University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc Digital Systems Security

MSc Dissertation

Integration of OpenID Connect with FIDO UAF for Android Environments

Supervisor Professor: Xenakis Christos

Name-Surname	E-mail	Student ID.
Makropodis Ioannis	mte1917@ssl-unipi.gr	mte1917

Piraeus

20/12/2021

Special Thanks to Prof.Xenakis for his guidance and to my colleague Ioannis Chouchoulis for his help.

Περίληψη

Το Single Sign-On μεσω του OpenID Connect πρωτοκόλλου είναι ένα ευρέως διαδεδομένο πρωτόκολλο ελέγχου ταυτότητας με ανάθεση. Είναι φτιαγμένο πάνω από το OAuth 2.0 το οποίο παρέχει εξουσιοδότηση. Αυτό το πλαίσιο πρωτοκόλλου επιτρέπει στους χρήστες να συνδέονται με πολλούς Παρόχους Υπηρεσιών με τους λογαριασμούς τους, που προσδιορίζονται από έναν Πάροχο Ταυτότητας. Τον τελευταίο καιρό όλο και περισσότερες αναφόρες γίνονται γύρω απο την ακαταλληλότητα του προτύπου αυθεντικοποίησης μέσω ονομάτος χρήστη και κωδικό , με τα ερευνητικά δεδομένα να δείχνουν το FIDO πρωτόκολλο ως την καταλληλότερη λύση για την αντιμετώπιση αυτού του προβλήματος.Το FIDO είναι ένας νέος μηχανισμός ελέγχου ταυτότητας που αντικαθιστά τους κωδικούς πρόσβασης, απλοποιώντας τη διαδικασία ελέγχου ταυτότητας νέου χρήστη. Σε αυτή τη μεταπτυχιακή διατριβή, θα περιγράψουμε πώς αυτά τα δύο πρωτόκολλα μπορούν να συνδυαστούν προκειμένου να δημιουργηθεί ένα πιλοτικό πλαίσιο διαχείρισης ταυτότητας που παρέχει τόσο ισχυρό έλεγχο ταυτότητας όσο και ισχυρή εξουσιοδότηση. Κύριο μέλημά μας είναι οι χρήστες, να μπορούν να χρησιμοποιούν ένα κινητό τηλέφωνο με βιομετρικό έλεγχο ταυτότητας για πρόσβαση στην υπηρεσία web της επιλογής τους.

Abstract

Single Sign-On with OpenID Connect is a widely adopted delegated authentication protocol. It is a layer above OAuth 2.0 which provides delegated authorization. This protocol framework allows users to connect to several Service Providers with their accounts, identified from a single Identity provider. Recently, more and more reports are being made about the inadequacy of username and password authentication scheme, with literature demonstrating the FIDO protocol as the most appropriate solution to address this problem. The FIDO is a new authentication mechanism that replaces passwords, simplifying the process of new user authentication. In this master thesis, we will describe how these two protocols can be combined in order to build a pilot Identity management framework that provides both strong authentication and strong authorization. Our main concern is that users, can use a mobile phone with biometric authentication to access the web service of their choice.

Table of Content

1.Introduction

Main technology providers have been trying for years to leverage existing user accounts in order to provide new services regarding identity and access management, while users have been looking for effortless solutions allowing them to consume different services from different devices with a Single Sign-On approach (SSO). Federated Identity Management (FIM) allows end users to access different resources, applications and services through a single Identity Provider (Idp) avoiding the need of having an account for each resource, application or service. FIM specifications have been massively adopted in mobile environments during the last years. Facebook, Google, LinkedIn, Microsoft, Amazon are some of the most important examples, which are actively supporting standards such as OAuth 2.0 or OpenID Connect, becoming in many cases identity providers. This last specification is one of the newest and most widely deployed single-sign-on protocols on the web. OpenID Connect (OIDC) is a protocol for delegated authentication in the web. A user can log into a relying party (RP) by authenticating himself/herself at an identity provider. It builds upon the OAuth 2.0 framework which define an authorization protocol.

Today's Federated Identity Management systems are diligent in vulnerabilities. Most common identity providers ask a user to authenticate through the username password scheme in order to issue short lived bearer identity assertions or tokens. By using these tokens, a user can have access to a Service provider (SP). Recent studies show that accounts on single sign-on (SSO) systems are a target for password spraying attacks. Targeting federated authentication can help mask malicious traffic. Additionally, by attacking these specific areas, hackers can obtain widespread access to networks and compromise or steal a greater amount of data. [1]

Fast Identity Online (FIDO) is an authentication mechanism that replaces passwords and simplifying the process of user authentication. FIDO Alliance specified three authentication frameworks and protocols: The Universal Authentication Framework (UAF) for password-less authentication from smart devices, the Universal Second Factor protocol (U2F) for two-factor authentication using a small hardware token to accompany a non-FIDO smart device having a FIDO compliant web, and FIDO 2 which is combined into the W3C Web Authentication Recommendation. FIDO UAF mechanism provides several important advantages as it offers strong authentication due to its reliance on public key cryptography, it simplifies the registration and the authentication, it reduces the need for maintaining passwords, and it strengthens user privacy since all identifying information is stored locally at the user's device.

Bearing in mind all the above, we built a pilot Identity management framework that is based on the combination of OpenID Connect and FIDO UAF protocols. In this way, we tried to maintain all the advantages offered by SSO systems by adding FIDO UAF usability and advanced security features. This framework is based solely on open-source solutions. We used Keycloak as Idp. Keycloak is an open-source Identity and Access Management tool, it provides single sign-on as well as session management capabilities, allowing users to access multiple applications, while only having to authenticate once. This approach provides a higher level of security as applications do not have direct access to user credentials. As FIDO UAF implementation, we used the FIDO UAF server provided by eBay, extending it to be able to connect and work harmoniously with Keycloak. In this implementation, the user will have to authenticate biometrically using his android phone in order for Idp to issue authentication tokens for access to web services. We implemented the whole system and proved its efficiency in a university student registration scenario.

2. Single Sign-On with OpenID Connect

2.1 OAuth 2.0

Identity delegation is a feature during which, an entity delegates his or her authority to use identity information to another entity. Some emerging technical specifications provide schemes for exchanging identity information using a short string token. OAuth 2.0 is by far a massively popular industry-standard protocol for authorization. Prior to Oauth 2.0 there was OAuth 1, but this approach was complex or not easily interoperable. Oauth 2.0 focuses on client developer while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

OAuth 2.0 is useful both in dealing with third-party applications since through the OAuth 2.0 framework it activates an ecosystem of websites to interact with each other, as well as in own applications since it allows the possibility for limited access. It is not uncommon for third-party applications to ask users to enter their username and password on other sites, which is also the case within a business. Applications would, for example, ask for your LDAP username and password, which would then be used to access other services within an enterprise. Such an example in practice could have dire consequences since if an application is compromised, all services within the business could also be compromised.

2.1.1 Roles in OAuth

There are four different factors or roles involved in making OAuth 2.0 work. These four roles should interact properly with each other in order for the protocol to work uneventfully:

- Resource Owner: As Resource Owner, we define, the end user that owns the resources, that an application wants to access.

- Resource Server: This is the service hosting the protected resources. No one can access these resources without the right credentials (OAuth2.0 access token).

- Client: Client is an application that is authorized by the Resource Owner to access its resources.

- Authorization Server: Authorization Server is the server that, once there is a successful authorization for Resource Owner resources, will issue an access token to the client.

In a brief description, in an OAuth 2.0 protocol flow, the client makes a request to the Authorization Server in order to access a Resource Owner resource. The Authorization server in turn will issue an access token that allows the user to access the resource for a limited time. After receiving the access token, the client by sending it with a request to the resource server, will gain access to the resource.

2.1.2 Authorization Grant

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) and it is used by the client to obtain an access token. In order to be able to cover different types of applications, OAuth 2.0 identifies five different protocol flows. Below we will describe these five flows, emphasizing in which cases is proper to use them [2]:

1. Client Credentials flow: It is used in cases where the application wishes to gain access to the resource on its behalf, i.e., if the application is also the source owner.
2. Device flow: It is used in cases where the application is running on a device without a browser or is input constrained such as smart TV where it would be difficult for the user to enter his/her credentials.
3. Authorization Code flow: In this type of grant flow, we use an authorization server as an intermediate between the client and the resource owner. The Client in order to obtain the authorization code, directs the Resource owner to the Authorization Server. The Authorization server authenticates the resource owner and gain authorization. The resource owner’s credentials are never shared with the client. Finally, the resource owner returns back to the client with an authorization code. If none of the preceding conditions are applicable it is recommended to use Authorization Code flow.
4. Implicit flow: The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In this process instead of issuing an authorization code

to the client, an access token is issued directly to the client. During the implicit authorization process, the authorization server does not authenticate the client. Despite the fact that the implicit authorization improves the responsiveness and the efficiency of certain clients, it is considered as insecure and should not be used.

5. Resource Owner Password Credentials flow: In this flow, the application collects the user’s credentials directly and exchanges them for an access token. Credentials such as authorization codes should only be used when there is a high degree of trust between the resource owner and the client. This grant type is likely to be used in cases where a browser is not available or even when you want the login form to be integrated directly with your application. It is inherently insecure as you are exposing the user's credentials directly to the application, and you will also run into other problems in the long run, when you want your users to use stronger authentication than only a password, for example.

2.1.3 Token and Endpoints

An access token is used to access protected resources and is generated by the Authorization Server. More specifically, the access token is a string that represents the client's authorization. This string is usually opaque to the client. The token represents a specific access scale and duration, granted by the resource owner, and enforced by the resource server and the authorization server. In order for a client to have access to the protected resources of the Resource Owner, each request it sends must pass the token either to the HTTP header or as a body parameter. For security reasons, the Access token should have different formats, structures, and usage methods (for example, encryption attributes). To allow clients to obtain new access tokens without going through the complete flow, a refresh token is used. A refresh token should be kept secured by the client and can be used to obtain a new access token when the current one is invalid or expired or to obtain additional access tokens with the same or narrower range.

In essence, for the Oauth 2.0 protocol to work properly there must be some specific endpoints that will guide the communication between Client and Authorization Server. These endpoints are [3]:

- Authorization Endpoint: The Authorization Endpoint is used by the Client to redirect the end-user to the Authorization Server in order to be identified and authorized for specific resources.
- Redirection Endpoint: In case of successful authentication and authorization, the Authorization Server should know the endpoint to redirect the end-user. The Redirection Endpoint should be registered in the Authorization Server. When a client sends the authorization request to the Authorization Server, it must also send the redirection endpoint. In order to accept this request, the Authorization Server will check on its basis whether the specific endpoint is declared for the specific client.
- Token Endpoint: In order for a client to obtain a Token, it must first obtain an Authorization Grant from the Authorization Server. It will then send the Authorization Grant via an HTTP POST request to the Token endpoint for verification so that if it is correct to issue a Token for the client.

2.1.4 Protocol Flow

The following diagram describes the communication of the roles when implementing Authorization Code flow in OAuth 2.0.

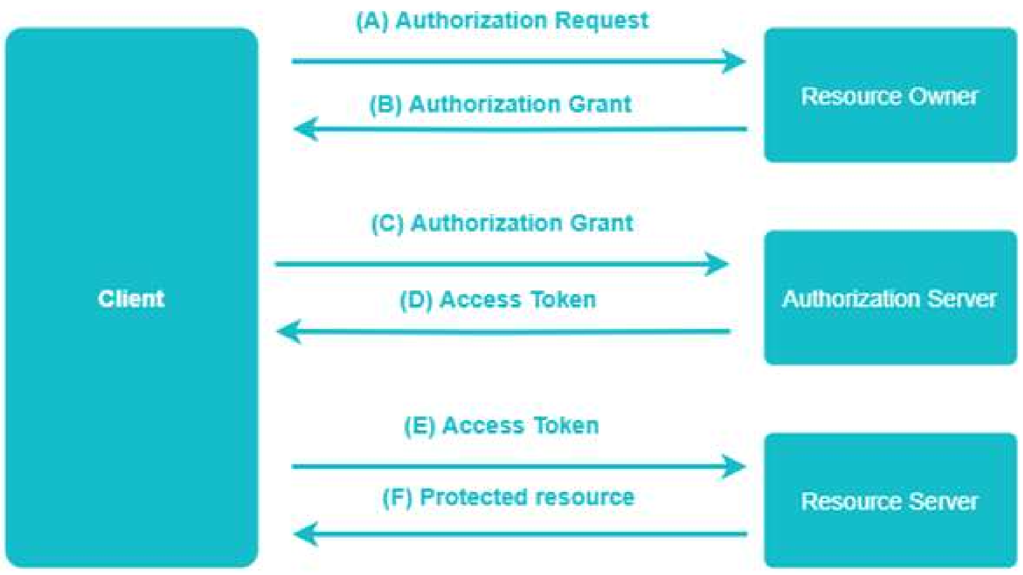


Figure 1. OAuth 2.0 Authorization Code grant type

A. Authorization Request: In order for the Client to gain access to Resource Owner's Protected Resources, an authorization request is sent to the resource owner directly from the client or indirectly through the Authorization Server (preferred).

B. Authorization Response: If the resource owner's authorization of the client is successful, an authorization grant will be sent as response to the client. There are four grant types in this specification.

C. Access Token Request: After acquires the Authorization Grant, the client requests the authorization server by sending the Authorization Grant in order to obtain an access token.

D. Access Token Response: The authorization server authenticates the client and validates the authorization grant. If the authorization grant is valid the authorization server response the client with an access token.

E. Resource Request: Client sends a request for accessing the protected resources to Resource Server which also contains the Access Token for authentication.

F. Protected Resource: Once the Resource Server receives a valid Access Token then it replies with the Protected resource.

OAuth 2.0 flow consists of two types of Clients, which are confidential and public clients. Confidential clients, on the one hand, are applications such as a server-side web application that is able to safely store credentials which they can use, to authenticate with the authorization server. On the other hand, public clients, are client-side applications that are not able to securely store credentials. As public clients cannot authenticate with an authorization server, there are two safeguards:

1. The authorization server will only send authorization code to an application hosted on a pre-configured URL, in the form of previously registered redirect URI.
2. Proof Key for Code Exchange (PKCE) [4] is an extension to OAuth2.0 which prevents anyone trying to steal an authorization code from exchanging it for an

access token. PKCE protects the authorization code in the redirect. In the authorization code flow the authorization server issues the authorization code, sends it in the URL back to the user's browser, the user's browser delivers it back to the application and the application exchanges it for an access token. So, when the authorization server issues that temporary authorization code and it's handing it off the user's browser, the main problem is that the authorization server can never actually be sure if it landed back at the right application or not. So, when the authorization server goes and issues this access token it is not sure if that it is actually the right application that's bring the code back. PKCE introduces a few new things to the authorization code flow, a code verifier and a code challenge. The code verifier is a random code and the code challenger is a hash transformation (SHA 256) of the code verifier or in some cases it can be the code verifier as plaintext (Not the correct way). Both the code verifier and the code challenger are created by the client App. So, the basic flow of the PKCE is:

A. The client sends the authorization request along with the code_challenge and the code_challenge_method.

B. The Authorization Server makes note of the code_challenge and the code_challenge_method and issues an authorization code.

C. The client sends an access token request along with the code_verifier.

D. The Authorization Server validates the code_verifier with the already received code_challenge and the code_challenge_method and issues an access token if the validation is successful.

The following diagram will help you understand how it works:

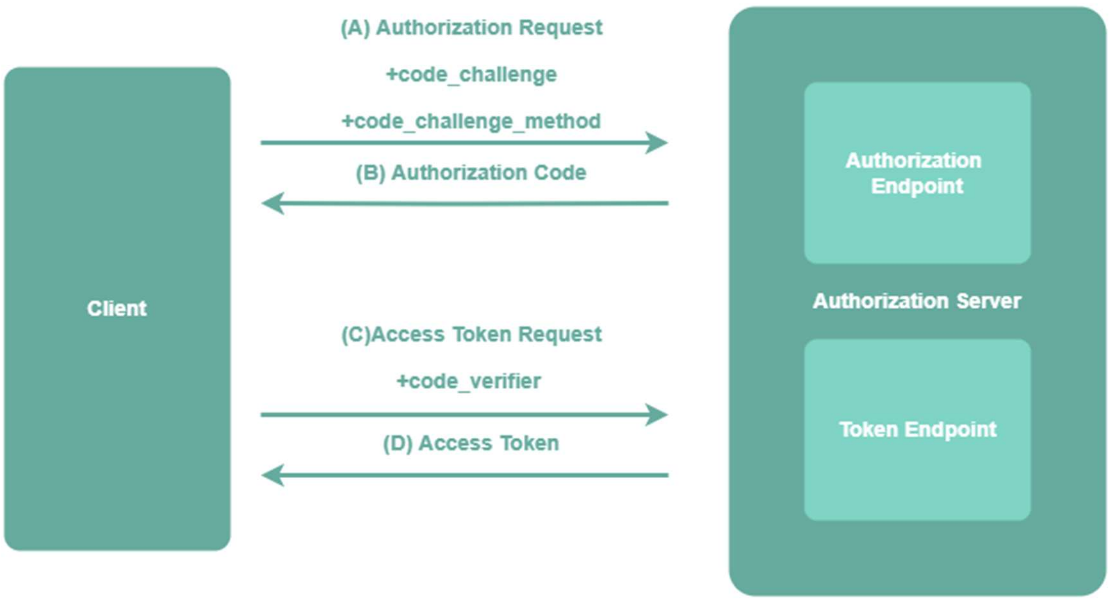


Figure 2. PKCE Flow

In addition to the core OAuth 2.0 framework there are a few additional specifications:

- Bearer Tokens [5]: Bearer tokens are the most commonly used type of access token and they are sent to resource servers through the HTTP Authorization header. They can also be sent in the form-encoded body, or as a query parameter (should be avoided).
- Token Introspection [6]: In Oauth 2.0 the access tokens are opaque for the application, which means that they have a format that is not intended to be read by the application. The token introspection endpoint allows the client to obtain information about the token access without having to understand its format.
- Token Revocation [7]: Token Revocation endpoint covers the issue of how the access tokens should revoke.

2.2 OpenID Connect

The OAuth 2.0 protocol, that we aforementioned above is an authorization protocol, so it does not cover authentication. OpenID Connect is built on top of the OAuth 2.0 protocol and adds the authentication layer that is missing. The combination of OAuth 2.0 and OpenID Connect is a Single Sign-On (SSO) mechanism that can offer both authentication and authorization to resources and services, through a third party, so that the end-user does not have to create new login credentials for a specific client [8]. It is widely used on the internet so that clients can authenticate and authorize their users to specific resources and services.

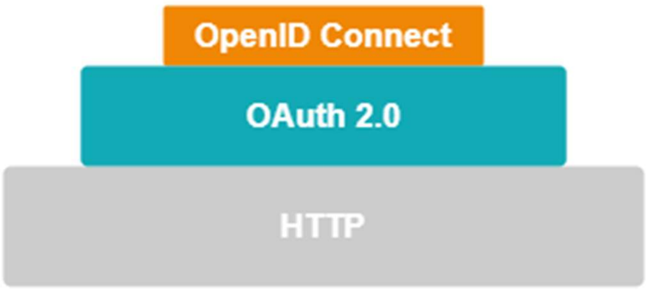


Figure 3 OAuth 2.0 & OpenID Connect

At the heart of OpenID Connect is the OpenID Connect Core Specification, which activates a whole ecosystem of websites to no longer need to deal with user management and the authentication of the users. The advantages that offers to a common user is to significantly reduce the number of times it needs to be authenticated, while at the same time it reduces the number of passwords it has to manage, especially if each website uses a unique password. The most widely used example of using OpenID connect is the sign-in on websites using Google or other social networks. However, OpenID Connect is not only activated through social login but is also very useful in the enterprise in order to have a centralized solution for authentication, supporting single sign-on. This significantly increases security since an application does not have access to the user credentials directly. It also enables the use of stronger authentication, such as OTP or WebAuthn, without the need to support it directly within applications. Finally, we would like to emphasize that OpenID Connect not only enables easy authentication within the enterprise but also allows third parties such as employees in partner companies, to access applications within your enterprise without having to create separate personal accounts.

2.2.1 Roles in OpenID Connect

There are four roles that participate in the OpenID Connect protocol [8]:

• End-User: Is the human who wants to be authenticated to the Relying Party through the OpenID . We could say that it is equivalent to resource owner in OAuth 2.0.

• Relying Party (RP): It is the application that is located between the End User and the OpenID Provider and wants to authenticate the user. It is called relying party since it is a party that relies on the OpenID provider in order to verify the identity of the user.

- OpenID Provider (OP): It is essentially the entity that identifies the End-User in the Relying party by generating an ID token and will authorize the Client to specific resources, issuing an OAuth 2.0 access token. As we will see later, this is the role of Keycloak.

The OpenID Connect protocol, in abstract, follows the following steps:

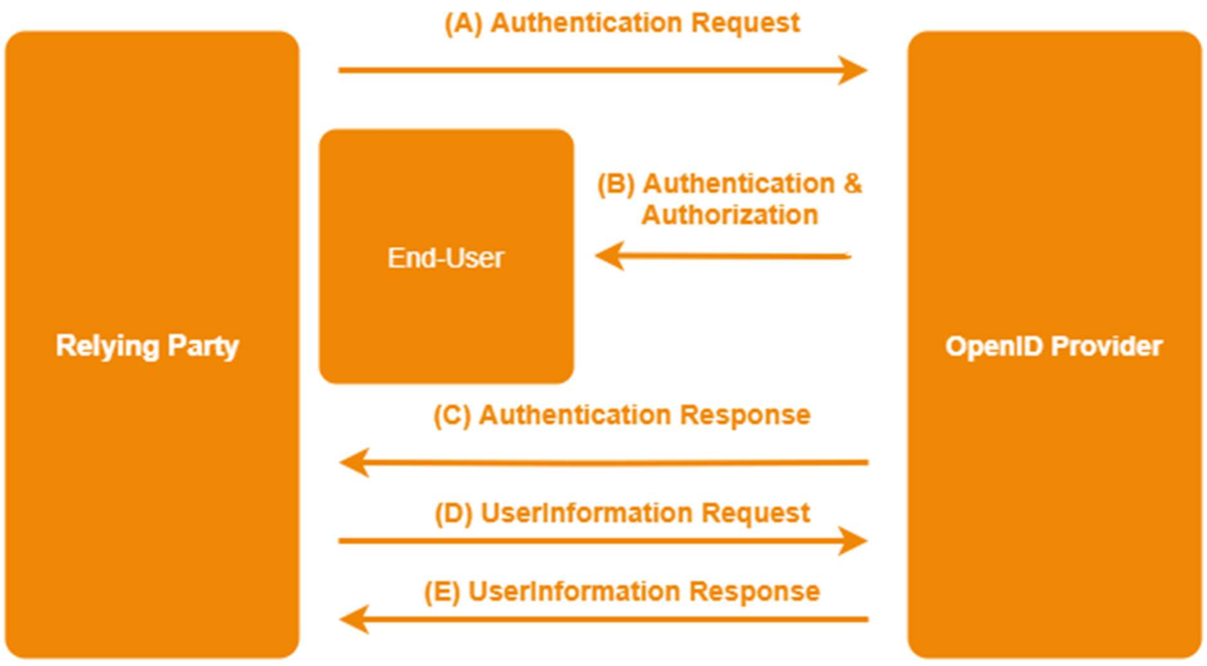
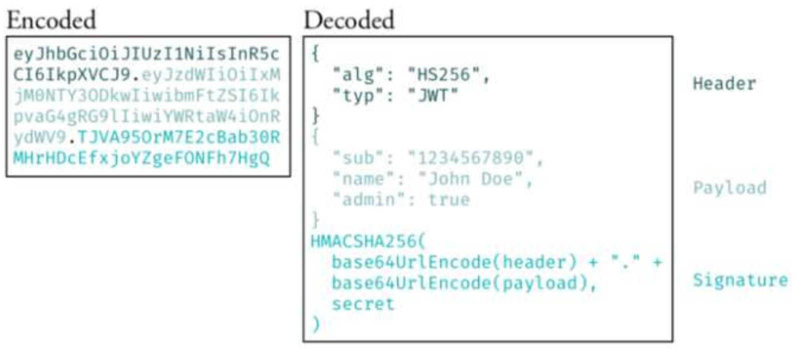

(A) Authentication Request
Relying Party
End-User
(B) Authentication & Authorization
OpenID Provider
(C) Authentication Response
(D) UserInformation Request
(E) UserInformation Response

Figure 4 OpenID Connect Abstract flow simplified

A. The RP (Client) sends a request to the OpenID Provider (OP).
B. The OP authenticates the End-User and obtains authorization.
C. The OP responds with an ID Token and usually an Access Token.
D. The RP can send a request with the Access Token to the UserInfo Endpoint.
E. The UserInfo Endpoint returns Claims about the End-User.

2.2.2 ID Token

The ID Token is a secure Token that contains values (claims) for the authentication of the End-User in the OP, thus proving the identity of the End-User to the Client. In essence, the ID Token data structure is the primary extension that OpenID Connect makes to OAuth 2.0 to enable End-Users to be Authenticated. OpenID Connect clearly specifies the format of the token ID by leveraging the Jason Web Token (JWT) specification, which unlike the access token in OAuth 2.0 is not opaque. ID token has a well-specified format, and the claims can be read directly by the client. In order to maintain the integrity and authenticity of Tokens, JSON Web Signatures (JWS) are used, while JSONWeb Encryption (JWE) is used to maintain confidentiality.



```
eyJhbGciOiJIUzI1NiIsInR5c
CI6IkpXVCJ9.eyJzdWIiOiIxM
jM0NTY3ODkwIiwibmFtZSI6Ik
pvaG4gRG9lIiwiYWRtaW4iOnR
ydWV9.TJVA95OrM7E2cBab30R
MHrHDcEfxjoYZgeFONFh7HgQ
```

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

Figure 5 ID Token Structure

2.2.3 OpenID Connect Endpoints

OpenID Connect has specific endpoints, each of which has a specific purpose in terms of communication of the roles within the protocol. Below we will mention the most important:

- Authorization Endpoint: In order to authenticate and give the Client access to specific resources, an End-User will be redirected to the authorization endpoint. This endpoint is located in the OP where the End-User has an account.

- Token Endpoint: The Token Endpoint is located in the OP and is deployed every time the Authorization Code or Hybrid Flow is used to obtain an ID Token or an Access Token or both.

- JSON Web Key Set Endpoint: This endpoint is located in the OP and is used by the Client to validate and decrypt an ID Token from the OP, each time is used asymmetric cryptography. From this endpoint, we can get the JSON Web Key Set (JWKS) which contains all the necessary public keys.

- UserInfo Endpoint: This endpoint can be invoked by the end-user with an access token and return the same standard claims as those contained in the ID Token.

- Dynamic Registration Endpoint: This particular Endpoint allows the Clients to dynamically register themselves with the OpenID Provider.

- Discovery: This Endpoint allows the Clients to dynamically discover information about the OpenID Provider.

- Session Management: Defines how the client can initiate a logout and also how to monitor the end user’s authentication session with the OP.

- Front-Channel Logout: Defines a mechanism for single sign-out of multiple applications using embedded iframes.

- Back-Channel Logout: Defines a mechanism for single sign-out of multiple applications using a back-channel request mechanism.

2.2.3 Protocol Flows

OpenID Connect is based on the relationship of trust between two entities, when a Relying party wants to authenticate an End-User it requests the OpenID provider for its identity, so we know that every time we use OpenID connect, an RP trusts one or more OPs for the identity of an End-user. OpenID Connect supports two types of clients, web clients, and mobile/native clients.

In the OpenID Connect protocol flow, Relying Party requests the identity of the end-user from the OpenID Provider. As aforementioned, OpenID Connect is built on top of OAuth 2.0 and for this reason, as the user's identity is requested, at the same time can be obtained an access token. OpenID Connect utilizes the Authorization Code grant type from OAuth 2.0, with the difference that the client defines scope=openid in the

initial request, so in this way, the request is more about authentication than authorization.
OpenID Connect has three different authentication flows to serve different types of applications. These three paths are:

1. **Authorization Code Flow**: This type of flow is used for server-side applications satisfying the role of the Client in the protocol. This flow is similar to OAuth 2.0 Authorization Code grant type because it returns an authorization code that is exchanged for an ID token, an access token, and a refresh token. In this flow, all Tokens are returned by the Token Endpoint and they are not appearing to the End-User. The issued tokens are not revealed to User Agent (Front-end application). Although when performing a request to the token endpoint of the OP, the client (RP) must authenticate with its credential.

2. **Implicit Flow**: This type of flow is used for end-user-sided applications that run in the web browser and are written in a scripting language (such as JavaScript). In this specific flow, all Tokens (ID and optionally access token) are returned by the Authorization Endpoint. This process happens via the front channel. We don't recommend the use of the Implicit flow at all because the token values are stored in the browser memory and thus are more vulnerable to security threats.

3. **Hybrid Flow**: This type of flow is used mainly in native applications. In Order to obtain Tokens both Authorization and Token Endpoints are used. In the Hybrid flow, the ID token is returned from the initial request alongside an authorization code. This process is practically a combination of the two previous methods and is rarely used.

Below we will analyse the Authorization Code Flow step by step. We chose this type of flow because it is used more by developers to build Clients in the protocol.

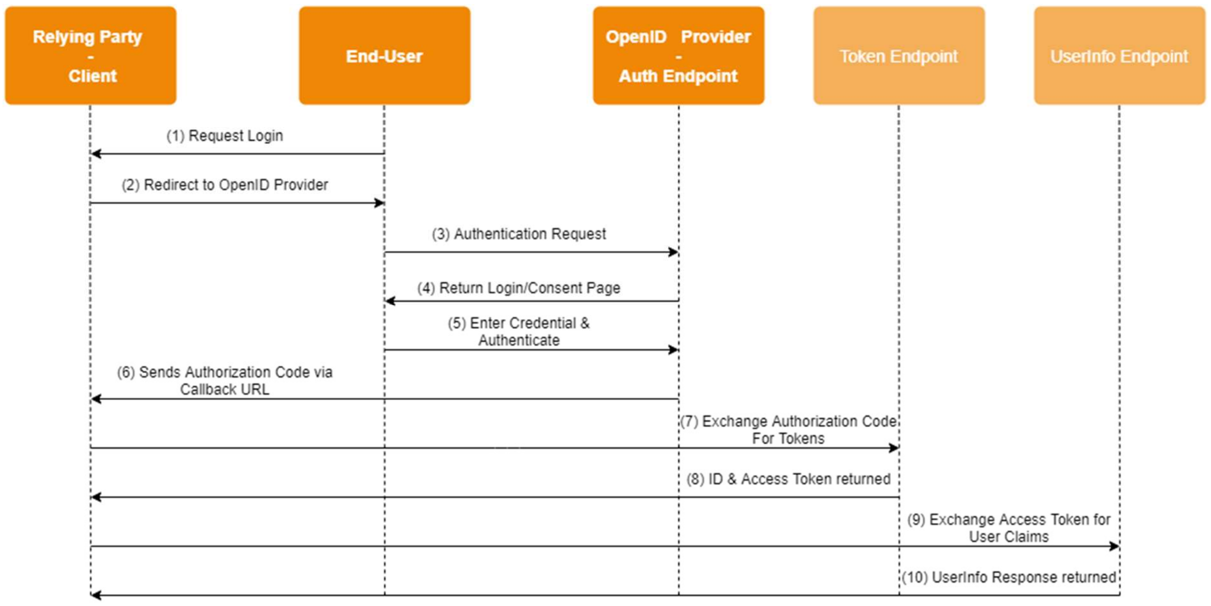

Relying Party - Client
End-User
OpenID Provider - Auth Endpoint
Token Endpoint
UserInfo Endpoint
(1) Request Login
(2) Redirect to OpenID Provider
(3) Authentication Request
(4) Return Login/Consent Page
(5) Enter Credential & Authenticate
(6) Sends Authorization Code via Callback URL
(7) Exchange Authorization Code For Tokens
(8) ID & Access Token returned
(9) Exchange Access Token for User Claims
(10) UserInfo Response returned

Figure 6- Authorization Code Flow

The Authorization Code Flow goes through the following steps:

1. End-User wants to Login to the Client so that it can be authenticated and access the protected resources.
2. The Client will be redirect to the Open id Provider.
3.The user request for authentication to the OpenID Provider.
4. The user will be redirected to the OP Login Page
5. The user will enter his credentials in the OP in order to be authenticated and authorized to specific resources or services.
6. The user after authenticating to the OP based on the Authorization Code will redirect to the Client.
7. The Client will then request the Token Endpoint in order to receive the Access Token and ID Token.
8. The OP and more specifically the Token Endpoint of the OP should respond the client’s request by returning the Tokens.
9. Then steps 9 and 10 are optional. If the Client wants to receive more information about the End-User will request then the OP and more specifically, the UserInfo Endpoint of the OP by sending the Access Token.
10. The OP will respond to the Client's request by sending him information about user.

3. FIDO UAF

FIDO UAF is an authentication mechanism whose architecture is based on public-key cryptography and was designed to replace password-based authentication [9].The protocol's architecture as shown in Figure 7 consists of 6 components, the User agent, the UAF Client, the UAF ASM, the UAF Authenticator, the Web server and the UAF server. The first four components are deployed on the user's device while the last two are based on the relying party. The user device represents the client and interacts with the user. Its main functions are to generate and store a unique pair of authentication keys as well as to respond to the challenge posed by the server. The Relying party is essentially a server that generates the challenge (in order to initiate the challenge-response mechanism), verifies and stores the user credentials (e.g. name, authentication keys) [10].These two entities communicate with each other using a secure transport protocol such as TLS/HTTPS.

The UAF authenticator is an entity that can either be inserted (such as a USB hardware device with a pin code protection) or embedded (such as fingerprint sensor or face recognition) in the user's device. In android environments, it is recommended that the development of the authenticator should be part of the TEE module. The UAF authenticator has an internal matcher for user verification, also it has a model identifier and an asymmetric attestation key stored (AAID, Uatt.pub). This asymmetric key is used in the registration operation. In addition, it generates the asymmetric authentication key pair (Uauth.pub, Uauth.priv) in the registration process, which are used during authentication.

The Authenticator Specific Module (ASM) is a software interface between the UAF client and the UAF authenticator that provided a uniform API to the upper layer. It resembles such as an abstract layer that helps the UAF client to serve a variety of UAF authenticators with different biometric factors. The first time the ASM is launched it creates a secret token (tok) [11].

The UAF Client (UC) is an application or a system service that implements the client-side logic of the UAF protocol [12]. It interacts with diverse authenticators through the ASM and with the UAF server through the Relying Party (RP). The ASM can retrieve the Caller ID value from the operating system which is used to identify the UAF client. On Android, the value of the CallerID is the result of the UAF Client's APK signing certificate hash.

The User Agent (UA) is a user application that interacts with the user and when the user enables biometric authentication, it initiates the whole operation. It is identified by a URI named FacetID. FacetID value can be the origin of the web page triggering the UAF operation (e.g. https://fidouaf.com) when the user agent is a browser or it can be the hash of the user agent's APK signing certificate when the user agent is an Application on android. Also, on android environments, the UAF Clients and the UAF ASM can be independent applications separated from the User Agent or built-in modules of the User Agent [10].

The Relying party consists of the web server and the UAF server. The web server provides the user application service and interacts with the UAF server to transfer the protocol messages. The UAF server is responsible for communicating with the client. It ensures that only trusted authenticators can be registered, also manages the association of the authenticators to user accounts by updating the public key related to the user, evaluates the user authenticator, and verifies the response message.

At the high level, the UAF protocol works as follows: A user wants to log in to a web service using a valid UAF authenticator e.g., fingerprint, facial recognition etc. The authenticator contains a pair of attestation keys (RSA or ECDSA). The user logs in to a web service using his original credentials e.g., his text-based username. The authenticator will record its authentication (e.g., fingerprint), generates a pair of authentication keys for this website, signs the public part of the new keys with the attestation key, and send them to the web service. The web service link's the user's online profile with the authentication key. If the above process is successfully completed, then a bond of trust between relying party and authenticator is established and the authenticator registration process comes to an end. In all login attempts that will follow, the user will have to prove his identity to the local authenticator, in a process in which, the relying party exercises the challenge–response technique with the authenticator by using the authentication key

3.1 UAF Attestation types

In FIDO protocol as attestation is referred the capability of a FIDO Authenticator to provide a cryptographic proof about its model to a Relying party. FIDO UAF have 3 types of attestation [13]:

1. **Basic Full**

   A group of authenticators with common characteristics (e.g., same model), occupies an attestation certificate and an attestation private key which they use to sign the registration object.

2. **Basic Surrogate**

   The key registration object is signed using Uauth.priv key and it does not provide any cryptographic proof of the authenticator's security characteristics. This attestation type is used when the authenticator is not able to own an attestation private key.

3. **ECDAA**

   In this attestation type, the trust in the authenticator is achieved by using Direct Anonymous Attestation cryptographic scheme (DAA) with elliptic curves. This is more secure than basic full attestation and the usage of "group keys" because in ECDAA if the key is stolen it does not affect other authenticators. An alternative solution to group keys is the use of individual keys combined with a Privacy-CA . Nevertheless, this kind of solution involves a third party and new risks such as threats on user's privacy and high availability requirements on behalf of the Privacy-CA [13].

3.2 UAF Client Trusted Model

According to the FIDO UAF Specification, we will describe how the entities that structure the client-side are authenticating each other. The FIDO UAF TRUST Model is shown in Figure 7. By distinguishing the relationship between the UAF client and the user agent, the authentication of the latter to the former is made through the value of FacetID, a platform specific identifier that points out how an application is implemented on various platforms. The UAF client is authenticated to the ASM via the CallerID and the KHAcceessToken, which is used to provide access control to the Authentication Key. The ASM must provide a specific KHAccesstoken to get access to the correct user Authentication Key and the UAF Authenticator have to validate it.

KHAccesstoken is computed by ASM. Its value is calculated as the hash of the values AppID, CallerID, tok and PersonalID concatenation.

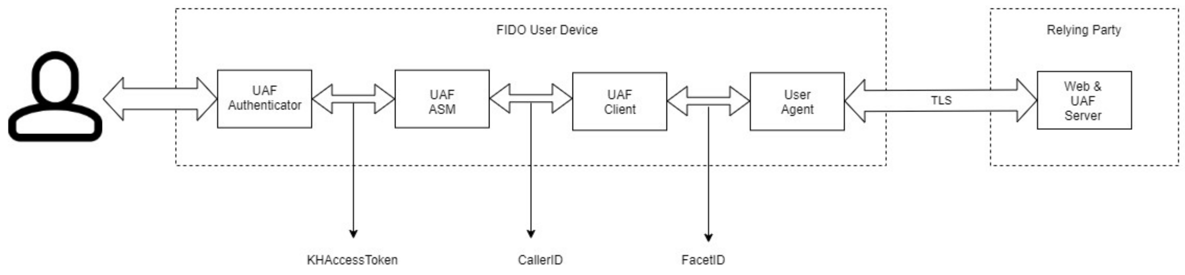


Figure 7 UAF Trusted Model

3.2 UAF Operations

Table I describes the acronyms used in this report. In the following sections, we will describe the three basic functions of FIDO along with their diagrams. For these diagrams, we will use one entity to represent the Web server and the UAF server, in order to make the description more concise.

Acronym	Full Name	Description
Rp	Relying Party	The server-side,which contains a web server and a UAF server.
UA	User Agent	A user application that supports the UAF protocol.
UC	UAF Client	A system service or application that implements the client-side of UAF protocol.
ASM	Authenticator Specific Module	An authenticator abastraction layer that provides a uniform API for the upper layer.
UName	Username	A human-readable string that identifying user's account at a Relying Party
HDR	Header	A dictionary which contains the type of UAF protocol Version, the type of the FIDO operation, AppID , a session identifier created by the relying party and exts.
POL	Policy	A set of match criteria made by the Relying Party, which are used to validate the suitability of an authenticator.
Chlg	Challenge	A random value that is provided by the FIDO UAF server in the UAF protocol requests in order to protect against replay attacks.
ReqType	Request Type	A value that specifies the type of the request ("Getinfo","Register","Authenticate","Deregister").
AsmVER	ASM Version	A description of the ASM version.
AIndex	Authenticator Index	A set of indicators for all authenticators discovered by the ASM "GetInfo" Request.
Args	Arguments	An object whose values depends on the type of the request ("Registration", "Authentication","Deregistration")
Exts	Extensions	A dictionary which contains parameters such as extension id, an arbitrary data which is agreed between the server and the client and fail_if_known value which indicates whether unknown extensions must be ignored.
SC	Status Code	A value that is returned by the Authenticator to inform about a problem in case of fail.
ApiVER	API Version	The version of the authenticator api.
Ainfo	Authenticator Info	A set of values describing the authenticator.
AAID	Authenticator Attestation Identifier.	A unique identifier assigned to a model, batch or class of FIDO UAF Authenticators that all shared the same characteristics.
AppID	Application Identifier	The application id is used by the UAF Server to determine if the application is authorized to use UAF protocol.
FacetID	Application Facet Identifier	A platform-specific identifier (URI) for an application facet to indicate how an application is implemented on various platforms. (Such as Web application, android applications, ios apllications).
TLSData	Channel binding Data	A channel binding allows applications to establish that the two end-points of a secure channel at one network layer are the same at a higher layer by binding the authentication to the higher layer to the channel at the lower layer.
tok	ASM Token	A randomly generated which is created when the ASM is launched for the first time. The ASM will maintain this secret until is uninstalled.
ak	Key handle access token	An access control mechanism for protecting an authenticator's FIDO UAF credentials from unauthorized use. It is created by the ASM by mixing various sources of information.

fc	Final Challenge	Hashed final Challenge parameter
UsrVERtkn	User Verification Token	A 32-bit flag referring to user verification method (fingerprint, voiceprint, passcode, etc).
h	KeyHandle	A key container created by the FIDO UAF authenticator, containing an authentication private key and optionally other data such as the username if it is a first factor authenticator).
Uauth.pub	Authentication public Key	User authentication public Key created by FIDO UAF Authenticator.
Uauth.priv	Authentication private Key	User authentication private Key created by FIDO UAF Authenticator.
Uatt.pub	Attestation public Key	The private asymmetric key used for FIDO UAF authentication attestation.
Uatt.priv	Attestation private Key	The public asymmetric key used for FIDO UAF authentication attestation.
regCounter	Register Counter	An increasing counter is maintained by the Authenticator. It is increased at the end of User registration.
signCounter	Signature Counter	An increasing counter is maintained by the Authenticator. It is increased every time that the authentication private key is used. FIDO server uses this value to detect cloned authenticators.
KeyId	Key Identifier	A unique identifier for an authentication key registered by an authenticator with a FIDO UAF Server.
AssertionInfo	Assertion Information	A dictionary which consists of authenticator version , signature algorithm and enconding of the signature , public key algorithm and enconding of the Uauth.pub key and the authenticationMode value which means that a user has verified her action.
attestationType	Attestation Type	The attestation scheme used by the authenticator to exchange data.

TABLE I. ACRONYMS AND DESCRIPTIONS.

3.2.1 Authenticator Registration

In order to be able to authenticate through the FIDO UAF protocol, the authenticator registration process must first be performed. In this process, the RP validates the authenticity of the FIDO authenticator and registers it by associating it with a user account.

1. Relying Party -> UAF Client

When the fido client launches the registration process the RP will create a Registration message (Uname, POL, HDR, Chlg) and send it to the UAF Client in order to officially start the registration process. Uname is used as an identifier for the user. POL refers to the built in RP policy, which consists of matchcriteria that indicate if an authenticator can be accepted or not. The header is a dictionary of informative values such as the UAF version, the type of operation, the AppID which is a URL that points to a list of trusted user agents and the server data that acts as a session identifier and is created by RP. Finally, Chlg is a random challenge value.

2. UAF Client -> ASM

After receiving the request message from the RP, the UAF client will retrieve the trusted user agent list from AppID and verify if the FacetID is on the list. Then, the UC will compose the GetInfo = (Reqtype, AsmVer, AIndex, Args, Exts) message, asking for information about the available authenticators in order to find out who is fulfilling the RP policy. ReqType specifies the type of the request ("register", "authenticate"," deregister"), AsmVer stipulates the asm version that is going to be used in this request, Aindex is a set of indices for all authenticators discovered by the ASM "GetInfo" request, args are the arguments, this object depends on the type of the request, in this case, arguments concerning the GETINFO type of message will be sent. Exts are the extensions dictionary which contains the extension id, the data which is an arbitrary data agreed between the server and the client and the fail_if_uknown value that indicates whether unknown extensions must be ignored.

3. Authenticator Specific Module -> Authenticator

ASM will forward the above message to the authenticator and will wait for its response.

4. Authenticator -> ASM

The Authenticator will respond to the GetInfo request made by UC by sending GetInfOut=(SC, ApiVER, Ainfo).SC is the status code that inform in case of a fault, where is the problem. ApiVer is a value that presents the version of the API.AInfo is a set of values which consists of AuthenticatorIndex, AAID which stands for authenticator attestation id and uniquely recognizes a specific authenticator model, Authenticator Metadata that represents the Metadata requirements for authenticator certification, the assertion scheme, the scheme used for the exchange of messages (e.g. registration) and the attestation type, which indicates the attestation scheme used by the authenticator to exchange data.

5. ASM -> UAF Client

The ASM after receiving the information from the previous message will send to the UAF Client a message with the following values (SC, ResponseData, Exts), where ResponseData will consist of the GetInfOut message sent by the authenticator.

6. UAF Client -> ASM

After the UAF Client receives the GetInfOut reply from ASM, it will check which authenticator satifies the prerequisites set by FIDO Server, comparing the matchCriteria dictionary received from the authenticator with the Policy sent to it at the beginning, by RP. If the policy is satisfied by the specific authenticator and the AppID has become validated, fido UAF Client will calculate the final challenge parameter fcp=(AppID, FacetID, Chlg, TLSdata).UC calculates TLSData by using TLS channel information to prevent MIMT attacks. Having calculated the fcp, FIDO UC will start the registration process of the specific authenticator by sending to ASM the following request (ReqType, AsmVER,Uname,fcp, Aindex, Args, Exts) where this time args will be structured so, as to serve the needs of the Register request.

7. ASM -> Authenticator

ASM obtains UserVerificationToken, a 32-bit flag that refers to the user-verification method, and then calculates a token ak=hash (AppID || PersonnalID || tok || CallerID) where || implied concatenation. The ak token is located under the KHAccessToken mechanism, which is an access control mechanism that protects UAF credentials from unauthorize use, the authenticator uses ak in the procedure of authentication to verify ASM. Finally, ASM will calculate the final Challege fc=hash(fcp) and send the (Uname, UseVERtkn, AppID, attestationType, Aindex, ak, fc) message to the authenticator.

8. Authenticator -> ASM

Initially, the authenticator will update the ak token = hash (ak || AppID), while then it will trigger its built-in matcher e.g., face-recognition or fingerprint sensor, in order to locally verify the identity of the user. An authentication key pair (Uauth.pub,Uauth.priv) will be generated for the specific user. The authenticator will even compute, a random KeyID which acts as a Key Identifier. Then the key Handle h= Enc (Uauth.priv, AK, Uname) will be calculated, where Enc is symmetrical encryption such as AES-GSM or AES-CCM. Continuing, the authenticator will calculate KRD = (fc, Uauth.pub, Aaid, RegCNTR, SignCNTR, keyID, Ainfo) where RegCNTR is a counter that increases in every registration and signCNTR is a random, signature counter which is synchronized with RP and is used by RP to detect cloned authenticators. The KRD will be signed with the Uatt.priv key and it will be converted to $S=sign_{Uatt.priv}(KRD)$. At the end of this step the authenticator will send the message (SC, fc, Uauth.pub, signCNTR, regCNTR, keyID, S, KRD, assertionInfo, Aaid,h) to ASM.

9. ASM->UAF Client

After saving the CallerID,AppID, key handle h and keyID values, the ASM will forward the message (SC, assertionscheme, S, KRD, exts) to UC.

10. UAF Client -> RP

UC will remove the Status Code value from the received message and in its place will add Header and fcp so the new message (HDR, fcp, AssertionScheme, S, KRD, exts) will be sent to RP in order to complete the registration process. For its part, RP will compare xfc = hash (AppID || Chlg || TLSData) which consists of stored values, with fc = h (fcp) where fcp is received from the previous message. Then the Relying party, will verify the fcp.AppID, fcp.Chlg, fcp.TLSData values based on the values it has stored, and it will check if the fcp.FacetID value is in the Trusted FacetIDs list. When it is finished, it will verify the S signature with the attestation public key (Uatt.key) of the authenticator. If the signature corresponds, then it will store the values of signCounter, AAID, KeyID, Uauth.pub, authenticatorVersion, and the registration process will be completed successfully.

Figure 8 depicts the message flows of the UAF authenticator registration operation.

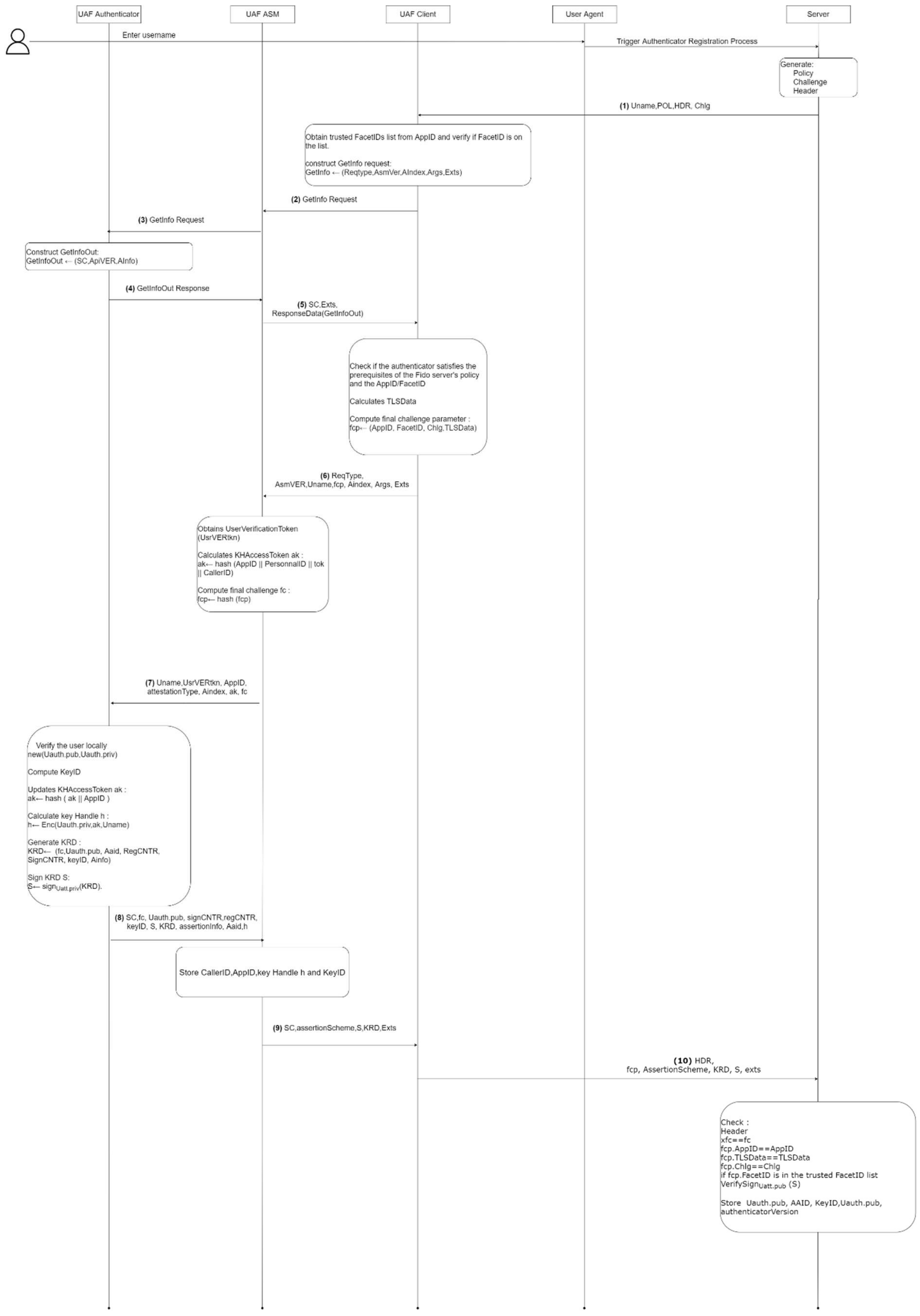

UAF Authenticator
UAF ASM
UAF Client
User Agent
Server
Enter username
Trigger Authenticator Registration Process
Generate: Policy Challenge Header
(1) Uname,POL,HDR, Chlg
Obtain trusted FacetIDs list from AppID and verify if FacetID is on the list.
construct GetInfo request:
GetInfo ← (Reqtype,AsmVer,AIndex,Args,Exts)
(2) GetInfo Request
(3) GetInfo Request
Construct GetInfoOut:
GetInfoOut ← (SC,ApiVER,AInfo)
(4) GetInfoOut Response
(5) SC,Exts, ResponseData(GetInfoOut)
Check if the authenticator satisfies the prerequisites of the Fido server's policy and the AppID/FacetID
Calculates TLSData
Compute final challenge parameter :
fcp← (AppID, FacetID, Chlg,TLSData)
(6) ReqType, AsmVER,Uname,fcp, Aindex, Args, Exts
Obtains UserVerificationToken (UsrVERtkn)
Calculates KHAccessToken ak :
ak← hash (AppID || PersonnalID || tok || CallerID)
Compute final challenge fc :
fcp← hash (fcp)
(7) Uname,UsrVERtkn, AppID, attestationType, Aindex, ak, fc
Verify the user locally
new(Uauth.pub,Uauth.priv)
Compute KeyID
Updates KHAccessToken ak :
ak← hash (ak || AppID)
Calculate key Handle h :
h← Enc(Uauth.priv,ak,Uname)
Generate KRD :
KRD← (fc,Uauth.pub, Aaid, RegCNTR, SignCNTR, keyID, Ainfo)
Sign KRD S:
S← sign$_{Uatt.priv}$(KRD).
(8) SC,fc, Uauth.pub, signCNTR,regCNTR, keyID, S, KRD, assertionInfo, Aaid,h
Store CallerID,AppID,key Handle h and KeyID
(9) SC,assertionScheme,S,KRD,Exts
(10) HDR, fcp, AssertionScheme, KRD, S, exts
Check :
Header
xfc==fc
fcp.AppID==AppID
fcp.TLSData==TLSData
fcp.Chlg==Chlg
if fcp.FacetID is in the trusted FacetID list
VerifySign$_{Uatt.pub}$ (S)
Store Uauth.pub, AAID, KeyID,Uauth.pub, authenticatorVersion

Figure 8 Registration of Authenticator

3.2.2 Authentication

The authentication of the user in the FIDO UAF protocol is a process, which is based on a cryptographic challenge-response scheme. More specifically, the UAF Server asks the user to authenticate himself/herself in the fido authenticator he/she used in the registration process. The authentication process can be extended (optional) to the transaction confirmation process. The transaction confirmation process offers the appropriate support to a user who desires to confirm a specific transaction with a secure display device. Since, we want to make a complete analysis of the FIDO UAF protocol the transaction confirmation related operation is marked with a '[]'.

1. RELYING PARTY -> UAF CLIENT

   When the fido client prompts to start the authentication process, the RP will generate the policy, the challenge, the transaction, and it will send the authentication request message (POL, HDR, Chlg, [TR]) to the UAF CLIENT. The TR is a text to be confirmed in the case of the transaction confirmation.

2. UAF CLIENT -> ASM

   The FIDO UAF Client after receiving the message will initially check the AppID and facetID. Then based on the RP's policy will search for the appropriate authenticator which will be used to authenticate the user. In order to do this, it will send a GetInfo=(ReqType, ASMVer, AIndex, Args, exts) message to the ASM.

3. ASM-> AUTHENTICATOR

   ASM will forward the above message to the authenticator and will wait for its response.

4. AUTHENTICATOR -> ASM

   As in the Registration process, the authenticator will respond to this step by sending the GetInfoOut = (Sc, ApiVer, AInfo) message.

5. ASM -> UAF CLIENT
   The ASM after receiving the GetInfoOut message from the authenticator will send the message (SC, ResponseData, exts) to the UC where the GetInfoOut will be located in the Responsedata field.

6. UAF CLIENT -> ASM
   Upon receiving the response from GetInfo, UC will check which of the available authenticators completes the RP policy requirements, in order to use it in the user authentication process. It will then calculate the final challenge parameter fcp = (AppID, FacetID, Chlg, TLSdata) and send (ReqType, ASMVer, authenticatorIndex, args, [TR]). Args includes fcp and keyID values.

7. ASM-> AUTHENTICATOR
   Once ASM receives the message, it will obtain the UserVerificationToken, calculate the final Challenge fc = hash (fcp) and the token ak = hash (AppID || PersonID || ak || CallerID). Then it will locate the key handle h from KeyID, which is a key container, containing the private key Uauth.priv, ak, and optionally other data such as the username (if there is a first-factor authenticator). Finally, the ASM will send (ak, fc, AppID, h, [TR], UserVerificationToken) to Authenticator.

8. AUTHENTICATOR -> ASM
   Upon receiving the message, the authenticator will initially update the token ak = hash (ak || AppID) and trigger its built-in matcher to verify the user identity. Then, the Authenticator will verify the UserVerificationToken and locate the keyhandle (if provided by ASM) so that it can be used to verify the ak, in order to make sure that the ASM is trusted. (If ak pass the ckeck, in case of Transaction the authenticator will display the transaction text Tr on the secure display for the user to confirm.) A random value n will be generated to protect the authenticator from replay attacks. Finally, the authenticator calculates the signature S = $\text{sign}_{\text{Uauth.priv}}$ (data), where data = (AAID, n, assertionInfo,fc, [hTR], signCNTR, KeyID) and will send (SC, data, S) to ASM .

9. ASM -> UAF CLIENT
   ASM will forward the message to UC by adding AssertionScheme and exts values

(SC,Data,S,assertionSheme , Exts) .

10. UAF CLIENT -> RELYING PARTY

    UC will check the status code of the message received from ASM with the aim to validate if the user verification was successful. Finally, it completes the exchange of messages by sending (HDR, fcp, data, S, assertionScheme, exts) as a response to the Relying Party.

Once the RP receives the response from the client it will follow a series of steps in order to verify the validity of the message. First, it will locate the user's Uauth.pub key via (UName, AAID, KeyID). It will then check the header, and the assertion it received from the message. It will continue its validation process by verifying fcp.AppID, fcp.Chlg and fcp.TLSdata corresponding to those stored in RP, and checking if fcp.facetID is in the trusted FacetIDs list. Then, it will compare the saved AAID with the one received in order to check if the authenticator from which the message originates is the same as the one registered by the user in the previous stage. The RP then computes the final challenge from the values received fc = hash (AppID || Chlg || TLSdata) and compares it with fc that is stored to make sure that the response is right. In Transaction case, Rp compares hTR with hash [Tr]. Finally, RP verifies the signature S and checks if the signCntr which was in the message received from UC is synchronized with the signCntr that is stored. If the message sent by the client goes through all the above steps, RP will update the signCntr stored with the one it received and will complete the authentication process of the user.

Figure 9 depicts the message flows of the UAF authentication operation.

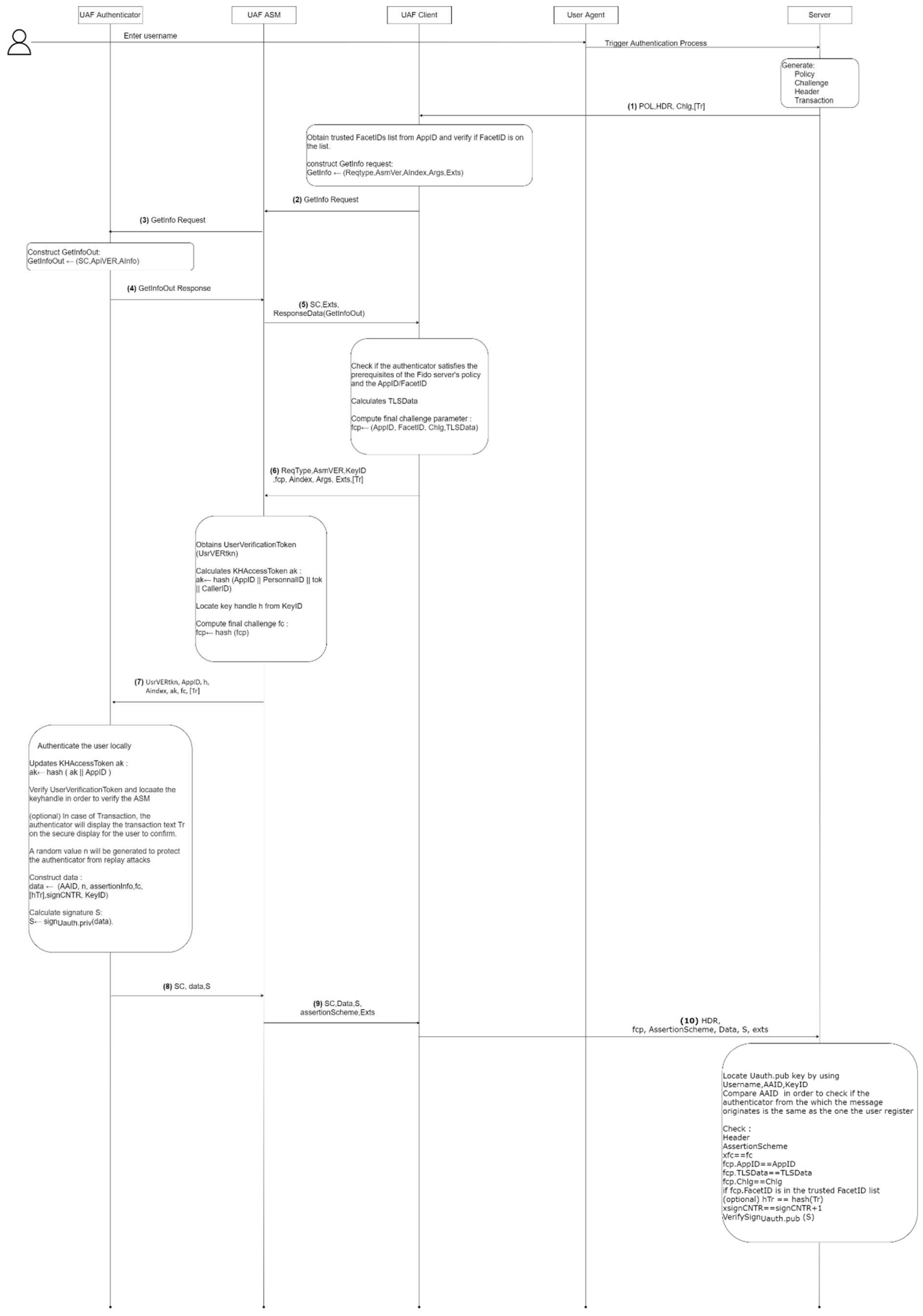

UAF Authenticator
UAF ASM
UAF Client
User Agent
Server
Enter username
Trigger Authentication Process
Generate: Policy Challenge Header Transaction
(1) POL,HDR, Chlg,[Tr]
Obtain trusted FacetIDs list from AppID and verify if FacetID is on the list.
construct GetInfo request: GetInfo ← (Reqtype,AsmVer,AIndex,Args,Exts)
(2) GetInfo Request
(3) GetInfo Request
Construct GetInfoOut: GetInfoOut ← (SC,ApiVER,AInfo)
(4) GetInfoOut Response
(5) SC,Exts, ResponseData(GetInfoOut)
Check if the authenticator satisfies the prerequisites of the Fido server's policy and the AppID/FacetID
Calculates TLSData
Compute final challenge parameter : fcp← (AppID, FacetID, Chlg,TLSData)
(6) ReqType,AsmVER,KeyID ,fcp, Aindex, Args, Exts,[Tr]
Obtains UserVerificationToken (UsrVERtkn)
Calculates KHAccessToken ak : ak← hash (AppID || PersonnalID || tok || CallerID)
Locate key handle h from KeyID
Compute final challenge fc : fcp← hash (fcp)
(7) UsrVERtkn, AppID, h, Aindex, ak, fc, [Tr]
Authenticate the user locally
Updates KHAccessToken ak : ak← hash (ak || AppID)
Verify UserVerificationToken and locaate the keyhandle in order to verify the ASM
(optional) In case of Transaction, the authenticator will display the transaction text Tr on the secure display for the user to confirm.
A random value n will be generated to protect the authenticator from replay attacks
Construct data : data ← (AAID, n, assertionInfo,fc, [hTr],signCNTR, KeyID)
Calculate signature S: S← sign$_{Uauth.priv}$(data).
(8) SC, data,S
(9) SC,Data,S, assertionScheme,Exts
(10) HDR, fcp, AssertionScheme, Data, S, exts
Locate Uauth.pub key by using Username,AAID,KeyID
Compare AAID in order to check if the authenticator from the which the message originates is the same as the one the user register
Check :
Header
AssertionScheme
xfc==fc
fcp.AppID==AppID
fcp.TLSData==TLSData
fcp.Chlg==Chlg
if fcp.FacetID is in the trusted FacetID list
(optional) hTr == hash(Tr)
xsignCNTR==signCNTR+1
VerifySign$_{Uauth.pub}$ (S)

Figure 9 User Authentication

3.2.3 Deregistration

The process of deregistration takes place when a user account is deleted from the Relying Party. In this process, the UAF Client logs in to the RP and asks for deregistration. The RP, triggers the deregistration process, in which it commands the authenticator to delete the UAF credentials associated with the user's account.

1. RELYING PARTY -> UAF CLIENT

In order to start the deregistration process, RP will send the deregistration request (HDR, AAID, KeyID) to UC. KeyId and AAID are provided to the authenticator with the intention of finding the credentials associated with its registration in the specific Relying Party.

2. UAF CLIENT -> ASM

After receiving the message, the UAF client will obtain the list of trusted FacetID's in order to check the specific AppID that was in HDR. If the AppID is verified, it will continue the process by sending the message (Reqtype, AsmVER, AIndex, AppID,KeyID) to Asm.

3. ASM -> AUTHENTICATOR

ASM will use the authenticatorIndex value of the message in order to locate the authenticator. Then it will generate the token ak=hash(AppID|| PersonalID|| tok || callerID) and will send the message (AIndex, AppID, KeyID, ak) to the authenticator.

4. AUTHENTICATOR -> UAF CLIENT

Once the Authenticator receives the message, it will first check the ASM's validity by checking the ak token, and then it will analyze the values that it received, in order to find and delete the credentials associated with its registration in this relying party. The authenticator deregistration will be finally completed with the authenticator sending the status code (SC) to the ASM and consequently to the UC informing them of the success or the failure of the process.

Figure 10 depicts the message flows of the UAF authentication operation.

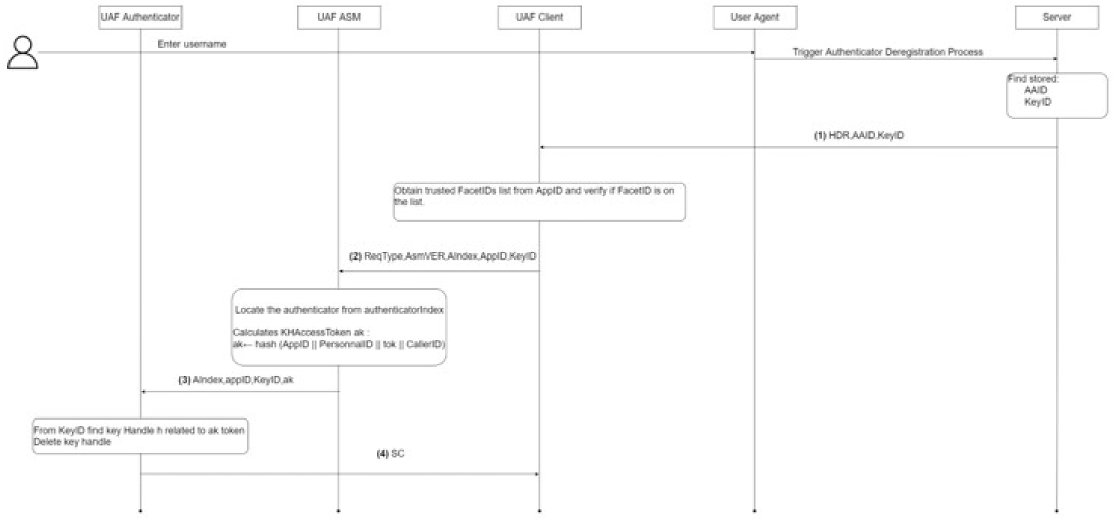

UAF Authenticator
UAF ASM
UAF Client
User Agent
Server
Enter username
Trigger Authenticator Deregistration Process
Find stored:
AAID
KeyID
(1) HDR,AAID,KeyID
Obtain trusted FacetIDs list from AppID and verify if FacetID is on the list.
(2) ReqType,AsmVER,AIndex,AppID,KeyID
Locate the authenticator from authenticatorIndex
Calculates KHAccessToken ak :
ak←hash (AppID || PersonnalID || tok || CallerID)
(3) AIndex,appID,KeyID,ak
From KeyID find key Handle h related to ak token
Delete key handle
(4) SC

Figure 10 Deregistration of Authenticator

4 Implementation

In this chapter we will present the analysis of the tool in terms of its setup, how it works, the conversions we had to do as well as a detailed description of the key components that contributed to this integration between the FIDO UAF server and the Keycloak identity provider. In essence, we will describe the reference architecture of the tool we construct, its components and how they interact with each other. Our main goal was to propose an architecture that follows the rules of device centric architecture and their interoperability. Initially, the hardware and software used for the implementation of the project are the following:

- For the servers
  - Ubuntu 20.04.2 LTS 64bit
  - 16 GB Ram
  - 21 GB disk
  - Intel Xeon Processor (Cascadelake) × 2
- For the client
  - Android 10
  - 6 GB Ram
  - 64 GB disk
  - Qualcomm SDM845 Snapdragon 845

The architecture of the tool was based on the following principles:

- The main goal of the architecture was to follow the triptych Something I have, something I am, and something I know, by offering to the user an authentication system that can be done exclusively from his mobile phone by using his/her biometrics without the dependence to interact with external factors beyond it (e.g., QR codes).
- The architecture is based on the OpenID Connect protocol. The information that needs to be exchanged between its various components will be done following the OpenID Connect specification.
- This tool uses the Fido UAF protocol, a multi-level in terms of assurances and with minimal user involvement (no need to remember any password) security

framework which provides more safety compared to the classic pattern username password or 2FA.

- The tool architecture is based solely on open-source solutions.

The main components of the architecture are:

1. Keycloak Identity provider.
2. Fido UAF Server with the custom keycloak library.
3. The Script Authenticator.
4. User Device.

4.1 Keycloak Identity provider

Keycloak is an open-source Identity and Access Management tool that focuses on modern applications such as single-page applications, mobile application and REST APIs. The Keycloak project started in 2014 and therefore has a huge and active community with a large user base. Keycloak is built on industry standard protocols that support OAuth2.0, OpenID Connect, and SAML2.0. The use of these protocols is particularly important both in terms of security and in terms of facilitating the integration of the Keycloak server in existing and new applications.

In an application or web service, keycloak completely undertakes the authentication process. Through this approach, a higher level of security is provided as the applications do not have direct access to the user's credentials. Instead, they are provided with special security tokens which give them access to what they need and only. Keycloak provides single sign-on and session management capabilities, thus allowing users to access multiple applications, by performing authentication only once. Both the users themselves and the administrators have full visibility of where the authentication takes place and can terminate it even remotely. When integrated into a project, Keycloak offers login pages that can be fully customized and include strong authentication and other features such as password recovery, acceptance of terms and conditions, regular password updates, etc.

Keycloak is a very light and easy-to-install solution. It contains its own database so it is very easy to use. It can be easily integrated into an existing identity infrastructure, while through identity brokering capabilities it can be connected to an existing user database that comes from social networks or identity providers. Finally, it has the ability to embody user directories, such as Active Directories and LDAP servers.

Importantly, Keycloak has a large number of expansion points where a developer can implement and develop custom code to modify existing functionality and add new features to suit his or her work needs.

In this section, we will show in detail how we set up the Keycloak server step by step so that we can perform a basic configuration that is needed to secure our web application. The admin console provided by Keycloak Vendor has an extensive and user-friendly interface for administrators and developers in order to manage and configure Keycloak. As Keycloak is implemented in Java, it is easy to run Keycloak on any operating system without the need to install additional dependencies. The only thing that you need to have installed is a Java virtual machine, such as OpenJDK.

Installing JDK

On Ubuntu, we install OpenJDK by executing the following command:

```
sudo apt-get install openjdk-8-jre
```

Installing Keycloak

Once you have the Java virtual machine installed on your workstation, the next step is to download the distribution of Keycloak from the Keycloak website. Open https://www.keycloak.org/downloads, and then download either the ZIP or the TAR.GZ archive of the server (standalone server distribution). Once downloaded, simply extract this archive to a suitable location. Then, open the terminal and go to the bin folder on the Keycloak's directory.

```
cd /path-to/keycloak-13.0.0/bin
```

On Linux or macOS, start Keycloak with the following command:

```
./standalone.sh -Djboss.socket.binding.port-offset=100 -b=0.0.0.0 -
bmanagement=0.0.0.0
```

Since we have installed Keycloak, to gain access, we will open the browser on the page http://localhost:8180/auth/admin. From there we will be redirected to the login page where we will be able to log in by entering the word Admin for username and password.

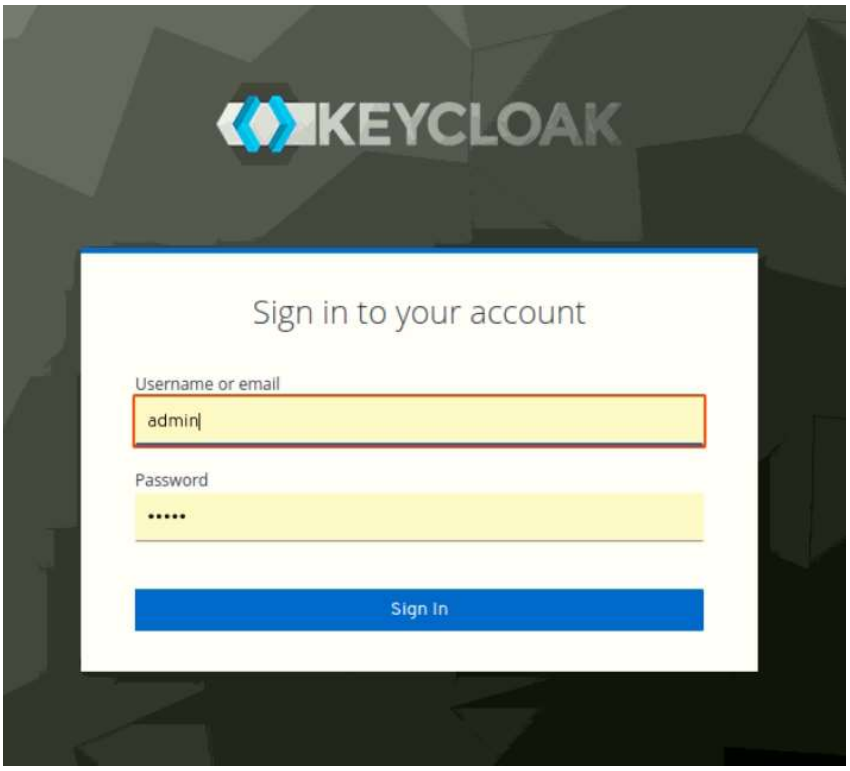
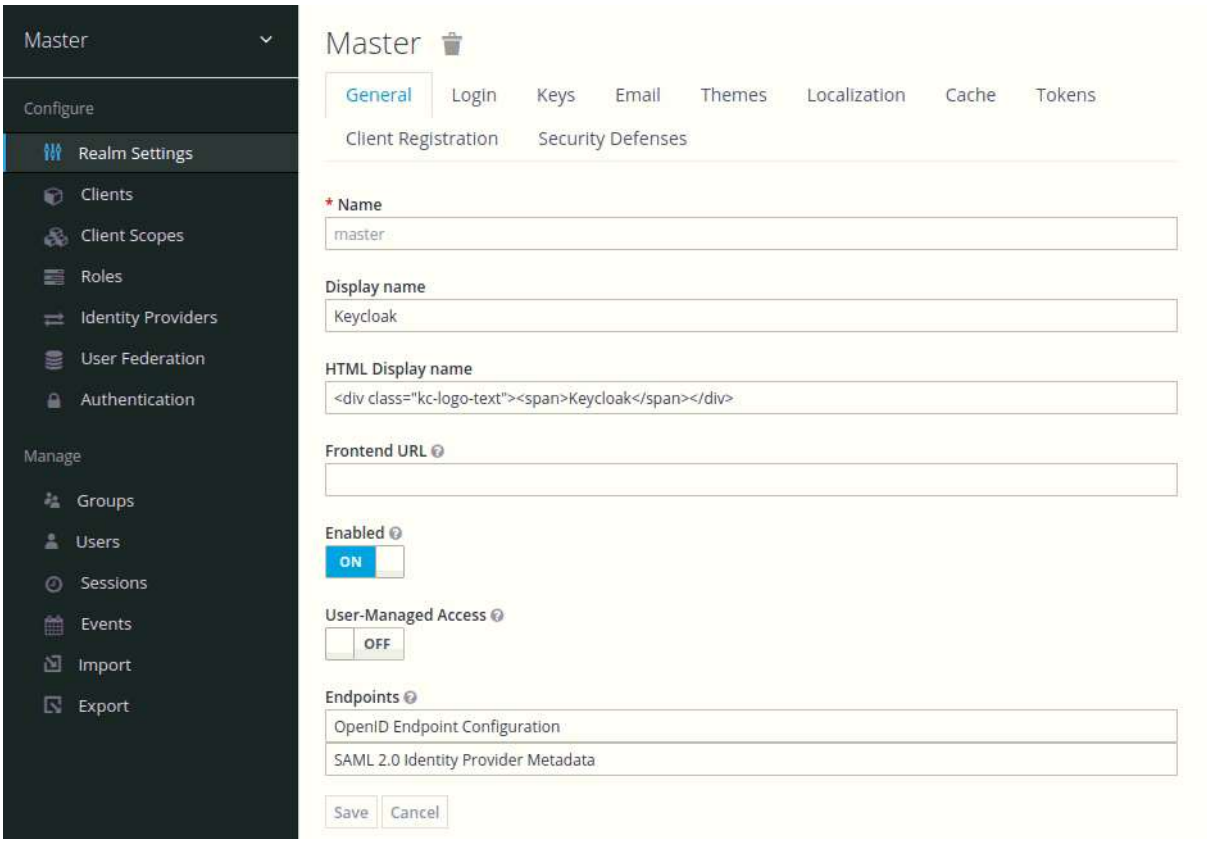


Figure 11 Sign in to Keycloak

Once logged in, the Master Realm configuration will be displayed.



* Name: master

Display name: Keycloak

HTML Display name: <div class="kc-logo-text"><span>Keycloak</span></div>

Frontend URL:

Enabled: ON

User-Managed Access: OFF

Endpoints: OpenID Endpoint Configuration; SAML 2.0 Identity Provider Metadata

Figure 12 The Keycloak Admin Console

Creating a Realm

Imagine realm as an apartment in a block of flats. Each realm is completely isolated from any other realm, has its own configuration as well as its own applications and users. In this way, we can use a Keycloak installation for different projects. Therefore, we will build a new realm for our project. To create the new realm, we just need to place the mouse pointer over the realm selector located on the top left, and by clicking on the Add Realm button a page with a field in which we must fill in the name of the realm will appear. Because the name is used in the URL, we would suggest to not use special characters. For the needs of our project, we used the word demo as a name.

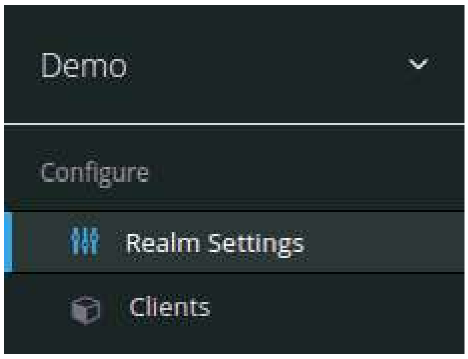


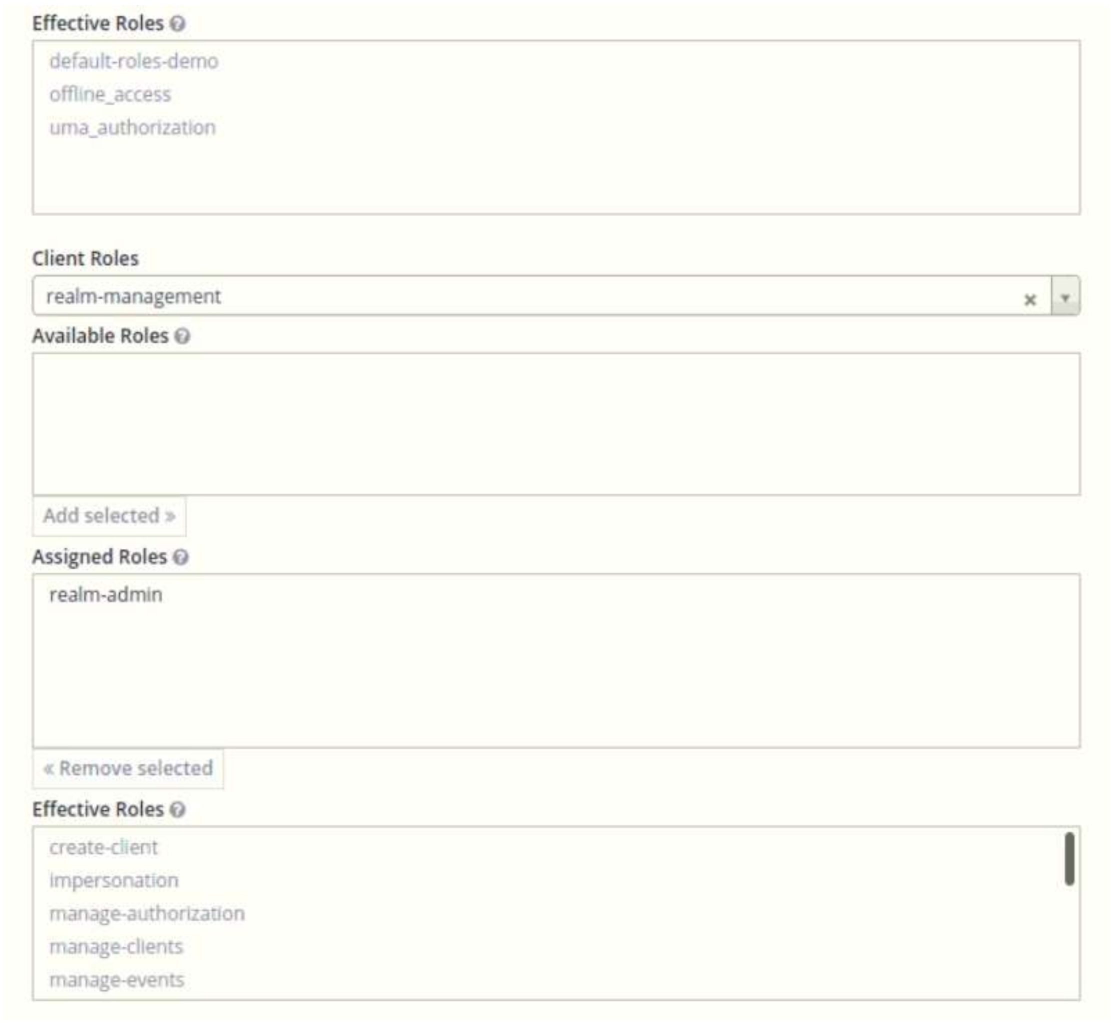
Figure 13 Demo Realm

Creating users

The next step after the realm construction, is to create a user administrator who will be used mainly by the integrated keycloak library on the UAF server in order to define in each user who registers a parameter named fidoauthenticationID and update it each time the user authenticates through the Fido UAF. The steps to create a user administrator are:

1. In the left vertical menu bar, click Users and then Add User.
2. Fill in the form that will appear, the name of your choice, you also provide the option to fill in the email, the first name, and the last name. (In our case we filled in the Username only). Once we fill in the fields, we select the save button.
3. Before the administrator be able to log-in, we need to create a password for him. To do this, click on the Credentials tab. In the Password Setting section, we enter a password while we also disable the Temporary option (this option is used in case the administrator assigns a password to a user in order to make his first log-in and then change it to a password of his choice.) Finally, press the Set Password button.

4. For the specific user to be an administrator, we must define new rights for him. In the Role Mappings section, in the client roles rollbar, we click on the realm-management option. Once we click on it, in the field Available roles we will see a rollbar with different roles for users, in our case, we will select the realm-admin role.


Effective Roles
default-roles-demo
offline_access
uma_authorization

Client Roles
realm-management
Available Roles

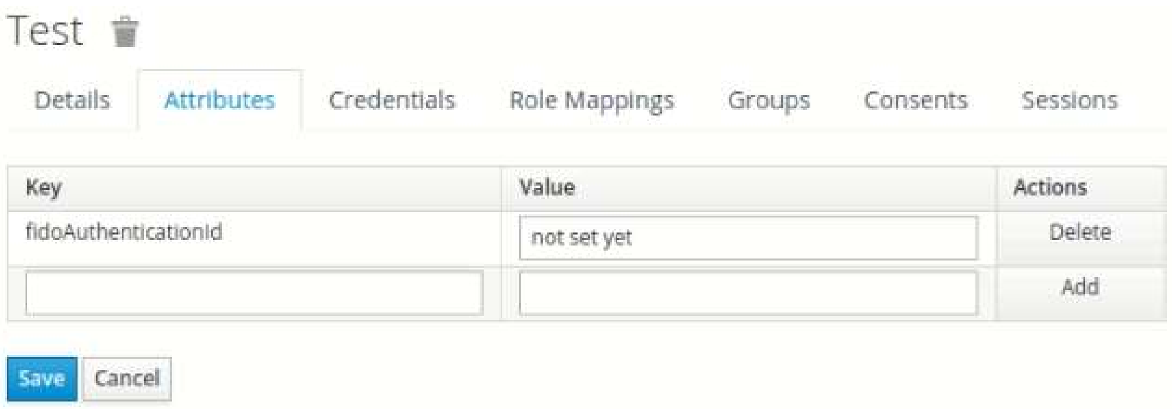
Add selected »
Assigned Roles
realm-admin

« Remove selected
Effective Roles
create-client
impersonation
manage-authorization
manage-clients
manage-events

Figure 14 Setting up realm-admin role

Once we have created the administration user, we can now create a regular user. We will use steps 1-3 from the above procedure and will add a new step that has to do with the creation of a unique custom authentication attribute on which the Integration between the FIDO server and keycloak will use as a common credential. Creating an attribute for a user is done as follows:

- In the options bar click on the Attributes tab. On the screen, two empty input fields will appear. In the Key column set the value fidoAuthenticationId, and in the Value column set the value not_set_yet. Then click Add and at the end click on the Save option.


Test

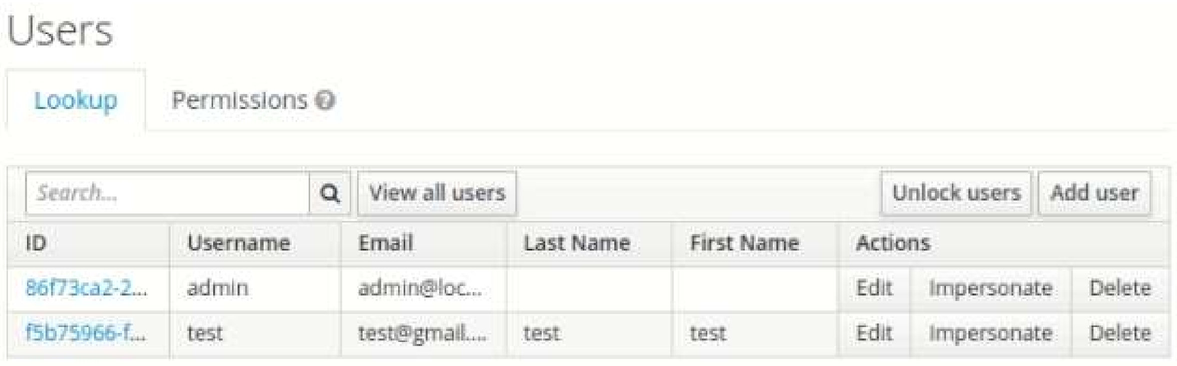
Details | Attributes | Credentials | Role Mappings | Groups | Consents | Sessions

Key	Value	Actions
fidoAuthenticationId	not set yet	Delete
		Add

Save Cancel

Figure 15 Adding a custom attribute to a user

The Keycloak server assigns a unique ID to each user that is created.


Users

Lookup | Permissions

Search... | View all users | Unlock users | Add user

ID	Username	Email	Last Name	First Name	Actions		
86f73ca2-2...	admin	admin@loc...			Edit	Impersonate	Delete
f5b75966-f...	test	test@gmail....	test	test	Edit	Impersonate	Delete

Figure 16 Keycloak Users example

Securing my application

In this section, we will show you, how to secure an application through Keycloak. For this purpose, we will use an already made application that allows us to view and interact with the tokens issued by Keycloak. As we can see in the Figure17, the web application is a single-page application that runs as a docker container while providing us with the following functions:

- Login with Keycloak.
- Displays the username.
- Introduces the ID Token.
- Introduces the Access token.
- Enables refresh tokens.

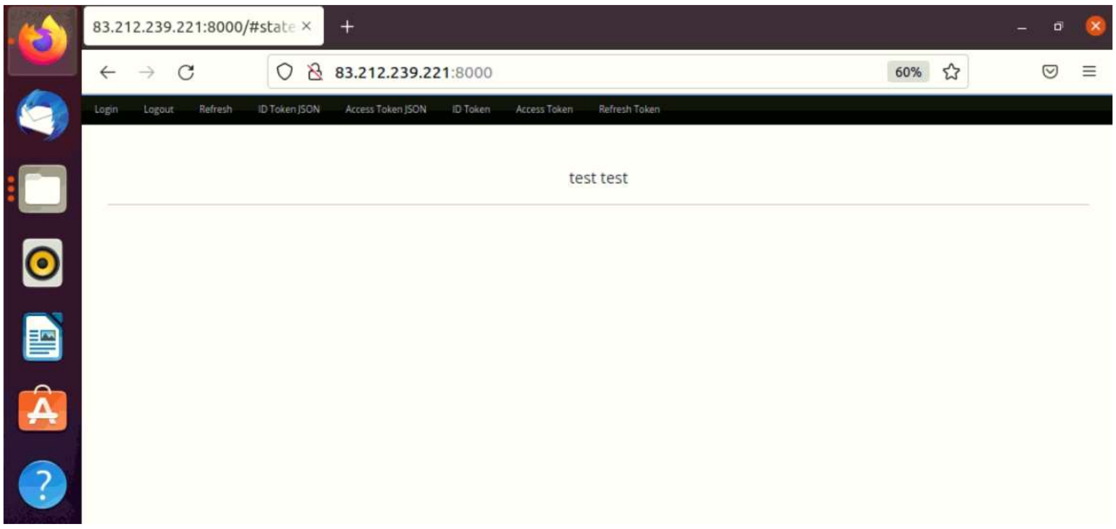


Figure 17 The web service

When the user clicks on the address where the application is located, the browser is redirected to the Keycloak login page. The user is authenticated via Keycloak and redirected back to the application along with a special code called authorization code. The application will then negotiate with Keycloak in order to exchange the authorization code with the following tokens:

- ID token: provides information about the authenticated user in the web application.
- Access token: The web application includes the specific token in each request to any service. In this way the service verifies whether the request is accepted or not.
- Refresh token: The ID token, as well as the Access token, have a very short lifespan, by default it is 5 minutes. Through the refresh token, the application acquires new tokens from Keycloak.

We would like to emphasize that by authenticating the user through Keycloak we automatically gain two advantages. First, the authentication mechanism does not affect our application at all and as we will see later, by changing the authentication mechanism, we do not add or remove code to the application. The second advantage and the most important is that the application has no contact with the user's credentials. Below you will find a diagram showing how the authentication process is done using Keycloak as Open ID Connect the identity provider.

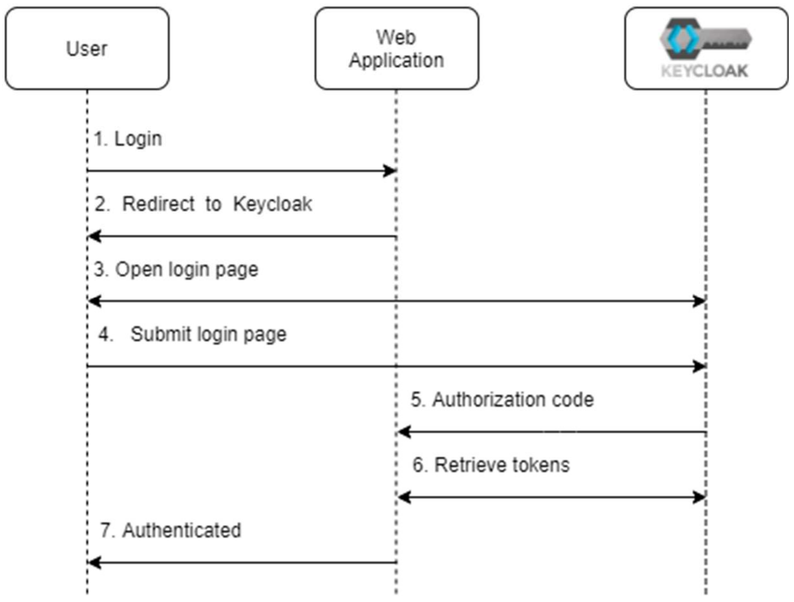


Figure 18 Authorization Code flow in OpenID Connect through Keycloak

The flow chart is as follows:

1. The user is directed to the web application page by entering the specific URL with the address.
2. The application redirects the user to the Keycloak login page.
3. The Keycloak login page is displayed to the user, with the latter entering his username and password.
4. After verifying the information entered by the user, Keycloak sends the authorization code to the web application.
5. The application exchanges the authorization code for an ID token, an Access token and a refresh token.
6. The application verifies the identity of the user by checking the ID token he received.
7. The user is authenticated and can access the web service.

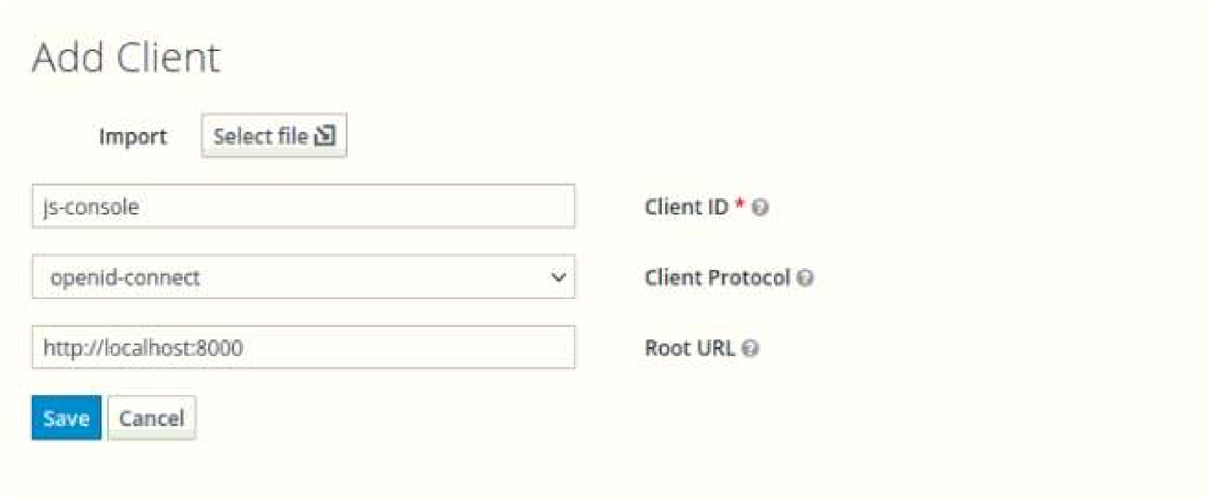
The specific flow described, is the authorization code flow defined by OpenID Connect.

Register the web service to Keycloak

In order for any application to connect to its users through Keycloak, it must first be registered to it as a Client. In case the application has not been registered and the user tries to connect to it, Keycloak will respond with an error page.

The registration of an application in Keycloak is done through the administrator console. At the top of the menu on the left, select Clients, and then click on Create option. Make sure that you are in the correct realm. From the Create option, you will see a form where you have to fill in the following values:

- Client ID : js-console
- Client Protocol : openid-connect
- Root URL: http: //localhost:8000



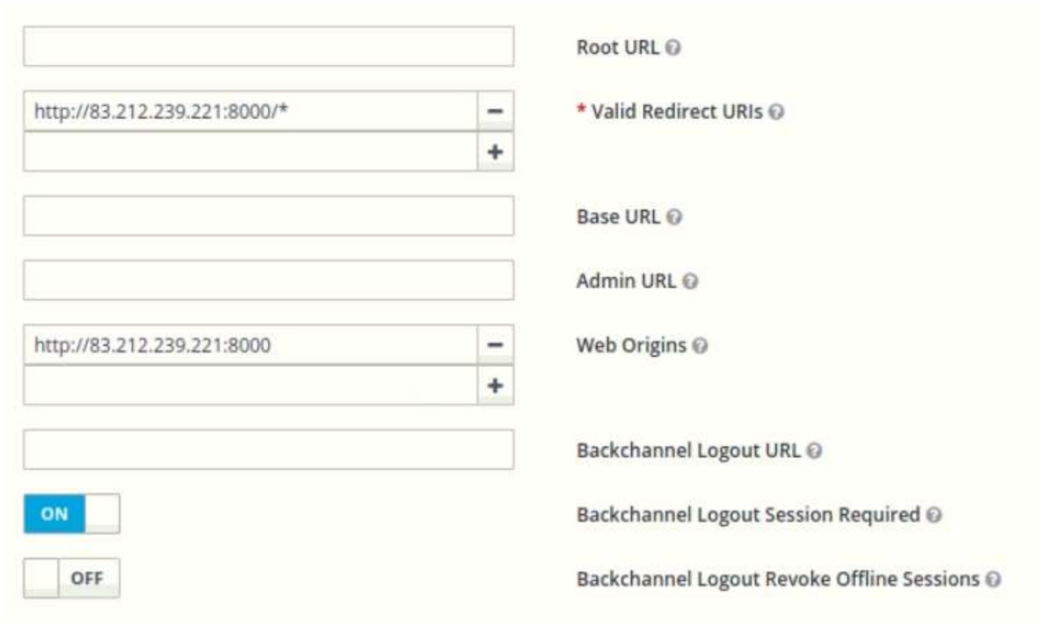
Add Client

Import	Select file
Client ID *	js-console
Client Protocol	openid-connect
Root URL	http://localhost:8000

Figure 19 Creating the client in the admin console

When the form is completed, click the save button. After clicking save, the full client configuration page will appear. There are two configuration options to look out for:

- Valid Redirect URIs: This field is used as a countermeasure for Masquerade attacks. This value is very important when a client-sidede application is used. A client-side application is not able to have any credentials, as these could be visible to the end-users of the application. To prevent any malicious applications from being able to masquerade as the real application, the valid redirect URIs instructs keycloak to redirect users only to URLs that match the valid redirect URL. In this case, after setting this value to point at http://localhost:8000/*, a malicious host on http://malware.com cannot disguise and authenticate like a real application.
- Web Origins: This option registers the valid web origins for the application for Cross-Origin Resource Sharing (CORS) requests. The use of CORS is particularly important in the keycloak token recovery process. To obtain tokens from Keycloak, the frontend application has to send an AJAX request to Keycloak, and browsers do not permit an AJAX request from one web origin to another, unless CORS is used.



Root URL

* Valid Redirect URIs: http://83.212.239.221:8000/*

Base URL

Admin URL

Web Origins: http://83.212.239.221:8000

Backchannel Logout URL

Backchannel Logout Session Required: ON

Backchannel Logout Revoke Offline Sessions: OFF

Figure 20 Configuring Client

Now we can go back to the frontend by opening http://localhost:8000. This time, when we click on the Login button, you will see the Keycloak login page. Log in with the username and password, you created during the previous chapter.

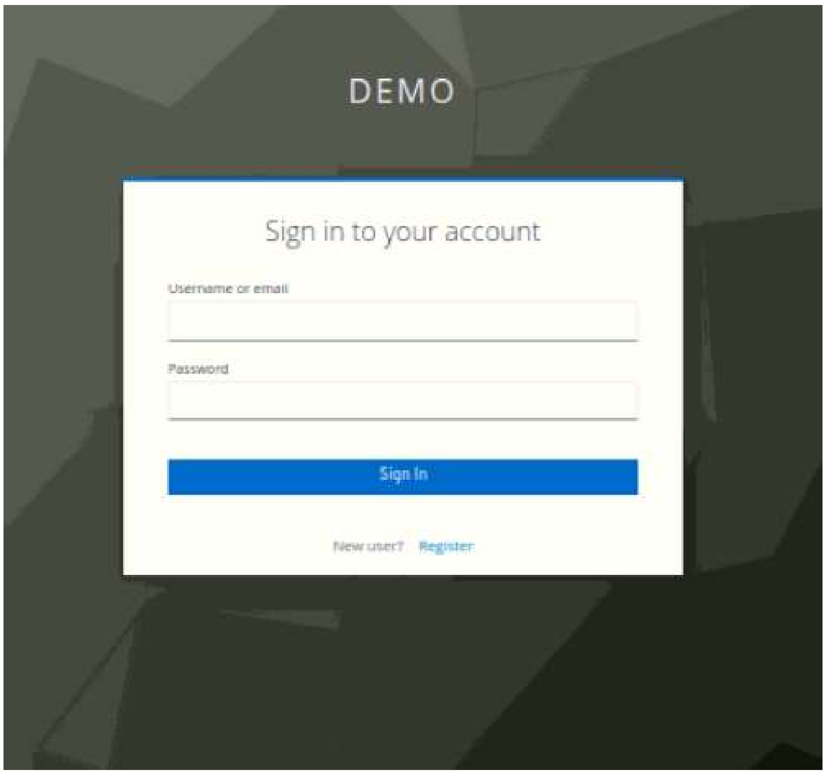
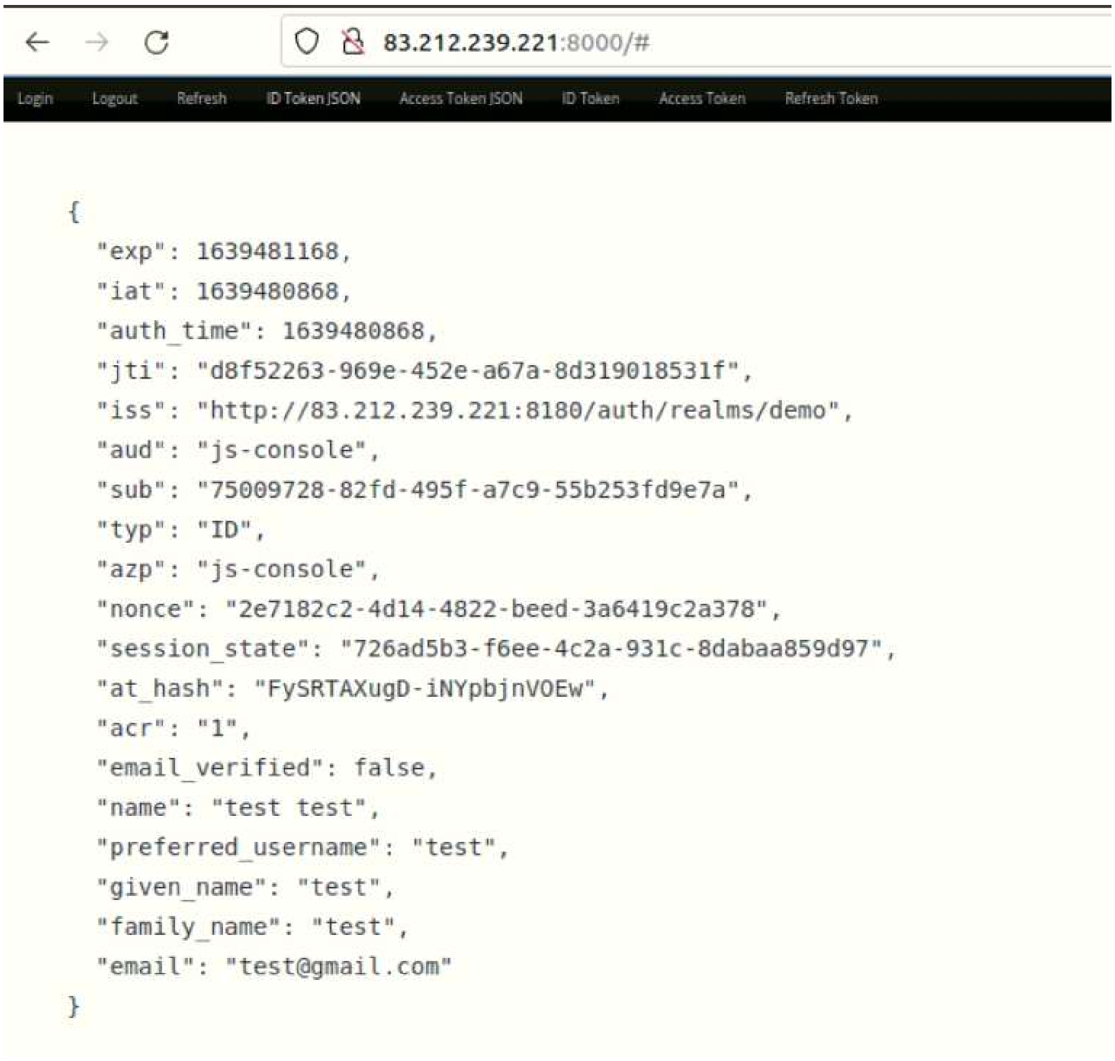


Figure 21 Login to the application through Keycloak

Let's take a look at the ID token that Keycloak issued. Click on the Show ID Token button. The ID token that is displayed will look something like the following:



```
{
  "exp": 1639481168,
  "iat": 1639480868,
  "auth_time": 1639480868,
  "jti": "d8f52263-969e-452e-a67a-8d319018531f",
  "iss": "http://83.212.239.221:8180/auth/realms/demo",
  "aud": "js-console",
  "sub": "75009728-82fd-495f-a7c9-55b253fd9e7a",
  "typ": "ID",
  "azp": "js-console",
  "nonce": "2e7182c2-4d14-4822-beed-3a6419c2a378",
  "session_state": "726ad5b3-f6ee-4c2a-931c-8dabaa859d97",
  "at_hash": "FySRTAXugD-iNYpbjnVOEw",
  "acr": "1",
  "email_verified": false,
  "name": "test test",
  "preferred_username": "test",
  "given_name": "test",
  "family_name": "test",
  "email": "test@gmail.com"
}
```

Figure 22 ID Token of the web service application that we used

4.2 UPRC Fido UAF Server with the custom keycloak library

Regarding the FIDO protocol, we used an open-sourced implementation from eBay in Java. This project is structured in three parts, a client module (Android), a server module (Apache tomcat), and the Fido module. The server part is certified by the FIDO Alliance [14]. The standard Fido protocol is implemented through three rest services:

- registrationRequest: Issues the UAF registration message to the client, i.e., the FIDO device, whenever requests it.
- authenticationRequest: Issues the UAF authentication message to the client whenever requests it.
- response: Receives the response sent by the client to the above messages. This response can be either a registration response which has the attested public key from the client or Authentication response which is signed by the private key of the client.

The Fido server is used in interaction with the client in order to perform the Registration, Authentication, and Deregistration procedures. All three procedures satisfy the requirements set by the protocol as we will see below. In the server, we have embedded a library that builds the connection between the FIDO UAF server with the Keycloak identity provider. The FIDO server has several APIs, one of them will be used as a means of verifying user authentication. There is detailed documentation of instructions for installation on the github of the project [15].

4.2.1 UPRC FIDO server Registration

We will divide the Registration process into two parts as many as the server participation in this process. The first part is called registration request and is the beginning of the registration process. The second part is called return verification result and in essence is the point where the server checks the registration response received from the client and sends a message of success or failure of the process.

Registration Request

The registration process allows the Server to verify the authenticity of the FIDO Authenticator and register it among with the user's account. Once an authenticator has been validated, the Relying Party can assign a unique identifier number (AAID) to the authenticator that can be used in future communication between the two parties.

The Fido UAF server will send to the Fido UAF client a registration request which consists of 4 parameters:

1.username: A human-readable string identifying a user's account at a relying party.

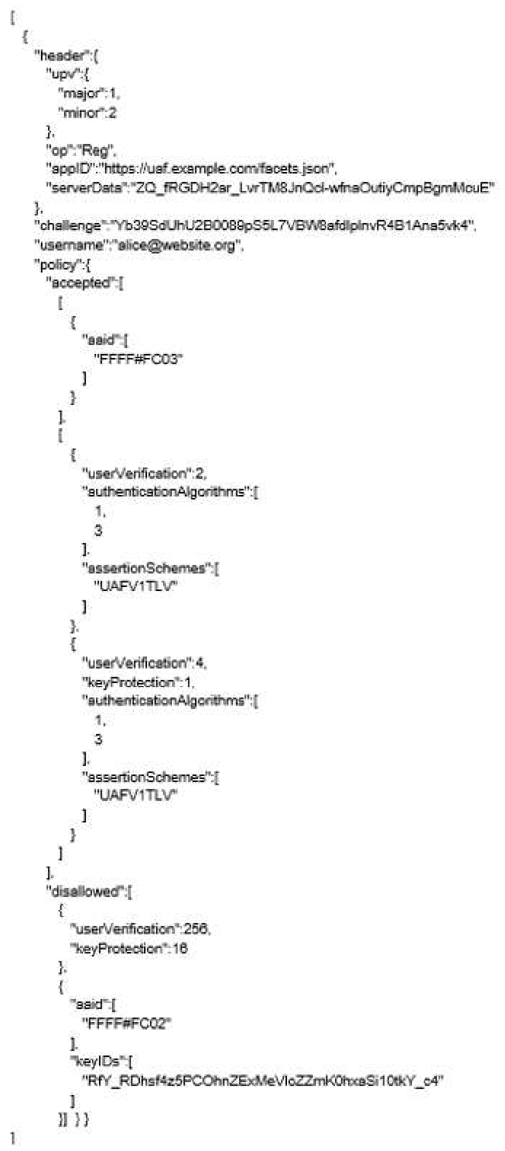
2.policy: The policy refers to the Relying party's set of match criteria concerning the acceptable authenticators.

3.Header: The Header refers to the operation header which is a dictionary containing the following values [13]:

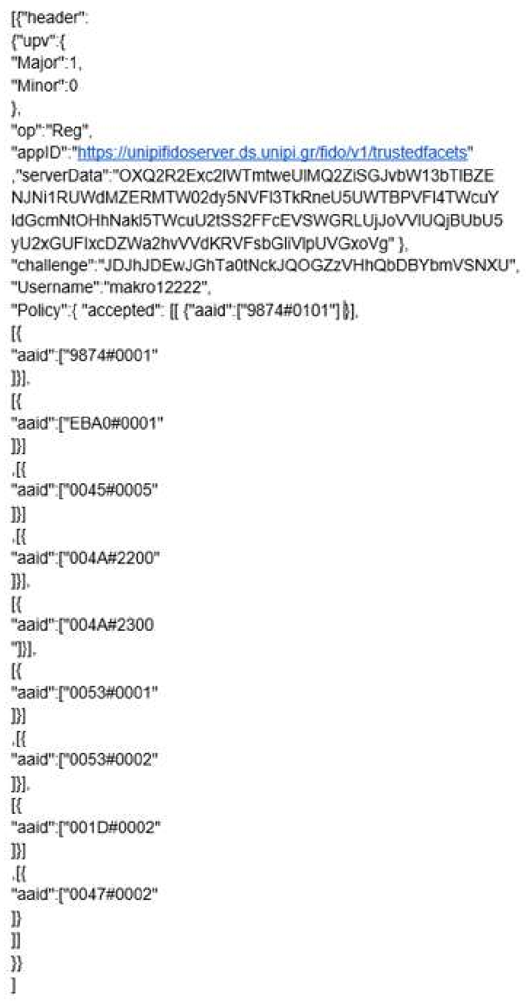
- upv
- operation
- appID
- serverData
- exts

4.challenge: A random value provided by the FIDO Server in the UAF protocol requests.

We will quote the registration request as described in the specification [16], as well as the registration request sent by the server of our implementation. Their main difference is in the policy in which as we see in the specification it consists of match criteria accepted and match criteria disallowed while our implementation has only the accepted match criteria which only consists of AAID value.



```
[
  {
    "header":{
      "upv":{
        "major":1,
        "minor":2
      },
      "op":"Reg",
      "appID":"https://uaf.example.com/facets.json",
      "serverData":"ZQ_fRGDH2ar_LvrTM8JnQcl-wfnaOutiyCmpBgmMcuE"
    },
    "challenge":"Yb39SdUhU2B0089pS5L7VBW8afdlpInvR4B1Ana5vk4",
    "username":"alice@website.org",
    "policy":{
      "accepted":[
        [
          {
            "aaid":[
              "FFFF#FC03"
            ]
          }
        ],
        [
          {
            "userVerification":2,
            "authenticationAlgorithms":[
              1,
              3
            ],
            "assertionSchemes":[
              "UAFV1TLV"
            ]
          },
          {
            "userVerification":4,
            "keyProtection":1,
            "authenticationAlgorithms":[
              1,
              3
            ],
            "assertionSchemes":[
              "UAFV1TLV"
            ]
          }
        ]
      ],
      "disallowed":[
        {
          "userVerification":256,
          "keyProtection":16
        },
        {
          "aaid":[
            "FFFF#FC02"
          ],
          "keyIDs":[
            "RfY_RDhsf4z5PCOhnZExMeVloZZmK0hxaSi10tkY_c4"
          ]
        }]] }}
]
```



```
[{"header":
{"upv":{
"Major":1,
"Minor":0
},
"op":"Reg",
"appID":"https://unipifidoserver.ds.unipi.gr/fido/v1/trustedfacets"
,"serverData":"OXQ2R2Exc2lWTmtweUlMQ2ZiSGJvbW13bTlBZE
NJNi1RUWdMZERMTW02dy5NVFl3TkRneU5UWTBPVFl4TWcuY
ldGcmNtOHhNakl5TWcuU2tSS2FFcEVSWGRLUjJoVVlUQjBUbU5
yU2xGUFlxcDZWa2hvVVdKRVFsbGlVlpUVGxoVg" },
"challenge":"JDJhJDEwJGhTa0tNckJQOGZzVHhQbDBYbmVSNXU",
"Username":"makro12222",
"Policy":{ "accepted": [[ {"aaid":["9874#0101"]}],
[{
"aaid":["9874#0001"
]}],
[{
"aaid":["EBA0#0001"
]}]
,[{
"aaid":["0045#0005"
]}]
,[{
"aaid":["004A#2200"
]}],
[{
"aaid":["004A#2300
"]}],
[{
"aaid":["0053#0001"
]}]
,[{
"aaid":["0053#0002"
]}],
[{
"aaid":["001D#0002"
]}]
,[{
"aaid":["0047#0002"
]}
]]
}}
]
```

Specifications Register request   UPRC Fido Register Request

Verification Result

The fido uaf client sends the registration response message to the UAF server. Once received, Fido Server will verify the KRD signature, the attestation and it will store a new public authentication key for the specific user. The response analysis will be continued by the server and will be completed by sending a success or failure message to the client. The message does not have any protocol limitation in terms of its structure [16].

Here is an example of a message verifying the result:

```
{
   "authenticator":{
      "AAID":"EBA0#0001",

"KeyID":"dU5zekI3VFYtZlFMeG5KaGZJNnZaMlJZR0l0ZElBTE1VUlFRYXNCaGlTdz
0"
   },

"PublicKey":"MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEi7gWzuYCbhQPac8yfa
l7dU-vGBmlTHNu0tq3qSfjQdkIBAAqLH1MBE6I-kX-
pcR0D7dUmwd6ro0i9XG1D2bcPg",
   "SignCounter":"0",
   "AuthenticatorVersion":"0.0",
   "username":"makro12222",
   "deviceId":"f15d084c57c60006�?\b\u0000SM-G965F",
   "timeStamp":"1619441493525",
   "status":"1200",
   "attestCert":"MIIB-TCCAZ-
gAwIBAgIEVTFM0zAJBgcqhkjOPQQBMIGEMQswCQYDVQQGEwJVUzELMAkGA1UECAwCQ0
ExETAPBgNVBAcMCFNhbiBKb3NlMRMwEQYDVQQKDAplQmF5LCBJbmMuMQwwCgYDVQQLD
ANUTlMxEjAQBgNVBAMMCWVCYXksIEluYzEeMBwGCSqGSIb3DQEJARYPbnBlc2ljQGVi
YXkuY29tMB4XDTE1MDQxNzE4MTEzMVoXDTE1MDQyNzE4MTEzMVowgYQxCzAJBgNVBAY
TAlVTMQswCQYDVQQIDAJDQTERMA8GA1UEBwwIU2FuIEpvc2UxEzARBgNVBAoMCmVCYX
ksIEluYy4xDDAKBgNVBAsMA1ROUzESMBAGA1UEAwwJZUJheSwgSW5jMR4wHAYJKoZIh
vcNAQkBFg9ucGVzaWNAZWJheS5jb20wWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAAQ8
hw5lHTUXvZ3SzY9argbOOBD2pn5zAM4mbShwQyCL5bRskTL3HVPWPQxqYVM-
3pJtJILYqOWsIMd5Rb_h8D-
EMAkGByqGSM49BAEDSQAwRgIhAIpkop_L3f0tm79Q2lKrKxea-
KcvA1g6qkzaj42VD2hgAiEArtPpTEADIWz2yrl5XGfJVcfcFmvpMAuMKvuE1J73jp4"
,

"attestDataToSign":"Az73AAsuCQBFQkEwIzAwMDEOLgcAAAABAgABAQouIAB5YvN
lEClglDhYE3XT0WqXaDsOlpMwcRzKPXKA3hFPewkuLAB1TnN6QjdUVi1mUUx4bkpoZk
k2dloyUllHSXRkSUFMTVVSUVFhc0JoaVN3PQ0uCAAAAAAAAAAAAwuWwAwWTATBgcqh
kjOPQIBBggqhkjOPQMBBwNCAASLuBbO5gJuFA9pzzJ9qXt1T68YGaVMc27S2repJ-
NB2QgEACosfUwETaj6Rf6lxHQPt1SbB3qujSL1cbUPZtw-
_D8QAGYxNWQwODRjNTdjNjAwMDb9PwgAU00tRzk2NUY",
   "attestSignature":"MEQCIFlnSOJiqweU-
NnC2s0LnRGGqgjciQWhhUsdWecNbgrJAiB8DGzfuP2jU26AUfZlKFDJJCQL8upnPlu0
FVZ0SKrwtw",
   "attestVerifiedStatus":"VALID"
}
```

UPRC Server Registration Verification Result

When the status value is 1200 it means that the process was successful. Respectively, when it has a value of 1404 the authenticator who is trying to register has already registered, in any other case the server gives an error message 1400 which means that the registration response received from the Fido Client had wrong structure

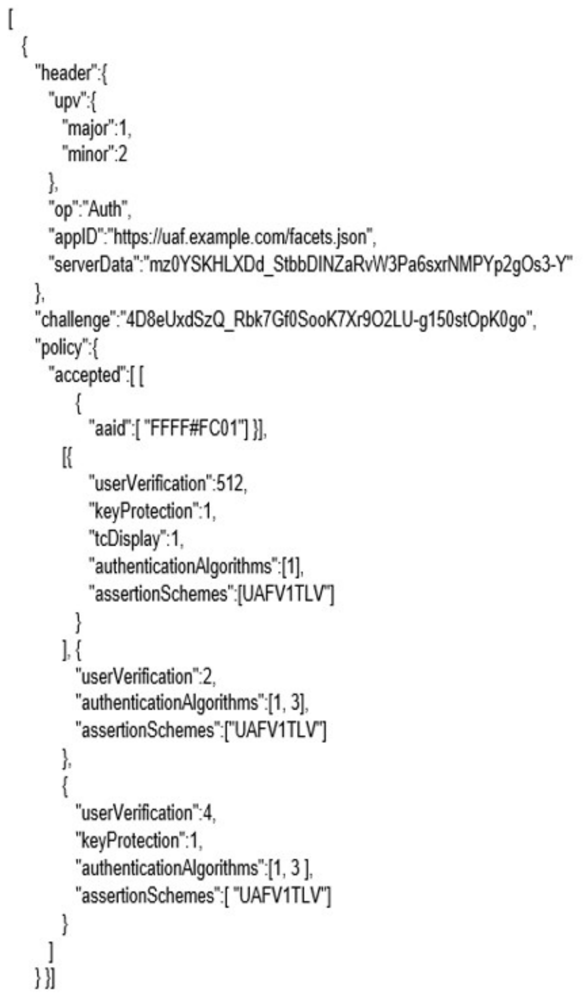
4.2.2 UPRC FIDO server Authentication

The authentication process is also divided into two parts as the server in this process also sends two messages, an authentication request which means the beginning of the authentication process, and the completion message of the user authentication.

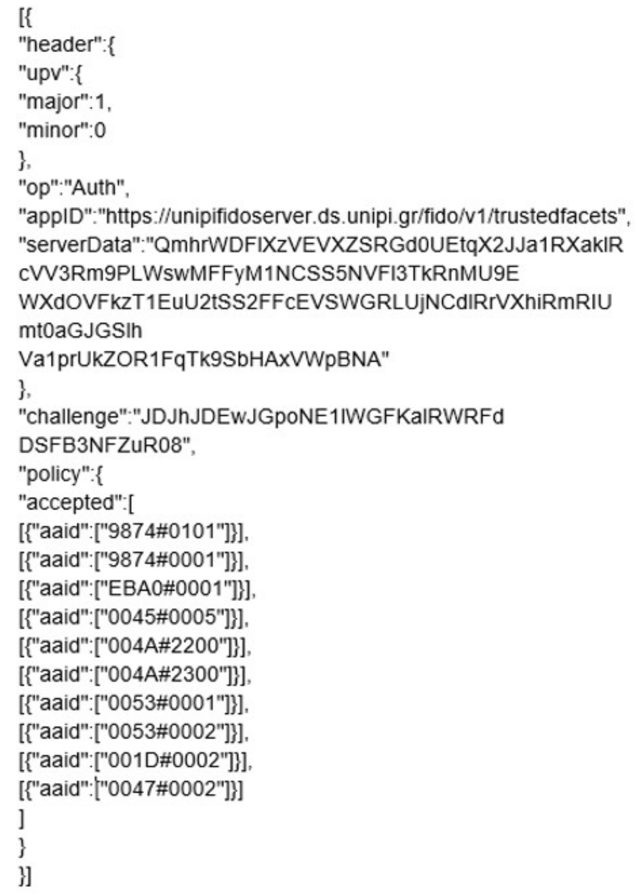
Authentication Request

The authentication process is based on a cryptographic challenge-response scheme in which the user is called by the FIDO UAF Server to authenticate himself/herself using the FIDO authenticator, that was used during the registration process. After the authentication endpoint is called by the FIDO Client, the UAF Server will generate the policy and the challenge with the purpose to send them as a reply to the Fido Client.
The authentication request that the FIDO UAF server will send to the client consists of the same parameters as the registration request.

We will quote the authentication request as described in the specification [16], and the authentication request sent by the server of the implementation. Their main difference again, is in the policy, in which as we see, in the specification it consists of a variety of match criteria, while the implementation's, has match criteria which consists only of AAID.



```
[
  {
    "header":{
      "upv":{
        "major":1,
        "minor":2
      },
      "op":"Auth",
      "appID":"https://uaf.example.com/facets.json",
      "serverData":"mz0YSKHLXDd_StbbDINZaRvW3Pa6sxrNMPYp2gOs3-Y"
    },
    "challenge":"4D8eUxdSzQ_Rbk7Gf0SooK7Xr9O2LU-g150stOpK0go",
    "policy":{
      "accepted":[ [
          {
            "aaid":[ "FFFF#FC01"] }],
        [{
            "userVerification":512,
            "keyProtection":1,
            "tcDisplay":1,
            "authenticationAlgorithms":[1],
            "assertionSchemes":[UAFV1TLV"]
          }
        ], {
          "userVerification":2,
          "authenticationAlgorithms":[1, 3],
          "assertionSchemes":["UAFV1TLV"]
        },
        {
          "userVerification":4,
          "keyProtection":1,
          "authenticationAlgorithms":[1, 3 ],
          "assertionSchemes":[ "UAFV1TLV"]
        }
      ]
    } }]
```



```
[{
"header":{
"upv":{
"major":1,
"minor":0
},
"op":"Auth",
"appID":"https://unipifidoserver.ds.unipi.gr/fido/v1/trustedfacets",
"serverData":"QmhrWDFIXzVEVXZSRGd0UEtqX2JJa1RXaklR
cVV3Rm9PLWswMFFyM1NCSS5NVFI3TkRnMU9E
WXdOVFkzT1EuU2tSS2FFcEVSWGRLUjNCdlRrVXhiRmRIU
mt0aGJGSlh
Va1prUkZOR1FqTk9SbHAxVWpBNA"
},
"challenge":"JDJhJDEwJGpoNE1lWGFKalRWRFd
DSFB3NFZuR08",
"policy":{
"accepted":[
[{"aaid":["9874#0101"]}],
[{"aaid":["9874#0001"]}],
[{"aaid":["EBA0#0001"]}],
[{"aaid":["0045#0005"]}],
[{"aaid":["004A#2200"]}],
[{"aaid":["004A#2300"]}],
[{"aaid":["0053#0001"]}],
[{"aaid":["0053#0002"]}],
[{"aaid":["001D#0002"]}],
[{"aaid":["0047#0002"]}]
]
}
}]
```

Specifications Authentication request UPRC Fido Authentication Request

Return Verification Result

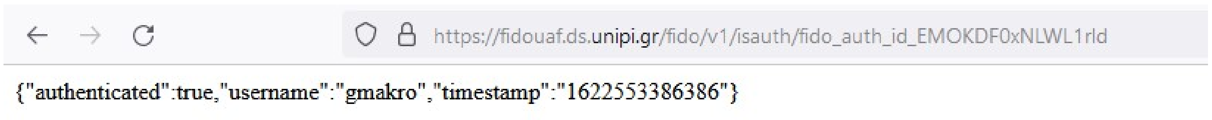
As soon as the Fido UAF Client receives from ASM, the confirmation message that the user has been verified successfully, it will send the authentication response to the Fido server. The server will process the message and more specifically the fields of the header, fcp, and the assertion. If everything proves to be correct then the authentication process will be completed successfully and the server will send to the Fido client a message confirming the result of the procedure. In this case, as in registration, the protocol does not have any restrictions on the structure of the message.

```
{
   "authenticator":{
      "AAID":"EBA0#0001",

"KeyID":"dU5zekI3VFYtZlFMeG5KaGZJNnZaMlJZR0l0ZElBTE1VUlFRYXNCaGlTdz
0"
   },
   "deviceId":"f15d084c57c60006�?\b\u0000SM-G965F",
   "username":"makro12222",
     "status":"1200",
   "timeStamp":"1619441493525",
   "authenticationId":"fido_auth_id_53x-zEWUz-LmnpFT",
   "radiusPassword":"SUkw3B4tw-vO_0hsf05Rq7Flvc0 "
}
```

UPRC Server Authentication Verification Result

In the above message, we can distinguish two very important new fields, authenticationId, and radiusPassword. AuthenticationId is a sole value that the server generates using base64enconding each time a user authenticates. The value in this field is unique, it corresponds to single-user and its lifespan is as long as the timestamp of this authentication. The fact that the value of authenticationId is renewed every time a user authenticates himself/herself, makes it safe in alteration or simulation attacks. The server for the specific field has an API which by calling it validates if the holder of the specific authenticationId is authenticated, its username and its timestamp. The call for this API can be requested through a GET request at /fido/v1/isauth/{auth} endpoint, where we replace {auth} with the authenticationId we want to check the validity (see Figure 23). RadiusPassword is also a unique value, that the server issue using base64enconding each time a user authenticates itself. This field can be used to integrate the Fido server with a Radius server for passwordless authentication of users in scenarios with services such as VPN and WI-FI.

The status in this case when it has a value of 1200, means that the process was performed successfully, while in any other case the server gives an error message 1400 which means that the authentication response received from the Fido Client had the wrong structure.


https://fidouaf.ds.unipi.gr/fido/v1/isauth/fido_auth_id_EMOKDF0xNLWL1rld

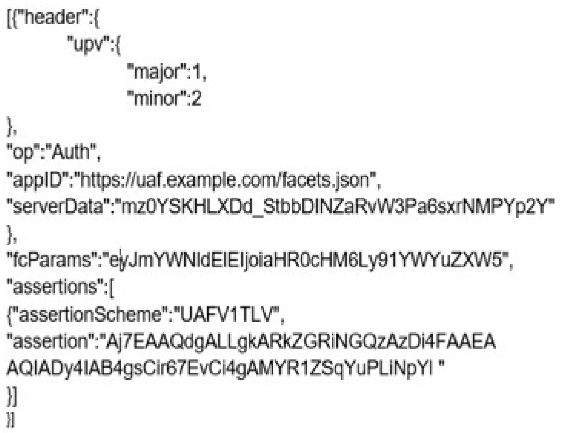
```
{"authenticated":true,"username":"gmakro","timestamp":"1622553386386"}
```

Figure 23 FIDO UAF isAuth API

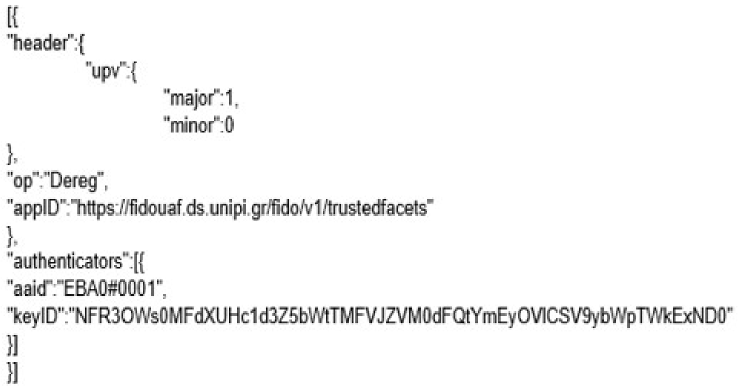
4.2.3 UPRC FIDO server Deregistration

The Deregistration process is used when a user wants to delete his/her account from Fido Server. The user by triggering the deregistration endpoint will enable the Fido server to send the deregistration request which will command the authenticator to delete the UAF credentials associated with the user's account.

In this process, the Fido UAF server does not send any confirming message about the result of the process. The deregistration request in terms of its structure is very similar to the requests of the two previous procedures, the only difference is that this one does not have the policy and the challenge fields. We are going to compare the deregistration request as described in the specification, with the deregistration request that the server of our implementation sends, in order to show that they don't have any difference.



```
[{"header":{
        "upv":{
                "major":1,
                "minor":2
},
"op":"Auth",
"appID":"https://uaf.example.com/facets.json",
"serverData":"mz0YSKHLXDd_StbbDINZaRvW3Pa6sxrNMPYp2Y"
},
"fcParams":"eyJmYWNldElEIjoiaHR0cHM6Ly91YWYuZXW5",
"assertions":[
{"assertionScheme":"UAFV1TLV",
"assertion":"Aj7EAAQdgALLgkARkZGRiNGQzAzDi4FAAEA
AQIADy4IAB4gsCir67EvCi4gAMYR1ZSqYuPLiNpYI "
}]
}]
```



```
[{
"header":{
        "upv":{
                "major":1,
                "minor":0
},
"op":"Dereg",
"appID":"https://fidouaf.ds.unipi.gr/fido/v1/trustedfacets"
},
"authenticators":[{
"aaid":"EBA0#0001",
"keyID":"NFR3OWs0MFdXUHc1d3Z5bWtTMFVJZVM0dFQtYmEyOVlCSV9ybWpTWkExND0"
}]
}]
```

Specifications Deregister request UPRC Fido Deregister Request

4.3 Fido Keycloak Library

In the FIDO UAF server, we have integrated a library written in java which is used as a gateway between Keycloak and the fido server. More specifically, this library is activated every time a user is authenticated through the Fido UAF protocol and offers the possibility to the UAF server to log-in to Keycloak as an administrator using the admin account and the credentials we created in section 4.1. Thus, the FIDO UAF server has the ability to add or renew the user's fidoAuthenticationId each time the user is authenticated. The library is based on three requests: a GET, a POST, and a PUT.

```
public interface AdminAPI {

    @FormUrlEncoded
    @POST("realms/" + APIConfiguration.REALM + "/protocol/openid-connect/token")
    Call<AccessToken> getAdminAccessToken(@Field("grant_type") String grant_type,
                                          @Field("client_id") String client_id,
                                          @Field("username") String username,
                                          @Field("password") String password) throws IOException;
    @GET("admin/realms/" + APIConfiguration.REALM + "/users")
    Call<List<User>> getListOfUsers(@Header("Authorization") String token);

    @PUT("admin/realms/" + APIConfiguration.REALM + "/users/{id}")
    Call<ResponseBody> updateUser(@Path("id") String id,
                                  @Header("Authorization") String token,
                                  @Header("Content_Type") String content_type,
                                  @Body User user);
}
```

The Code of the Requests

The Fido Server in order to be able to log in and make various changes to the user profiles, it must first obtain an access token. The access token can be obtained using the OAuth 2.0 Authorization Code grant type. The server will send a POST request to the RestFul API of keycloak which is located at the URL: **http://localhost:8180/auth/realms/demo/protocol/openid-connect/token**. The body of the request will consist of: client -id where in this case will be the Admin command-line interface, the username, the password, and the grant-type which symbolizes the way that we want to obtain the request. The prices we will enter are the following:

Client-id: “admin-cli”

Username: admin

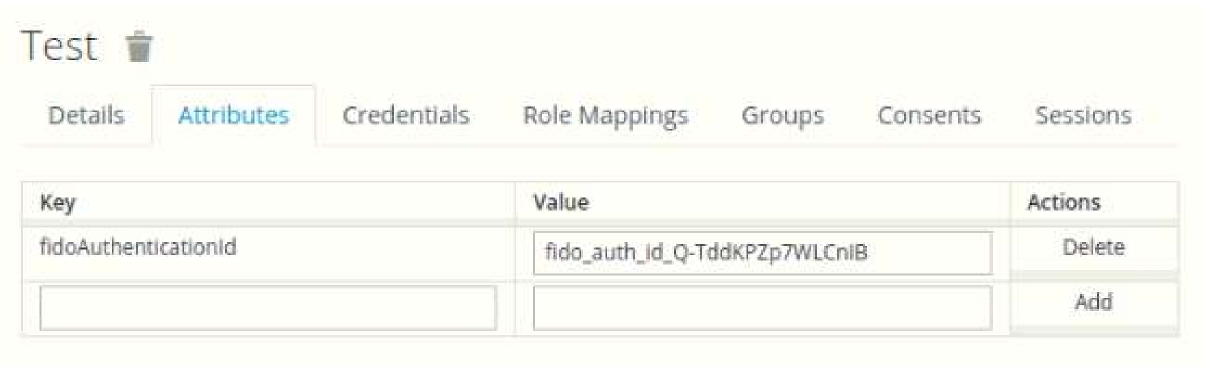
Password: admin

grant-type: password

```java
public class APIClient {
    public AdminAPI client;
    public APIClient() {
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl(APIConfiguration.API_BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
        client = retrofit.create(AdminAPI.class);
    }

    public AccessToken getAccessToken() throws IOException {
        Call<AccessToken> call = client.getAdminAccessToken(
                APIConfiguration.GRANT_TYPE,
                APIConfiguration.CLIENT_ID,
                APIConfiguration.USERNAME,
                APIConfiguration.PASSWORD);
        Response<AccessToken> response = call.execute();
        if (!response.isSuccessful()) {
            throw new IOException(response.errorBody() != null
                    ? response.errorBody().string() : "Unknown error");
        }
        return response.body();
    }
    public List<User> getUsers(String token) throws IOException {
        String access_token = "Bearer " + token;
        Call<List<User>> call = client.getListOfUsers(access_token);
        Response<List<User>> response = call.execute();
        if (!response.isSuccessful()) {
            throw new IOException(response.errorBody() != null
                    ? response.errorBody().string() : "Unknown error");
        }
        return response.body();
    }
    public void updateUserAuthenticationId(String id, String token, User user) throws
IOException {
        String access_token = "Bearer " + token;
        Call<ResponseBody> call = client.updateUser(id, access_token,
APIConfiguration.JSON_CONTENT_TYPE, user);
        Response response = call.execute();
        if (!response.isSuccessful()) {

            throw new IOException(response.errorBody() != null
                    ? response.errorBody().string() : "Unknown error");
        }
    }

}
```

The Functions of the Keycloak UAF library.

Once the FIDO UAF server obtains the access token from Keycloak, it will execute a GET request in which it will request a list of all the users who are registered in the realm that we made for the specific project. Having received the list and knowing from the UAF authentication process the username, it will search for the specific user in order to add it (if it is the his/her first authentication) or to renew the fidoAuthenticationId. This process is done through a PUT request that sends the FIDO UAF server as admin in keycloak. Below we present an image with the updated fidoAuthenticationId attribute after the authentication of the user with Username Test in the Fido UAF server.



Key	Value	Actions
fidoAuthenticationId	fido_auth_id_Q-TddKPZp7WLCnIB	Delete
		Add

Figure 24 Update of the fidoAuthenticationId From FIDO server

4.4 Keycloak Rest Authenticator

The last component from the side of relying parties is the assimilation of new authentication mechanism to the registered client, which will combine Keycloak with the Fido UAF server. Keycloak offers a script authenticator, that enables javascript processing using its basic class model. It can be used to build HTTP requests and act on the response. Using this authenticator, we do not need to program in difficult frameworks such as in java as there is the possibility to use JavaScript as a basic programming language. At the same time, it is easier to build micro-services, which implement the authentication logic that the authenticator is required to have in order to fulfill the needs of the project [17].

This solution is based on a single NPM package, the keycloak-rest-authenticator which was designed to be used with the express js framework. First, we will demonstrate the code and its logic.

```
const FidoAuthenticator = createAuthenticator(
  function () {
    this.fidoAuthenticationIdAttribute = "fidoAuthenticationId";
  },
  {
    processNew: async function (req) {
      console.info(`\n[${new Date()}] New request from Keycloak
received for User: ${req.user.username}`);

console.info(`\n[${req.user.attributes[this.fidoAuthenticationIdAttr
ibute]}`);

          await
axios.get(`http://localhost:8080/fido/v1/isauth/${req.user.attribute
s[this.fidoAuthenticationIdAttribute]}`).then((resp) => {

        var response=JSON.stringify(resp.data);
        var obj=JSON.parse(response)
         console.log(obj.authenticated);
               this.latestFidoAuthenticationId = obj.authenticated;
               this.usersuname=obj.authenticated=obj.username;

      });


      if (
         req.user.username==this.usersuname &&
          this.latestFidoAuthenticationId==true
      ) {
        return {} // meaning successful verification
      } else {
        return {

          failure: "invalidCredentials",
        };
      }
    }
```

The programming logic of the authenticator was based on the following. Once the user has already authenticated via the Fido UAF server, the fidoAuthenticationId attribute has been supplemented with a unique value created by the Fido server. The user is directed to a login page for which Keycloak is responsible in order to connect to the web service of his/her choice. On the background of the user login page, the script authenticator will run and it will ask the user to enter the username and press the authentication button. Once the user presses the button, the authenticator will receive the username from Keycloak along with the fidoAuthenticationId attribute that the FIDO UAF server has been completed for that particular user. The authenticator will then perform a GET request, on /isauth/{auth} API of the server using the user's fidoAuthenticationId, to verify if the user that tries to log in, has

actually been authenticated through the FIDO UAF server and if authentication is still active. The answer to this request is a Json with the values: authenticated, username and timestamp . In case the Boolean authenticated value is true and the username is the same as the username entered by the user, then the authenticator allows the user to enter the web application. In case the authentication through the Fido protocol has expired or the username of the owner of the fidoAuthenticationId does not match the username of the user trying to authenticate in keycloak or a combination of the two, the authenticator will block the authentication of the user, and the page login will display an error message. By adding code to Keycloak, we have set, for security reasons, the deletion of the value contained in the attribute fido authenticationId, each time a successful authentication is performed, in order to prevent any authentication processes from being tampered with reuse attempts.
The KeycloakScript Packager must be initialized with the configuration that will enable the node service to develop its definition within the Keycloak environment, so that the Keycloak server can also use the service endpoint.
We will define in the configuration the keycloakDeploymnetLocation field with the path keycloak/standalone/deployments so that the auto-deploy jar files are placed in this folder. Then we will fill in the keycloakAccessibleBaseUrl value and set it with the absolute path in which the keycloak will be able to reach the node service.

```
const packager = new KeycloakScriptPackager({
  keycloakDeploymentLocation: "/home/fido-
server/Downloads/keycloak-13.0.0/standalone/deployments",
  keycloakAccessibleBaseUrl: "http://localhost:10000",
});
```

Finally, we need to instruct the node service to create and deploy the jar file in Keycloak. With the make () function the packager constructs the jar and places it in the desired folder.

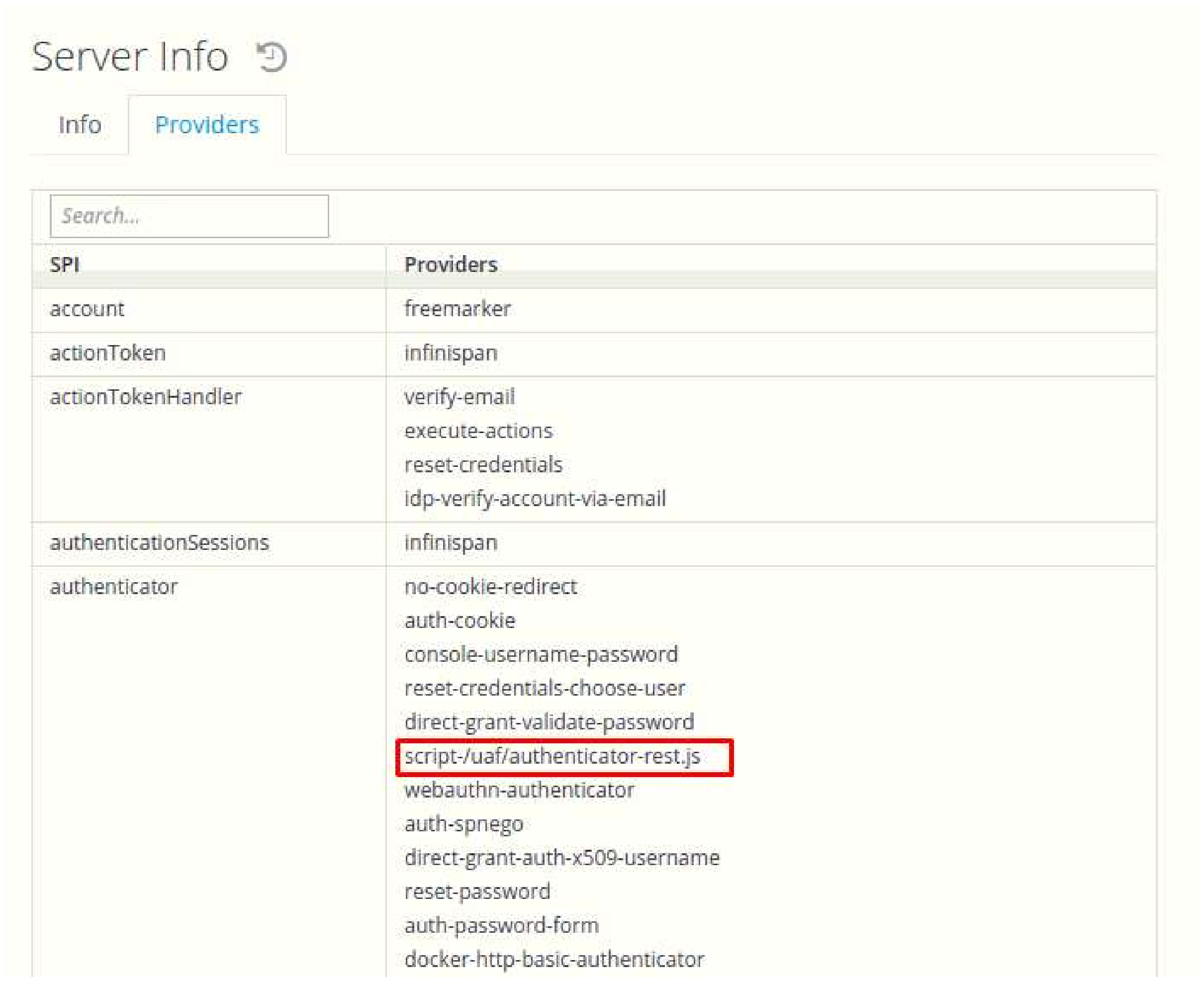
```
app.listen(10000, "0.0.0.0", ()=>{
  console.log("Keycloak-FIDO UAF REST Authenticator running on
http://localhost:10000");
  packager.make();
}
```

We run the node service as normal nodejs process by using the command:

```
node index.js
```

In order to define the authenticator that we built as the only way to authenticate the user when he/she tries to connect to a client that is protected by Keycloak we will follow the following steps:

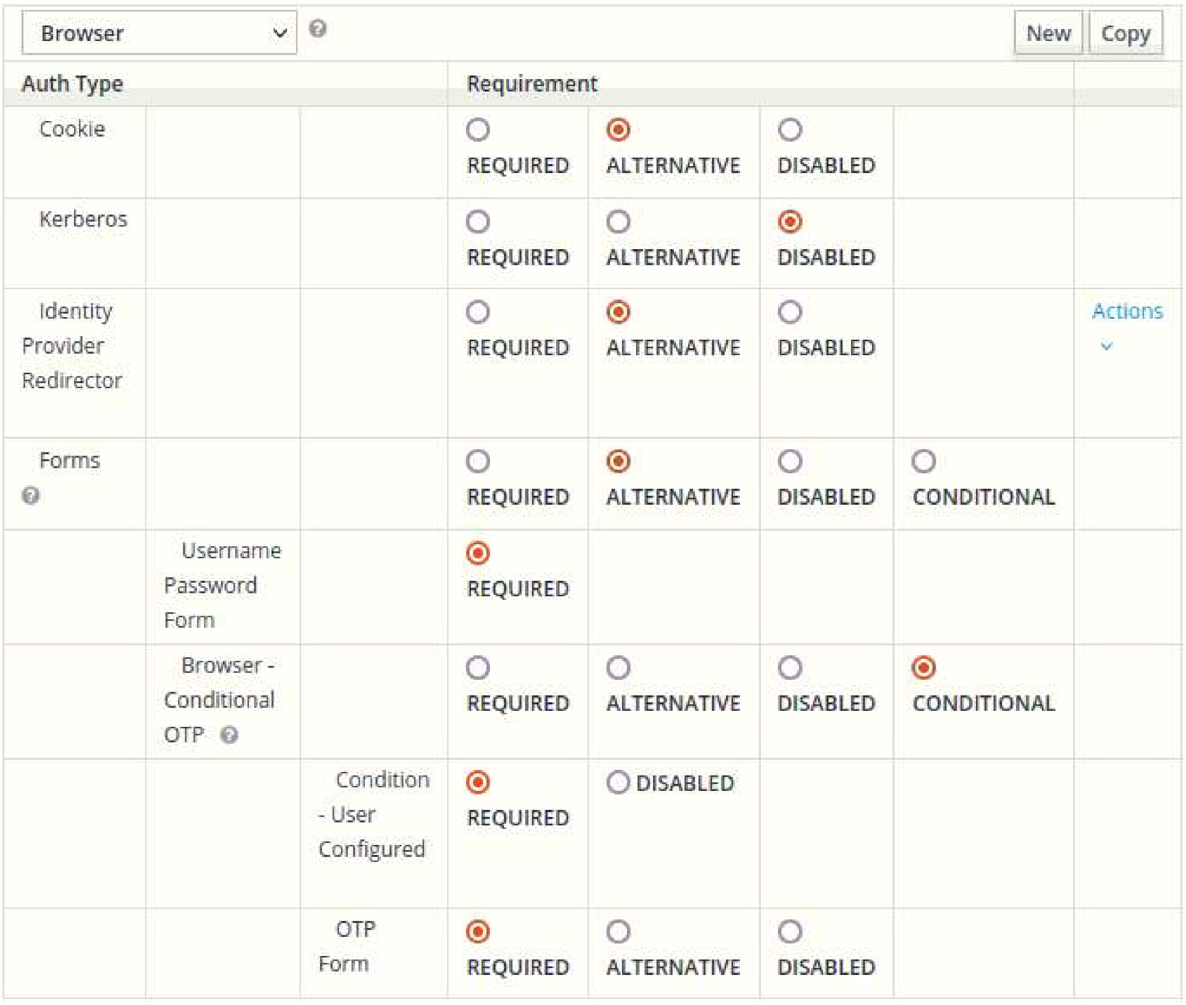
1. Firstly, we will log in to the admin console of Keycloak and we will go to the page ServerInfo/Providers in order to check if the authenticator we created has been integrated into Keycloak. In case the provider with value /script/uaf/authenticator-rest.js is not found in the authenticator field, check the keycloak, service logs, and also check if the jar file has been created.



SPI	Providers
account	freemarker
actionToken	infinispan
actionTokenHandler	verify-email execute-actions reset-credentials idp-verify-account-via-email
authenticationSessions	infinispan
authenticator	no-cookie-redirect auth-cookie console-username-password reset-credentials-choose-user direct-grant-validate-password script-/uaf/authenticator-rest.js webauthn-authenticator auth-spnego direct-grant-auth-x509-username reset-password auth-password-form docker-http-basic-authenticator

Figure 25 Intergration of the authenticator to Keycloak.

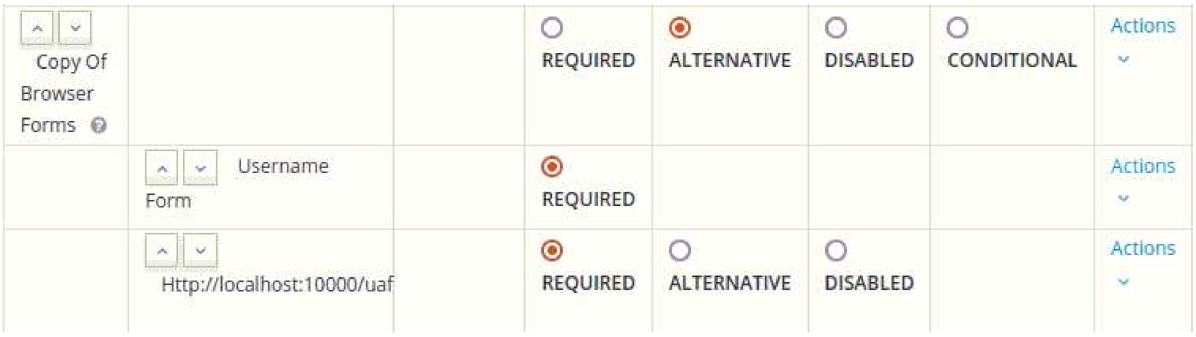
2. Having selected the correct realm, go to the Configuration field and press the Authentication option.


Browser New Copy

Auth Type			Requirement				
Cookie			REQUIRED	[✓ ALTERNATIVE]	DISABLED		
Kerberos			REQUIRED	ALTERNATIVE	[✓ DISABLED]		
Identity Provider Redirector			REQUIRED	[✓ ALTERNATIVE]	DISABLED		Actions
Forms			REQUIRED	[✓ ALTERNATIVE]	DISABLED	CONDITIONAL	
	Username Password Form		[✓ REQUIRED]				
	Browser - Conditional OTP		REQUIRED	ALTERNATIVE	DISABLED	[✓ CONDITIONAL]	
		Condition - User Configured	[✓ REQUIRED]	DISABLED			
		OTP Form	[✓ REQUIRED]	ALTERNATIVE	DISABLED		

Figure 26 The default Browser authentication flow

3. From the Authentication/Flows dropdown choose the Direct Grant and using the "Copy" button on the right create a copy of the Direct Grant Flow and name it "Copy of Browser".
4. Select the newly created "Copy of Browser" authentication flow and delete the default authenticators.
5. Click the "Add execution" button and from the provider dropdown try to find the username form select it and save it.
6. Click the "Add execution" button and from the provider dropdown try to find the FIDO UAF JS Authenticator provider (e.g., "Http://localhost:10000/uaf") select it and save it.
7. Set the newly added FIDO UAF JS Authenticator Auth Type as ALTERNATIVE



Copy Of Browser Forms			REQUIRED	[✓ ALTERNATIVE]	DISABLED	CONDITIONAL	Actions
	Username Form		[✓ REQUIRED]				Actions
	Http://localhost:10000/uaf		[✓ REQUIRED]	ALTERNATIVE	DISABLED		Actions

Figure 27 modified Browser flow with our new authenticator

8. Edit the Keycloak Client that you want to enable FIDO UAF authentication (e.g., the "oidc-client" client) as follows:
   a. Go to Clients > js-console.
   b. Scroll down and open the "Authentication Flow Overrides" and for the Direct Grant Flow select "Copy of Browser".

Now if a user wants to log in to the web service, he/she will be faced with the new authentication flow (figure 28).

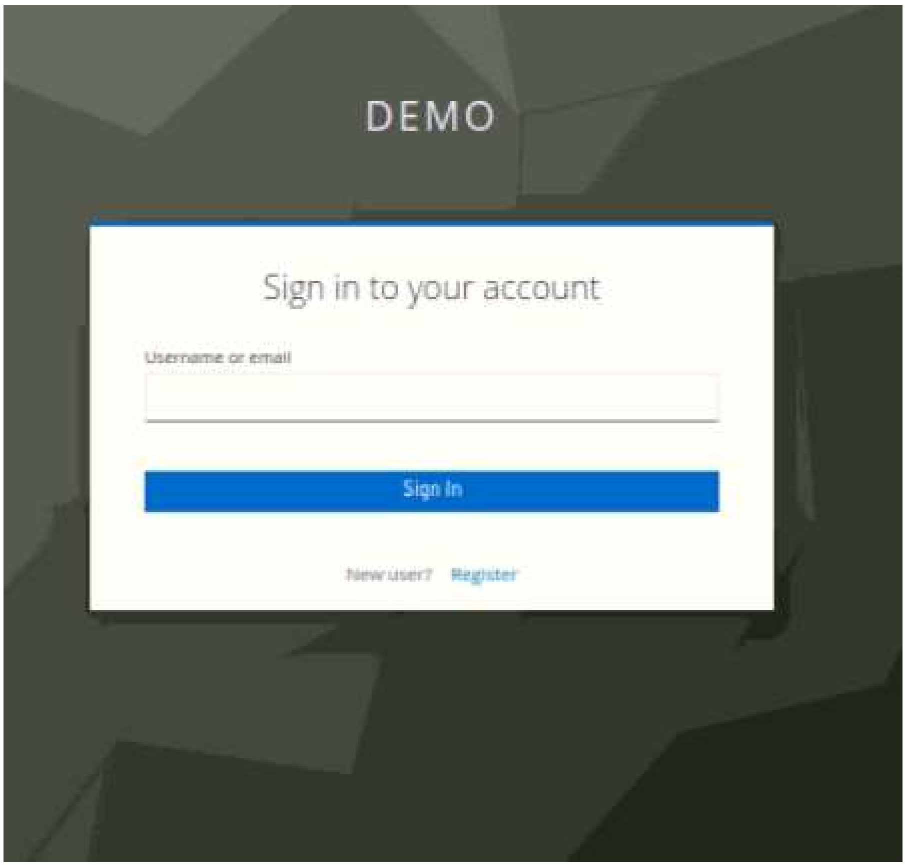


Figure 28 The new authentication flow.

4.5 Android Client

The android client of this work has been divided into two parts that work together to achieve a common result, while their structure is designed to provide usability. The first part consists of the fido UAF client which is responsible for the user's communication with the UAF server so that the user can be authenticated through the Fido protocol. The second part consists of the Keycloak android client which provides the user with the ability to access the web service he/she wants to enter using as a means of authentication the FIDO UAF collaboration with Keycloak Idp. The User’s device consists of 2 applications:

1. UAF Client
2. Keycloak Client

The user's mobile device is the most important component in the architecture, as it allows us to achieve device-centric architecture. During authentication, the user is asked to perform a local authentication on his/her mobile phone using his/her biometric features. This is achieved through the FIDO UAF client application. This application is designed to meet the requirements of the FIDO UAF protocol and furthermore, to support all types of human-to-device authentication that are required by FIDO. Finally, this mobile device also includes the application that allows the user to interact with the identity provider (idp) in order to store and confirm the user's identity so that he/she can connect respectively to the web service.

4.5.1 Unipi FIDO UAF Android Client

The Unipi Fido UAF client application consists of three parts: the client, the Authenticator Script Module (ASM), and the authenticator. The Client executes the Discovery, Registration, Authentication, and Deregistration functions. ASM is located between the Client and the Authenticator and is responsible for the communication between them. The Authenticator makes the key to the cryptographic challenge by combining the supported algorithms with the desired authentication input (e.g. fingerprint).

The verification methods that can be used for authentication through the Unipi UAF Client are the following:

- Fingerprint
- 4-digit-PIN
- Eye print

- Faceprint
- Pattern

The programming language in which the application was programmed is in java. The software architecture in Android implements FIDO ASM as a separate APK packaged application, as suggested by the UAF documentation. ASM protects the keys in the Android Keystore System

As aforementioned, the FIDO UAF client implements four functions. Below we will describe what these functions do, while we will present the messages sent by the client to the server. These messages are fully in line with the instructions of the protocol specification.

1. Discovery: This function runs only once when the application starts and allows the relying party to verify the availability of FIDO capabilities in the client, at the same time through the metadata statement, it is checked the availability of authenticators.
2. Registration: The client generates a new set of keys and associates it with an account on the relying party server. The set of keys is based on the policy set by the server and the acceptable attestation that the authenticator and the registration process, in general, are in line with this policy.

```
[
   {
      "assertions":[
         {
            "assertion":"AT5LAwM-
9wALLgkARUJBMCMwMDAxDi4HAAAAAQIAAQEKLiAA87bSDCc2ayLukgSyPV2NE
cta4E48\nNX6ouww5JKueigoJLiwAcUlCbUZDc0RUcE91ZVhjVV9aT0U1WGFZ
VW5iX2xSbXgyN3RyZ1lCUlVI\nbz0NLggAAAAAAAAAAAAMLlsAMFkwEwYHKoZ
Izj0CAQYIKoZIzj0DAQcDQgAEBm5TRGTnK6vkb32B\nHulZHDpKFnOVSqKSDs
WYTcm4DMM0nc9yT_kJPF7wFBBwty8kMhDSGwXfZeyWCZnnIOaIw_w_EABm\nM
TVkMDg0YzU3YzYwMDA2_T8IAFNNLUc5NjVGBz5MAgYuRwAwRQIhAOY6quf0iO
H4g6rov80LpnMb\nB6L99q1Df1f1SD03oSaKAiBtlWEZZbu3PmIlxll3mYkiH
k1osbi3Qvj6E_so_kBrMAUu_QEwggH5\nMIIBn6ADAgECAgRVMUzTMAkGByqG
SM49BAEwgYQxCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJDQTER\nMA8GA1UEBww
IU2FuIEpvc2UxEzARBgNVBAoMCmVCYXksIEluYy4xDDAKBgNVBAsMA1ROUzES
MBAG\nA1UEAwwJZUJheSwgSW5jMR4wHAYJKoZIhvcNAQkBFg9ucGVzaWNAZWJ
heS5jb20wHhcNMTUwNDE3\nMTgxMTMxWhcNMTUwNDI3MTgxMTMxWjCBhDELMA
kGA1UEBhMCVVMxCzAJBgNVBAgMAkNBMREwDwYD\nVQQHDAhTYW4gSm9zZTETM
BEGA1UECgwKZUJheSwgSW5jLjEMMAoGA1UECwwDVE5TMRIwEAYDVQQD\nDAll
QmF5LCBJbmMxHjAcBgkqhkiG9w0BCQEWD25wZXNpY0BlYmF5LmNvbTBZMBMGB
yqGSM49AgEG\nCCqGSM49AwEHA0IABDyHDmUdNRe9ndLNj1quBs44EPamfnMA
ziZtKHBDIIvltGyRMvcdU9Y9DGph\nUz7ekm0kgtio5awgx3lFv-
HwP4QwCQYHKoZIzj0EAQNJADBGAiEAimSin8vd862bv1DaUqsrF5r4\npy8DW
DqqTNqPjZUPaGACIQCu0-lMQAMhbPbKuXlcZ8lVx9wWa-kwC4wq-
4TUnveOng\u003d\u003d\n",
            "assertionScheme":"UAFV1TLV"
         }
      ],

"fcParams":"eyJhcHBJRCI6Imh0dHBzOi8vdW5pcGlmaWRvc2VydmVyLmRzL
nVuaXBpLmdyL2ZpZG8vdjEvdHJ1\nc3RlZGZhY2V0cyIsImNoYWxsZW5nZSI6
IkpESmhKREV3SkRoNWVWTXliRkJ2TnpaemJFbHhUa3RN\nWmxOUU1uVSIsImN
oYW5uZWxCaW5kaW5nIjp7fSwiZmFjZXRJRCI6ImFuZHJvaWQ6YXBrLWtleS1o
\nYXNoOnl1MjNGNk4wMTBoSFp5Q25aTVplRVFEZnNpN2hfcmFqaElTdUZhenp
iZGNcdTAwM2QifQ\u003d\u003d\n",
      "header":{

"appID":"https://fidouaf.ds.unipi.gr/fido/v1/trustedfacets",
         "op":"Reg",

"serverData":"TGJsRW9ZZjdQSlVrdmRibFh1ZUlmZS1JN2tmUW9XSEdsQTd
2dmlhT21lNC5NVFl3TmpVMk5EZzFNVE14TlEuZFc1cGNHbGZabWxrYncuU2tS
S2FFcEVSWGRLUkdnMVpWWk5lV0pHUW5aT2VscDZZa1ZzZUZScmRFMWFiRTVSV
Fc1Vg",
         "upv":{
            "major":1,
            "minor":0
         }
      }
   }
]
```

Registration Response Message

3. Authentication: This function allows the user to list an account identification, as proof of the previous process and potentially any other certified data on the relying party server.

```
[
   {
      "assertions":[
         {
            "assertion":"Aj7SAAQ-
ggALLgkARUJBMCMwMDAxDi4FAAAAAQIADy4IAITQrNPbLZaJCi4gAAIKx87iP
If2adG1\nDdt3OHOF9ZdIbuCX-
LFNASzPJmfkEC4AAAkuLABxSUJtRkNzRFRwT3VlWGNVX1pPRTVYYVlVbmJf\n
bFJteDI3dHJnWUJSVUhvPQ0uBAAAAAAABi5IADBGAiEAtTtfI3KaSsxTuNsrF
AMLD3YM-
g8DazHk\ndKkcDTTXxvQCIQDWvVIFNMatWPAVkVSiZvh6J50zTXdIROUItfwr
pb5RpQ\u003d\u003d\n",
            "assertionScheme":"UAFV1TLV"
         }
      ],

"fcParams":"eyJhcHBJRCI6Imh0dHBzOi8vdW5pcGlmaWRvc2VydmVyLmRzL
nVuaXBpLmdyL2ZpZG8vdjEvdHJ1\nc3RlZGZhY2V0cyIsImNoYWxsZW5nZSI6
IkpESmhKREV3SkV3d1RrTnBTbTFIV1RObmFVWTFVR05E\nZVROcVkwOCIsImN
oYW5uZWxCaW5kaW5nIjp7fSwiZmFjZXRJRCI6ImFuZHJvaWQ6YXBrLWtleS1o
\nYXNoOnl1MjNGNk4wMTBoSFp5Q25aTVplRVFEZnNpN2hfcmFqaElTdUZhenp
iZGNcdTAwM2QifQ\u003d\u003d\n",
      "header":{

"appID":"https://fidouaf.ds.unipi.gr/fido/v1/trustedfacets",
         "op":"Auth",

"serverData":"SC1PcUNUaXhiMlJ4bWRwbUFnUW1kYjdXUzFQN0VkWG9qMG9
yQ3VkQ2ktby5NVFl3TmpVMk5EZzFOVEUzTkEuU2tSS2FFcEVSWGRLUlhkM1ZH
dE9jRk50TVVoWFZFNXVZVlZaTVZWSFRrUmxWRTV4V1RBNA",
         "upv":{
            "major":1,
            "minor":0
         }
      }
   }
]
```

Authentication Response Message

4. Deregistration: This operation occurs when an account needs to be deleted by the relying party. The relying party on its part informs the authenticator to delete the associated UAF credentials stored for a specific user.

All communication between the application and the server takes place over TLSv1.3. A channel binding allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication at the higher layer to the channel at the lower layer [18].
The installation of the Unipi FIDO UAF client can be done either by downloading the APK on a phone or by downloading and executing the whole project on a virtual device through Android Studio [15].

The interface of the android application (figure 30) is very simple in terms of its design in order to be easy in use. It includes two fields in which the user fills in his/her username and the URL of the UAF SERVER, a field which displays the server response and seven buttons. These buttons are used as follow:

1. SET LOCAL FIDO/ SET PUBLIC FIDO: These buttons autocomplete the Server field with the URL of the Unipi FIDO UAF server.
2. REGISTER: This button triggers the authenticator registration process.
3. AUTHENTICATION: This button triggers the authentication process.
4. DEREGISTER: This button triggers the authenticator deregistration process.

5. CLEAR FIELDS: This button clears the username and the server fields.
6. CLEAR DATA: This button clears the data and the cache memory.

Figure 29 The start of the Unipi FIDO UAF client.

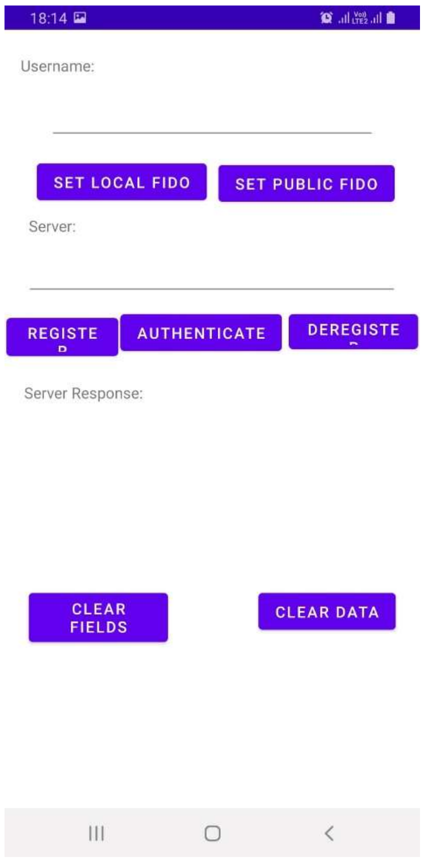


Figure 29 The interface at the start of the Unipi FIDO UAF client.

4.5.2 Unipi Keycloak Android Client

Keycloak Android Client is a custom application made for the user to access web applications that are protected by the Keycloak Identity Provider. The application is designed in such a way that it interacts with both the Keycloak server and the FIDO UAF client in order to serve the needs of the user, with the maximum utility. Due to the innovation that exists in the whole project, it was not possible to use the classic supported platforms offered by Keycloak, in order for a user to obtain his access token, therefore we created a new approach in terms of architecture for the construction of this implementation. The structure of the application for the purposes of the project was based on the structure of the web browsers that exist for the android devices, with the difference that the application for security reasons is directly connected to the web service protected by Keycloak. Furthermore, the application has for utility reasons, a button that automatically transfer the user to the FIDO UAF client in order to perform biometric registration or authentication. After the user follows the prerequisite steps needed the application will successfully log in him/her to the web service.

The installation of the Unipi FIDO UAF client can be done either by downloading the APK on a phone or by downloading and executing the whole project on a virtual device through Android Studio [15].

Below we will demonstrate the interface of the application:

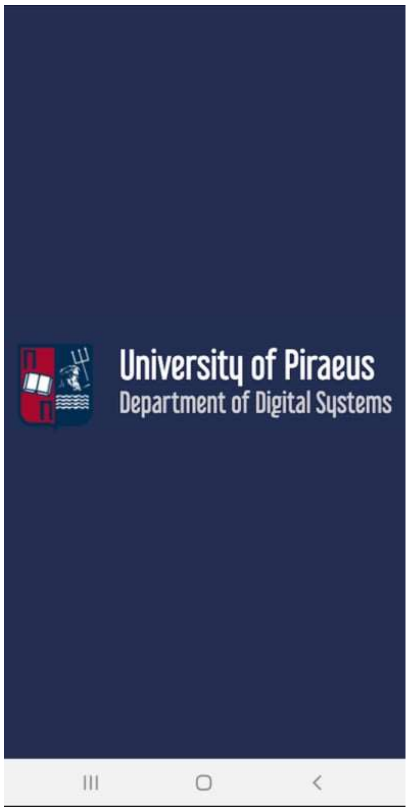
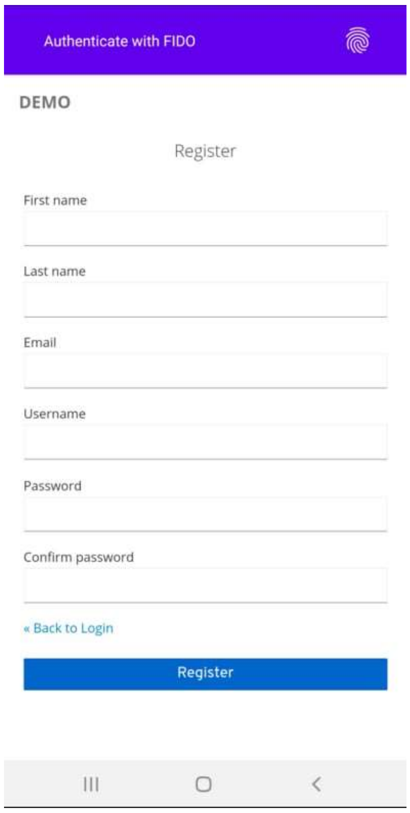


Figure 31 The start of the SSL Keycloak Client.


Authenticate with FIDO

DEMO

Register

First name

Last name

Email

Username

Password

Confirm password

« Back to Login

Register

Figure 30 The registration interface of the SSL Keycloak Client.

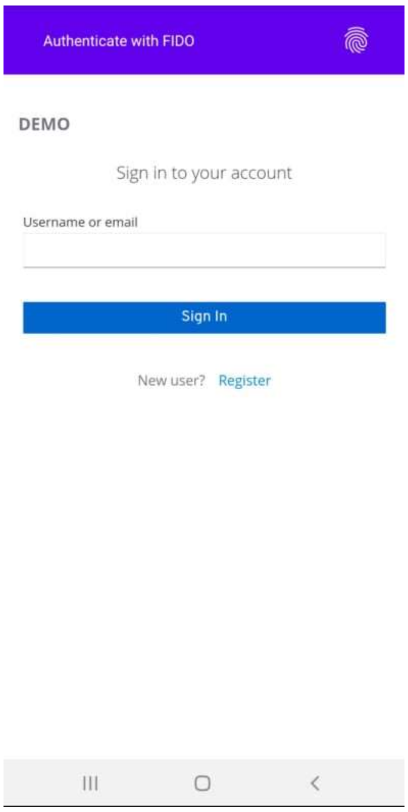


Figure 32 The authentication interface of the SSL Keycloak Client.

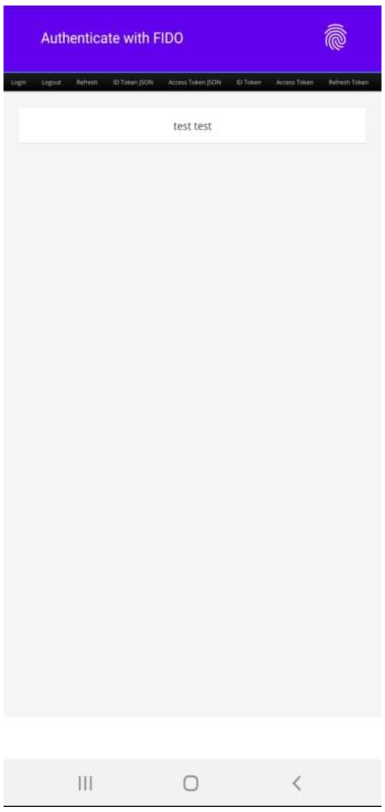


Figure 33 Access to the web service from the SSL Keycloak Client.

4.6 FIDO UAF & OpenID Connect Integration High Level Architecture

In this section, we will present through diagrams the detailed exchange of messages required between the implementation components for the registration, authentication and deregistration processes.

4.6.1 User Registration Flow

The registration flow needs to create credentials using the authenticator and store the public-key in user's profile. Let's take a closer look at the sequence of calls and involved components to implement this flow. The following diagram describes the custom registration flow.

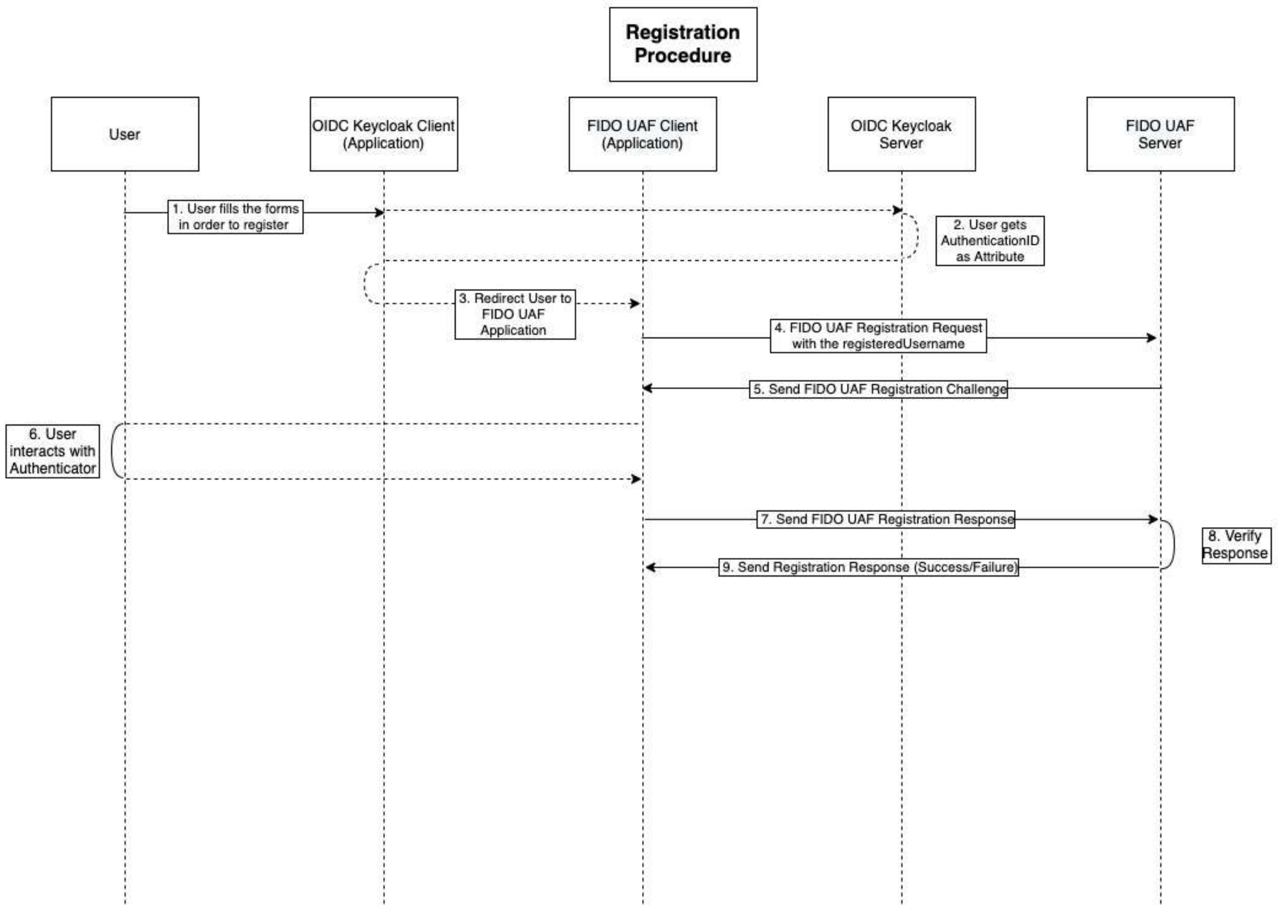

Registration Procedure
User
OIDC Keycloak Client (Application)
FIDO UAF Client (Application)
OIDC Keycloak Server
FIDO UAF Server
1. User fills the forms in order to register
2. User gets AuthenticationID as Attribute
3. Redirect User to FIDO UAF Application
4. FIDO UAF Registration Request with the registeredUsername
5. Send FIDO UAF Registration Challenge
6. User interacts with Authenticator
7. Send FIDO UAF Registration Response
8. Verify Response
9. Send Registration Response (Success/Failure)

Figure 34 Registration Procedure

1. The user through the Keycloak Client application fills in a form with his/her data in order to register on the Keycloak OIDC server.
2. Once the user completes the registration process, he/she will be registered in the Keycloak server database too. During this process, a default Authentication ID will be added as an Attribute for the specific user.
3. The user will be redirected to the FIDO UAF Client Application.
4. By filling in the username that he/she set during his/her registration in Keycloak and defining the Unipi FIDO UAF server, the user will send the FIDO UAF Registration Request through FIDO UAF client.
5. The FIDO UAF server will respond by sending the registration challenge.
6. The user will be authenticated locally in the authenticator.
7. The FIDO UAF Client will send the FIDO UAF registration response to the server.
8. The UAF Server will verify the message sent by the UAF Client.
9. The UAF Server will send the result of the registration process to the UAF CLIENT.

4.6.2 User Authentication Flow

The following diagram describes the custom authentication flow:

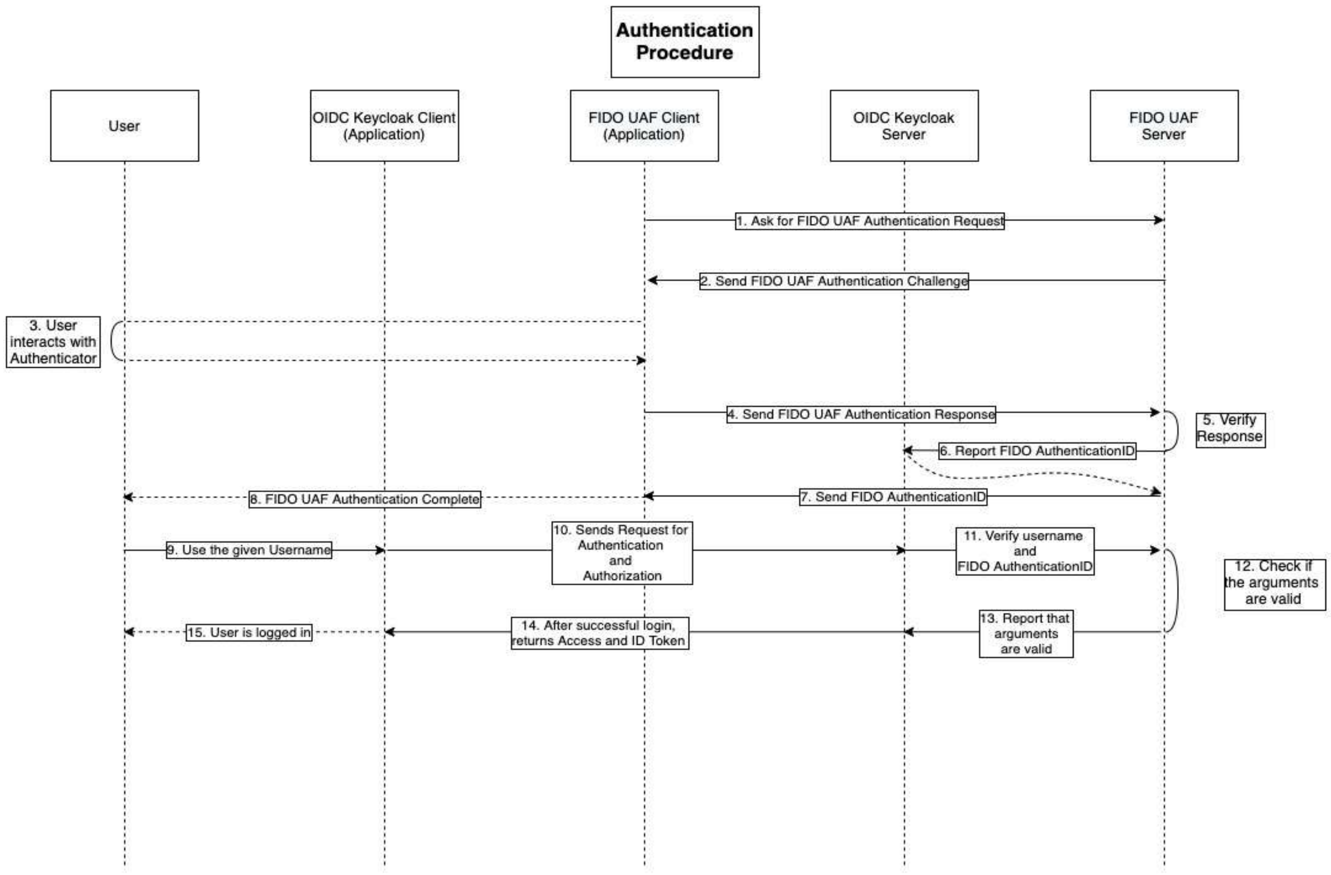

Authentication Procedure
User
OIDC Keycloak Client (Application)
FIDO UAF Client (Application)
OIDC Keycloak Server
FIDO UAF Server
1. Ask for FIDO UAF Authentication Request
2. Send FIDO UAF Authentication Challenge
3. User interacts with Authenticator
4. Send FIDO UAF Authentication Response
5. Verify Response
6. Report FIDO AuthenticationID
7. Send FIDO AuthenticationID
8. FIDO UAF Authentication Complete
9. Use the given Username
10. Sends Request for Authentication and Authorization
11. Verify username and FIDO AuthenticationID
12. Check if the arguments are valid
13. Report that arguments are valid
14. After successful login, returns Access and ID Token
15. User is logged in

Figure 35 Authentication Procedure.

1. By filling in the username that he/she defined during his registration in Keycloak and defining the Unipi FIDO UAF server, the user will send the FIDO UAF Authentication Request through the FIDO UAF Client.
2. The FIDO UAF server will respond by sending the authentication challenge.
3. The user will be authenticated locally in the authenticator.
4. The FIDO UAF Client will send the authentication response to the UAF server.
5. The UAF Server will verify the response sent by the UAF Client.
6. In case the authentication process has been completed successfully the FIDO UAF server will renew the FIdoauthenticationID for the specific user in the Keycloak's database.
7. The Fido UAF server will send the result of the authentication process to the UAF Client.
8. The User has completed the FIDO authentication process and he/she will open the Keycloak Client application.

9. The User will enter the username on the login page in order to authenticate via OpenID Connect.
10. The OIDC Keycloak Client application will send a request to Keycloak for authentication and authorization.
11. Keycloak through the script authenticator will send a get request to the /isAuth/{auth} API of the server in order to verify the username and fidoauthenticationID.
12. The Script authenticator will check the JSON given by the UAF server as a response.
13. The Script authenticator informs Keycloak that the user has passed the check as the username and fidoauthenticationID are valid.
14. The Keycloak server returns the access token and id token to the OIDC client.
15. The User through these tokens can have access to the web service.

4.6.3 User Deregistration Flow

The following diagram describes the custom deregistration flow

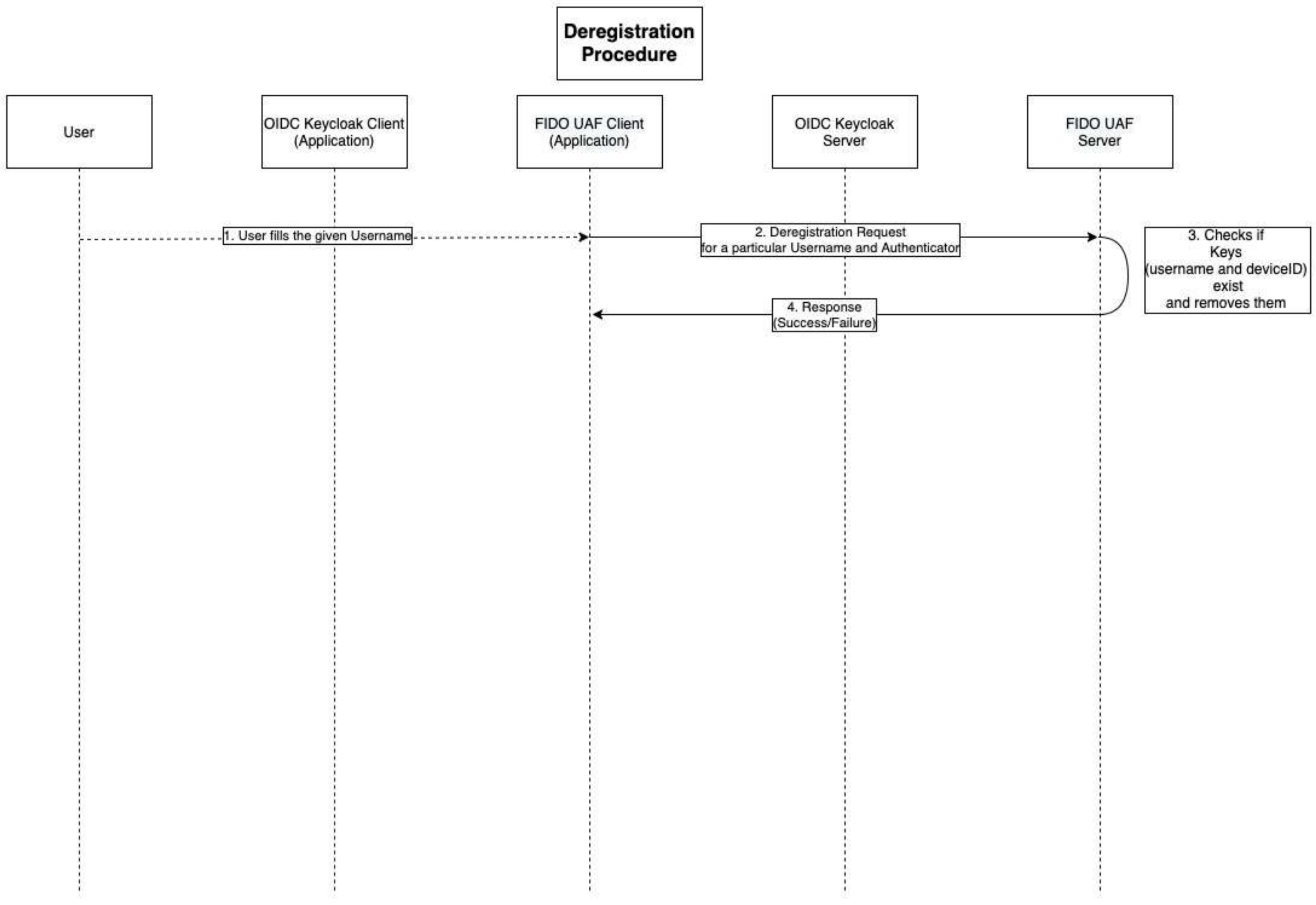

Deregistration Procedure
User
OIDC Keycloak Client (Application)
FIDO UAF Client (Application)
OIDC Keycloak Server
FIDO UAF Server
1. User fills the given Username
2. Deregistration Request for a particular Username and Authenticator
3. Checks if Keys (username and deviceID) exist and removes them
4. Response (Success/Failure)

Figure 36 Deregistration Procedure.

1. The user fills in the given username and sets the FIDO UAF Server in order to start the deregistration process.
2. The FIDO UAF Client will trigger the Deregistration process.
3. The UAF server will delete the username, deviceID, and the key set for the specific user from the database.
4. The server will send to UAF client as response a message with the result of deregistration process and the instruction to the authenticator for deleting the credentials stored for that user.

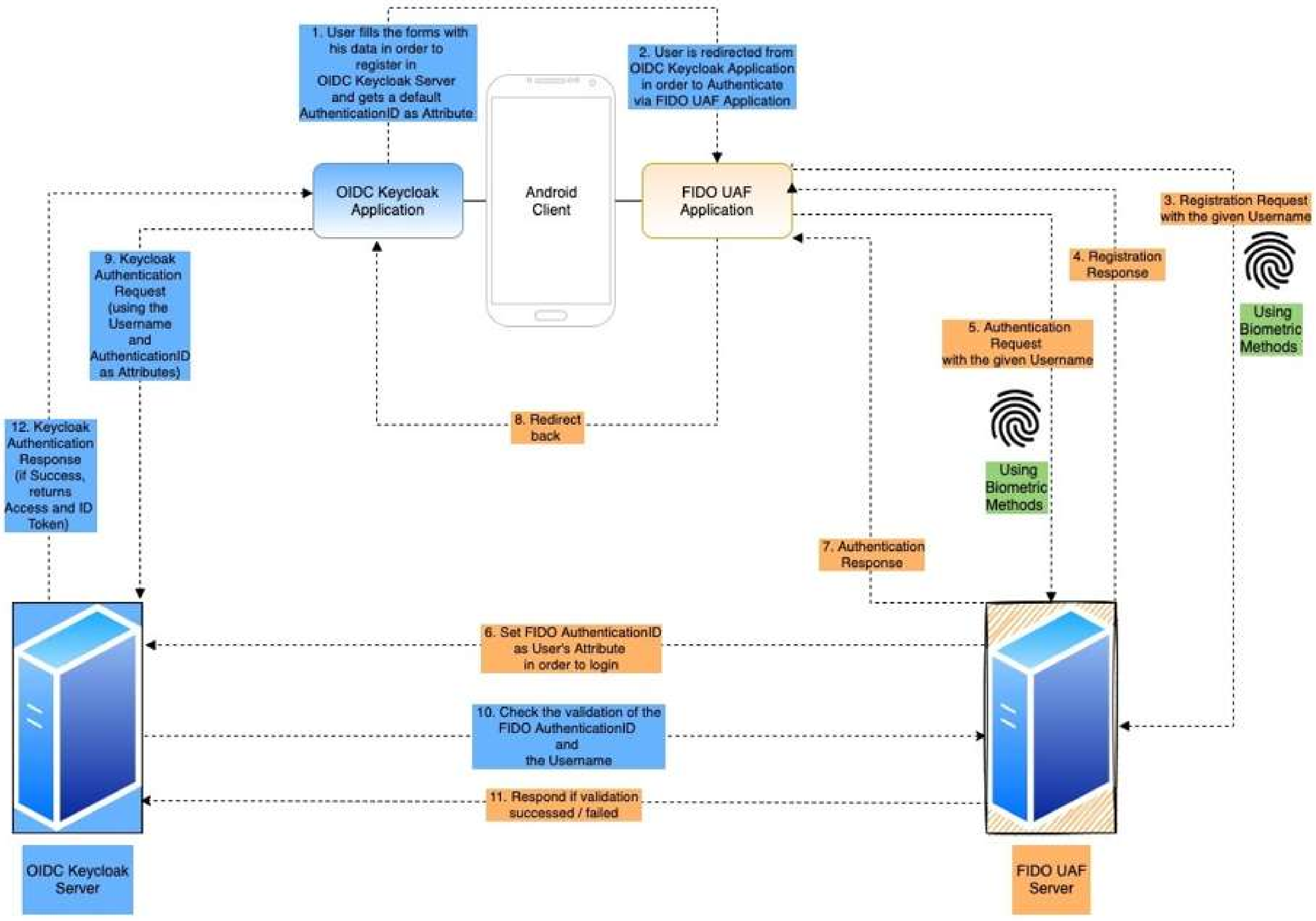
4.7 Analysis as a use case

The Scenario

It is a fact that universities and colleges are easy targets for cyber security attacks. Although they have a lot of resources at their disposal, they often have to spread these resources over a much wider and more complex system. Universities and colleges have an exponential variety of needs. Each academic department has specific networking and computer needs and in the case of many departments, it may include the research equipment, which can be used in obsolete operating systems. The percentage of academic departments under attack in 2021 is increasing exponentially, HAN University [19], University of Malaya [20] and BAR Ilan [21] are the most recent examples, with the latter even losing 20 terabytes of information from research lab documents, paper lists and personal information from staff and students. Some students are said to be planning to sue the university after their details were leaked and their online passwords were changed, locking them out of some systems. Such attacks are usually prevented by paying ransoms of millions as in the case of Bar Ilan where the amount reached 2.5 million and was an economic black hole for the country's education system. In addition, we would like to mention that according to the 2020 Data Breach Investigations Report (DBIR) published by Verizon every year over 80% of hacking-related breaches involve the use of lost or stolen credentials [22]. Considering the above and seeing that most services in universities work using the very old and dangerous username password login scheme, we will present a use case in which students will connect to a web service of the university using as a means of authentication and authorization the password-less system we made by combining FIDO UAF with Open ID Connect in the android environment.

The students will be given by the secretariat the University registration number which is unique for each student and will be used as a username. At the same time they will be asked to download the UPRC FIDO UAF client applications in order to be authenticated through FIDO and the SSL Keycloak android client for their login to the university service. As a web service, we have installed a docker which we present as the students-web. For our use case when the user logs in to the service his name displays as a welcome and there are tabs of options that display the access token, id token, and refresh token to make it clear that the OpenID Connect protocol is used.

Demonstration Example of FIDO UAF & Keycloak Integration using Custom Android Applications



1. User fills the forms with his data in order to register in OIDC Keycloak Server and gets a default AuthenticationID as Attribute
2. User is redirected from OIDC Keycloak Application in order to Authenticate via FIDO UAF Application
OIDC Keycloak Application
Android Client
FIDO UAF Application
3. Registration Request with the given Username
Using Biometric Methods
4. Registration Response
5. Authentication Request with the given Username
Using Biometric Methods
9. Keycloak Authentication Request (using the Username and AuthenticationID as Attributes)
12. Keycloak Authentication Response (if Success, returns Access and ID Token)
8. Redirect back
7. Authentication Response
6. Set FIDO AuthenticationID as User's Attribute in order to login
10. Check the validation of the FIDO AuthenticationID and the Username
11. Respond if validation successed / failed
OIDC Keycloak Server
FIDO UAF Server

Figure 37 FIDO UAF & Keycloak High Level Architecture

In the figure above, one can notice the high-level architecture of the FIDO UAF & Keycloak Integration for password-less authentication on Android phones. The Identity and Access Management component (Keycloak) is directly connected with the Password-less Authentication that FIDO UAF Protocol provides, in order to exchange information regarding the Registration, User Authentication and De-Registration processes.

In our case a new postgraduate student wants to connect to the university service which has stored the personal card of each student. The student, knowing his/her university registration number (Mte2101) and having installed the two client applications, will initially be asked to register in the SSL application. After launching the custom Keycloak Android Application, the user must fill in his/her personal details in the Keycloak Identity Provider and defining his/her university registration number as a username. This step can be skipped, if the admin console of the Keycloak has already access to the user's personal information.

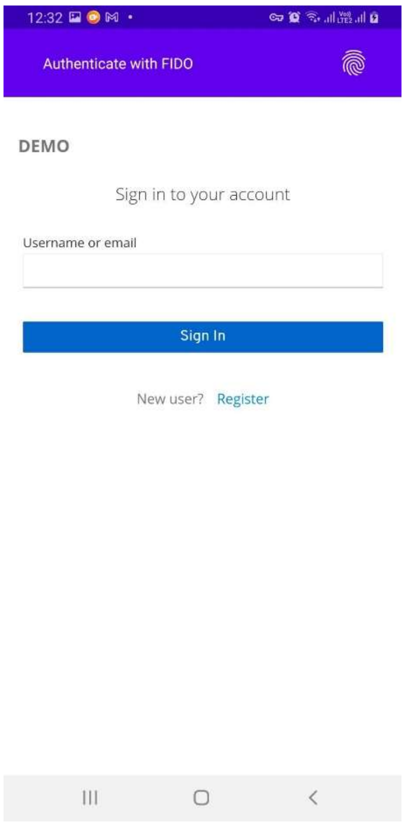


Figure 39 The student launch the Keycloak Android Application.

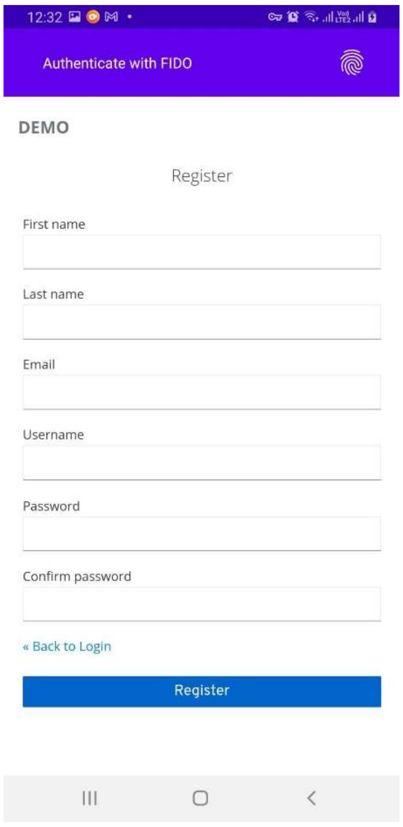
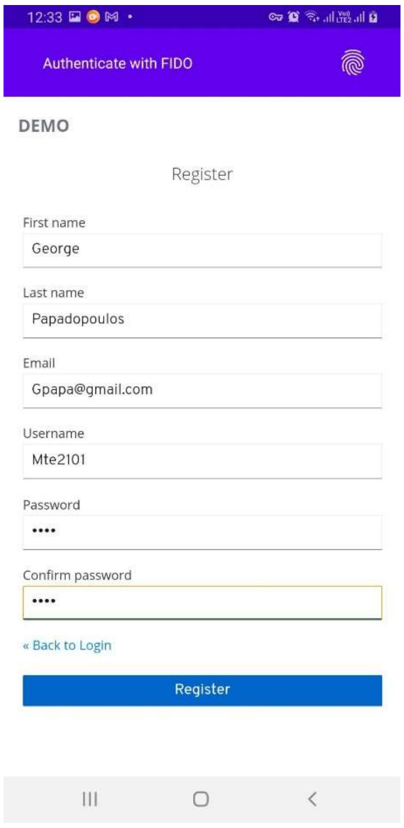


Figure 40 The Keycloak's Registration form.



12:33

Authenticate with FIDO

DEMO

Register

First name: George

Last name: Papadopoulos

Email: Gpapa@gmail.com

Username: Mte2101

Password: ••••

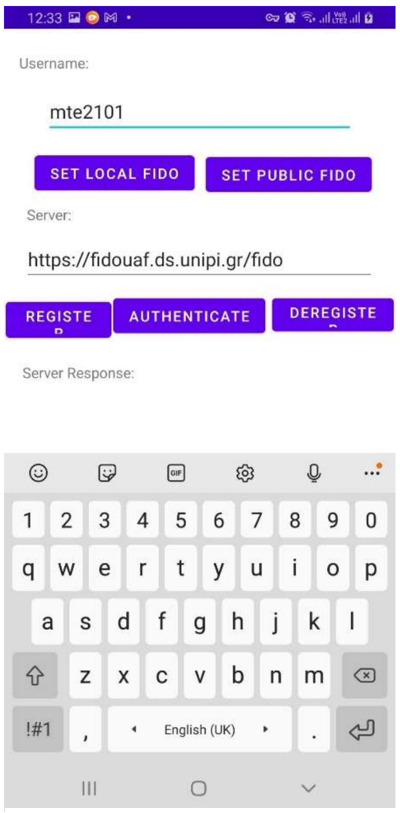
Confirm password: ••••

« Back to Login

Register

Figure 41 The student fill in his/her personal details in the keycloak Identity Provider and defining his/her university

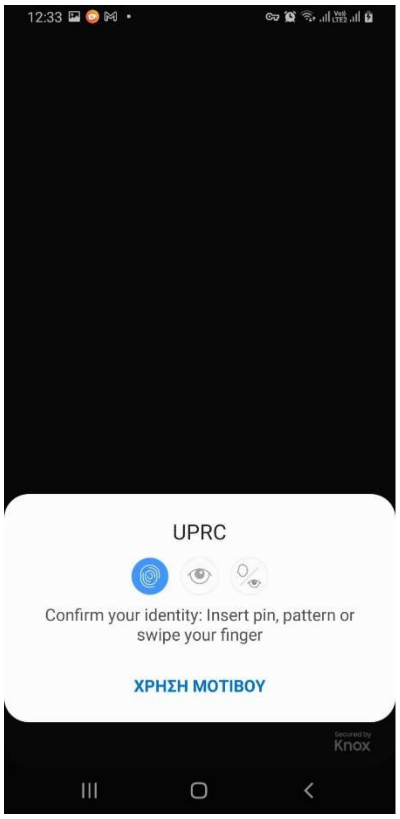
The Registration process will be completed when the student is redirected to the FIDO UAF Android Application, where he/she will be asked to fill the username, set the dedicated FIDO Server and tap on the Register button. Then he/she is prompted to confirm his/her identity using Biometrics (or even PIN /Pattern). As long as the Registration process is completed, the student is provided a special Attribute generated from FIDO and Keycloak communication, which is necessary for the User's Authentication process.


Username: mte2101

Server: https://fidouaf.ds.unipi.gr/fido

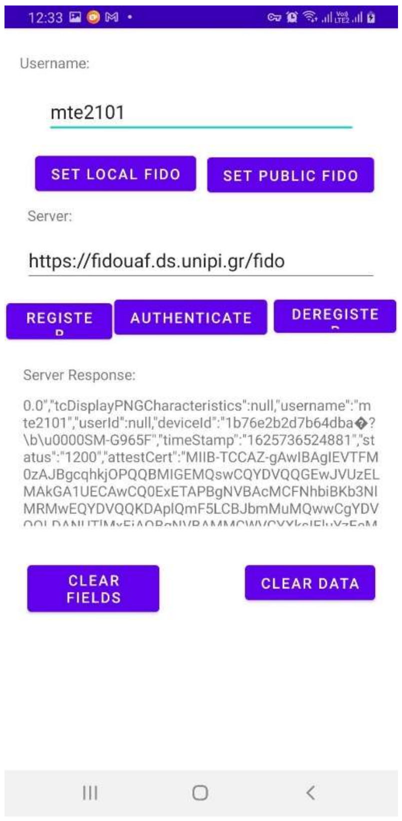
Server Response:

Figure 42 The student is redirected to the FIDO UAF Android Application.


UPRC

Confirm your identity: Insert pin, pattern or swipe your finger

Figure 43 The user confirms his/her identity using Biometrics in order to register through the FIDO UAF protocol.


Username: mte2101

Server: https://fidouaf.ds.unipi.gr/fido

Server Response:

```
0.0","tcDisplayPNGCharacteristics":null,"username":"m
te2101","userId":null,"deviceId":"1b76e2b2d7b64dba�?
\b\u0000SM-G965F","timeStamp":"1625736524881","st
atus":"1200","attestCert":"MIIB-TCCAZ-gAwIBAgIEVTFM
0zAJBgcqhkjOPQQBMIGEMQswCQYDVQQGEwJVUzEL
MAkGA1UECAwCQ0ExETAPBgNVBAcMCFNhbiBKb3NI
MRMwEQYDVQQKDAplQmF5LCBJbmMuMQwwCgYDV
```

Figure 44 The Registration Process is completed.

After the Registration process in order to successfully connect to the web service the student should tap on the Authenticate button. Then, he/she will be asked once again to confirm his/her identity using the same method as before.

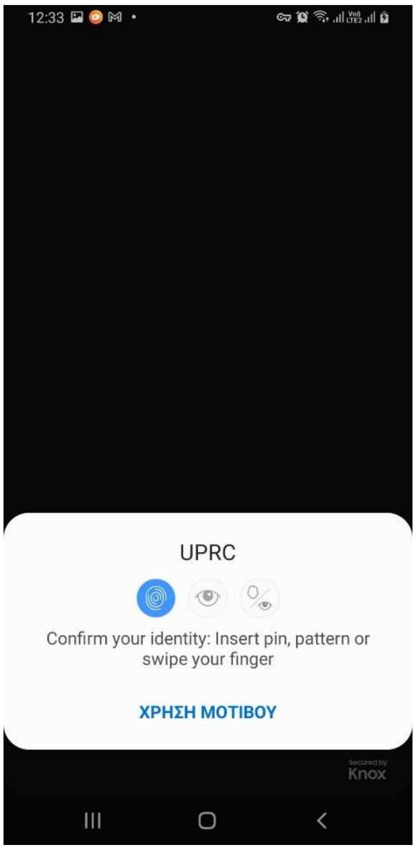
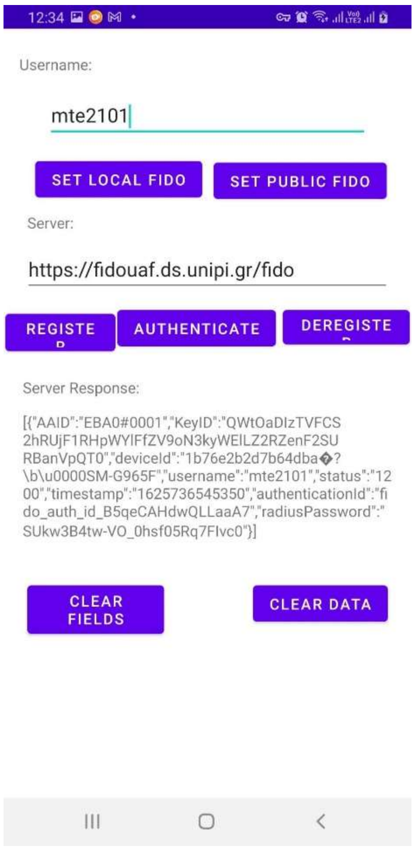

12:33
UPRC
Confirm your identity: Insert pin, pattern or swipe your finger
ΧΡΗΣΗ ΜΟΤΙΒΟΥ
Secured by Knox

Figure 45 The user confirms his/her identity using Biometrics in order to register through the FIDO UAF protocol.



Username:

mte2101

Server:

https://fidouaf.ds.unipi.gr/fido

Server Response:

```
[{"AAID":"EBA0#0001","KeyID":"QWtOaDIzTVFCS
2hRUjF1RHpWYlFfZV9oN3kyWElLZ2RZenF2SU
RBanVpQT0","deviceId":"1b76e2b2d7b64dba�?
\b\u0000SM-G965F","username":"mte2101","status":"12
00","timestamp":"1625736545350","authenticationId":"fi
do_auth_id_B5qeCAHdwQLLaaA7","radiusPassword":"
SUkw3B4tw-VO_0hsf05Rq7FIvc0"}]
```

Figure 46 The Authentication Process is completed.

At this point, the student is redirected back to the custom Keycloak Android Application and will be asked to fill his/her username in order to sign in. In between, Keycloak and FIDO UAF components validated the special Attribute needed for the User's Authentication.

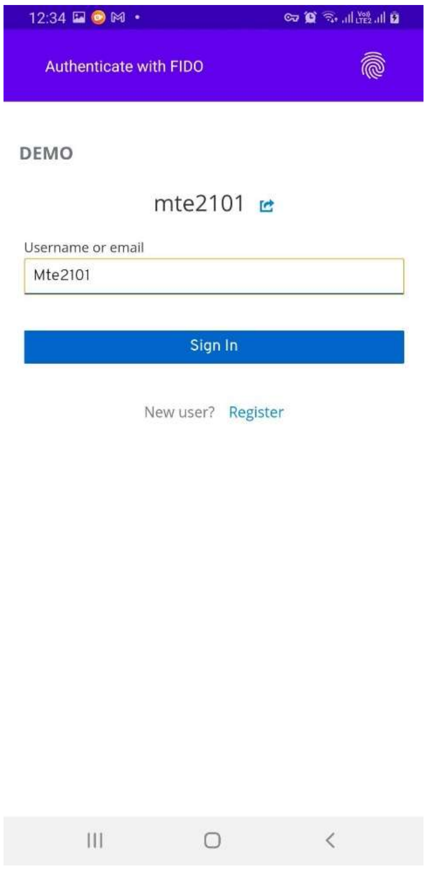


Figure 47 The Keycloak Android Application will ask the student to fill his/her username in order to sign in.

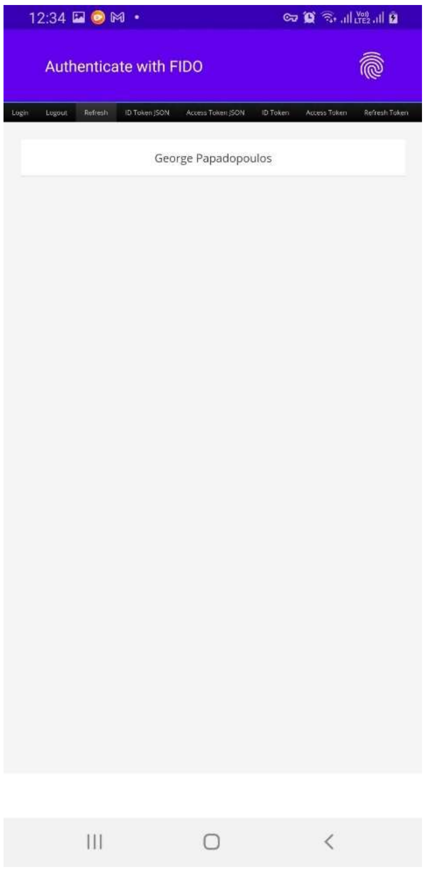


Figure 48 The Student have access to the web service.

The DeRegistration process is completed as long as the student launches the FIDO UAF Android Application, fills his/her username, sets the dedicated FIDO Server and taps on the Deregister button. At this point, the user will not have permission to sign in, unless he/she re-authenticate him/herself.

5 Security Considerations

OpenID Connect and FIDO UAF protocols, each separately, have many security valves, which is why they are so widespread at the moment. In this chapter we will present for each of these two protocols the countermeasures they have against known attacks and the security parameters that are taken into account, thus wanting to demonstrate why their union can create a complete authentication ecosystem of authentication.

5.1 OpenID Connect

SSO has been debated much in the last years, leading to the emergence of many solutions aimed at creating a user-centric environment. Among these solutions the most promising was the combination of OpenId 2.0 and OAuth 2.0, and resulted in its implementation and study. The wide usage of this solution was the spark to understand the security necessities of SSO mechanisms based on these protocols, resulting in the creation of OpenId Connect which addresses many of the vulnerabilities identified in OpenId 2.0 and OAuth 2.0. At this point, we will present a set of remedies that OpenId Connect Core uses against attacks, in order to authenticate and authorize securely a user. First of all, the factors that involved in the protocol communicate through the use of TLS with cipher suite so as to provide confidentiality and integrity protection and keep secure the protocol against information disclosure and tampering. In cases where there is a risk of Token Modification, even though the tokens are sent through a secure channel (TLS), OIDC requires them to be digitally signed by the OpenId Provider, so that the RP can be sure of who issued each token [23]. Each Server is used must be authenticated through the usage of Signed or Encrypted JWTs with an appropriate key and cipher for the purpose of dealing with Server Masquerading attacks. The handling of the request or request_uri parameters, whose content is an Encrypted JWT, prevents Request Disclosure attacks [24]. In OpenId Connect the requests sent by the client, as well as, the server responses, are digitally signed to address the cases of Request and Response Repudiation. In addition, through the use of a signed request, the end-user is sure that the desired request parameters have not been altered and they have been delivered correctly to the OP. OIDC requires, the requests sent to the OP to be encrypted, in order to prevent such potentially sensitive information from being revealed. Overall, since all JWE encryption algorithms provide integrity protection, there is no need to separately sign the encrypted content.

5.2 FIDO UAF

The main goal of the FIDO UAF protocol is to be equipped with security properties so as to provide strong authentication. Strong authentication is defined as the authentication of a user and/or a device by a Relying party using possible cryptographic schemes. The main attacks from which a user is in danger are DoS (Denial of Service) attacks, Forgery Resistance (Impersonation Attacks), Parallel Session Resistance, Forwarding Resistance, and Replay attacks. The protocol in order to indicate Dos resistance and prevent attackers from entering invalid registration information for a legitimate user to affect its authentication, uses the signature counter so that the authenticator and the RP are synchronized and they provide resilience against cloned authenticators. For Forgery Resistance, Parallel Session Resistance, Forwarding Resistance and Replay attacks, the FIDO UAF protocol imposes a state of integrity regarding message exchange which it is achieved either through randomly generated challenge or mutual agreement after the authentication, between RP and authenticator for the values of AAID, KeyID, Uname and AppID. The mutual agreement between RP and the authenticator after the authentication for the mentioned values, also prevents the situations in which information is leaked by the authenticator or by a verifier. In addition to user authentication, the fido UAF protocol is also used as a means for secure transactions through the Transaction option [12]. During the Transaction Confirmation process, the What You See Is What You Sign (WYSIWYS) Concept is used, which makes necessary the usage of the protocol in the marketplace since it addresses the problem of the Transaction Non-Repudiation [25]. As provision for achieving the goal of confidentiality, FIDO UAF uses two pairs of asymmetric encryption keys, the attestation, and the authentication keys. Moreover, it uses KHAccesstoken (ak) as an access control mechanism that protects the UAF credentials of the authenticator from unauthorized use. Comparing FIDO UAF with FIDO2 we see that by linking the keys to the Relying parties which is achieved through the appID and ChannelBindingInformation values, the UAF offers protection against phishing attacks. Regarding the communication between client and server, the UAF specification requires this to be done through a secure channel, and for this purpose, the use of the TLS protocol is essential. The rest of the software entities communicate through inter-process communication methods or hardware and software communication. Despite the above, as well as, all the options offered according to the security policy criteria, there is intense interest in research work that reveals theoretical (Xenakis et all) [26]and applicable attacks. The most recent attack is [10], in which researchers observed a lack of effective authentication between the entities in the implementations of the UAF protocol. The result of this vulnerability was the implementation of the Authenticator Rebinding attack in which the attacker rebinds the victim's identity to a misused authenticator and not to the victim's authenticator thus by passing the UAF protocol authentication mechanisms.

6 Related Work

There is a big importance in creating federated identity solutions that can provide increased levels of security. The dilemma for developers and IT professionals lies in choosing the standard that should be deployed to keep federated identity safe. This option is not straightforward. Most stakeholders find it difficult to decide between OAuth 2.0, OpenID, and Security Assertion Markup Language (SAML), each of which brings structure to the federation process. As aforementioned OAuth 2.0 is a framework that controls authorization to a protected Resource, in contrast, OpenID and SAML are both industry standards for federal authentication. Using either OpenID or SAML independently enterprises can achieve user authentication and deploy single-sign-on, but we must mention that they have different strengths and weaknesses. OpenID Connect as described in the previous chapters is built on the OAuth 2.0 protocol, focuses specifically on user authentication, and is widely used to enable user logins on consumer websites and mobile apps. SAML is independent of OAuth, relying on an exchange of messages to authenticate in XML formats instead of JWT. It is mainly used by companies so that their employees can sign in to multiple applications using a single login. We could emphasize that a protocol like FIDO can complete any needs of a federated identity management system, as it increases security and usefulness by removing the pattern of username password. Below we will present other related research studies that we found. It is worthwhile to point out that all solutions, like ours, are based on Fido Alliance certified [14] eBay UAF open-source implementation.

In [27]for the needs of the recred European project, a privacy-preserving architecture for device-centric and attribute-based authentication is proposed on the integration of FIDO UAF with the access management platform of OpenAM which implements OpenID Connect together with other technologies. This solution uses FIDO UAF as an authentication module in OpenAM. Although we noticed in the GitHub [28] of the project, that OpenAM is no longer an Open-source solution, so we do not know if this project is a functional solution anymore.

In [29] a solution is proposed which focuses on security, privacy, and usability, using the FIDO UAF, TEE, and SAML standards. The paper describes a mobile electronic identity (eID) Management strategy which can be aligned with Brazil's electronic government (e-GOV) programs that try to promote government transparency and improve interaction with its citizens. A citizen during the eID registration phase performs the FIDO UAF registration process along with his / her identity card to confirm his / her identity. From the asymmetric key pair that will be produced by FIDO, the public key is sent to the government system where it will be associated with the government attributes database, while the private key will be stored in the user's device TEE. SAML is used to exchange authentication and authorization data between the parties. So every time a user wants to use a government service he will be able to do it remotely, authenticating himself through FIDO, in fact, a using scenario is presented in which citizens by identifying himself/herself by using his/her mobile eID, can vote in elections remotely from their mobile.

In [30] Laborde et all, present a user-centric and decentralized digital identity system based on FIDO UAF and the data model proposed by W3C Verifiable Credentials WG, which allows anyone to easily benefit from an enriched digital identity made of multi-propose and multi-origin attributes. The Verifiable Credential is defined as a set of one or more tamper-resistant claims made by an issuer, where each claim assets as a set of properties about a subject. This system also makes this digital identity highly trustworthy both for the user and the service provider who requires highly certified information about the user being enrolled to and/or authenticated on its services. In the paper, the contributors apply the implementation of the specific project in an integrated banking system with the aim of the easiest and safest service for a user.

In [31] it is used the aforementioned project which combines FIDO UAF with W3C verifiable credentials to build a pilot application for UK NHS. In this implementation, patients were able to use a mobile phone with a fingerprint reader to access restricted NHS sites in order to make, cancel appointments and order repeat prescription drugs. In the paper, it is emphasized that after a trial with 10 UK NHS patients using the application, the conclusion was, the system with biometric authentication methods is easier to use and preferable than the classic username and password.

7 Conclusion

Federated Identity management refers to standards-based approaches for handling authentication, single sign-on (SSO), role-based access control and session management. Many standards and protocols have been specified in the last few years following this kind of scheme. OpenID is an authentication protocol providing a way to prove that an end user controls a specific identifier. The FIDO protocol suite aims at allowing users to log in to remote services with a local trusted authenticator. This thesis contributed to developing, a single sign-on system combining OpenID Connect and FIDO UAF protocols, in which users in order to enter a web service must perform biometric authentication on their mobile phone. First, we analysed the involved technologies used, then we described in detail the components that surround the tool and how they communicate with each other. After presenting the flows of registration, authentication, deregistration, we were able to compile a use case in which a university can use this implementation in order to provide its students with secure access to its web services. After research, we mentioned the security considerations for the protocols we use in order to have a complete understanding of the security that this implementation can offer. Finally, we were able to find similar implementations, but there is no open-source implementation for OpenID Connect and FIDO UAF protocols that is applicable, except ours.

References

[1] "Identity Management Institute," [Online]. Available: https://identitymanagementinstitute.org/7-hacking-password-attack-methods/. [Accessed 18 December 2021].

[2] "OAuth Grant Types," Oauth2, [Online]. Available: https://oauth.net/2/grant-types/. [Accessed 26 November 2021].

[3] D. Hardt, "The OAuth 2.0 Authorization Framework," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6749#page-18. [Accessed 26 november 2021].

[4] N. Sakimura, J. Bradley and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients," September 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7636. [Accessed 26 November 2021].

[5] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," October 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6750. [Accessed 26 November 2021].

[6] J. Richer, "OAuth 2.0 Token Introspection," October 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7662.html. [Accessed 26 November 2021].

[7] T. Lodderstedt and M. Scurtescu, "OAuth 2.0 Token Revocation," August 2013. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7009. [Accessed 26 November 2021].

[8] N. Sakimura, J. Bradley, M. Jones, B. Medeiros and C. Mortimore, "OpenID Connect Core 1.0," OpenID Foundation, 2014 . [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html. [Accessed 30 November 2021].

[9] S. Machani, R. Philpott, S. Srinivas, J. Kemp and J. Hodges, "FIDO UAF Architectural Overview," FIDO Alliance, 28 November 2017.

[10] L. Hui, P. Xuesong, W. Xinluo, F. Haonan and S. Chengjie, "Authenticator Rebinding Attack of the UAF Protocol on Mobile Devices," *Wireless Communications and Mobile Computing ,* vol. 2020, p. 14, September 2020.

[11] R. Lindemann and J. Kemp, "FIDO UAF Authenticator-Specific Module API," 28 November 2017. [Online]. Available: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-asm-api-v1.2-rd-20171128.html.

[12] F. Haonan, L. Hui, P. Xuesong and Z. Ziming, "A Formal Analysis of the FIDO UAF Protocol," 2021.

[13] A. Angelogianni, "FIDO & TEE," 2018.

[14] F. ALLIANCE, "FIDO Certified," [Online]. Available: https://fidoalliance.org/certification/fido-certified-products/. [Accessed 5 December 2021].

[15] M. giannis, "FIDO UAF intergration with OpenID," 24 August 2021. [Online]. Available: https://github.com/giannismakro/FIDO_UAF_Intergration_with_OpenID.

[16] R. Lindemann, E. Tiffany, D. Baghdasaryan, D. Balfanz, B. Hill, J. Hodges and K. Yang, "FIDO UAF Protocol Specification," Fido Alliance, 20 February 2018. [Online]. Available: https://fidoalliance.org/specs/fido-uaf-v1.2-id-20180220/fido-uaf-protocol-v1.2-id-20180220.html. [Accessed 8 December 2021].

[17] G. Cadoudal, "How to create a Keycloak authenticator as a microservice?," 28 June 2020. [Online]. Available: https://medium.com/application-security/how-to-create-a-keycloak-authenticator-as-a-microservice-ad332e287b58. [Accessed 7 December 2021].

[18] N. Williams, " On the Use of Channel Bindings to Secure Channels," November 2007. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5056.html. [Accessed 10 December 2021].

[19] "HAN UNIVERSITY," 3 September 2021. [Online]. Available: https://hanuniversity.com/en/news/2021/09/data-leak-at-han/. [Accessed 20 December 2021].

[20] Y. Y. Mae, "Mashable SE Asia," 19 October 2019. [Online]. Available: https://sea.mashable.com/article/6978/nearly-45000-university-malaya-login-ids-and-passwords-were-leaked-by-an-anonymous-hacker. [Accessed 20 December 2021].

[21] "The times of ISRAEL," 9 September 2021. [Online]. Available: https://www.timesofisrael.com/liveblog_entry/mass-data-leak-after-bar-ilan-university-refuses-to-pay-hacker-2-5m/. [Accessed 12 December 2021].

[22] "Enzoic," [Online]. Available: https://www.enzoic.com/credential-vulnerabilities/. [Accessed 20 December 2021].

[23] D. Fett, R. Kusters and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines," in *IEEE 30th Computer Security Foundations Symposium (CSF)*, Santa Barbara, CA, USA, 2017.

[24] J. Navas and M. Beltran, "Understanding and mitigating OpenID Connect threats," *Computers & Security,* no. Elsevier, 2019.

[25] A. Angelogianni, I. Politis and C. Xenakis, "How many FIDO protocols are needed? Surveying the design, security and market perspectives," Piraeus, 2021.

[26] C. Panos, S. Malliaros, C. Ntantogian, A. Panou and C. Xenakis, "A Security Evaluation of FIDO's UAF Protocol in Mobile and Embedded Devices," 2017.

[27] K. Papadamou, S. Zanettou, B. Chifor, S. teican, G. Gugulea, A. Caponi, A. Recupero, C. Pisa, G. Bianchi, S. Gevers, C. Xenakis and M. Sirivianos, "Killing

the Password and Preserving Privacy with Device-Centric and Attribute-based Authentication".

[28] S. Zanettou, "ReCRED FIDO UAF OIDC," [Online]. Available: https://github.com/zsavvas/ReCRED_FIDO_UAF_OIDC. [Accessed 15 December 2021].

[29] G. Verzeletti, E. Ribeiro de Mello and M. Silva Wangham, "A National Mobile Identity Management Strategy for Electronic Government Services," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, New York, NY, USA, 2018.

[30] R. ,. LABORDE, S. Wazan, F. Barrere, A. Benzekri, D. Chadwick and R. Venant, "A User-Centric Identity Management Framework based on the W3C Verifiable Credentials and the FIDO Universal Authentication Framework," in *IEEE 17th Annual Consumer Communications &networking Conference* , 2020.

[31] D. Chadwick, R. Laborde, A. Oglaza, R. Venant, S. Wazan and M. Nijjar, "Improved Identity Management with Verifiable Credentials and FIDO," *IEEE Communications Standards Magazine,* p. 7, 2019.