIDO2 Authentication schema,the integration with auth.js and finally, the evaluation using a real-life scenario(wordpress)

Chapter 2

Background

In this section, some common terminology and background knowledge will be mentioned to help with the entire paper understanding. Fast IDentity Online (FIDO) Alliance is a non-profit organization that seeks to reduce the world's overreliance on passwords by developing authentication standards and specifications.This paper focuses on one of their latest standards called FIDO2 Authentication.

The FIDO Alliance's specification CTAP and the W3C WebAuthn standard are combined in FIDO2 authentication, which allows users to authenticate to online services using an ondevice or external cryptographic authenticators from both mobile and desktop devices.This passwordless schema relies on publickey cryptography.More indepth details about FIDO2 Authentication on section 4

2.1 Cryptographic Authenticator

An cryptographic authenticator is a device that confirms a user's identity by performing digital authentication using symmetrickey or publickey cryptography. There are no memorized secrets(such as passwords) involved, regardless of the key used.

2.2 PublicKey Cryptography

Asymmetric cryptography (also referred to as Publickey cryptography) is a type of cryptography that uses a keypair consisted of two keys, a public (which others maybe know) and a private key (which no one knows except the owner). The science behind asymmetric cryptography relies on the fact that these two

keys are mathematically related to eachother, with their generation to be the outcome of mathematical problems known as oneway functions. The usage of such keys is that the owner of these keys allows others to send him encrypted messages using his public key. These encrypted messages can be decrypted only using the private key linked with the specific public key.

2.3 Web Authentication (WebAuthn)

WebAuthn is an open standard that uses PublicKey Cryptography to create a standardized interface for passwordless authentication to webbased services.

2.4 Client to Authenticator Protocol (CTAP)

CTAP is a specification that describes how an application (such as a browser) and operating system communicate with a authentication device through USB, NFC, or Bluetooth

- *CTAP1 or U2F*: CTAP1 (also known as Universal 2nd Factor) is a standard that defines how the communication is established between FIDO2-enabled browsers and operating systems and a FIDO U2F device to achieve *secondfactor* authentication.
- *CTAP2*: Specifies how to communicate between FIDO2enabled browsers and operating systems and external authenticators (FIDO Security Keys, mobile devices) to empower *passwordless,second or multifactor* authentication.

Chapter 3

Architecture

This section focuses on how the components communicate to complete the registration and accession procedure.

3.1 Components

- *Replaying Party*: It is a trusted service that runs on the server side. RP is the service that will decide whether the assertion and registration are successful.
- *Internal Authenticator*: Located within the client device. Uses device built-in sensors/procedures in order to authenticate the user. IA consists of Fingerprint/FaceID/Iris Recognition and anything else that does not require the Client's device to interact with another physical component.
- *External Authenticator*: This is a physical device that the user owns and uses against the client device to authenticate themselves. A physical authenticator may be a security key (e.g. Yubikey) that uses NFC, WiFi or USB to communicate with the client's device.
- *Auth.js*: A framework that communicates with the RP in order to complete registration and assertion procedures.

All necessary components interact with each other in order to complete a task. At this time being, auth.js supports four tasks; registration, authentication, new key registration to a pre-existing webauthn enabled user and finally, the ability to delete all the saved keys.Registration and authentication are the only non-restricted procedures.Figure 3.1 shows the categorization of the available *auth.js* procedures accosted with *FIDO2 Authentication*.

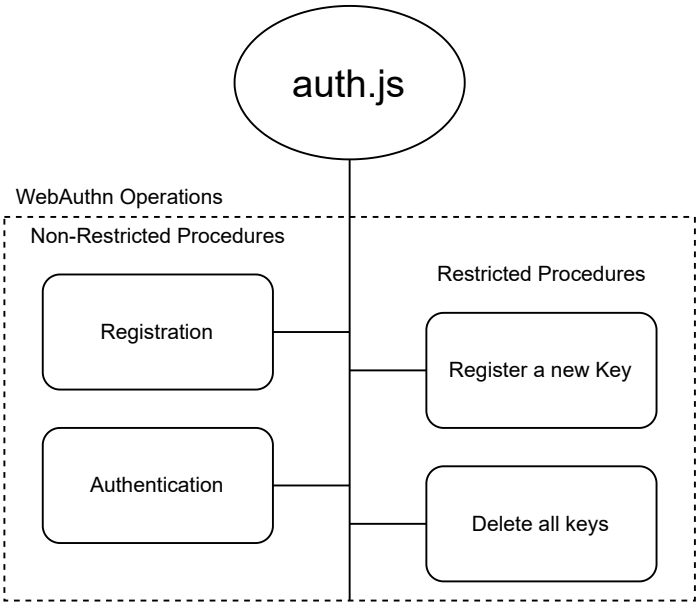

Figure 3.1: auth.js (WebAuthn implementation) Structure.

3.2 Procedure

Shown at Figure 3.2 is the higher level idea about how *FIDO2 Authentication* works.

- At first, the web page will request from the relying party to start the matching ceremony (2).
- After the ceremony initialization, the RP will send some data to the client's device, requesting the authenticator to use and include some settings provided within the received data (3).
- Based on the received data, the client's device will ask the user to identify themselves using an authenticator in order to finish the procedure (4).
- The used authenticator will generate some new data containing some key information from the data that were sent from the RP (5).
- After the response generation, the new data will be sent to the RP in order to verify or register the new data (6-8).

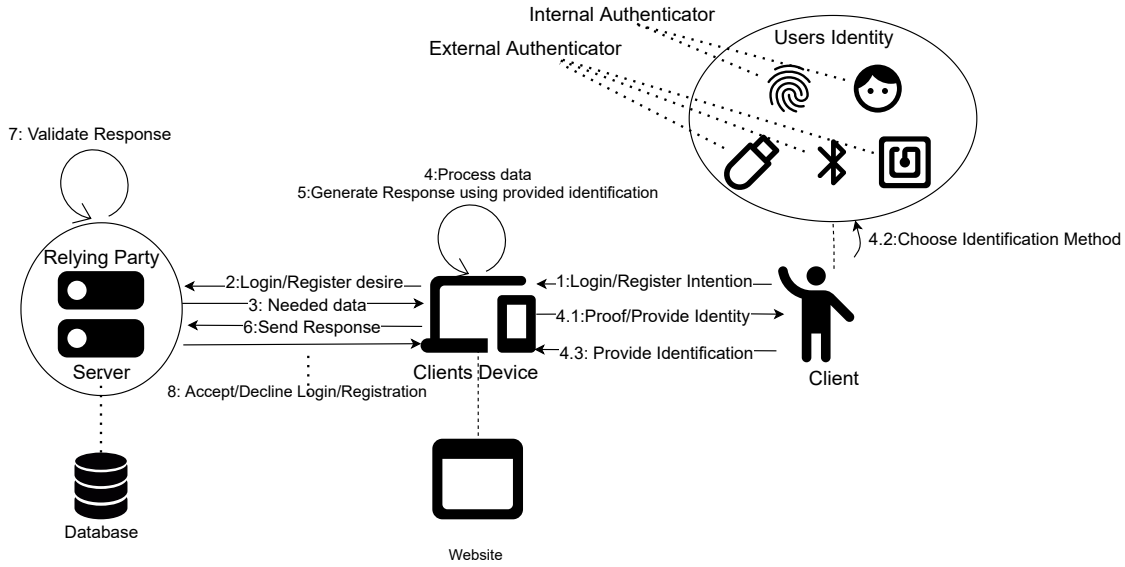

Figure 3.2: auth.js, webauthn registration/authentication procedure

3.3 Auth.js

The first step to use webauthn implementation is to include auth.js in the source code of the web page. *auth.js* uses some external libraries such as *base64.js* for encoding and *Jquery* for AJAX requests.Listing 3.1, shows The client that is being directed to get *auth.js* from a trusted source.*base64.js* and *jQuery.js* can be included from trusted online-sources.

```
<html>
<head>
 ...
<script type = "text/javascript" src = "https://trusted_domain.com/auth.js"></
    script>
<script type = "text/javascript" src = "https://trusted_domain.com/base64.js"
    ></script>
<script type = "text/javascript" src = "https://trusted_domain.com/jQuery.js"
    ></script>
 ...
</head>
<body>
/* Registration and login form */
</body>
</html>
```

Listing 3.1: *Web application html header file*

The programmer has the freedom to use all the available authentication methods (default,scrypt_seed_ed25519_keypair and webautn) because webau-

thn is a stand-alone implementation that does not require any initialization unlike scrypt_seed_ed25519_keypair.Listing 3.2 both authentication methods being used. The only parameter that webauthn methods take as an input is a username. If the parameter 'credential' is given instead of null during the registration procedure, the new Credential will be registered as another authorized device.

```
initializeCredentialType({
  passwordMinLength: 8,
  passwordProccessMethod: "scrypt_seed_ed25519_keypair",
});
  let username = document.getElementById("username");
  let password = document.getElementById("password");
    /* On registration action using scrypt_seed_ed25519_keypair */
  let credential = register(password);
  /* On registration action using WebAuthn */
  webauthn_registration(username, null).catch(function (promise){
      console.log(promise)
  });
    /* On login action using scrypt_seed_ed25519_keypair */
  let message = document.getElementById("nonce");
   /* On Authentication action using WebAuthn */
  webauthn_authentication(username).catch(function (promise){
      console.log(promise)
  });
  let credential = authenticate(password, message);
/* Send credential and other necessary information to the server */
```

Listing 3.2: *auth.js* authentication scheme initialization

Chapter 4

Implementation

FIDO2 authentication relies on two components, *WebAuthn* and *CTAP* standards. Keeping in mind that most of today's browsers support *CTAP*, the only missing part to complete the FIDO2 authentication puzzle is the Relying Party that serves based on WebAuthn Standard. This section will describe the implementation of *WebAuthn* standard in *Python3* and the integration of *auth.js* with *WordPress* in technical details.

4.1 Python3 and WebAuthn

As mentioned before, using the username and password authentication schema, the only process that needs to be done server-side is the password hashing and the association between the user and the hashed password or comparing the stored hash with the calculated one when authenticating.

4.1.1 RP configuration

The responsibilities of RP do not stop at validating incoming responses. One other main task is to send the configuration options for the authentication to use when creating or retrieving the credentials.

The first steps for both ceremonies require that the RP has been configured correctly. Some constant global settings can be found within the *c*onstant.py script file, such as Database credentials, supported hashing algorithms and public/private key file names for cookie signing procedures. At the beginning, a GET request will be placed to the server requesting a JSON structured response containing settings needed by the authenticator.

RP ceremony settings can be configured using the *ServerConfigRegistration* class located within the *Classes.py*. Usage of the *ServerConfigRegistra-*

tion class can be seen at listing 4.2. The comments show the available values that each parameter can take. After the initialization of the RP server using the *to_json()* function against the object type *ServerConfigRegistration*, a JSON structured configuration will be generated.

The basic structure of a JSON response can be seen at listing 4.1. This structure consists of some information identifying the RP, the user and the session. *RP/.id* represents the domain that RP is in charge of serving, while *pubKeyCredParams* represents the supported, by the RP, *type* of credentials. Also, the *alg* contains the supported cryptographic algorithms represented by their COSE identifier and the type of the credential. User data contain the user's entered username as a value to the displayName field along with a unique identifier encoded using base64. Challenge is a one/-time use nonce encoded in base64 used for validating each different session.

```
{
  "rp": {
    "id": "example.com"
  },
  "user": {
    "displayName": "user",
    "id": "yf2dzb2QFGGxOg=="
  },
  "challenge": "nA+lhg4461nAp9K6sLxXtIAQwPAGXftGsdN2v0EB75Q=",
  "pubKeyCredParams": [
    {
      "alg": /-7,
      "type": "public/-key"
    }
  ]
}
```

Listing 4.1: Registration Ceremony Server's basic requirements for authenticator

```
ServerConfigRegistration(username,          #Users entered username
    type,                                   #Ceremony type, "registration" OR
    "login"
    usernamelength=10,                      #Length Of the user.id that needs
    to be generated
    challengelength=32,                     #Length Of the challenge that
    needs to be generated
```

```
5     timeout=600000,                          #Time in milliseconds
6     attestation="none",                      #RP preferred attestation "none"
      OR "indirect" OR "direct"
7     requireResidentKey=False,                #Currently Not Supported
8     userVerification="discouraged",          #RP requirement Verification"
      required" OR "preferred" OR "discouraged"
9     excludeCredentials=None,                 #Array containing the credential
      that the authenticator excludes
10    pubKeyCredParams=constant.SUPPORTED_ALG, #Supported Algorithms for
      credential creation
11    extensions=None,                         #Currently Not Supported
12    webpageid=constant.WEBPAGEID)            #The domain name
```

Listing 4.2: ServerConfigRegistration Avaliable Options

4.1.2 Registration

Before committing the user's public key to the database and associating it with the user, the Relying Party must first decide whether the provided information is legitimate or not. The Validation procedure consists of 24 critical steps. It is mandatory mentioning that if any of the 24 censorious validation steps fails, so does the registration procedure. In case something has been committed to the database, then the specific transaction associated with the mentioned procedure rollbacks.Figure 4.1 shows an overview about how registration ceremony works.

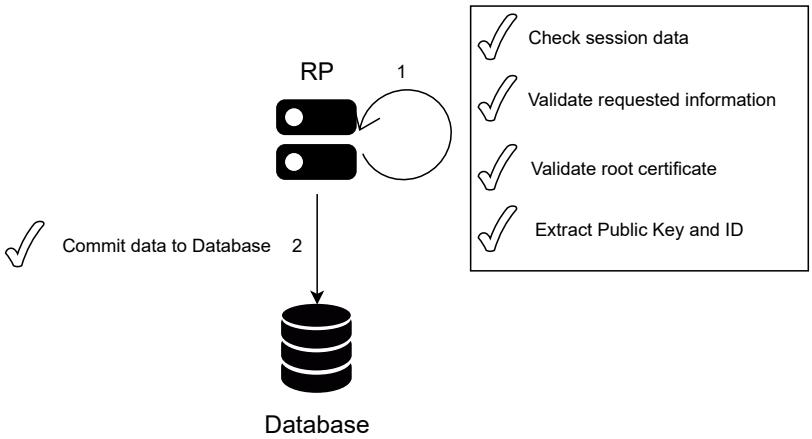

Figure 4.1: Registration Ceremony overview

4.1.3 Assertion

Pre-Validation Procedure

To start the registration ceremony, the user must initiate the procedure by providing their username via completing a form. In this part the website must have already set up auth.js (listing 3.2).

The handler responsible for the interaction between RP and Client is the *webauthn_registration()* function. After the *webauthn_registration()* function call, a GET request will be placed to the RP to retrieve the configuration JSON object. After receiving the JSON object, the next step is to decrypt the BASE64 encoded fields. The data that need to be decoded are the *challenge* and *User.id*.

After the prementioned fields are successfully decoded, the data is ready to passed to the Authenticator (*navigator.credentials.create()* function). Based on the configuration file that RP sent, the user will be prompted to interact with the authenticator using external media (CTAP Protocol) or provide their identity using the U2F protocol. After the user's interaction with the authenticator, a new key pair will be generated. Finally, the Public key will be returned among with other metadata to *auth.js*

The next step is to POST the generated data to the RP to decide about their integrity. The RP only accepts a pre-defined JSON structure that can be seen in Figure 4.3. After the RP receives the response, the validation of the received response begins.

```
1 {
2    "id": Contains Authenticator assigned unique credential
     identifier encoded in BASE64  ,
3    "type": Describes Credential key type.Currently only "Public
     key" is supported,
4    "transport": Describes the authenticator used. Values:"usb","
     nfc","ble" or"internal",
5    "response":{
6       "attestationObject":Contains an attestation object, which
     is opaque to, and cryptographically protected against
     tampering by, the client.,
7       "clientDataJSON": Contains a JSON formatted string,
     representing the client data that was passed to the navigator
     function.
8    }
```

```
}
```

Listing 4.3: Registration Ceremony Server's accepted JSON structure

As mentioned before, the Client initiates the registration procedure. After *auth.js* requests the configuration JSON structure, the received configuration will be taken as an input by the Authenticator (*navigator.credentials.create()* function). The majority of today's browsers supports the authenticator function. The Authenticator will then generate the Public Key Pair. The Private key will be safely stored on the client-side, while the public key will be encapsulated within some extra meta-data and send back to the RP. Now the RP must validate the received data. It is important to mention that in order for the *navigator.credentials.create ()* function to work, the web page must be running on an HTTPS domain with a valid certificate or in localhost.

Credential Validation Procedure

After the RP receives the POSTed data, the RP will validate the data using session data, predefined server configuration and trusted certificates. The primary data session holds consists of the user's username, challenge and page.id. Before the validation starts, the RP deconstructs the received data to JSONtext, attestationObject and transports by extracting the clientDataJSON, attestationObject and attestationObject, respectively. Moreover, if needed, a BASE64 or CBOR decoder will be applied to the extracted data.

Firstly, the JSONtext data must be validated. Listing 4.4 shows the internal structure of the clientDataJSON after it run through a BASE64 decoder. Each field must be matching with the corresponding session data within the placed cookie. During registration, the "*type*" field can take only the value *webauthn.create*. The origin must be the same or a sub-domain of the provided by *rp.id*. To continue with the next validation steps, the challenge must be the same as the one provided previously by the RP, and both type and origin match the pre-described values.

```
"clientDataJSON":{
"type":"webauthn.create" or "webauthn.get"
"challenge":"h5xSyIRMx2IQPr1mQk6GD98XSQOBHgMHVpJIkMV9Nkc"
"origin":"https://example.com"
}
```

Listing 4.4: clientDataJSON Stracture

The data that next needs to be verified is the information within the attestationObject. Firstly, in order to access the data, a BASE64 decoder must be applied and then the result must be parsed into a CBOR2 using a compatible loader. Figure 4.2 shows the structure of attestationObject. As can be seen, attestationObject consists of three sub-structures. FMT holds the specific attestation format used. Attestation statement (attStmt) carries information about the generated credential and the authenticator used. Attestation signature is also contained in the attStmt. The final part of the attestationObject is the authenticator data (authData). AuthData contains bindings made by the authenticator. The most important field of attestationObject is attestedCredentialData. The attestedCredentialData field contains the credentialId and credentialPublicKey.

The validation of attestationObject begins by checking authData. In order for authData to be valid, the length must be greater than 37 bytes. After that, the rpIdHash is checked against the hashed version of the rp.id stored in the session. The user's presence is checked using the zeroth bit(UP) of flags buffer. If the RP requested the authenticator to check about "user verification" then the third bit(UV) must have the value of 1.

After the successful validation of authData, the attStmt must be checked. The first part is to check whether the RP supports the attestation format used. A list of all supported FMTS can be found within the constant.py file. If the FMT is supported, then the RP will verify the attStmt based on the FMT. The primary approach used for validating attStmt is first to check if the syntax of the given statement is correct by checking if some specific fields are contained (each FMT attStmt can be found on the W3C website). After that, if an X5C certificate chain is found, the root certificates will be checked against the RP saved certificates from multiple vendors saved on the root_certificates folder.

After the procedures mentioned above are completed successfully, the public key and credential id will be gathered from attStmt and stored in the database among the user's unique assigned id. If no errors occur, then a success code will be sent from the RP.

4.1.4 Assertion

The assertion ceremony, also known as the Login procedure, works with the same principles as the registration ceremony. In order to proceed using the assertion ceremony, the registration ceremony must be successfully completed. As before, in order for the assertion ceremony to successfully complete, the

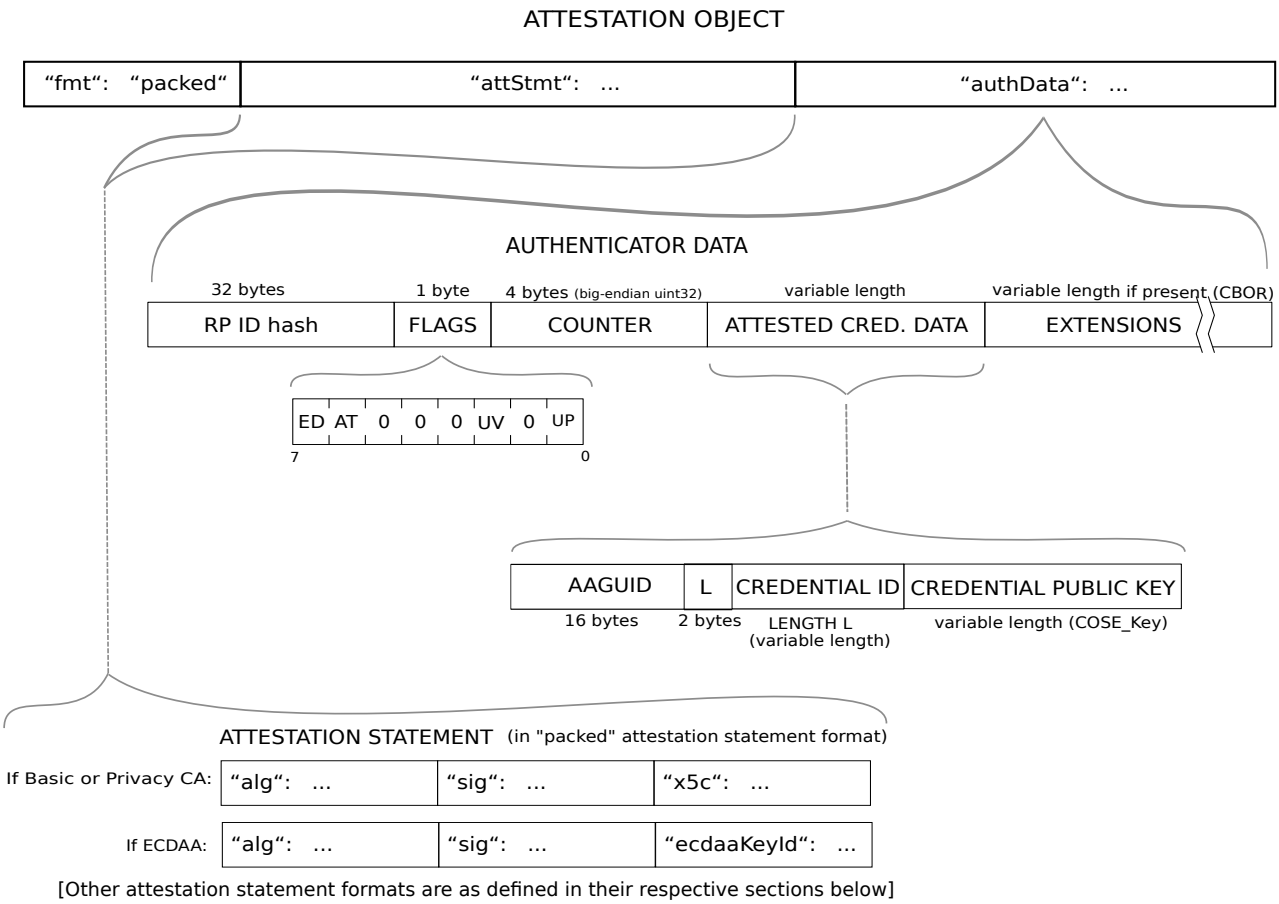

Figure 4.2: attestationObject Structure, Source :https://www.w3.org/TR/webauthn-2/images/fido-attestation-structures.svg

authenticator’s response must go through all 22 RP checkpoints.Figure 4.3 shows an overview about how authentication ceremony works.

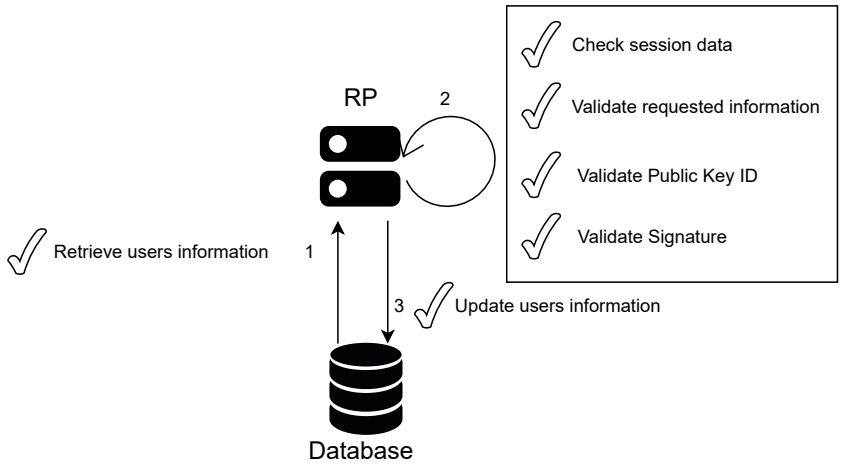

Figure 4.3: Registration Ceremony overview

Pre-Validation Procedure

To start the assertion ceremony, the user must initiate the procedure by clicking the corresponding button on the web form.

The only information needed from the user in order to initiate the assertion ceremony is the username. Supposedly that the website has integrated the auth.js framework as demonstrated in Chapter 3 and successfully connected the *webauthn_authentication()* handle, the pre-validation procedure will be similar to the pre-validation procedure of the registration ceremony.

Firstly, a GET request will be placed to the server to retrieve the configuration file, which is different from the registration ceremony. The basic server JSON respond file can be seen on listing 4.5. The only new field that the authenticator needs for retrieving the previously generated credential is the allowCredentials. As the name suggests, this field is an array containing information regarding the associated keys with the given account. ID represents the unique credentials identifier assigned by the authenticator at registration.*RP.ID* field that was used on registration is now renamed to *rpid*. The server configuration can be seen at listing 4.2.In order to use the *ServerConfigRegistration* for the Assertion, the parameter *type="login"* must be given. The rest of the configuration is as described at the Registration ceremony.

After the configuration is retrieved by the client and the necessary parts were decoded, the next task is to retrieve the credentials stored in the authenticator, supposed that the user has successfully registered. The *navigator.credentials.get()* will retrieve the credentials from the authenticator using the configuration file provided by the RP. Then *navigator.credentials.get()* will ask the user to interact with the authenticator to prove his identity and release the stored credentials associated with the user's chosen authentication method. After collecting and formatting the results gathered from *navigator.credentials.get()* the JSON string will be sent to the RP to finish the Assertion ceremony. The structure that the RP accepts can be seen at listing 4.6

```
{
  "allowCredentials": [
    {
      "id": "QxmXKURjzSjqtqSyXEI0G9Xd3Y02D1fZo40tqeMs3i8=",
      "type": "public-key"
    },
    {
```

```
      "id": "AXIPSeUq0vYLSzOi1WiS0DHiImal64g8mfFrFb06E8OXh0gT8yIP
   55ldj9NkE5dv8W85K9XZwsUtkLkaD76gbXc=",
      "type": "public-key"
    }
  ],
  "challenge": "vvxBGQyzkp6u4yUnv5rmloejF6YinRSfNHcVwbwBgMU=",
  "rpid": "mariosfidotest.com"
}
```

Listing 4.5: Assertion Ceremony Server's basic requirements for authenticator

```
{
   "id": Contains Authenticator assigned unique credential
   identifier encoded in BASE64 ,
   "type": Describes Credential key type.Currently only "Public
   key" is supported,
   "response":{
      "authenticatorData":Also known as AuthData, contains
   bindings made by the authenticator,
      "clientDataJSON":" Contains a JSON formatted string,
   representing the client data that was passed to the navigator
   function,
      "userHandle": Users ID given by RP on the registration
   ceremony,
      "signature": Contains the signature of the authenticator
   for both authenticatorData and a SHA-256 hash of the
   clientDataJSON.
      "clientExtensionResults":contains a map between the
   extensions identifiers and their results after having being
   processed by the client if any extensions are enabled by RP.
   }
}
```

Listing 4.6: Registration Ceremony Server's accepted JSON structure

Credential Validation Procedure

After the RP receives the data POSTed, the validation procedure will start. The first step is to extract the needed data from the response. The *response.authenticatorData*, *clientDataJSON* and *signature* must be extracted

from the received JSON string and stored to the authData, cData and sig, respectively. Because *auth.js* encodes all the data using *BASE64* encoder, all data must be decoded before continuing the validation procedure.

The validation of the received data starts by retrieving all users data from the database associated with FIDO2 authentication.The database schema will be described later in this section. In order to continue with the procedure, the received *id* and *uhandle* must be checked against the database records. In the case of *id* not matching with any *credential ID* retrieved from the database or *uhandle* not matching with the user ID that *id* is linked to, then the procedure is failed.

The next step is to verify the data provided by the RP for the authenticator with the data that the authenticator included, located within the cData. The basic structure of the *clientDataJSON* can be seen once again to the listing 4.4.*clientDataJSON*. More specifically,*cData.challenge* must match the one that RP sent at the beginning of the authentication session. *cData.type* must be equal to "*webauthn.get*" and lastly, as before, the origin must the the same as the domain or a valid sub-domain of the provided one.

The next step is to validate the data within the *authData*. The *authData* structure can be seen at figure 4.2 as part of the Attestation Object. Firstly *rpIdHash*, located within the first 32 bytes of *authData*, must be the same as the expected domain stored in session data. In order to validate the *rpIdHash*, the session stored domain must be hashed using SHA-256 and compare the digest with the *rpIdHash*. The next step is to verify the flags that RP requested on the configuration file. It is mandatory to check for the *User Present*, bit is equal to 1. If RP requests no other flags, then the procedure will continue to the next validation step.

The final step is to verify the received signature using the public key retrieved from the database at the beginning of the ceremony with the received *id*. In order to verify the signature, a hashed version of *cData* must be created using SHA-256. After that, using the public key and the *COSEAlgorithmIdentifier* used on the specific key, a signature verifier will be applied on the concatenation of the digest on the previously generated *cData* hash and the *authData*.

The only process left after all the previous verifications are successfully completed is to update the signature counter,*SignCount* value from the Database. *SignCount* is located within the 34-38 bytes of *authData*. In order to successfully complete the ceremony, the value of *SignCount* stored with the database must be strictly grater that the *SignCount* value that *authData* has.

This is an extra layer of protection that *W3* uses to ensure that each time an assertion ceremony occurs, the clients authenticator used and not some other impersonator.

After the assertion ceremony finishes successfully, the RP will place a secure session cookie named *webauthn_login*, on the client's browser. The cookie structure can be seen on listing 4.7 as a JSON structure. The cookie contains the username of the successfully authenticated user as well as a signature for the digested username. The signature is generated using the RSA public/private key-pair that RP owns. The specific key pair is only used for encrypting and decrypting the cookie.

The successful cookie placement signals the end of the assertion ceremony.

```
{
   "Username": Authenticated users username ,
   "Signature": The signed username digest
}
```

Listing 4.7: RP Authentication secure cookie

4.1.5 Database Schema

The data generated or released from the authenticator during the registration or assertion ceremony respectively need to be stored in a database for the next ceremony to use. Like the traditional authentication methods that store the hashed version of the password, in this case, an encoded version of the public key is stored along with some other information.

At figure 4.4 the database schema can be seen. The database schema follows a minimalistic design approach. There are only two tables. The first one is the *FIDOUSER* where the association between *username*, user *fidoid* (also referred as *userid*) and *origin* is held. The *FIDOCREDENTIAL* table stores the multiple credentials for each user. The two tables are linked together using a foreign key constrain between *FIDOUSER.FIDOID* (Primary Key) and *FIDOCREDENTIAL.FIDOID* (Foreign Key). *FIDOCREDENTIAL* consists of an auto indexing field(*ID*), the *FIDOID* Foreign Key and some basic information about the credential such as the credential ID, the attestation type, the transport, the public key as well as the signature counter. The use of each field was described in the previous subsection.

Insert, Update and Delete transactions are done by stored procedures located within the database and not by standalone statements from the RP.

The RP only needs to call the procedure corresponding to the action needed and pass the correct parameters. There are five stored procedures: *RegisterUserAndCred*, *RegisterCredential*, *FindUser*, *UpdateSig* and *Deleteusercred*. All procedures that execute multiple statements will rollback and throw an exception if one statement within the transaction fails.

The first stored procedure to be explained is the *RegisterUserAndCred*, the implementation can be seen in listing 4.8. This specific procedure takes as input the *FIDOID* (unique user ID encoded in BASE64), *ORIGIN*(Web Page, used when RP serves multiple domains), *USERNAME*, *CREDID* (the unique credential identification encoded in BASE64), *CREDKEY* (the corresponding Public Key assigned to CREDID encoded in BASE64), *CREDTYPE* (the credential type, in this case is "Public-Key"), *CREDATTTYPE* (the attestation used for the specific credential), *CREDTRANSPORT* (the transport used for the specific credential) and *COUNTER* (signature counter). This procedure is only used in the registration ceremony.

```
START TRANSACTION;
    INSERT INTO FIDOUSER(‘FIDOID‘,‘USERNAME‘,‘ORIGIN‘)  VALUES (
   FIDOID,ORIGIN,USERNAME);
    IF CREDTRANSPORT = ’None’ THEN
    INSERT INTO FIDOCRETENTIAL(‘FIDOID‘,‘CREDID‘,‘CREDKEY‘,‘
   CREDTYPE‘,‘CREDATTTYPE‘,‘COUNTER‘) VALUES (FIDOID,CREDID,
   CREDKEY,CREDTYPE,CREDATTTYPE,COUNTER);
  ELSE
    INSERT INTO FIDOCRETENTIAL(‘FIDOID‘,‘CREDID‘,‘CREDKEY‘,‘
   CREDTYPE‘,‘CREDATTTYPE‘,‘CREDTRANSPORT‘,‘COUNTER‘) VALUES (
   FIDOID,CREDID,CREDKEY,CREDTYPE,CREDATTTYPE,CREDTRANSPORT,
   COUNTER);
  END IF;
COMMIT;
```

Listing 4.8: RegisterUserAndCred Implementation

The next procedure is the *RegisterCredential*. This specific procedure is used when a preexisting FIDO user needs to enrol a new key/Device to their account. The implementation can be seen in listing 4.9. The set of parameters that this procedure takes as an input is the same as the *RegisterUserAndCred* but without the *FIDOID*, *ORIGIN* and *USERNAME*. The order of the arguments is the same as before.

```
IF CREDTRANSPORT = ’None’ THEN
```

```
  INSERT INTO FIDOCRETENTIAL(‘FIDOID‘,‘CREDID‘,‘CREDKEY‘,‘
   CREDTYPE‘,‘CREDATTTYPE‘,‘COUNTER‘) VALUES (FIDOID,CREDID,
   CREDKEY,CREDTYPE,CREDATTTYPE,COUNTER);
ELSE
  INSERT INTO FIDOCRETENTIAL(‘FIDOID‘,‘CREDID‘,‘CREDKEY‘,‘
   CREDTYPE‘,‘CREDATTTYPE‘,‘CREDTRANSPORT‘,‘COUNTER‘) VALUES (
   FIDOID,CREDID,CREDKEY,CREDTYPE,CREDATTTYPE,CREDTRANSPORT,
   COUNTER);
END IF;
```

Listing 4.9: RegisterCredential Implementation

The *FindUser* is used in both procedures. During the registration ceremony, *FindUser* is used to validate whether the given username and the generated *FIDOID* is already registered. In the assertion ceremony, it is used for retrieving the information regarding the user that needs to be authenticated. The way that this procedure works is by applying a query to both *FIDOCREDENTIAL* and *FIDOUSER* tables and returning the retrieved information from *FIDOCREDENTIAL* for the given user. The implementation can be seen in listing 4.10. The procedure takes as an input the username of the user that needed to gather information.

```
select cred.*
from FIDOCRETENTIAL cred, FIDOUSER users
where users.USERNAME=input and users.FIDOID=cred.FIDOID;
```

Listing 4.10: FindUser Implementation

UpdateSig is only used during the assertion ceremony. As the name implies, this specific stored procedure aims to update the signature counter for a specific credential. The implementation can be seen in listing 4.11. The procedure takes as an input the signature counter number (*count*) and the row identification number(*input*). The row identification number, which is set as auto-increment, is returned when the procedure *FindUser* is called.

```
update FIDOCRETENTIAL set counter=count where ID=input;
```

Listing 4.11: FindUser UpdateSig

Finally, the *Deleteusercred* procedure can only be accessed when the *webauthn_login* cookie is set, meaning that the user must be authenticated first. The purpose of the procedure is to remove all user data from the database. The implementation can be seen in listing 4.12. The procedure takes as an input

the username,*UNAME*, of the user to be deleted. Implementation-wise, it is seen that the delete statement removes only the records from the *FIDOUSER* table. The data within the *FIDOCREDENTIAL* will be removed automatically due to the *ON DELETE CASCADE* option on the foreign key constraint.

```
1 DELETE FROM FIDOUSER WHERE USERNAME=UNAME;
```

Listing 4.12: Deleteusercred implementation

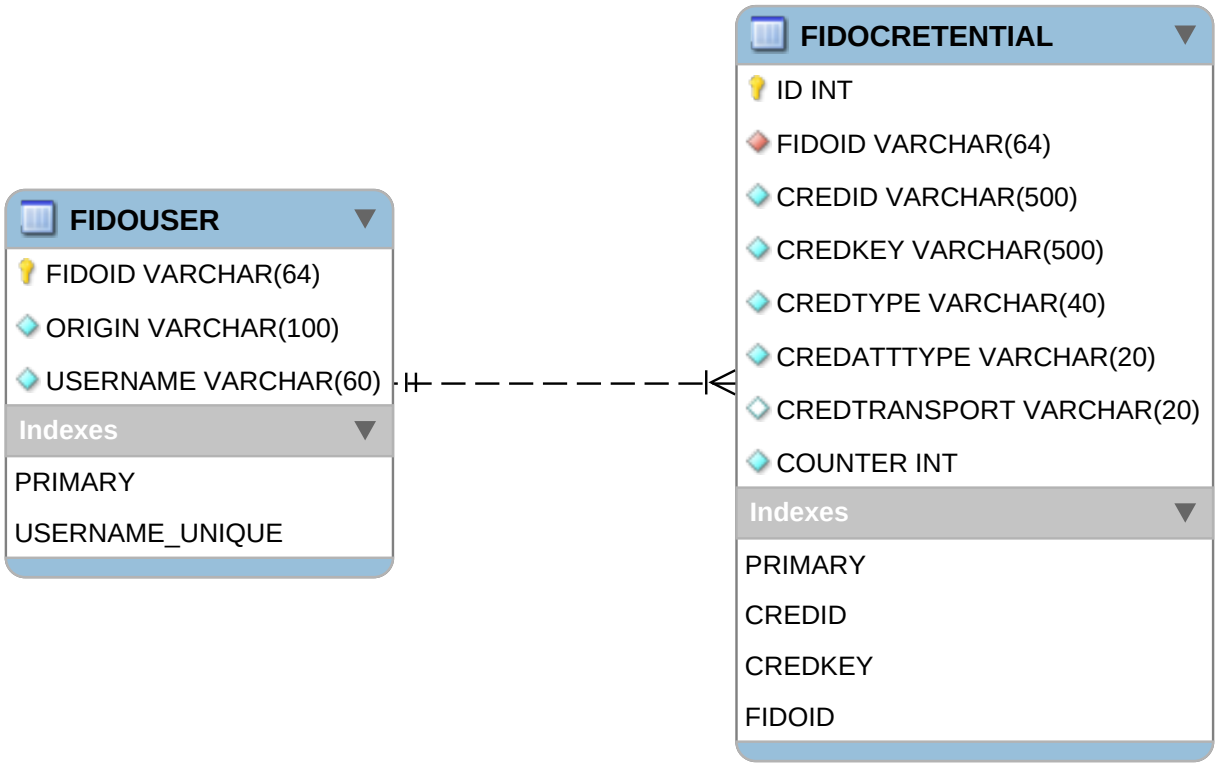

Figure 4.4: Database Schema

4.2 Real Life Scenario (*WordPress*)

This last subjection of the Implementation section focuses on allowing *WordPress* to use *FIDO2 Authentication* from *auth.js* as the main authentication schema.

4.2.1 Environment Setup

Setting Up WordPress is a straightforward procedure. In order to allow FIDO2 Authentication to function as expected, some necessary changes must be first

made in both the web-server and the database.

In this specific implementation, the webserver used was NGINX. In listing 4.13 the configuration settings of the webserver can be seen. In order for the *Navigator.credentials* API to work an *HTTPS* connection is needed, thus a *self-signed certificate* is being added to the server. Now that the domain runs under a *secure channel* the server is redirecting the connection from the standard webserver port(80) to the secure port(443).

```
server {
    listen 443 ssl;
    listen [::]:443 ssl;
    ssl_certificate /etc/ssl/certs/localhost2.crt;
    ssl_certificate_key /etc/ssl/private/localhost2.key;
    ssl_protocols TLSv1.2 TLSv1.1 TLSv1;
    server_name example.com www.example.com;
    include fcgiwrap.conf;

    root /var/www/wordpress.com;
    index index.php index.html index.htm;
    error_log /var/log/nginx/mysite.com_error.log;
    access_log /var/log/nginx/mysite.com_access.log;
    client_max_body_size 100M;

    location / {
      try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
      include snippets/fastcgi-php.conf;
      fastcgi_pass unix:/run/php/php7.4-fpm.sock;
      fastcgi_param SCRIPT_FILENAME
      $document_root$fastcgi_script_name;
    }

    location /req/ {
        include proxy_params;
        proxy_pass http://localhost:5000/;
    }
}
server {
    listen 80;
    listen [::]:80;
    server_name example.com www.example.com;
    include fcgiwrap.conf;

```

```
38     return 301 https://$server_name$request_uri;
39 }
```

Listing 4.13: Web-server Configuration FIle

The way that *RP* interacts with WordPress is by a reverse proxy.*RP* in this implementation is a Python3 server implemented using the *FLASK WEB FRAMEWORK*. The client can access the *RP* using API calls to the *example.com/req/* URL. Registration ceremony, Assertion ceremony, Register a new credential and Delete a key can be accessed through adding the corresponding sub-category to the *example.com/req/* URL. The sub-category for the previously mentioned procedures are *register*, *login*, *register/cred* and *del/all* respectively.

4.2.2 Registration Procedure

In order to allow users to register using the *FIDO2 Authentication*, some changes must be made within the WordPress Source code. The first modification must be made within the *wp-login.php* file.

The first step is to make sure that *WordPress* can read and load the included webauthn files successfully. In order to load these files, a function must be added and executed, before the predefined *login_head* is called. At total, four scripts are required for *FIDO2 Authentication Schema* to function properly. *authjs* needs the *jquerymin̈js* and *base64js*. The final JavaScript file that is needed is the *webauth_interruption.js*. The prementioned script is used to link the *authjs* with the preexisting forms. In listing 4.14 the way that the necessary scripts are linked with *WordPress* is shown.

```
1  /**
2  * Enqueue scripts and styles for the login page.
3  *
4  * @since 3.1.0
5  */
6  add_action( 'webauthn_enqueue_scripts', 'enqueue_authjs' );
7  function enqueue_authjs( $page ) {
8      wp_enqueue_script('jquery-webauthn', 'https://ajax.googleapis.com/ajax/libs
       /jquery/3.5.1/jquery.min.js', array(), null, true);
9      wp_enqueue_script( 'webauthn', home_url() . '/wp-includes/js/webauthn/auth.
       js', array('jquery-webauthn'), null, true );
10     wp_enqueue_script( 'base64', home_url() . '/wp-includes/js/webauthn/base64.
       js', null, null, true );
11     wp_enqueue_script( 'webauth_interruption', home_url() . '/wp-includes/js/
       webauthn/webauth_interruption.js', array('jquery-webauthn'), null, true );
```

```
12 }
13 do_action( 'webauthn_enqueue_scripts' );
14
15 /**
16  * Fires in the login page header after scripts are enqueued.
17  *
18  * @since 2.1.0
19 */
20   do_action( 'login_head' );
```

Listing 4.14: Include auth.js scripts to WordPress

To allow the end-user to choose the authentication schema that they wish to use, the HTML code of the registration screen has to be slightly modified. A new check box will be added to allow the user to use the *FIDO2 Authentication Schema*. In listing 4.15 the addition of the *FIDO2* check box to the preexisting *WordPress* source code is shown (lines 14-16).

```
1                                                .
2                                                .
3                                                .
4 <form name="registerform" id="registerform" action="<?php echo esc_url(
      site_url( 'wp-login.php?action=register', 'login_post' ) ); ?>" method="
      post" novalidate="novalidate">
5       <p>
6         <label for="user_login"><?php _e( 'Username' ); ?></label>
7         <input type="text" name="user_login" id="user_login" class="input"
      value="<?php echo esc_attr( wp_unslash( $user_login ) ); ?>" size="20"
      autocapitalize="off" />
8       </p>
9       <p>
10        <label for="user_email"><?php _e( 'Email' ); ?></label>
11        <input type="email" name="user_email" id="user_email" class="input"
      value="<?php echo esc_attr( wp_unslash( $user_email ) ); ?>" size="25" />
12      </p>
13            <p>
14                <input name="registerusingwebauthn" type="checkbox" id="
      registerusingwebauthn"/>
15                <label for="registerusingwebauthn"><?php esc_html_e( 'Use
      Webauthn Standart' ); ?></label
16            </p>
17                                               .
18                                               .
19                                               .
```

Listing 4.15: FIDO2 Authentication Checkbox for registration in HTML code

The linking between *WordPress* and *auth.js*, as mentioned before, is happening within the *webauth_interruption.js*. The linking is achieved by interrupting the form submission, finishing the *FIDO2 Registration* and resuming the form submission. The above procedure can function properly because *RP* is an independent service running alongside with *WordPress*. Listing 4.16 demonstrates how the interruption is happening during the registration form submission and how the *auth.js* is called. If any error occurs during the registration ceremony, the error will be printed on the console. This procedure is no blocking, meaning that if the *FIDO2 Authentication* fails, then the registration procedure that *WordPress* follows will be resumed.

```
1  jQuery("#loginform").on("submit", function (e) {
2      e.preventDefault(); // Stop form submission
3      let self = jQuery(this);
4
5      if (jQuery("#loginusingwebauthn").is(":checked")) {
6          let username = jQuery("#user_login").val();
7          webauthn_authentication(username).then(function (promise){
8              console.log(promise)
9          });
10     }
11     self.unbind().submit();
12 });
```

Listing 4.16: WordPress Registration Interruption

4.2.3 Authentication Procedure

Allowing the user to authenticate using the *FIDO2 Authentication* schema is a bit more challenging than allowing them to register using the same schema.

Due to the reason that both registration and authentication pages are both under the same PHP page, in this case, the *wp-login.php*, the required JavaScript files are already linked with *WordPress* authentication page. The next step is to add a button that will allow the user to authenticate using the *FIDO2 Authentication* schema. This addition will be made at the HTML code within the *wp-login.php*, but at the authentication screen section. In listing 4.17 the addition of the *FIDO2* check box to the preexisting *WordPress* source code is shown (lines 8-11).

```
1  .
2  .
3  .
```

```
<form name="loginform" id="loginform" action="<?php echo esc_url( site_url( 'wp-login.php', 'login_post' ) ); ?>" method="post">
                                        .
                                        .
<p class="forgetmenot"><input name="rememberme" type="checkbox" id="rememberme" value="forever" <?php checked( $rememberme ); ?> /> <label for="rememberme"><?php esc_html_e( 'Remember Me' ); ?></label></p>
    <p>
        <input name="loginusingwebauthn" type="checkbox" id="loginusingwebauthn" />
        <label for="loginusingwebauthn"><?php esc_html_e( 'Use Webauthn Standart' )?></label>
    </p>
                                        .
                                        .
                                        .
```

Listing 4.17: FIDO2 Authentication Checkbox for authentication in HTML code

To pass the authentication flow to the *auth.js*, an interruption before submitting the login form is needed. Listing 4.18 shows how the interruption is declared within the *webauth_interruption.js* file. Same as before, if the user checks the *'Use Webauthn Standart'* then the control will be given to the *webauthn_authentication* until the *RP* accepts or declines the authentication. After the *RP* returns a response, then the form submission is resumed. *WordPress* procedures will continue the authentication process by verifying the placed cookie.

```
//login interruption
jQuery("#loginform").on("submit", function (e) {
    e.preventDefault(); // Stop form submission
    let self = jQuery(this);

    if (jQuery("#loginusingwebauthn").is(":checked")) {
        let username = jQuery("#user_login").val();
        webauthn_authentication(username).then(function (promise){
            console.log(promise)
            self.unbind().submit();
        });

    }else{
        self.unbind().submit();
    }
});
```

Listing 4.18: WordPress Authentication Interruption

A new authentication procedure must be added to *WordPress* source code to allow the authentication process to not only check for username and password but also for a specific cookie. The basic idea behind this new procedure is that given the cookie that *RP* placed and the username submitted from the registration form, this procedure will retrieve the user's data from the *WordPress's* database and lead the authentication procedure that *WordPress* is using to continue executing. Listing 4.19 shows the procedure definition within the *user.php* file. This procedure will first read the data within the *webauthn_login* cookie. Using the *$username* parameter that is passed from the submitted form, the external *cookie.py* script will be called. This external script using the public-private key pair that *RP* is using for signing the cookie will try to verify that the signature within the *webauthn_login* cookie using the provided cookie. If the verification is completed successfully, then the procedure will return an *WP_User* object for the provided user. If the verification fails, then an error will be returned. Finally, to link the aforementioned function with the authentication procedure, a new filter must be added within the *default-filters.php* file. The filter can be seen at listing 4.20.

```
function wp_authenticate_using_webauthn( $user, $username ) {
    if ( $user instanceof WP_User ) {
        return $username;
    }
    if ( empty( $username )) {
        if ( is_wp_error( $user ) ) {
            return $user;
        }

        $error = new WP_Error();

        if ( empty( $username ) ) {
            $error->add( 'empty_username', __( '<strong>Error</strong>: The username field is empty.' ) );
        }
        return $error;
    }
    $user = get_user_by( 'login', $username );
    if ( ! $user ) {
        return new WP_Error(
            'invalid_username',
            __( 'Unknown username. Check again or try your email address.' )
```

```
22          );
23      }
24
25      if (!isset($_COOKIE['webauthn_login'])) {
26          $error = new WP_Error();
27          $error->add( 'missing_webauthn_cookie', __( '<strong>Error</strong>:
    Webauthn Authentication Cookie Missing' ) );
28          return $error;
29      }
30      $data= $_COOKIE["webauthn_login"];
31      $key_path="/var/www/wordpress.com/wp-content/public_key.pem";
32      $command = escapeshellcmd("python3 /var/www/wordpress.com/wp-content/cookie
    .py verify $key_path $data");
33      $output = exec($command);;
34      if(strpos($output, "ERROR:") == true){
35          $error = new WP_Error();
36          $error->add( 'verification_webauthn_cookie', __( '<strong>Error</strong
    >: Unable to verify Signature' ) );
37          return $error;
38      }
39      if(strcmp($username,$output)!==0){
40          $error = new WP_Error();
41          $error->add( 'verification_webauthn_cookie', __( '<strong>Error</strong
    >: Cookie Username and given username does not match' ) );
42          return $error;
43      }
44
45      return $user;
46  }
```

Listing 4.19: WordPress function definition for allowing the use of cookie for registration

```
1  // Default authentication filters.
2  add_filter( 'authenticate', 'wp_authenticate_using_webauthn', 20, 3 );
```

Listing 4.20: Enabling wp_authenticate_using_webauthn function using a new filter in WordPress

4.2.4 After the Authentication

This subsection focuses on the actions that the user can do, related with *FIDO2 Authentication* after they successfully logged in to the system using both the traditional authentication schema or the *FIDO2 Authentication*.

After the successful user authentication, the user can enrol a new key or delete all the saved ones. The way that *RP* distinguishes whether a user is authenticated is through the *webauthn_login* cookie. If the default username and password schema handled the authentication, then *WordPress* must generate and place the cookie to the browser to allow the user to manage its keys.

The pre-existing function that handles the cookie generation when authenticating is the *wp_set_auth_cookie* located within the *pluggable.php* file. Listing 4.21 shows some additional code that must be added which creates the *webauthn_login* cookie. The additional code after retrieving the user's username, the *cookie.py* script will be called. The script, using the given username and the pre-mentioned public/private key, will generate the cookie data. The structure of the cookie can be seen at listing 4.7.

```
1  function wp_set_auth_cookie( $user_id, $remember = false, $secure = '', $token
       = '' ) {
2                                              .
3                                              .
4                                              .
5      if (!isset($_COOKIE['wwebauthn_login'])) {
6          $data=get_userdata($user_id)->user_login;
7          $key_path="/var/www/wordpress.com/wp-content/private_key.pem";
8          $command = escapeshellcmd("python3 /var/www/wordpress.com/wp-content/
       cookie.py sign $key_path $data");
9          $output = exec($command);
10         if (!empty($output))
11             setrawcookie ( 'webauthn_login',"$output", $expire, COOKIEPATH,
       COOKIE_DOMAIN, false, true );
12     }
13 }
```

Listing 4.21: Generating *webauthn_login* cookie using WordPress

To allow users to access the *Remove All Keys* and the *Add This Device as a key* functions, some buttons must be added to the user's profile settings page. Listing 4.22 shows the HTML code additions within the *user-edit.php*. These changes include the placement of one new page subsection and two new buttons. The next step is to connect the buttons with the corresponding handle. As happened before, the handles are located within the *webauth_interruption.js*. Listings 4.23 and 4.24 demonstrate the *Add This Device as a key* and *Remove All Keys* interruption declaration. As before, both handles connect the corresponding function with the correct button.

```
                                        .
                                        .
                                        .
?php elseif ( ! IS_PROFILE_PAGE && $sessions->get_all() ) : ?>
  <tr class="user-sessions-wrap hide-if-no-js">
    <th><?php _e( 'Sessions' ); ?></th>
    <td>
      <p><button type="button" class="button" id="destroy-sessions"><?php _e( '
    Log Out Everywhere' ); ?></button></p>
      <p class="description">
        <?php
        /* translators: %s: User's display name. */
        printf( __( 'Log %s out of all locations.' ), $profileuser->
    display_name );
        ?>
      </p>
    </td>
  </tr>
<?php endif; ?>
  </table>
    <h2><?php _e( 'WebAuthn Keys' ); ?></h2>
    <table class="form-table" role="presentation">
        <tr id="Webauthn_key">
            <th><label><?php _e( 'Register New Key' ); ?></label></th>
            <td>
                <button id = "webauthnregcred" type="button" class="button wp-
    generate-pw hide-if-no-js" aria-expanded="false"><?php _e( 'Add This Device
     as a key' ); ?></button>
            </td>
        </tr>
        <tr id="Webauthn_key_del">
            <th><label for="pass1"><?php _e( 'Delete all Webauthn Keys' ); ?></
    label></th>
            <td>
                <button id = "webauthnredallkeys" type="button" class="button
    wp-generate-pw hide-if-no-js" aria-expanded="false" style="background-color
    :red; color:white"><?php _e( 'Remove All Keys' ); ?></button>
            </td>
        </tr>
    </table>
                                        .
                                        .
                                        .
```

Listing 4.22: WordPress HTML buttons addition

```
jQuery("#webauthnregcred").click(function(){
    let username = jQuery("#user_login").val();
    webauthn_registration(username, "credential").then(function (promise){
        console.log(promise)
        self.unbind().submit();
    });
});
```

Listing 4.23: WordPress Add New Key Interruption

```
jQuery("#webauthnredallkeys").click(function(){
    webauthn_delete_credentials()
        .then(function (promise){
        console.log(promise)
        alert("All keys Deleted")
    })
        .catch(function (promise){
            console.log(promise)
        });
});
```

Listing 4.24: WordPress Delete All Keys Interruption

Chapter 5

Evaluation

This section focuses on the performance evaluation of *auth.js*. More specifically, the *FIDO2 Authentication* schema will be used for evaluation purposes.

5.1 Setup

For the performance evaluation, the *WordPress* project and *Python3 RP service* were used as described in section 4. All necessary components such as Website, Server and Database were setup into an Ubuntu virtual server with the following specs:

- Ubuntu Version: 20.04.1 LTS
- Available Memory: 8GB
- Number Of Processors: 2
- Python Version: 3.8.5
- PHP Version: 8.0.1
- MYSQL Version: 8.0.25-0
- WebServer Version: NGINX 1.18.0

5.2 RP Evaluation

Table 5.2 shows the average time that the RP took to complete the registration and assertion ceremony. The browsers used to test the *auth.js* framework were Google Chrome, Microsoft Edge and Mozilla Firefox on both Windows

and Android based systems. Apple devices did not participate in the testing phase due to the *WebAuthn* standard not being fully supported yet. The testing was done using real users rather than an automated script due to the complexity of replaying the Authenticator output.

Table 5.1: *RP*:Average time for registration and assertion ceremony.

Ceremony	RP Average Time
Registration	263 ms
Assertion	43 ms

5.3 auth.js & WordPress Evaluation

Table 5.3 shows the total execution time the registration and authentication procedures need to complete. The total time is calculated by adding the time the RP took to complete each procedure and the time that *PHP* needed to verify the cookie. In this specific set of tests, the communication time between server and client is considered negligible. The testing method is as described in the previous subsection. The timing results are highly dependent on the user's authenticator. Thus, the results will vary for each user.

Table 5.2: *Total*:Average time for registration and assertion ceremony.

Ceremony	Average WordPress Time	Average RP Time	Total Time
Registration	256ms	- ms	256 ms
Assertion	300ms	43 ms	343 ms

Chapter 6

Related Work

In recent years, there is a push to move from the default authentication schema to more secure ones. In the previous sections, the *FIDO2 Authentication* was described as a part of the *auth.js* framework. There are also other approaches that strengthen web authentication using different mechanisms and approaches.

Authentication schemes based on *PAKE* protocol allow the end-user to prove their identity by securely exchanging a secret. Using this method, the user does not send his actual password. Another password base authentication schema that increases the security of such schemes is the usage of *HoneyWords*. *HoneyWords* are slightly modified versions of the real password that are being stored along with the actual password. If an attack occurs and a *HoneyWord* is used, then a trigger will be enabled.

Some papers suggest that if some modifications are made to the Authenticator's components, it can increase the security level of the authentication schema. More specifically, *simTPM* aims to replace the vulnerable *TPM* on mobile platforms with a newer User-centric implementation that uses the *SIM card*. This approach allows for a mobile *TPM* that avoids additional hardware and allows credential portability between devices.

Others took a different approach and created a multi-factored authentication schema using *QR-Code*. Just as *FIDO2 Authentication*, *3CAuth* supports similar authentication methods and additionally supports users' registered numbers and smart cards. The *QR-Code* functionality is replacing the *SMS* method for retrieving the *OTP*.

Chapter 7

Conclusion

This paper focused on implementing the *FIDO2 Authentication* schema as one of the available authentication schemes that the *auth.js* framework supports. More specifically, *FIDO2 Authentication* uses *asymmetric cryptography* instead of *secrets*. *auth.js* focuses on enabling web developers to use newer and more advanced authentication schemes effortlessly. The framework itself is written in *Javascript,* and it is considered a trusted component of the authentication schema. At this point, *auth.js* supports two authentication schemes based on *asymmetric encryption* and the primary password base schema. Finally, *auth.js* framework was tested by allowing WordPress to use *FIDO2 Authentication* as the primary authentication schema. This testing required some crucial modifications and additions to the preexisting authentication schemes. Performance-wise, the RP can take up to 300 ms to complete each ceremony. This timing will vary for different *FTMS* as verifications differ. The most considerable timing overhead comes from the user's side and specifically from the authenticator.

Bibliography

[1] A demo of the webauthn specification. https://webauthn.io/.

[2] Enabling strong authentication with webauthn | google developers. https://developers.google.com/web/updates/2018/05/webauthn.

[3] Guide to web authentication. https://webauthn.guide/.

[4] Mdn web docs. https://developer.mozilla.org/en-US/.

[5] D. Chakraborty and S. Bugiel. simfido: Fido2 user authentication with simtpm: Semantic scholar. https://www.semanticscholar.org/paper/simFIDO:-FIDO2-User-Authentication-with-simTPM-Chakraborty-Bugiel/7ee68251acb59237fe215e88a3d0a2d4dd4a75c5, 2019.

[6] R. S. Chowhan and R. Tanwar. Password-less authentication. *Machine Learning and Cognitive Science Applications in Cyber Security Advances in Computational Intelligence and Robotics*, page 190–212, 2019.

[7] D. D. H. Latha, A. Mubeen, and D. D. R. K. Reddy. Password substantiation with negative password encryption. *International Journal of Advanced Research in Science, Communication and Technology*, page 184–190, 2020.

[8] K.-C. Liao and W.-H. Lee. A novel user authentication scheme based on qr-code. *Journal of Networks*, 5(8), 2010.

[9] R. Libfeld. What is client to authenticator protocol (ctap)?: Security wiki. https://doubleoctopus.com/security-wiki/protocol/client-to-authenticator-protocol/, journal=Secret Double Octopus.