OLYMPUS PROJECT

OBLIVIOUS IDENTITY MANAGEMENT FOR PRIVATE USER-FRIENDLY SERVICES

D5.4 OLYMPUS Framework and Tools

PROJECT NUMBER	PROJECT ACRONYM
786725	OLYMPUS
CONTACT	**WEBSITE**
contact@olympus-project.eu	http://www.olympus-project.eu/

Due date of deliverable: 30-07-2021
Actual submission date: 30-07-2021

Dissemination Level	
PU = Public, fully open, e.g. web	✓
CO = Confidential, restricted under conditions set out in Model Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC.	
Int = Internal Working Document	

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

OLYMPUS

REVISION HISTORY

The following table describes the main changes done in the document created

Revision	Date	Description	Author (Organization)
V0.1	18/06/2021	Initial draft	Michael Stausholm
V0.2	23/06/2021	Complete CF content	Noelia Martínez
V0.3	28/06/2021	W3C, verifier-side and use-case-3 input	Jesús García, Rafael Torres
V0.4	28/06/2021	mDL content	Georgia Sourla
V0.5	05/07/2021	Attribute hiding towards IdPs	Tore Frederiksen
V0.6	05/07/2021	MSO validation content	Nuno Ponte, Nuno Martins, Fátima Cristiana Conceição
V0.7	07/07/2021	Review comments	Georgia Sourla
V0.8	08/07/2021	Review comments addressed	Noelia Martínez
V0.9	16/07/2021	Final review changes	Michael Stausholm

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

INDEX

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

OLYMPUS

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

Table of figures

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

OBLIVIOUS IDENTITY MANAGEMENT FOR PRIVATE AND USER-FRIENDLY SERVICES

ABSTRACT

Deliverable D5.4 gives a technical overview of the OLYMPUS framework. This overview may be used by architects, and to some degree developers, wishing to integrate the OLYMPUS framework into some application. This deliverable is an updated version of the previous deliverable D5.2, documenting various changes and improvements of the framework. The deliverable gives a description of how the framework can be integrated with other related technologies, how to integrate as a server and client application and gives a description of how the framework is used by the two use cases described in deliverable D6.1 and a third use case demonstrating functionality not showcased by the other use cases. This deliverable furthermore discusses a theoretical approach as to how the OLYMPUS framework can be extended to offer privacy of the user's attributes towards the partial IdPs.

KEYWORDS

API documentation, Integration, Implementation, OAuth, SAML, IdP

AUTHORS (ORGANIZATION)

Michael Stausholm (ALX), Tore Frederiksen (ALX), Georgia Sourla (SCY), Noelia Martínez (LOG), Rafael Torres (UMU), Jesús Garcia (UMU), Nuno Ponte (MUL), Nuno Martins (MUL) and Fátima Cristiana Conceição (MUL).

DISCLAIMER

1. EXECUTIVE SUMMARY

In order to ease integration of the OLYMPUS framework for external developers, this deliverable gives a technical description of how the OLYMPUS framework may be integrated with applications. The framework is still (and will continue to) undergoing minor improvements; hence this deliverable is not guaranteed to form a complete documentation of the framework, but rather gives a description of the current state and how the improvements are planned to be implemented. Certain parts will give technical descriptions of implemented code, where other parts will give descriptions of how other components are planned to be designed and implemented.

In addition to this deliverable, the deliverable D3.3 may be referred for a high-level description of the architecture and D4.1 may be referred for details regarding the cryptographic protocols used. For concrete details regarding the implemented code, the JavaDoc documentation of the framework may be referenced.

The deliverable is structured as follows:

Section 2 gives a high-level description of how the framework works.

Section 3 describes how the framework may be integrated with applications and how it can be used to implement identity providers in established authentication systems such as OAuth or SAML.

Section 4 focuses on the IdP server part of the framework. The section describes how the framework may be deployed, how to interact with existing persistent storage and how HSM support can be achieved.

Section 5 focuses on the client part of the framework. The section describes how the API may be used by client applications, such as smartphone apps, and discusses how the framework may be deployed in a trusted environment to simulate a traditional IdP.

Section 6 describes how the framework can be used in relation to verification of issued tokens.

Section 7 describes how the two use cases have integrated with the framework and also outlines a third use case for demonstrating features of the framework not utilized by the two main use cases.

Section 8 describes a theoretical approach to how the framework may be expanded in the future. Specifically, this involves the description of how to hide the users' attributes from the partial IdPs.

Section 9 concludes the deliverable.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

2. HIGHLEVEL DESCRIPTION

The OLYMPUS framework is implemented as a few Java projects, built using Maven, i.e., the command *'mvn install'* will download the required dependencies, compile the code, run a test suite and produce a Java Jar file. The project can be split into three segments: A client Java library segment, a server component segment and an optional verifier component segment. The server segment may be used as a Java library for a custom server application or deployed directly as a self-contained REST-based server.

The client library exposes the basic methods (such as *createUser(username, password)* and *authenticate(username, password, policy)*) for client applications. These calls are then translated into appropriate protocol messages and transmitted to the vIdP servers, and a client understandable output is returned to the client application. Hence the business logic of the client can be completely oblivious regarding *how* the cryptographic protocols are executed in order to authenticate the user. Note that the different protocols may yield different tokens as output, hence the client cannot be completely oblivious as to *what* is produced by the protocols. The OLYMPUS framework supports three main protocols, where two of these are intended for practical use and one is only included for benchmarking purposes:

1. A traditional password verification scheme, where the password is sent to a single server and compared to a stored (salted and hashed) password. If the verification succeeds, a JWT token is generated and returned to the client.

2. A distributed scheme based on PESTO, where a protocol is executed between the client and minimum 2 distinct servers. It consists of two conceptual steps: Distributed Password Verification (DPV) and Distributed Token Generation (DTG). In PESTO, if the DPV passes, a distributed RSA signature on user attributes is the result of the DTG. This allows the client to combine the result into a JWT token consisting of its attributes and a combined RSA signature.

3. A scheme based on distributed pABC, where the DPV from PESTO is used to verify the identity of a user towards a minimum of 2 distinct servers. However, instead of constructing a distributed RSA signature as a result, the servers construct a short-lived distributed pABC for the client. The client can then combine these to construct a regular pABC which can be used to issue tokens based on its attributes.

The server component can either be deployed as a self-contained webserver, as a servlet or be used as a library for a custom server application. It contains a set of protocol endpoints, matching the protocols supported by the client, i.e. traditional password verification, PESTO and pABC credentials.

In addition to executing the cryptographic protocol, the server component has two additional responsibilities, independent of the protocols:

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

1. Manage stored user attributes. This can be handled purely using the supported authentication protocols, but may also be supplemented by interacting with some existing database solution or other resources external to the vIdP instance.
2. Validating user claims, i.e., identity proofing. In order to attach attributes to a user account, the correctness of these attributes must be validated. Since identity proofs may have many forms, this is done in a modular fashion, allowing different vIdPs to support different types of identity proofs such as X.509 certificates, JWT tokens or custom eID tokens.

These two responsibilities are encapsulated into modules, allowing for easier integration with the concrete application. In the case of managing user attributes, this minimally involves integration with some database for persistence, but may also involve interaction with some external data source.

Regarding the validation of user claims, this will vary depending on the concrete scenario and a wide range of validation methods should be supported. This functionality will be implemented using an *IdProofer* interface. This allows the rest of the application to be oblivious regarding the type of identity proof.

The Java projects are built using Maven, hence running the command '*mvn install*' will build the project and run the included unit tests.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

OLYMPUS

3. INTEGRATION WITH EXISTING TECHNOLOGIES

As there are multiple existing technologies related to authentication, it is relevant to note how these functions and how the OLYMPUS solution can be used in connection with these. In order to ensure a broad adaptation of the work done in OLYMPUS, the effort required by the Identity and Service providers to support OLYMPUS should be minimized. If the framework can fit into existing ecosystems, it is likely that the service providers will not have to make any changes to their implementations at all, hence the technology will only have to be adapted by the IdPs and to some extent by the end users. Note that there is no guarantee that all OLYMPUS supported cryptographic protocols can be used in these settings. For instance, the tokens/proofs produced as part of the pABC functionality, cannot be verified using standard COTS products and can therefore not be used in e.g., OIDC.

3.1. OAUTH

One of the most common technologies for web-based authorization is OAuth 2.0 [4]. The objective of OAuth 2.0 is to allow a resource owner (the user) to grant some service provider access to a protected resource without exposing the user's credentials (primarily passwords) to the service provider. The abstract flow of OAuth 2.0 can be found in Figure 1. OAuth 2.0 is designed to work specifically with the Hypertext Transfer Protocol (HTTP) and works with a number of different *grant type* flows. Of these *grant types*, the *implicit* and *authorization code* flows are of primary concern in relation to OLYMPUS. In both cases, the flow can be summarized as follows:

1. A user wishes to gain access to some service (e.g., Spotify). This service (Client) requires the user to prove its identity (which is done by having the service provider contact a restricted resource), and redirects the user to an IdP service (able to authorize access to the resource). This redirection URL contains (amongst others) some id for the service provider and a call-back URL, to where the user should be sent after the authorization is achieved and what operation to authorize.

2. The client follows the redirect to the IdP, authenticates and authorises the operation requested. The IdP produces some token (depending on the *grant* type). This token is returned to the service provider.

3. In case of the *authorization* code flow, the service provider contacts the IdP and authenticates using the provided token.

4. The IdP generates a proper access token which is returned to the service provider.

5. The service provider can now utilize the access token to gain access to the restricted resource.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

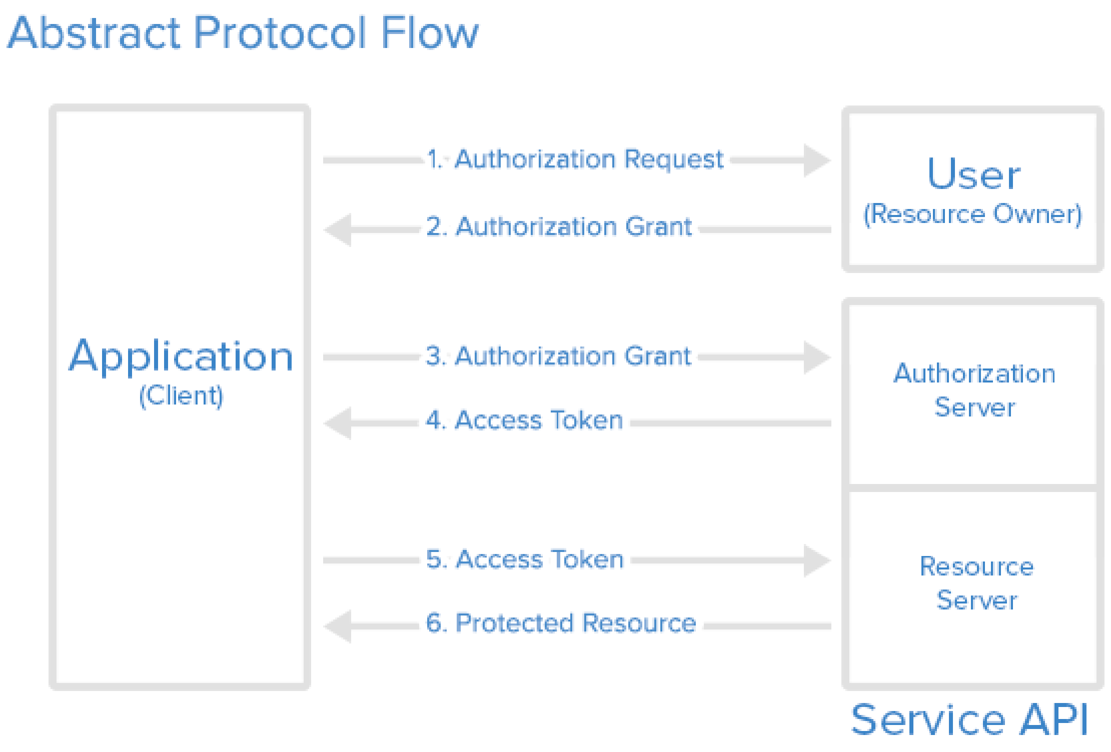


Figure 1. The abstract OAuth flow

For the OLYMPUS framework to be compatible with the above sketched workflow, the main challenge lies with redirecting to the IdP. As the OLYMPUS vIdP consists of multiple individual servers, the redirection must include all of these. Furthermore, the client must perform a cryptographic protocol, hence simply having a standard browser follow the URL will not suffice. On the positive side, the OLYMPUS vIdP is flexible enough to produce a wide range of tokens, hence once the redirection problem is solved, the complete flow can be implemented using an OLYMPUS vIdP. Note that in order to support the *authorization code* grant flow, the issued token is only temporary and the Service Provider must contact the vIdP and exchange the temporary token to a proper access token. While possible in theory, this will require the Service Provider to combine the distributed signatures to a combined access token. As this requires additional functionality from the Service Provider application, it will likely hinder adaptation. Furthermore, we note that the *authorization code* flow will also make it more challenging to avoid traceability from a user to a service provider by an IdP. For these reasons, the *authorization code* flow is not supported by the OLYMPUS framework, however *implicit* flow is.

The problem of multiple servers representing the OLYMPUS vIdP is solved by creating an OLYMPUS client that runs locally on the client machine, depicted in Figure 2. This client application exposes a local OAuth IdP REST interface, i.e., provides functionality similar to that of a traditional IdP. For this to work, the service provider generates a redirect URL to the client's localhost. When the

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

client browser follows the redirect, it is taken to the local hosted OLYMPUS application, which in turn communicates with the vIdP. In this way, the OLYMPUS based protocols would be oblivious for the user. This solution is generic enough to be suited for other schemes, e.g., SAML. However, there are some practical issues:

1. X.509 certificates are extensively used to validate both HTTPS connectivity and issued tokens.
2. Custom software is required to be installed on the client.

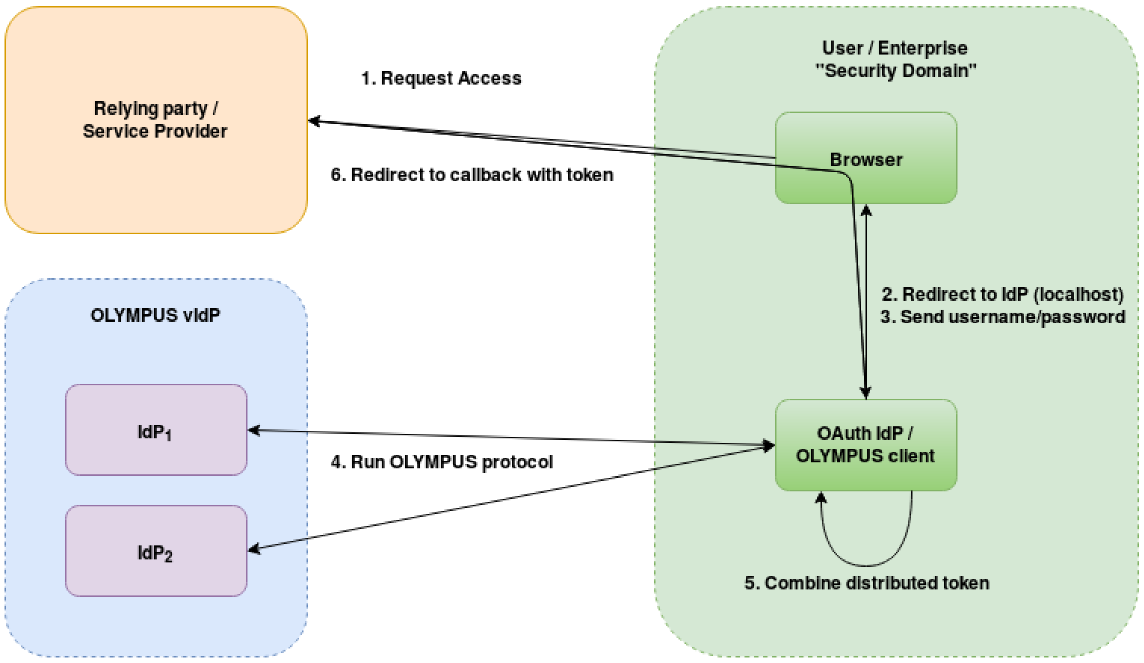

Relying party / Service Provider
1. Request Access
6. Redirect to callback with token
User / Enterprise "Security Domain"
Browser
2. Redirect to IdP (localhost)
3. Send username/password
OLYMPUS vIdP
IdP1
IdP2
4. Run OLYMPUS protocol
OAuth IdP / OLYMPUS client
5. Combine distributed token

Figure 2. The OLYMPUS client acting as a proxy OAuth IdP

When verifying the signature of a token, the SP will ensure that the signing certificate matches the URL of the IdP. Since this URL is *localhost*, a valid X.509 certificate for *localhost* must be obtained. As this is not possible for CA's to issue certificates for *localhost*, we are limited to self-signed certificates (or potentially some creative use of DNS/hostnames), which do require more effort on the SP side. Furthermore, this also limits the SP to only support 1 OLYMPUS vIdP, as only a single certificate can be used.

The other downside of the sketched solution is that the client machine must have software installed. This requirement could, in theory, be removed in various ways by having the software run on some trusted server or hosting a JavaScript application and having the service provider link to this. Both of these solutions do, however, introduce a single-point-of-trust, as either the user credentials (password) must leave the client machine or the user must trust the JavaScript delivered by the server. Hence, while this solution is not entirely as secure as the previous solution, it could be usable in an enterprise setting,

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

where the OLYMPUS client/OAuth IdP could be hosted within some trusted perimeter.

Note that the OAuth specification does not specify the format of the access tokens or the validation process. This is completely decided by the involved parties. Hence any of the OLYMPUS supported protocols can be used for generic OAuth.

3.2. OpenID Connect and JWT

OpenID Connect (OIDC) [5] is built on top of OAuth 2.0 and is used for user authentication rather than authorization. Rather than having the IdP authorize the user's access to some resource, OIDC uses the IdP to authenticate the user and lets the Service Provider handle the authorization. While both OAuth *implicit grant* and *authorization code* flows can be used in connection with OIDC, only *implicit grant* is supported by the OLYMPUS framework.

In addition to building on OAuth, OIDC also relies on JSON Web Tokens (JWT) [7] for representing the identity of the user. JWT is a lightweight encoding of a set of claims, which supports both signatures and encryption. A JWT token consists of a header describing the encryption or signature algorithms used, a JSON data payload and a signature. This is Base64 URL encoded to be easily sent using HTTP(s).

OIDC follows the basic flows of OAuth, but specifies how the access token sent from the IdP should be formatted and what it should contain, i.e., rather than being some arbitrary unique value, it must now be a JWT token containing a number of specific fields. The following JSON is a sample of a JWT token issued by "vidp.olympus.eu", to the service provider "service-provider-1" attesting the username attribute "alice":

```
{
  "sub":"alice",
  "iss":"https://vidp.olympus.eu",
  "aud":"service-provider-1",
  "auth_time":1311280969,
  "iat":1311280970,
  "exp":1311281970
}
```

In addition to the "sub" (username) attribute, OIDC specifies a series of standard user attributes or claims such as "email", "name", "given_name", "family_name", "birthdate", etc.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

Since the OIDC builds on OAuth, the challenges (and solutions) for the OLYMPUS framework are identical to those of OAuth. Since OLYMPUS (particularly the PESTO protocol) provides a generic signature scheme, it is easy to provide distributed RSA signatures on JWT tokens and recombining these. In fact, JWT is the default output of the PESTO protocol implementation in OLYMPUS.

3.3. SAML

In addition to OAuth, the other major authentication framework is SAML (Security Assertion Markup Language) [6]. Similar to OAuth and OIDC, it is a framework for authentication and authorisation between an identity provider and a service provider. It is based on XML and used (amongst others) for Web-based applications, where it offers Single-Sign On (SSO) functionality, i.e., using a single login, a user may access a set of web applications.

The SAML framework is large and can accommodate a number of different scenarios. A common scenario is the "Web Browser SSO Profile", where a user, wishes to access a resource located at a Service Provider. The flow is roughly as follows:

1. A user agent (e.g., user browser) requests a resource from the service provider.
2. The service provider generates a *SAMLRequest* and redirects the user agent to the SSO service at the IdP.
3. The user agent follows the redirect and provides the *SAMLRequest* to the IdP.
4. The user now authenticates with the IdP.
5. The IdP validates the *SAMLRequest* and builds and signs a *SAMLResponse*. The *SAMLResponse* is embedded in an XHTML form which is presented to the user agent.
6. The user agent processes the XHTML, then sends (using HTTP POST) the *SAMLResponse* to the Service Provider.
7. The Service provider validates the *SAMLResponse* and if successful, redirects the user agent to the resource requested in step 1.

The abstract flow is identical to the OAuth *implicit grant* flow described in Section 3.1. Hence, a similar solution can be used when integrating with SAML, i.e., Figure 2. In addition to this challenge, another challenge regarding SAML is the complexity of the mark-up language itself; While OAuth and OIDC are fairly simple and lightweight, SAML requires a lot of processing outside of what is in scope of the OLYMPUS project. While there are COTS solutions for handling SAML, these are not intended to be modified or their cryptographic components

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

replaced. Therefore, support for SAML will require custom XML handling, assertion verification, etc. which is not related to the core problems addressed by the project, namely cryptographic protocols focused on password handling and signatures.

3.4. FIDO, WebAuthn and CTAP

YubiCo [8] and their YubiKey solution has been addressing the problem of passwords for some time. Their approach involves the use of a secure authenticator for second-factor authentication. This work has resulted in the forming of the FIDO alliance and the development of the FIDO2 framework, consisting of the WebAuthn [9] and CTAP protocols [10]. These protocols allow for, respectively, a ServiceProvider to access a user authenticator using JavaScript and the user's browser to access the physical authentication device. Since these protocols do not involve an IdP and are solving the password problem in a completely different fashion, the work done in OLYMPUS does not relate to these protocols.

3.5. W3C Verifiable Credentials

Recently W3C has been working on a specification for credentials to be used within the W3C domain. This work has been focused on building a data model on top of JSON-LD and, optionally, JWT. Based on the work done by the working group so far, the PESTO protocol inherently supports this specification, as it is "just" a matter of building the proper JSON to sign. The introduction of the "domain" and "challenge" attributes as part of a linked proof might further make it possible for future improvements to the PESTO protocol to improve privacy by reducing linkability: To prevent a malicious relying party from reusing a valid JWT with another relying party in the OIDC scenario, the JWT contains an "aud" attribute describing the intended audience of the token. This attribute must be known by the IdP, making it impossible to make an unlinkable IdP solution OIDC compliant, as the IdP thereby knows the who the relying party is. While not thoroughly investigated, it is possible that the W3C Verifiable Credentials specification might make it possible for IdPs to make unlinkable verifiable credentials.

In the case of pABCs, adoption of the W3C Verifiable Credential format is even more interesting. One of the key factors against pABC adoption has been the difficulty of integrating them with existing systems or even other pABC schemes. In this case, the integration of the project’s pABC functionality with the Verifiable Credentials specification has been thoroughly investigated. In fact, we have a working implementation that uses the specification models for serializing pABC credentials and presentations used in the framework. Apart from implementation work, this integration has required some theoretical work

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

on definitions and how to adopt the specification. More details on how this integration has been realized can be found in deliverable D4.3 [11].

The W3C Verifiable Credential specification is still evolving and suffering changes, and the integration has some functionalities that will not be implemented within the project (e.g., no verifiable data registry will be deployed). What is more, the integration of W3C Verifiable Credentials may not be necessary for various use cases and it demands extra work on part of the deployer and heavier serialization processes. Because of these reasons, we decided to keep the original project-specific serialization in the framework, offering the framework+W3C VC integration as an alternative branch.

3.6. Multifactor Authentication

As discussed in Deliverables D3.3 and D4.3, the basic OLYMPUS framework is somewhat vulnerable to client-side attacks such as keyloggers. To mitigate this type of attacks, support for multifactor authentication is implemented in the framework. As there are numerous ways to implement multifactor authentication, e.g., email, phonecalls, sms, push-notifications and various OTP schemes, a generic approach is taken in the framework.

The deployed IdP must create one or more implementations of the *MFAAuthenticator* interface. This interface contains 4 methods:

- generateSecret() - Used as part of the initialization protocol for the MFA device.
- combineSecrets(List<String>) - Used optionally as part of the initialization protocol for the MFA device.
- generateTOTP(String) - Used to generate a one-time key, given a secret key.
- isValid(String, String) - Used to validate that a one-time key and secret key matches.

While the interface is quite generic, it should enable most MFA schemes to be implemented and may not even require all methods to be implemented, ie., the *combineSecrets* method is only relevant if the partial IdP should share common key material. Furthermore, the implementations of the interface can be used on both client and server side, i.e., the *generateTOTP* method is mainly relevant for clients, whereas *isValid* is mainly relevant for servers.

We do note that any MFA support is likely to contain out-of-band communication, i.e., the OLYMPUS clients have a method obtaining a secret key from the vIdP, however this secret key is used to initiate the MFA device out-of-band and is therefore out of scope of this document.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

The framework does contain a sample, working implementation of how a Google Authenticator type MFA implementation can be done.

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

4. SERVER-SIDE INTEGRATION

While an OLYMPUS based client is oblivious with regard to the cryptographic protocols used, this is not the case for the vIdP server. Due to the modular approach of the OLYMPUS architecture, an OLYMPUS vIdP consists of a number of components that must be configured during initialization in order to achieve the intended functionality. This section of the deliverable gives an overview of the main components, for more detailed documentation, we refer to the JavaDoc documentation of the implementation.

Depending on the concrete application/deployment, the vIdP will consist of an *AuthenticationHandler* and a *TokenGenerator*. In order for these two components to function, the vIdP will also require a storage component for persisting data, a cryptographic component (preferably in an HSM (Hardware Security Module) or alternatively a software based solution), one or more *IdentityProofing* components and (if multifactor authentication is desired) one or more *MFAAuthenticator* components.

Once the vIdP is properly configured and instantiated, its functionality must be exposed to the client. How this can be done in practice, is explained in Section 4.3.

Note that server implementation of PESTO currently relies on a secure key distribution assumption, i.e., someone must securely generate and distribute private keys to each vIdP server. While there are algorithms for doing this [12], these are not implemented in OLYMPUS.

In addition to the PESTO integration, the server contains the appropriate modules to work with distributed pABC (d-pABC) technologies. These modules include the implementation of the underlying cryptography based on Pointcheval-Sanders (PS) multi-signatures introduced in D4.1.

More specifically, the implementation of d-pABC introduces interfaces for setup, key generation, key aggregation, signing, combining signatures, verification and zero-knowledge proofs support. Unlike PESTO, the implementation of d-pABC does not need a key distribution phase, but the necessary parameters (like number of servers or set of attribute names) must be setup in all IdPs.

For assisting in key generation (and configuring the vIdP in general) the class *ConfigurationUtil* can be used. The utility will generate key material and package various information, provided by the developer, into a format understandable by the framework.

4.1. Persisting data

When a vIdP must store data such as user attributes or public keys, a storage interface is used. For testing purposes, a simple in-memory implementation of

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

the interface can be used, however this will erase all data when the server is restarted and is therefore not suitable for use beyond testing.

In order to persist data, a proper database should be used. This is done by making an implementation of the relevant storage interfaces, e.g., *PestoDatabase* if the cryptographic protocol is based on PESTO. Note that the storage implementation might have to implement multiple interfaces in order to allow for multiple components to store data. This may especially be relevant with different Identity proofing components.

For example, the pABC component needs access to the database to carry out its task. However, the only actual functionality it needs (and should use) is the retrieval of the attributes of a specific user to generate a credential from them. In this case, the "best" way to access the database would be through an interface that contemplates precisely that functionality.

The implemented components are oblivious to the storage type; hence the interface implementation should handle all responsibilities regarding the database handling, e.g., opening/closing the connection, error handling, etc.

4.2. HSM Integration

HSMs (Hardware Security Module) are a way to improve security, by encapsulating certain functions, such as signing and encryption, in tamper-proof hardware and under strict access rules requiring dual control and 2-factor authentication. This prevents a hacker from accessing sensitive material such as cryptographic keys and makes it impossible to create uncontrolled copies of the keys or even export them in clear text.

In order to support the usage of HSMs, the cryptographic operations used in the various protocols have been encapsulated in a couple of interfaces for that purpose. Primarily use of HSMs is a concern for the server components, however the framework does offer similar support for client functionality. While the framework comes with a standard Java based implementation of these cryptographic primitives, a custom implementation of the *ServerCryptoModule* (or *ClientCryptoModule)* interface can be implemented, replacing the software functionality with an HSM based version. The sensitive functionality encapsulated in this interface, consists of the following responsibilities:

- Store keys (Storage of private keys)
- Store certificates (Storage of public keys)
- Hashing (SHA-256)
- Verify signature (SHA256withRSA)
- Pairing (Perform standard operations in relation to bilinear pairing)

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

- Signing (Perform a SHA256withRSA signature)
- Combining signatures (Combine partial signatures to form a proper signature)

Not all of these methods may require HSM involvement. In particular combination of signatures and pairing are not likely to require HSM involvement.

The reason for using an HSM is to increase the level of security even further, by having the key material of each partial IdP stored and used securely. Thus, even if an adversary gains *read* access to a partial IdP's database or RAM, it will not be sufficient for it to fully compromise, and thus impersonate, that partial IdP.

Distributed signing can be done on an HSM that allows import of a private RSA key (share). Concretely, each partial IdP will have their own HSM which they each use to generate a public/private key pair. They then give the public key to the trusted party that is generating the distributed RSA key used for distributed token generation. The trusted party will encrypt (wrap) the private key share for each partial IdP under their unique public key (generated in their HSM) and send it back. Each partial IdP can then import the encrypted key share into the HSM, where it can be decrypted (unwrapped), stored and used securely. This is possible since distributed RSA signing is *almost* syntactically identical to non-distributed RSA signing as described in the academic OLYMPUS paper, PESTO [3]. However, some caveats are present which are described in an internal technical document.

One issue of particular interest is that PESTO supports updates of key shares after they have been issued. This can also be achieved securely outside HSM if the private key shares are encrypted using an additive homomorphic encryption scheme, like unpadded El Gamal. However, if we are working with an HSM model that does not support unpadded El Gamal, then this can instead be done in plain outside of the HSM, followed by import of the updated key share in a key ceremony. Even though doing this update in plain outside the HSM is not as secure as doing it inside the HSM, it will still increase security significantly over not using an HSM. This is because the only point in time where a compromise could happen is during key refreshing, which happens rarely and under the scrutiny of a multi-party key ceremony, compared to signing of tokens which happens continuously on a live system.

The pairing operation, along with exponentiation in the pairing source groups, which is needed for computing the dpOPRF in PESTO, are not possible to do in the HSM without programming custom firmware. This is because these operations are based on mathematical operations, which are currently not supported by default by any HSM manufacture to the best of our knowledge. Fortunately, we note that a compromise of the key used in the dpOPRF does *not* allow an adversary to directly construct false tokens or learn users' passwords even if *all* such keys are compromised on all partial IdPs. This is because PESTO requires knowledge of a user's password in order to construct

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

a token signing request that an honest server will accept. So even if the adversary has all the servers' dpOPRF keys, the adversary must further execute an offline attack to find the password of *each* specific user it wishes to impersonate. We thus believe that it is not critical to construct custom firmware in order to support this component.

In the pABC scenario, which relies on PS multi-signatures [13], secret keys are involved in unsupported group exponentiations and pairing operations. Each of the components of a PS secret key could be stored in an HSM. However, the process of key generation cannot be carried out within the HSM as the derivation of the public key is done through group exponentiations. Even then, executing key generation should be a rare occurrence and no sensitive material has to be exchanged between IdPs (thanks to the use of a multi-signature scheme), so corruption of the key would be difficult even if it is used outside the HSM.

More worrying is the exposure of the secret key when generating a signature share, as the frequency of this process is really high. In addition, an attacker that compromised the secret keys of all IdPs could create credentials for any attribute values it wanted. Again, the problem emerges from the use of the components of the secret key in a pairing group exponentiation. Nevertheless, the problem could be mitigated by not exposing each component of the secret key for the calculation. Instead, we could calculate the final exponent, which is a linear combination that depends on the components of the secret key and the message that will be signed, inside the HSM. Although this does not add too much security, it would help in case of a compromise of a single signing instance.

While HSM support is mainly of interest for the server components, the pABC client does need to (securely) store and manage the issued pABCs. This is done using the *CredentialManager* and *CredentialStorage* interfaces. The framework does include software versions of both, however if a client would wish to use secure storage on e.g., a smartphone, a custom implementation of these can be implemented.

4.3. Deployment

In order to have a functioning vIdP deployment, the various components of the framework must be combined, configured and exposed to the client. The combination and configuration of the various components may be done using an implementation of the *PestoIdP* interface. The framework comes with a couple of standard implementations that can be used either directly or as inspiration for custom deployments, the standard implementations are:

- PestoIdPImpl - The most basic implementation offering Distributed Password Verification and Distributed Token Generation, using generic JWT.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

- PabcIdPImpl - Similar to PestoIdPImpl, except it issues Distributed pABCs rather than signing JWT tokens.
- CombinedIdP - Offers both functionality for both distributed pABC issuance and DTG using generic JWT.
- OIDCPestoIdPImpl - Similar to the PestoIdPImpl, except an OIDCTokenGenerator is used to issue OIDC compatible JWT.
- PasswordJWTIdP - A standard non-distributed IdP, used for benchmarking.
- DistributedRSAIdP - Uses non-distributed password verification, but distributed token generation. Used for benchmarking.

Since the combination of components is done in Java code, each partial IdP is essentially a Java program whose methods must be exposed to the client. A common way to do this is to create a REST interface, which is also what the default client software assumes. This REST interface can be implemented and deployed in a manner of ways, depending on the priorities of the concrete application, e.g., a stand-alone jar with an embedded webserver can be deployed or the interface may be deployed as a servlet on a larger webserver.

The framework does come with an implementation of a basic generic IdP, exposed as a REST interface and deployed using an embedded webserver. The class *RestIdPServer* is given an instantiation of one of the above mentioned IdPs and will expose it as a REST service.

In addition to combining the various components in a single program, the partial IdP must also be configured, i.e., each partial IdP must know what ports to expose the REST interface on, how to contact the other partial IdPs, what keys to use, etc. The framework operates with a few *configuration* interfaces, e.g., *ServerConfiguration* and *PABCConfiguration,* describing relevant values for a specific implementation of the *PestoIdP* interface. This allows each deployment to represent configurations in the way they prefer, however in practice, a JSON format is likely to be used, which is also natively supported by the framework.

In order to ease the process of creating configurations, a configuration tool is included in the framework. The tool is implemented as an abstract class (*eu.olympus.util.ConfigurationUtil),* containing a number of fields. A child class should populate these fields and may then use the inherited functionality to generate configurations for the vIdP.

As an alternative to the configuration util, configurations can also be written or edited by hand. The following is an example of one such configuration:

```
{"keyStorePath":"src/test/resources/keystore.jks",

"keyStorePassword":"server1",

"trustStorePath":"src/test/resources/keystore.jks",

"trustStorePassword":"server1",
```

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

```
"port":9080,
"issuerId":"https://olympus-vidp.com/issuer1",
"myAuthorizationCookie":"some_cookie",
"authorizationCookies":{
"some_cookie":{"id":"server1", "roles":["SERVER"]},
"some_cookie":{"id":"server2", "roles":["SERVER"]},
"some_cookie":{"id":"administrator", "roles":["ADMIN"]}},
"servers":["http://127.0.0.1:9081","http://127.0.0.1:9082"],
"tlsPort":9933,
"keyMaterial":{"modulus":<BigInteger>,"privateKey":<BigInteger>,"publicExponent":<BigInteger>},
"oprfBlindings":{"1":<BigInteger>,"2":<BigInteger>},
"rsaBlindings":{"1":<BigInteger>,"2":<BigInteger>},
"localKeyShare":<Base64 encoding of a key share>,
"remoteShares":{"1":<Base64 encoding of a key share>, "2":<Base64 encoding of a key share>},
"oprfKey":<BigInteger>,
"id":0,
"cert":<PEM encoding of the certificate>,
"waitTime":1000,
"attrDefinitions":[{"type":"String","id":"Nationality","shortName":"Nationality","minLength":0,"maxLength":16,"type":"String"},
        {"type":"Integer","id":"Age","shortName":"Age","minimumValue":0,"maximumValue":123,"type":"Integer"},
        {"type":"String","id":"Name","shortName":"Name","minLength":0,"maxLength":16,"type":"String"}],
"seed":"cmFuZG9tIHZhbHVlIHJhbmRvbSB2YWx1ZSByYW5kb20gdmFsdWUgcmFuZG9tIHZhbHVlIHJhbmRvbQ==",
"lifetime":72000000,
"allowedTimeDifference":10000}
```

4.3.1. Example of an vIdP deployment

To demonstrate how the various components all fit together, the *oidc-demo-idp* sub-project can be investigated:

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

It is built using the command '*mvn install*' which will produce a *.jar* containing the IdP application. The *.jar* file can be found in '*./target/OIDC-IdP-jar-with-dependencies.jar*' and may be run either using the command *'java -jar OIDC-IdP-jar-with-dependencies.jar <configuration path>'* or (assuming the environment variable CONFIG_FILE is set) simply *'java -jar OIDC-IdP-jar-with-dependencies.jar'*.

The class *eu.olympus.oidc.server.RunOIDCServer.java* specifies which components, e.g., Storage, CryptoModule, IdentityProviders, type of IdP implementation, etc., are used in the vIdP deployment. Furthermore, the class *eu.olympus.oidc.TestConfigurationUtil* can be used to generate a valid set of configuration files.

Finally, the project also contains a Docker configuration for the demo IdP setup. The *'mvn install'* will also build Docker containers for the demo vIdP, hence running *'docker compose up'* should spawn a 3-server vIdP (running HTTP on ports 9080, 9081 and 9082 or HTTPS on 9933, 9934 and 9935).

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

5. CLIENT-SIDE INTEGRATION

5.1. Client deployment

The framework contains a Java based library containing the functionality required for a client to interact with an OLYMPUS based virtual IdP setup. The following sections will introduce the main functionality of the client library. In addition to this introduction the JavaDoc documentation may be referenced for further detail.

The main entry point to the framework for a client application, is the *UserClient* Java interface. This interface may be instantiated as a subclass of the *PestoAuthClient* class in order to perform authentication according to the PESTO protocol suite (i.e., *PestoClient* or *PabcClient* for distributed signatures or distributed pABC issuance). For benchmarking reasons, *DistributedRSAClient* and *PasswordJWTClient* are also included in the framework. They perform authentication according to the classic username + password verification and will, respectively, produce distributed and non-distributed JWT tokens.

The *UserClient* interface exposes the following method signatures:

- void: createUser(String username, String password)
- void: createUserAndAddAttributes(String username, String password, IdentityProof identityProof)
- String: authenticate(String username, String password, Policy policy, String token, String type)
- void: addAttributes(String username, String password, IdentityProof identityProof, String token, String type)
- String: requestMFAChallenge(String username, String password, String type)
- void: confirmMFA(String username, String password, String token, String type)
- void: removeMFA(String username, String password, String token, String type)
- Map<String, Attribute>: getAllAttributes(String username, String password, String token, String type)
- void: deleteAttributes(String username, String password, List<String> attributes, String token, String type)
- void: deleteAccount(String username, String password, String token, String type)

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

- void: changePassword(String username, String oldPassword, String newPassword, String token, String type)
- void: clearSession()

The *UserClient* implementations may take relevant parameters, such as a list of the partial IdPs to use and a *ClientCryptoModule* implementation, as input during construction. In the pABC case, a *CredentialManagement* implementation must also be included. The *CredentialManagement* component is responsible for storing and managing ABC's, e.g., it can decide if obtained credentials will be kept while they are valid or if they should be discarded after the derivation of a presentation token.

The *createUser* method will create a new user account on the vIdP, i.e., relevant data will be stored on the servers, attached to the specified username.

The *createUserAndAddAttributes* is similar to the *createUser* method, in that it will store key material on the servers. However, in addition to storing the key material, the method will also validate the attached identity proof and store the contained attributes in the vIdP database.

The *authenticate* method will produce an access token. The concrete content of this access token will vary depending on the *UserClient* implementation, i.e., in the PESTO case a JWT token may be returned, while the pABC implementation will produce a *PresentationToken.* How the access token is produced will also vary depending on the concrete *UserClient* implementation; In the PESTO case an authentication protocol is run with the vIdP. If the protocol is successful, the server will generate a signed access token and return it to the client. In the pABC case, a valid credential may be stored locally, allowing the access token to be produced without interaction with the vIdP. The *policy* parameter determines what the contents of the access token should be i.e., it specifies a list of predicates, which are essentially what attributes should be revealed or compared with some other value. If, for instance, a policy specifies that the attributes "*name*" and "*age*" should be revealed, the access token will contain a vIdP attested proof of these attributes. Details on the *policies* can be found in Deliverable D4.3.

The *addAttributes* method assumes a user account already has been created. The method will first authenticate the user against the vIdP and then send the *IdentityProof* to the vIdP for validation. If both the user authentication and *IdentityProof* validation are successful, the attributes contained in the *IdentityProof* will be stored in the vIdP database.

The *requestMFAChallenge*, *confirmMFA* and *removeMFA* methods are used in connection with either attaching or removing a multi-factor authentication method to an account. Details can be found in deliverable D4.3, but in general, the client will start the process of attaching an MFA device with a call to *requestMFAChallenge*, which will generate some data. These data are used to

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

initialize the MFA device, after which the device will produce some short-lived token. The token is used with the *confirmMFA* method to prove possession of the initialized MFA device and if the call succeeds, MFA is activated for the account. Finally, the *removeMFA* method may be used to de-attach an MFA device for the account. Note that the user must prove possession of the device in order to remove it.

In order to do account management, the methods *getAllAttributes*, *deleteAttributes*, *deleteAccount* and *changePassword* can be used to fetch all stored attributes belonging to an account, delete a list of attributes, delete the account entirely or change the password.

In order to improve user experience in connection with MFA, the client caches a session cookie and private key material, allowing the user to only use MFA for the method call. The method *clearSession* is included to have functionality to remove this cached information.

The client library itself is implemented in plain Java and does not rely heavily on non-standard libraries outside of a library for Bilinear pairings, for which we use Apache Milagro AMCL [2]. Utilizing the user client in Android should therefore be trivial.

Native support for iOS is not implemented in the framework, however there are tools for porting Java applications to iOS.

Similarly, the implementation is not planned to be ported to JavaScript within the project. There are libraries for the required bilinear pairing functionality, so an implementation is theoretically possible, but has not been further investigated.

5.2. Proxy alternative

If an application is not able to utilize the Java library, i.e., is based on a different programming language and porting the client code is not feasible, it is possible to deploy the client code as a proxy server. This would involve creating a small Java-based web server, wrapping the functionality of the *UserClient* interface, and exposing a REST interface instead. This would allow the application to utilize the interface using standard web communication.

An example of how this can be done, suitable only for local deployment, can be found in the sub-project *rest-wrapper.* The sub-project is built using *'mvn clean install'*, which will produce a *.jar* file, that can be started using the command *'java -jar /target/REST-client-wrapper-jar-with-dependencies.jar'*

The proxy approach does however change the trust model of the client. Rather than the password never leaving the client, it will now be transmitted to the client proxy, hence a compromised proxy will allow the attacker access to the user's password. This problem is however somewhat reduced:

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

- If the proxy is deployed locally on the client, the attacker must successfully compromise the client. This will create a security setting identical to the no-proxy setup.
- The proxy application would be very simple, hence doing a secure deployment should be less problematic.
- The passwords are never persisted in the proxy and an attacker will therefore only be able to learn passwords if the proxy is compromised.
- If MFA is used, the user must supply a fresh MFA token for every operation.

A somewhat similar approach would involve the proxy server to host a JavaScript version of the *UserClient*. While this would allow the application to execute the code without needing Java support, it would require the client to trust the code delivered by the proxy server (as there is no native support for code signing in JavaScript). Essentially that would make the proxy server the new single point of trust.

While both approaches are possible in certain use cases, they do cause extra complexity to the overall system deployment.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

6. VERIFIER-SIDE INTEGRATION

6.1. Token verifiers

The tokens generated using the PESTO approach (both the original and the ones adapted for the OIDC flow) are standard JWTs and can be validated with any JWT verifier (or OIDC verifier) without using project-specific code.

6.2. pABC verifier deployment

The pABC approach requires custom software for managing cryptographic operations like verification of presentation tokens. The framework contains a Java-based library for performing verification. The entry point is defined in the interface *PABCVerifier* (for the alternative with W3C VC serialization, we offer a *W3CPresentationVerifier* with an equivalent method):

- VerificationResult: verifyPresentationToken(String token, Policy policy)

The *verifyPresentationToken* method receives a token (serialized as String) and an OLYMPUS policy, and verifies whether the token is a valid pABC presentation, not expired, and fulfils the requested policy. Note that the *policyId* must be the same that was used to generate the token, or the verification of the signature will fail. *VerificationResult* is an enumerate with the following values:

- *VALID*: The token was validated and fulfils the policy.
- *INVALID_SIGNATURE*: The cryptographic operations (signatures, Zero-Knowledge proofs) failed.
- *BAD_TIMESTAMP*: Expired token or credential.
- *POLICY_NOT_FULFILLED*: The token does not fulfil the requested predicates.

The verification library does not rely heavily on non-standard libraries outside of a library for Bilinear pairings, for which we use Apache Milagro AMCL [2]. Although verification is expected to be used by a relying party in a common “server” in many use cases, using the verification library in Android should be trivial. Native support for iOS is not implemented in the framework, however there are tools for porting Java applications to iOS.

Similarly, the implementation is not planned to be ported to JavaScript within the project. There are libraries for the required bilinear pairing functionality, so an implementation is theoretically possible, but has not been further investigated.

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

6.3. pABC external verifier deployment

Akin to the client library, a service for verifying the presentation tokens generated with the pABC approach may also be externally hosted, with the goal of helping in cases where the Java library cannot be applied directly, e.g., in applications based in other programming languages. It simply consists of a Java-based webserver that exposes as a REST interface two methods: a verification method (same signature as the Java library) and a method for setup (which receives the endpoints of the partial IdPs that form the vIdP for which we want to verify tokens).

Again, the trust model is modified, as the relying party's application must rely on this webserver. As the idea is to have the proxy as a helping tool for integration with the application, the proxy verifier is expected to be deployed locally along the application. In that case, an attacker would need to compromise the same device as if the Java library was used, essentially avoiding the issue.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

7. USE CASE INTEGRATION

7.1. CFP Integration

The aim of the Credit File use case is to ease the process of getting some financing from a specific financial entity using the information from a financial report. The idea is to generate a token containing only the minimal financial information required for that particular entity and without any data that could be used to identify the user. The financial entity determines the required information (attributes) for a specific financing service or product and the required attributes are stored, at the financial entity, in a database catalogue accessible via API.

The financial report has to be generated in previous steps because it is a complex process that involves the interaction of several actors. The report has to be signed first by the CTI organisation, i.e., the company in charge of the financial report generation and the credit file platform, to guarantee the integrity of the information and then by the user with a qualified certificate, confirming that the information belongs to her or him.

The user selects a service or product from an entity and authenticates through an external ID platform called Certy, using a qualified certificate. Once the financial report has been generated, encrypted and stored in the user's device, there are two possible models of integration used, depending on the type of the smartphone used by the user.

If the device platform is Android, the client is integrated into the CF app and the following steps take place:

1. Using the CFP app, the user provides a username and a password and calls the method *Olympus.userclient.createUser(username, password)* to create a new user account.
2. Then, the username and password are used together with the identity proof, which in this case is the public part of an X.509 certificate, as parameters to invoke the method *Olympus.userclient.addAttributes(username, password, identityProof)*.
3. The OLYMPUS vIdP validates the certificate and marks the identity as valid. The UserCredentialIdentityProver is used here to check the identityProof and determine whether the user certificate associated with the public key is valid or not. If it is, both the public key and the certificate's serial number are stored in the vIdP's database.
4. The CFP app gathers the financial statement (the signed XML document previously stored in the user's device) and calls *Olympus.userclient.addAttributes(username, password, financial statement)*.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

OLYMPUS

5. The server validates the signature on the XML document, using the public part of the X.509 certificate sent earlier, and the public certificate of the CTI organisation. This is performed by the identity prover called CreditFileIdentityProver whose first step is to verify if the serial number of the credit file signature matches with the one stored in the vIdP's database. If it matches, a Xades validation takes place to verify the validity of the user's signature of the financial report and the CTI's signature is validated too. If both signatures are correct, the financial report is parsed and the relevant attributes with their values from the financial report are added to the user account.

6. The CFP app provides the token that identifies the request in the CF database by performing a new invocation to the method *Olympus.userclient.addAttributes(username, password, token)*. The identity prover used here is TokenIdentityProver which simply stores the token in the server's database.

7. The list of the attributes to reveal is decided by the entity and it depends on the specific profile that has been selected and its specific documentation requirements. This information is defined for the entities and also stored in the CF database. From there, using an API, the CFP app gets the list of attributes to reveal. This gets translated and used as input to *Olympus.userclient.authenticate(username, password, policy)*.

8. Once the client method is executed, a token is generated with the information of the revealed attributes that represents all or only a part of the defined relevant attributes. This token contains only the financial information required for the entity and no personal data is included here, e.g. Only "income=55000, employmentYears=4" or something similar. If the pABC protocol is used, a credential is issued in addition to the token. This credential will be stored in the user's device and allows for later use.

9. After the token is generated, the verification process is performed invoking the methods *JWTVerifier.verify (token)* or *PABCVerifier.verifyPresentationToken(token, policy)*, depending on whether the PESTO or pABC protocol is used. Its purpose is to check if the user's attributes comply with the requirements that the entity established and translated as a policy.

10. This token is sent to the entity and the entity evaluates the user's information. The entity can accept or deny the user's request and, if the request is approved, the user can send his or her identity to the entity. In that case, the unique ID and ID proof of the user is sent to the bank.

If the device platform is iOS, the client invocations are performed by means of the REST wrapper mentioned in Section 5.2. Using this REST wrapper, the invocations from the client app use several endpoints to invoke the methods that were used in the direct integration for the Android platform:

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

1. As the first step, to provide a username and a password, the idp/user/createUser endpoint is used.
2. Then, the different proofs that the process requires are gathered and used as input of the idp/user/addAttributes endpoint.
3. In order to generate the token, the endpoint used is idp /user/authenticate.
4. Finally, as there is no exposed method to perform the verification process in the REST wrapper, a CF Java backend invocation is required to verify the generated token. This backend contains a REST interface that is called from the client app. This interface instantiates a Verifier or a PSPABCVerifier object with cryptographic material retrieved from the vIdP and calls the *verify* or *verifyPresentationToken* methods, depending on whether PESTO or pABC module is used.

7.2. MDL Integration

BASIC FLOW

The aim of the mDL use case is that a user can be verified (i.e., age verification, address verification etc) by a third party, maintaining his privacy using anonymous credentials through the vIdP. Similarly, to the use case described in Section 7.1, this can be achieved by generating a token that contains only the minimum required information and without any data that could be used to identify the user. Further details on the use case can be found in Deliverable D6.1.

An mDL holder is a user that possesses a mobile driver's licence in his device, digitally signed by the Issuing Authority (IA) of his country. The data that contain the personal information of the mDL holder are included in a structure called Mobile Security Object (MSO), which is signed by the mDL Document Signer certificate issued by the Issuing Authority PKI. This certificate is used by the vIdP during registration, in order to check the authenticity of the data provided by the mDL holder.

When the mDL holder needs to register to the Olympus vIdP, the following procedure takes place:

1. The mDL holder app gathers username & password and makes a request for registration, providing an encoded version of the MSO to the server, by utilizing the method *client.createUserAndAddAttributes(username, password, MSO)*.
2. The server decrypts the received MSO and checks if it is issued by a trusted IA. More specifically, it checks whether the MSO was signed by a

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

certificate that the server already possesses from the IA which issues mDLs. If this is the case, server accepts the registration of the user.

3. If the mDL holder wishes to download his credentials and store them in his device for later (offline) usage, he may do so by invoking the method *client.authenticate(username, password, policy)* providing an empty policy. *CredentialManagement* module is responsible for dealing with the credential storage.

Next the verification process continues as follows:

4. When the mDL holder needs to be checked by a verifier, the verifier asks from the holder the pieces of information that needs to be revealed. If the mDL holder accepts to reveal them, then the mDL holder app will either make a request to the vIdP for an access token, invoking the method *client.authenticate(username, password, policy)* or will create the token from the already stored credentials, utilizing the method *credentialManagement.generatePresentationToken(policy)*.
5. Whichever method shall be used for authentication, an access token containing the revealed attributes is created. This token is anonymized in the sense it only contains attributes specified in the “reveal list”, eg. Only “name=John, dateOfBirth<2003-01-01” or something similar.
6. The mDL holder app sends the token, as well as information about which method was used to generate this token to the mDL verifier, in order to verify the user info.
7. Depending on the token generation method (PESTO for online mode or pABC for offline), the verifier will use either *JWTVerifier.verify(token)* method or *PABCVerifier.verifyPresentationToken(token, policy)* respectively, in order to verify the provided token. Note, that both methods are executed in the mDL verifier app and no communication with the vIdP is needed for this last step.

The following diagram shows the flow for the registration and verification of an mDL holder.

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

OLYMPUS

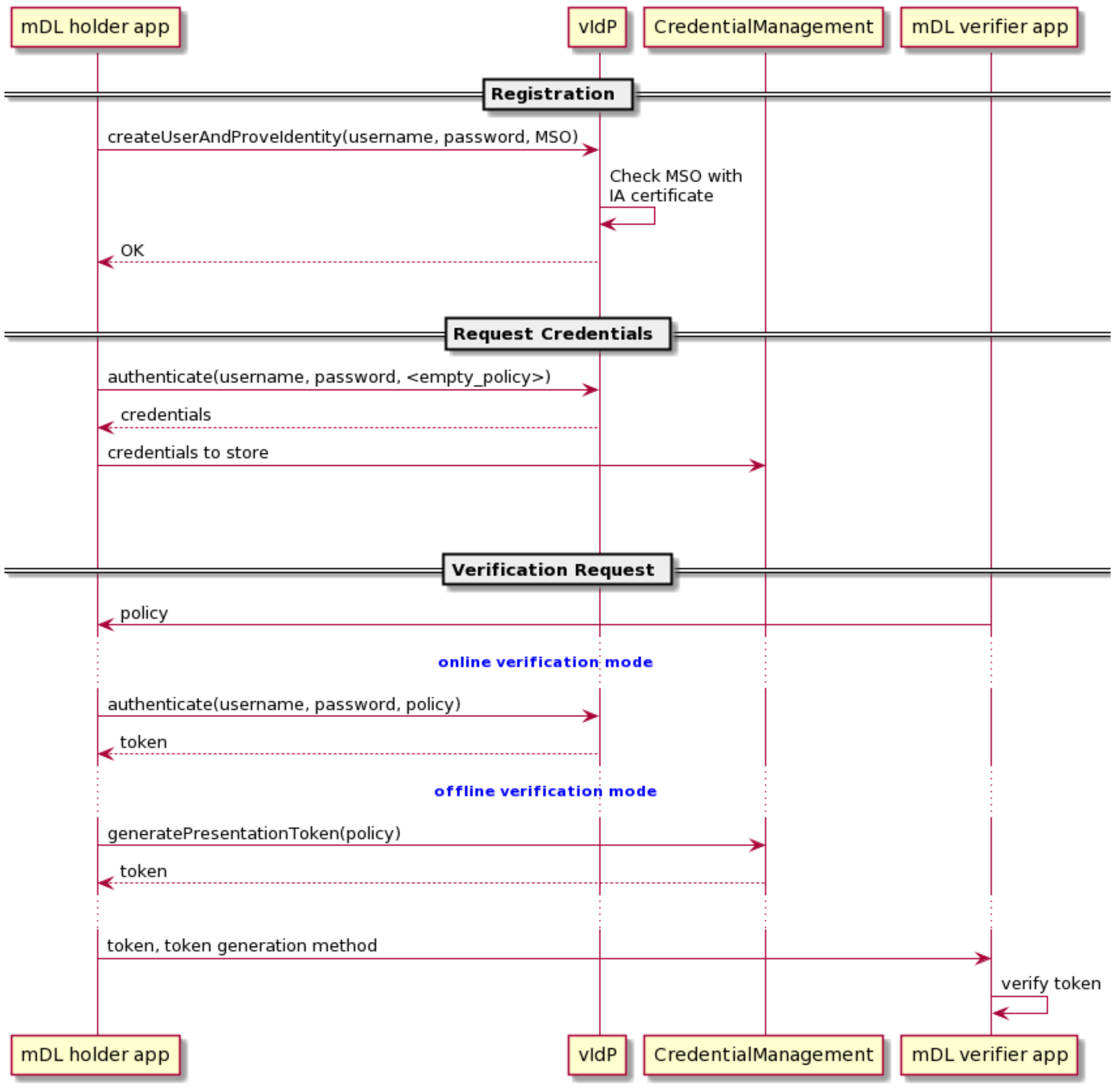

mDL holder app
vIdP
CredentialManagement
mDL verifier app
Registration
createUserAndProveIdentity(username, password, MSO)
Check MSO with IA certificate
OK
Request Credentials
authenticate(username, password, <empty_policy>)
credentials
credentials to store
Verification Request
policy
online verification mode
authenticate(username, password, policy)
token
offline verification mode
generatePresentationToken(policy)
token
token, token generation method
verify token

Figure 3. The mDL basic flow

ACCOUNT MANAGEMENT - MFA AUTHENTICATION

In the mDL use case, we are utilizing some of the account management options that are provided in the Olympus solution. More specifically, the mDL holder can change his password and delete his account, by making the corresponding requests to the vIdP.

MFA authentication is also available for the mDL holder as a means of enhancing security in his transactions with the vIdP. The mDL holder has the option to add or remove MFA anytime he wishes, as long as he is online in order to perform the corresponding requests to the vIdP. The MFA implementation used is the sample implementation of a Google Authenticator type that is provided by the Olympus framework.

As described in section 5.1, the attachment of the MFA authentication in a user's account is completed in two steps, invoking first the *requestMFAChallenge()* method to obtain a challenge from the vIdP, which will

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

be used in MFA token generation. Then, in the second step, this MFA token shall be sent to the vIdP through *confirmMFA()* request and, upon a successful response, the MFA authentication is activated. Removal of MFA authentication is performed in one step, with the invocation of *removeMFA()* method, providing again an MFA token generated from the obtained challenge.

The utilization of these interfaces is depicted in the following figure.

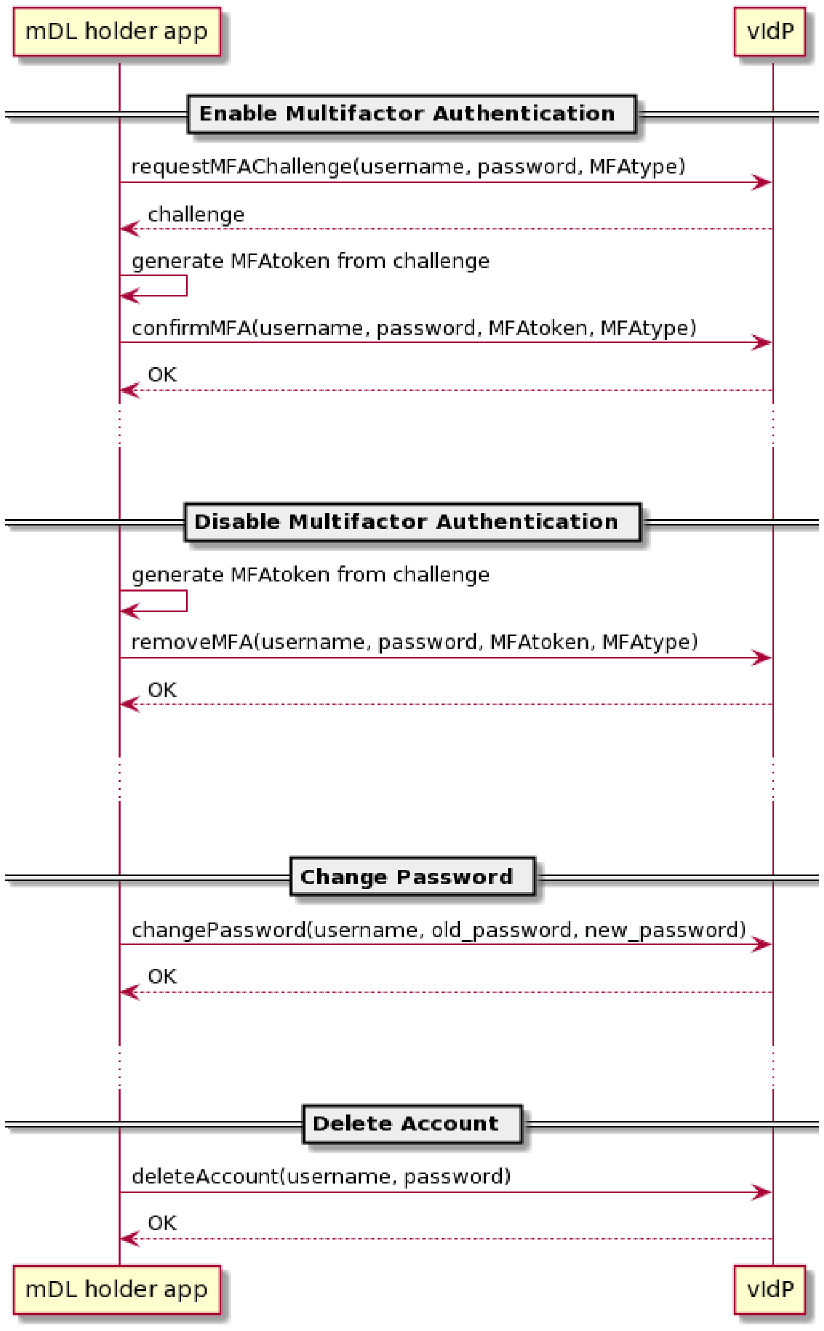

mDL holder app
vIdP
Enable Multifactor Authentication
requestMFAChallenge(username, password, MFAtype)
challenge
generate MFAtoken from challenge
confirmMFA(username, password, MFAtoken, MFAtype)
OK
Disable Multifactor Authentication
generate MFAtoken from challenge
removeMFA(username, password, MFAtoken, MFAtype)
OK
Change Password
changePassword(username, old_password, new_password)
OK
Delete Account
deleteAccount(username, password)
OK
mDL holder app
vIdP

Figure 4. mDL MFA and Account Management

MSO VALIDATION

The ID proofing service provided by the Issuing Authority takes an mDL as input and checks the validity of the mDL signed data, by verifying the digital signature from the respective Issuing Authority. The process is as follows:

1. Check MSO integrity: the *issuerAuth* element of the MSO is parsed and the COSE [17] signature is verified.
2. Check MSO authenticity: a certification path validation of the MSO signing certificate up to the trusted anchors of IAs is looked up.
3. Check MSO signing certificate is valid: the status of the MSO signing certificate is verified against the published CRL of the IA CA.

The IA validation is available as a REST service at endpoint */validate*.

7.3. Third use-case

While the CFP and MDL use cases show case the majority of functionality of the framework, we have designed a third use case for the OLYMPUS framework. The main goal is a generic, but relevant use case, that allows the demonstration of the functionality of the OLYMPUS framework, focusing on the aspects that were not completely covered in the other two use cases. In particular, the main interests have been: W3C Verifiable Credential specification for serialization in the pABC approach, Multi-Factor Authentication, and OIDC interoperability. As a side-effect, this use case shows the flexibility and interoperability of the OLYMPUS identity management system.

7.3.1 Scenario description

The use case establishes the OLYMPUS vIdP as a general-purpose authentication tool, with commonly needed attributes like name, date of birth, phone/email of contact and, more specific to the current situation, COVID vaccination date. Users can take advantage of this tool in a user-friendly manner in two ways: online (from home PC) and in-person (using a mobile phone application). In both cases, the tool can easily be adopted by any relying party that wishes to do so. In the former, the relying party will simply use the OLYMPUS vIdP as an OIDC provider. In the latter, the relying party will need a server that has the OLYMPUS verification library and start the presentation process by having the user scan a QR code.

The specific scenario. To particularize, we use the generic tools to book and attend a venue (i.e., a restaurant). In this case, an online reservation prior to attending is needed. This is done via a standard webpage, relying on a OIDC provider. Once we arrive at the venue, there is an extra policy check: the user

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

needs to show that the name corresponds to the person that made the reservation, as well as that he/she was vaccinated at least 14 days prior. This is done as a V3C based pABC presentation using a smartphone.

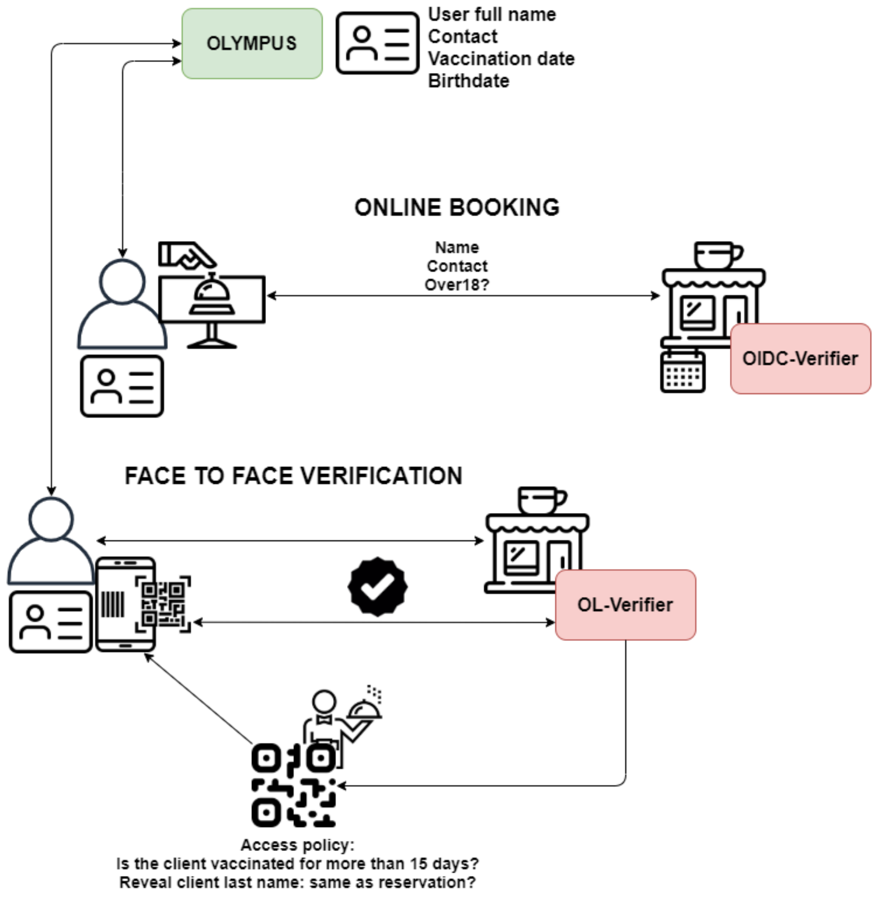


Figure 5. Use case 3 scenario

7.3.2 Components

The components involved in the scenario are:

- **OLYMPUS vIdP:** Provides privacy-preserving identity management to users, while still giving relying parties the security they need on the presented information. Supports both online authentication following the OIDC flow and issuance of pABCs (serialized following W3C's Verifiable Credentials specification) for "offline" interactions. Optionally (as identity proving is not the focus of the use case), an external service will play the role of third-party identity provider for user enrolment in the vIdP. From a technical perspective, the vIdP is simply a 3-server deployment of the *CombinedIdP* class, which combines pABC and OIDC functionality.
- **User side:** Two tools, one for each approach, are needed. A mobile application for pABC presentations ("offline") and, in addition to a web browser, a local hosted REST based implementation of the OLYMPUS

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

Pesto client. In the future the REST based implementation may be replaced by a browser plugin. In addition to providing OIDC functionality, the local host REST client also offers some account management functionality, such as creating accounts and changing passwords.

- **Relying-party side:** In this case, there is a single “service provider organization” (the restaurant/venue). However, it would act differently depending on the specific flow, hence it may be separated into two services/functions: A booking service, offering online interaction where a standard OIDC implementation is needed. An offline hostess/waiter function, where a PDA/mobile device relies on the OLYMPUS verification library to validate the pABC presentation, as the guest arrives. The service provider is implemented as a simple NodeJS application, utilizing standard components and libraries as much as possible. The OIDC support is done using *redux-oidc*, a common NodeJS library, allowing the OLYMPUS vIdP to be used interchangeably with a KeyCloak IdP, demonstrating the ease of integration. In order to utilize pABC presentations, the NodeJS application will also spawn a small REST server, as described in Section 6.3.

7.3.3 Flows

While the scenario involves a single overarching flow (booking and making use of a service), it can be divided into two separate sub-processes.

ONLINE RESERVATION

Users can create a reservation to their name using the OIDC flow for authentication.

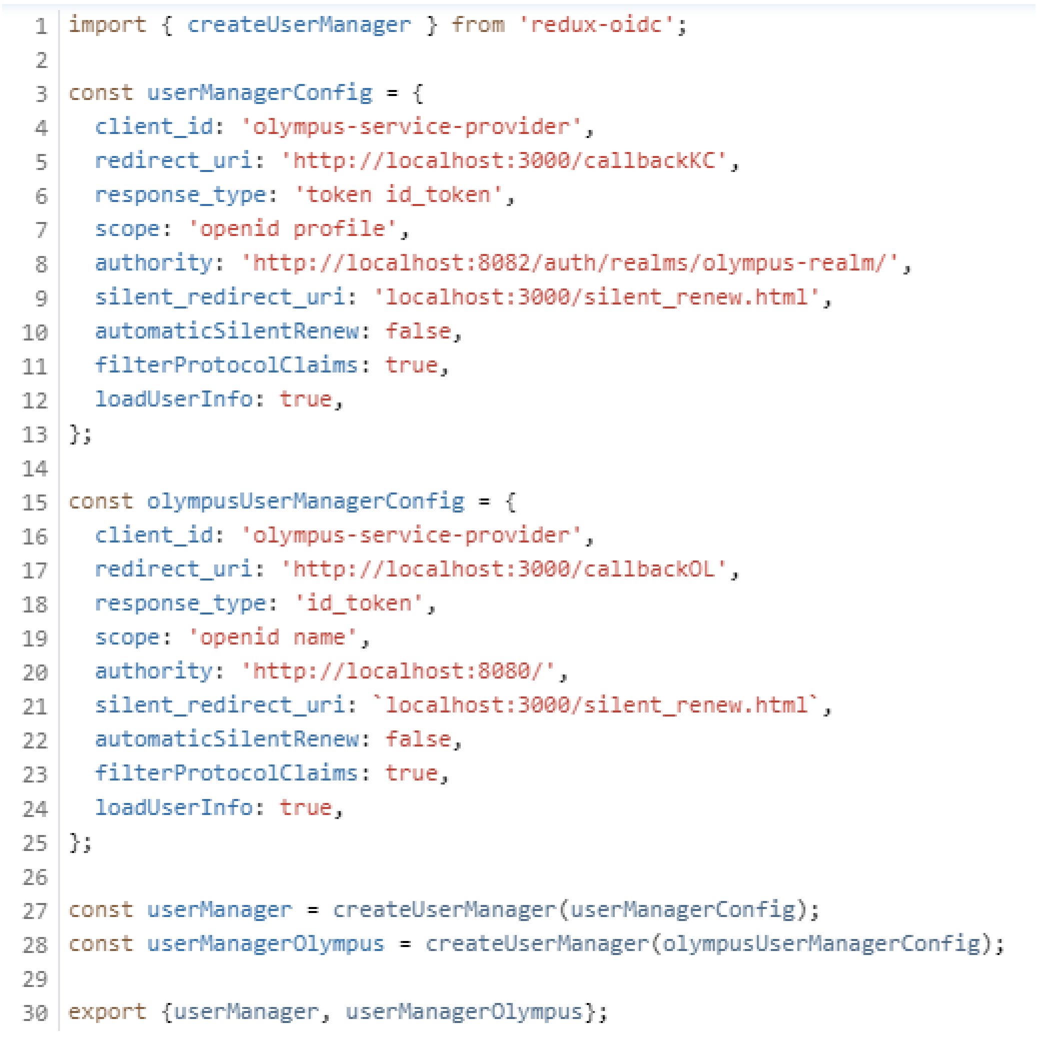
1. First the user accesses the service provider webpage, using a standard browser.
2. The user then selects login with OLYMPUS in the service provider webpage.
3. The OLYMPUS login screen requests username and password[1].
4. After login, an OIDC token with the requested attributes is generated and sent, via the browser, to the service provider.
5. The service provider verifies the token and grants access/confirms the reservation.

Advantages of using OIDC for this operation: Users are accustomed to OIDC (and similar) flows when browsing in the web. It is easy to use and understand.

[1] If the user does not already have an account, the “signup” button can be clicked, offering functionality for creating an account.

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

From the service providers' perspective, supporting an additional OIDC provider is just a matter of adding an additional configuration, as can be seen in figure 6. Also, this step is much faster/more efficient than the credential approach.



```
import { createUserManager } from 'redux-oidc';

const userManagerConfig = {
  client_id: 'olympus-service-provider',
  redirect_uri: 'http://localhost:3000/callbackKC',
  response_type: 'token id_token',
  scope: 'openid profile',
  authority: 'http://localhost:8082/auth/realms/olympus-realm/',
  silent_redirect_uri: 'localhost:3000/silent_renew.html',
  automaticSilentRenew: false,
  filterProtocolClaims: true,
  loadUserInfo: true,
};

const olympusUserManagerConfig = {
  client_id: 'olympus-service-provider',
  redirect_uri: 'http://localhost:3000/callbackOL',
  response_type: 'id_token',
  scope: 'openid name',
  authority: 'http://localhost:8080/',
  silent_redirect_uri: `localhost:3000/silent_renew.html`,
  automaticSilentRenew: false,
  filterProtocolClaims: true,
  loadUserInfo: true,
};

const userManager = createUserManager(userManagerConfig);
const userManagerOlympus = createUserManager(olympusUserManagerConfig);

export {userManager, userManagerOlympus};
```

Figure 6 OIDC configuration of the Service provider

IN-PERSON CHECK-IN

Users can rely on the application to prove necessary predicates to make use of their reservation.

1. The user selects the option for getting a new credential and performs a login process (which may involve multi factor authentication). This step may be done in advance, as the credential will be safely stored during its lifetime.
2. The user uses the app to scan the QR presented by the service provider.
3. The app uses the result to retrieve the policy asked and shows it to the user.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

4. If the user accepts, a presentation for that policy is generated and sent to the service provider.
5. The service provider verifies the validity of the presentation.

Advantages of using Verifiable Credentials for this operation: Offline authentication is possible while avoiding physical ids and the privacy problems that come from them. Also, the application is easy to use, and it is simpler for users to understand the usage of Credentials with their phone.

OLYMPUS ACCOUNT MANAGEMENT

During the previous processes, optional steps can be taken by the user to manage its account in the OLYMPUS vIdP.

In the **online (web-browser)** reservation, when the user is in the OLYMPUS login screen (step 2), he/she can start a registration process. In our specific scenario, for simplicity (as it is not a goal to test the identity proving functionality), the user will simply input the attributes that will be added to his/her account. In a more realistic setting, these attributes would have to be validated, using another IdP or similar role for the bootstrapping process.

When performing login from the **Android application**, the user can opt to add a Multi-Factor-Authentication method (in our specific scenario, based on Google Authenticator). The process will involve introducing user and password of the OLYMPUS account, adding the generated shared secret to the desired MFA authenticator (e.g., Google Authenticator app) and confirming the choice by inputting the TOTP generated by the authenticator.

This project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 786725

8. ATTRIBUTE HIDING

Currently the users' attributes are stored in plain at *each* of the partial IdPs. This means that *each* partial IdP becomes a potential point of failure of user-privacy. This is in stark contrast, not only to the overall security goals of the project, but also to the spirit of adding distribution to *enhance* security.

Specifically, regarding the current framework, a compromise of a *single* partial IdP will compromise the privacy of all the users' attributes, whereas an attack on *all* partial IdPs is needed in order to brute-force *any* user's password or issue illegitimate credential or token.

To protect the user's sensitive attributes, we here present an approach for storing these in a distributed manner, such that *all* partial IdPs must be corrupted at the same time in order for an adversary to learn anything about the content of the users' attributes. Even with this strong security, this approach will still allow the partial IdPs to collaborate in order to verify that a certain public policy is fulfilled and thus allow them to issue a JWT token. The approach will also allow them to issue credentials based on the users' attributes, without ever learning the value of these.

The overall idea is to have each partial IdP store a "zero-knowledge-proof-friendly" certification of the users' attributes. By this we mean that the certification itself does not leak information about the users' attributes. However, the certification can be secret shared in such a way that the shares can be used to do efficient zero-knowledge proofs on the content of the certification. The secret shares are then distributed among the partial IdPs. After a user has authenticated, the partial IdPs can return the relevant shares to the user and hence allow the user to do efficient zero-knowledge proofs based on the certified attributes. This will allow the user to convince the partial IdPs that the certification they hold fulfil a certain public policy or are consistent with a partial credential that the user wishes the partial IdPs to sign.

For example, the certification could be a Pedersen commitment with the opening information being shared between the servers. This allows the user to carry out zero-knowledge proofs efficiently using Bulletproofs [15]. Bulletproof code is already implemented in the OLYMPUS framework for constructing range proofs. Specifically, this code is used when doing proofs. See section 2.1.3.1 in Deliverable 4.3 for details.

The attribute certification come from an Issuing Authority. However, since these will often only construct signed documents using a standard digital signature scheme, the user must instead construct a zero-knowledge proof that it is in possession of such a signature and the attributes contained in the signed messages match those of a commitment the user has constructed.

This involves a rather expensive type of zero-knowledge proofs involving both algebraic and non-algebraic statements [16]. However, this is only needed

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

during enrolment and not during issuance of credentials or tokens. Thus, we claim this is not a big issue.

We expect to complete and submit a research paper that works out the details of this approach by M38. For now we provide a preliminary version of our worked-out ideas in an appendix to this deliverable. This annex will be made available as confidential in this stage until published in order to avoid possible conflicts in the future publication. In that sense it is not added as annex to this deliverable but in a separate document.

8.1. An alternative approach

While still under active investigation, we note that other user attribute hiding can also be implemented using secure multi party computation. This has the advantage that the user client will not need to perform any heavy computations. These computations will instead be left for the partial IdPs to carry out.

Concretely, the partial IdPs will store the users' attributes in an authenticated and secret shared manner. After a user authenticates and supplies a public policy to the partial IdPs, they will input the authenticated and secret shared user attributes into a secure multi party computation, which will verify that the public policy is fulfilled, and that the attributes have not been manipulated. If the verification is successful, then they will issue a JWT or credential accordingly.

For enrolment there would however still be the need to ensure that whatever document the Issuing Authority have signed will match whatever authenticated secret share the partial IdPs have. This can, in theory, also be handled using secure multi party computation. However, it is still an open question how to do so efficiently.

In general, we are still actively researching how to handle attribute hiding in the most efficient way for the attributes we expect to be used in the OLYMPUS system. We expect to complete and submit a research paper that works out the details of this approach by M38. For now we provide a preliminary version of our worked-out ideas in an annex to this deliverable but in a separate document.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

9. CONCLUSION

This document has described how it is possible to integrate an application with the OLYMPUS framework. The description has shown how it is possible to use the vIdP as part of an OAuth/OIDC based architecture, requiring only changes on the IdP and to some extent the client entities, making the service providers oblivious to the choice of (v)IdP.

The document has shown how to technically modify the software in order to persist data and integrate with HSMs on the IdPs, as well as how the framework allows different cryptographic protocols to be used, oblivious to the application client software.

The document has also shown how the framework is to be used in connection with the two use cases of the project. Demonstrating how the APIs can be used and how the modular approach of the vIdP server allows various instantiations depending on the concrete application.

In addition to the two use cases, a third (minor) use case has been outlined, demonstrating how OIDC and V3C is supported in the framework.

Finally, it has also been discussed how the framework could be expanded to allow hiding of the users' private attributes from the partial IdPs.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

10. REFERENCES

[1] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) Official Journal of the European Union, Vol. L119 (4 May 2016), pp. 1-88

[2] Apache Milagro AMCL, https://milagro.apache.org/docs/amcl-overview/

[3] Carsten Baum, Tore K. Frederiksen, Julia Hesse, Anja Lehmann and Avishay Yanai. Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server. In Cryptology ePrint Archive, Report 2019/1470. https://eprint.iacr.org/2019/1470, 2019.

[4] OAuth 2.0, https://oauth.net/2/

[5] OpenID Connect, https://openid.net/connect/

[6] SAML, https://www.oasis-open.org/standards#samlv2.0

[7] JSON Web Tokens, https://jwt.io/

[8] YubiCo, https://yubico.com/

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

[10] CTAP, https://fidoalliance.org/specifications/

[11] D4.3 Second Reference Implementation of Oblivious IdM, OLYMPUS version 1.0, 2021-05-31

[12] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, Benny Pinkas: Fast Distributed RSA Key Generation for Semi-honest and Malicious Adversaries. CRYPTO (2) 2018: 331-361

[13] Jan Camenisch, Manu Drijvers, Anja Lehmann, Gregory Neven and Patrick Towa. Short Threshold Dynamic Group Signatures. In Cryptology ePrint Archive, Report 2020/016. https://eprint.iacr.org/2020/016

[14] Torben Pedersen: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing**.** CRYPTO 1991: 129-140

[15] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., & Maxwell, G. Bulletproofs: Short proofs for confidential transactions and more. En 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018. pp. 315-334

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725

[16] Melissa Chase, Chaya Ganesh, Payman Mohassel: Efficient Zero-Knowledge Proof of Algebraic and Non-Algebraic Statements with Applications to Privacy Preserving Credentials**.** CRYPTO (3) 2016: 499-530
[17] CBOR Object Signing and Encryption (COSE), RFC 8152, July 2017

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 786725