OLYMPUS

Release v0.1

OLYMPUS consortium

May 19, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

Internet usage is higher than ever and it seems like it will only rise further. In this context, users need to access many services protected by different entities, which demand some form of authentication. What is more, it is quite common that users need to prove some information about themselves, like being over a specific age. Current Identity Management (IdM) systems still fail to truly fulfil security and privacy management requirements, such as unlinkability of users across service providers (SP), hiding SPs from Identity Providers (IdP), selective disclosure of personal data, usability and performance.

User and password systems remain the most accepted and widespread way to authenticate users. In this regard, users are experiencing an explosion of usernames and passwords making it difficult for them to remember all the credentials they have. In most cases, this leads to the use of low quality passwords, and the reuse of passwords (or slight modifications) in different services.

So far, the most successful identity federation systems available are online Single Sign-On solutions. This is convenient for users who do not have to establish separate accounts with separate passwords and are presented with a consistent interface for logging into services. Nevertheless, it is detrimental for security and privacy. Traditional solutions introduce a single-point-of-failure in the system, since the IdP is involved in every authentication to a service provider. The IdP is able to impersonate its users if it acts maliciously. Furthermore, it may act as a "Big Brother" that can track the browsing behaviour of its users and link their accounts across different services. If the IdP is compromised by an attacker, the attacker gains similar privileges, making the IdP a high value target for attacks.

This project implements the core infrastructure for deploying an Single Sign-On service using an OAuth flow using standard JWTs, but offering stronger security guarantees than previous solutions through distributed security and optional cryptographic proofs.

The latest version of this document can always be found here.

1.1 OLYMPUS

OLYMPUS is a project funded by EU under the H2020 programme, whose main objective is establishing an oblivious identity management framework that ensures secure and privacy-friendly virtual identity management interactions for citizens accessing services in Europe, based on novel cryptographic mechanisms. In particular, OLYMPUS employs distributed cryptographic techniques to split up the role of the online IdP over multiple authorities, so that no single authority can impersonate or track its users.

More information can be found on the project website.

The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement No. 786725 (OLYMPUS).

CHAPTER

TWO

KEY CONCEPTS

The main concept of the OLYMPUS framework, is the concept of a distributed virtual IdP (vIdP), replacing the traditional IdP concept.

A traditional IdP is given a username and password (or similar type of credential) and produces an access token. The access token may then be used by third parties, ie. serviceproviders, to verify the identity of the user.

In the OLYMPUS framework, the traditional IdP is replaced by a vIdP (consisting of 2 or more OLYMPUS partial IdPs (pIdP)). By distributing IdP functionality in the vIdP, the OLYMPUS framework offers distributed password verification (where a single pIdP never learns the user's password) and distributed signatures (where all pIdPs must cooperate, in order to produce a valid access token). These features forces an attacker to compromise _all_ pIdPs in the vIdP setup in order to impersonate users or learn their passwords, improving security compared to an traditional IdP.

Furthermore, the OLYMPUS project is currently investigating, how it would be possible to split user attributes in the vIdP, such that a single pIdP does know more about the user, then the username.

These new features do however add to the complexity of the overall federated identity setup. While it is a goal for OLYMPUS to ensure compability with existing IdM technologies, such as OpenID Connect and SAML, the distributed protocols for password verification and signatures, does require a dedicated client application rather than native browser support.

In addition to the vIdP concept, the OLYMPUS framework operates with a couple of other important concepts described in the following sections:

2.1 ID proof

The concept of Identity proofing is the process of getting user attributes into the vIdP. Initially when a user account is been created, it only consists of a username. In order to attach attributes, such as a name, birthdate or nationality to that user account, a user sends an ID Proof to the vIdP. The vIdP can then validate the proof and attach the attributes to the user account.

ID Proofs may take many forms and be anything ranging from X.509 certificates and identity tokens from other IdPs to self claimed (non-validated) user attributes.

The concrete vIdP deployment is responsible for defining which types of ID proofs to accept, as described in the vIdP deployment section.

2.2 Policies

Different service providers, may need different user attributes depending on the usecase. In some cases, the user's age might be relevant, while other cases may be focused on possesion of a drivers license or the nationality of the user. The concept of policies is what allows a relying party to specify what should be revealed as part of the vIdP issued JWT token.

The OLYMPUS framework uses simple policies, serialized as JSON objects, to describe the these requested attributes or properties. Policies are comprised of a *policyId*, used in the dP-ABC approach as a nonce, and a list of predicates. Each predicate defines an operation over a specific attribute, using two mandatory fields (*attributeName* and *operation*) and two fields with attribute values that will be necessary depending on the operation (*value* and *extraValue*).

- Policy Java class

```
public class Policy {

        private List<Predicate> predicates;
        private String policyId;

    public Policy(List<Predicate> predicates, String policyId) {
                super();
                this.predicates = predicates;
                this.policyId = policyId;
        }

        public Policy() {
    }

        public List<Predicate> getPredicates() {
                return predicates;
        }

        public void setPredicates(List<Predicate> predicates) {
                this.predicates = predicates;
        }

        public String getPolicyId() {
                return policyId;
        }

        public void setPolicyId(String policyId) {
                this.policyId = policyId;
        }

}
```

- Serialized policy example

```
{
  "predicates" : [ {
    "attributeName" : "Name",
    "operation" : "REVEAL"
  }, {
    "attributeName" : "Age",
    "operation" : "INRANGE",
    "value" : {
      "attr" : 18,
```

(continues on next page)

(continued from previous page)

```
      "type" : "INTEGER"
    },
    "extraValue" : {
      "attr" : 25,
      "type" : "INTEGER"
    }
  } ],
  "policyId" : "SignedMessage-9235621539"
}
```

2.2.1 Supported predicates

Current implementation supports predicates for the following operations:

- EQ: Check if corresponding attribute is equal to the value in field *value*.
- REVEAL: Reveal the value of the corresponding attribute.
- GREATER_THAN: Check if the corresponding attribute is greater or equal than the value in field *value*.
- LESSTHAN: Check if the corresponding attribute is lesser or equal than the value in field *value*.
- INRANGE: Check if the corresponding attribute is within the range [*value*,*extraValue*].

The last three predicates need to be applied to a "numeric" attribute. Currently, this means it must be an attribute of type *Integer* or *Date*. All of these operations are supported by both PESTO token generation and dP-ABCs, except EQ in dP-ABCs (currently, though it could be partially supported by levering attribute revelation).

2.2.2 Range predicates for dates

Usage of these predicates is pretty straightforward. However, *date* attributes require some extra consideration because of typical use cases for them. Complex proofs of age (or time periods in general) can be created using range (GREATER_THAN, LESSTHAN, INRANGE) predicates, but it is necessary to take into account some ideas.

- If no Age (e.g. 25 years old) attribute is included in the credential (which is better in the long run, allowing more flexibility, avoiding inconsistencies in credential values and real age...), it is necessary to rely on more "static" and "meaningful" attributes like Date of Birth.
- This means "Age" predicates are not as straightforward (e.g., proving someone's age is over 18 is not possible with a simple Age GREATER_THAN 18 predicate).
- However, it is quite simple to generate a predicate with equivalent meaning using the Date of Birth. We can take today's date, subtract 18 years (i.e., res = today – 18 years), and prove that the Date of Birth value in the credential, x, fulfils: x <= res.
- The same process can be applied for "LESS THAN" predicates (and consequently, for "IN RANGE"). In the predicate definition, inequalities are inverted (with respect to the "ideal" abstract predicate we want to check).
- This may seem a bit confusing at first, but it opens up many possibilities. Predicates are not limited to any "timescale", but can use years, months, days and even hours, minutes or seconds (if the correspondent attribute gives such precision). This allows creating predicates for any time frame, which can be useful not only for ages, but any date attribute considered in a use case.

2.3 Offline usage (dp-ABCs)

In some mobile app usecases, it might be relevant to support offline use. In this case, the OLYMPUS framework offers distributed privacy-attribute based credentials (dp-ABCs). A dp-ABC is (in principle) a set of attributes with a special signature from the vIdP. Given a policy, the special signature allows the user client to build a cryptographic proof satisfying the policy, without contacting the vIdP.

This feature does come at a cost: If an attacker gains knowledge of the dp-ABC, it will be able to impersonate the user, hence it must be stored securely. Furthermore, the cryptographic proof constructed from the dp-ABC is not a standard signature (such as RSA or ECDSA), requiring the relying party to use the verifier supplied by the OLYMPUS framework in order to verify the proof.

CHAPTER

THREE

ARCHITECTURE AND CODE

An overview of the OLYMPUS framework architecture can be found in the architecture section while a concrete description of the code can be found in the code repository section. The concrete APIs in the respective API sections.

3.1 Architecture

The following image shows the architecture of the OLYMPUS solution and the flows in an application scenario.

3.1.1 Roles and components

OLYMPUS issuer

The OLYMPUS issuer is a virtual Identity Provider (**vIdP**) comprised of multiple IdPs, offering two alternatives for issuance. An IdP has three key components:

- Distributed Authentication: Handles authentication of users averaging PESTO and other related functionality like managing identity provers.
- Distributed Token Management: It is in charge of distributed token generation (PESTO approach). Each IdP generates a token share given an access policy. When all the necessary token shares have been generated, the user client will be able to combine them into a valid access token to be presented to the Relying Party (RP, also called Service Provide (SP)).
- dP-ABC Credential Management: It is responsible for the management of distributed credentials (dp-ABC approach). As in the distributed tokens approach, each of the IdPs will generate a share (credential share). The user will be able to combine them in a full credential which will contain all the user attributes. Unlike in the previous approach, the credential can be reused several times to derive crypto-proofs to be presented to the relying party.

User

OLYMPUS offers an **user client** which can be integrated in applications willing to use the solution. The client includes:

- Client logic: Handles general client logic like interaction with IdPs...
- Credential management: Present only if the dp-ABC approach is used, manages credential usage (combination presentation generation...).

Relying party (RP), or Service Provider (SP)

Protects access to a range of resources or services. It will generate the access policies that the user will need to satisfy to access to the service. The RP will rely on the verification library provided by OLYMPUS (though for the PESTO approach a standard JWT verifier would suffice) to verify the validity of user presentations.

3.1.2 Flows

The clients will keep an active session such that the password and optional MFA token is only needed to the first API call in a session. After that the password (or retrieved signing key in case of Pesto and dp-ABC) will be cached locally along with a server generates session cookie. For the rest of the session only the username and session cookie is needed by the server to authenticate the client. Exceptions are critical tasks such as deleting the user's or adding/removing MFAs. Furthermore, if distributed password authentication is used (through the Pesto or dp-ABC clients) then signatures on every REST call is needed using the user's retrieved signing key. It should be noted that once an MFA has been associated to a user's account it must **always** be used in order for the user to authenticate. Furthermore, session cookies are used regardless of whether an MFA is associated with the user's account.

3.2 Code repository

The code for the Olympus distributed Single Sign-On service is freely available here where you can also find instructions on compilation. More build instructions are available *here*.

3.2.1 Code organization

The code consists of 3 separate modules: **cfp-usecase**, **core** and **rest-wrapper**. The **cfp-usecase** is a wrapper for applying the Olympus-core to a specific use-case and the **rest-wrapper** is a REST-wrapper for the client-code to allow usage of the client code directly, through REST calls. The main part of Olympus can be found in **core**, which contains code implementing both a client, servers (partial IdPs) and a verifier (Relying Party).

Note that the **core** codebase includes some client and IdP code primarily intended for comparison/benchmarking (the *password jwt* and *distributed rsa* components).

More details about the architecture can be found here

Client

The client code is implemented in the package *eu.olympus.client* and offers lightweight and simple client code used to interact with the servers through REST. Specifically 4 different clients are implemented.

Pesto Client

This client is implemented in the class PestoClient and realizes a JWT client in the setting of multiple servers with distributed password authentication. That is, it implements a new and advanced cryptographic approach to Single Sign-On where a user sends a hidden version of their passwords to multiple servers, which verifies this through an interactive protocol. After verification the servers use a *threshold RSA signature scheme* to construct *partial signatures* on a JWT token they return to the user. The user then combines these partial signatures to construct a *standard* RSA signed JWT token. Details about this can be found in this paper, published at IEEE Euro Security and Privacy 2020 as part of the Olympus project.

dp-ABC Client

This client is implemented in the class PabcClient and realizes a *distributed privacy Attribute Based Credential* client in the setting of multiple servers with distributed password authentication. That is, it implements a new and advanced cryptographic approach to Single Sign-On where a user sends a hidden version of their passwords to multiple servers, which verifies this through an interactive protocol. After verification the servers use multi-signature Pabc scheme to issue a *partial credential* to the user. The user can then combine these partial credentials into a standard credential. The user can store this locally and later non-interactive construct randomized proofs that it holds such a valid credential and optionally also to prove that the credential contains certain attributes. This proof can be given to a verifier to assert that a user is authenticated without any linkability across different verifiers or traceability to the credential issued by the servers. Details about the dp-ABC friendly multi-signature scheme (using Pointcheval–Sanders signatures) can be found in this paper, published at SCN 2020 as part of the Olympus project.

Password JWT Client

This client is implemented in the class PasswordJWTClient and realizes a standard JWT client in a setting where only a *single* server is in play. That is, it implements the standard, non-distributed, approach to Single Sign-On where a user sends their password to a server that verifies against a salted hash stored for that user's account. The server then signs a JWT token accordingly to the user's request. This class is purely used for testing and benchmarking to compare the advanced cryptographic solution with the typical approach to Single Sing-On.

Distributed RSA Client

This client is implemented in the class DistributedRSAClient and realizes a standard JWT client in the setting of multiple servers. That is, it implements the standard approach to Single Sign-On where a user sends their password to multiple servers that each verify the user's password against a salted hash stored for that user's account. However, instead of each server individually signing a JWT token, they use a *threshold RSA signature scheme* to construct *partial signatures* on a JWT token which they return to the user. The user then combines these partial signatures to construct a *standard* RSA signed JWT token. This class is purely used for testing and benchmarking.

Server

The server code is implemented in the package *eu.olympus.server* and offers code to interact will all the clients mentioned above through 3 different IdP classes, each using one of two possible Authentication Handlers underneath. One authentication handler called *PasswordAuthenticationHandler* for handing classical password verification where a password is sent in plain to the server and verified locally (which is the case for the Password JWT and Distributed RSA clients). Another authentication handler called *PestoAuthentication* used for distributed and secure password verification (which is the case for the Pesto Client and dp-ABC Client).

Pesto IdP

This server is implemented in the class PestoIdpImpl and realizes a distributed password verification and token issuance service. If the Pesto Client is used, then the token is a JWT token constructed using distributed RSA through the class *ThresholdRSAJWTTokenGenerator*, whereas if the Pabc Client is used, then the token is a partial credential constructed using *ThresholdPSSharesGenerator*.

Password JWT IdP

This server is implemented in the class PasswordJWTIdP and realizes a standard JWT issuance by verifying the user's password against a salted hash digest.

Distributed RSA IdP

This server is implemented in the class DistributedRSAIdP and realizes a distributed JWT issuance service based on a local verification of the user's password against a salted hash digest.

Verifier

The verifier code is implemented in the package *eu.olympus.verifier* and offers code to verify a token constructed by the server(s) and processed at the client. Two different verifiers are implemented depended on the type of token

JWT Verifier

This verifier is implemented in the class *JWTVerifier* and verifies a standard, RSA signed JWT token and is agnostic to whether it has been constructed in a distributed manner or by a single server. Since this is based on standards, this class is simply a wrapper for methods from the JWT library offered by Auth0.com.

dp-ABC Verifier

This verifier is implemented in the class *PSPABCVerifier* and verifies that a randomized credential is valid. It is furthermore also able to verify certain proofs of the content of the credential the user holds, according to what the user wishes the verifier to learn.

3.2.2 REST

The server contact is implemented via REST with optional TLS integration. Furthermore, inter-server communication is also implemented through a (mutually authenticated) REST connection. Mapping Java classes to REST and JSON friendly objects are done through the classes in package *eu.olympus.model*, whereas the actual REST wrapping is realized through the servlet classes in package *eu.olympus.rest*, in particular *PasswordIdPServlet* and *PestoIdPServlet*.

3.2.3 Libraries

Both the distributed password verification and credentials use advanced elliptic curve constructions, hence the code depends on multiple cryptographic libraries. Specifically Bouncy Castle and AMCL Miracl Core.

3.3 Client API

We here outline the different methods afforded by the clients in the Olympus project.

3.3.1 Global client API

All clients have the following API handles available. Each can be made accessible through REST by using the *rest-wrapper* component. Otherwise they can be accessed through calls to the specific client classes in package *eu.olympus.client*.

Create User (Username, Password)

Creates a new user account based on a Username and Password. Fails if an account already exists with the given Username.

Create User and Add Attributes (Username, Password, idProof)

Creates a new user account based on a Username and Password and associates certain attributes with this user's identity based on the data contained in idProof. Fails if an account already exists with the given Username or if the proof of the user's attributes is not accepted.

Add Attributes (Username, Password, Token, TokenType, idProof)

Authenticates an already existing user based on their Username, Password and optionally a MFA token of a certain type. If the authentication succeeds then the attributes in idProof is added to the user's account, assuming that these attributes can be verified.

Authenticate (Username, Password, Policy, Token, TokenType)

Authenticates an already existing user based on their Username, Password and optionally a MFA token of a certain type. If the authentication succeeds then a token is constructed and returned to the user, based on the user's stored attributed in accordance with the Policy supplied. That is, the token may or may not contain information based on the user's attributes as specified by the Policy.

GetAllAttributes (Username, Password, Token, TokenType)

Authenticates an already existing user based on their Username, Password and optionally a MFA token of a certain type and returns all the verified attributes a user has stored with their account.

DeleteAttributes (Username, Password, Attributes, Token, TokenType)

Authenticates an already existing user based on their Username, Password and optionally a MFA token of a certain type and deletes the Attributes specified from the user's account.

DeleteAccount (Username, Password, Token, TokenType)

Authenticates an already existing user based on their Username, Password and optionally a MFA token of a certain type and deletes the user's account completely.

ChangePassword (Username, OldPassword, NewPassword, Token, TokenType)

Authenticates an already existing user based on their Username, Password and optionally a MFA token of a certain type and changes the the user's password from OldPassword to NewPassword.

RequestMFA (Username, Password, TokenType)

Authenticates an already existing user based on their Username and Password and returns an MFA challenge for an MFA authenticator of a certain type.

ConfirmMFA (Username, Password, Token, TokenType)

Authenticates an already existing user based on their Username and Password and an MFA token constructed based on the a MFA challenge received by a call to RequestMFA. If the token is verified then the MFA of the TokenType is added to the user's account such that it is always required to use this to authenticate in the future.

RemoveMFA (Username, Password, Token, TokenType)

Authenticates an already existing user based on their Username and Password and an MFA token of TokenType. If the verification is successful then the MFA of TokenType is removed from the user's account such that it is not needed in the future.

ClearSession

Removes the current session stored on the client. The client is able to cache the cryptographic material associated with a given user along with a cookie for an active session between the user and the IdPs. This call removes all state of the client, including credentials in the case of a dp-ABC client.

3.4 IdP API

We here go through the REST end-points available at the servers realizing a (virtual) IdP.

3.4.1 Global IdP API

All IdPs have the following API handles available. Each are accessible through REST. Otherwise they can be accessed through calls to the specific IdP classes in package *eu.olympus.server*.

GetCertificate

Returns a certificate containing the server's public Signature verification key, used to verify client tokens. If a distributed IdP is in place, this will be the same key for all servers.

3.4.2 Administration

Start Refresh

The server will initiate a key-share refresh with the other servers.

3.4.3 Pesto and pABC IdPs

The following methods are specific for the Pesto and pABC IdP setting and are intended to be call by the relevant client.

Perform OPRF (Username, Nonce, Cryptographic value, Token, TokenType)

Performs an OPRF operation on a cryptographic value based on a nonce (constructed using the user's password). If an account for a user with the given Username already exists then the optional Token of TokenType is also validated before carrying out the OPRF operation. The user will use the output to (re)generate a public/private signing key pair.

Finish Registration (Username, Cookie, Public Key, Signature, Nonce, idProof)

Finishes a user account registration by storing a Public Key associated with Username and adding attributes in the optional idProof (assuming they can be verified). However the Signature on the nonce, idProof and Username must first be verified with the Public Key. Furthermore, it must also be verified that Cookie is valid (i.e. registration phase for user has been initiated). The Public key will be associated with Username and used to verify future authentication attempts.

AddAttributes (Username, Cookie, Nonce, Signature, idProof)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce and idProof. If the authentication succeeds then the attributes in idProof is added to the user's account, assuming that these attributes can be verified.

GetAllAttributes (Username, Cookie, Nonce, Signature)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username and Nonce and returns all the verified attributes a user has stored with their account.

DeleteAttributes (Username, Cookie, Nonce, Signature, Attributes)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce and the Attributes. It then deletes the Attributes specified from the user's account.

DeleteAccount (Username, Cookie, Nonce, Signature)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce and deletes the user's account completely.

ChangePassword (Username, NewSignature, OldSignature, New Public Key, Cookie, Nonce)

Authenticates an already existing user based on their Username, a session Cookie and two signatures (NewSignature and OldSignature) on the Username, Nonce and New Public Key. If the check succeeds it associates New Public Key with Username.

RequestMFA (Username, Cookie, Nonce, Signature, TokenType)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce and TokenType. If successful it returns an MFA challenge for an MFA authenticator of TokenType.

ConfirmMFA (Username, Cookie, Nonce, Signature, Token, TokenType)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce, Token and TokenType. If successful it verifies the MFA Token based on the a MFA challenge received by a call to Request MFA. If the token is verified then the MFA of the TokenType is added to the user's account such that it is **lways** required to authenticate in the future.

RemoveMFA (Username, Cookie, Nonce, Signature, Token, TokenType)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce, Token and TokenType. If the verification is successful then the MFA of TokenType is removed from the user's account such that it is not needed in the future.

RefreshCookie (Cookie)

Validates that the Cookie is still valid and returns a new cookie if so which renewed validity time.

3.4.4 Extra calls to Pesto IdPs

On top of the methods above a Pesto IdP also affords the following methods:

Authenticate (Username, Cookie, Nonce, Signature, Policy)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce and Policy. The Signature is verified against the public key associated with Username. If the authentication succeeds then a token is constructed and returned to the user, based on the user's stored attributes in accordance with the Policy supplied. That is, the Policy specifies which user attributes are included in the token (if any).

3.4.5 Extra calls to pABC IdPs

On top of the methods above a pABC IdP also affords the following methods:

Get dp-ABC Public Parameters

Returns the dp-ABC specific parameters of the servers.

GetCredential (Username, Cookie, Nonce, Signature)

Authenticates an already existing user based on their Username, a session Cookie and a Signature on the Username, Nonce and Policy. The Signature is verified against the public key associated with Username. If the authentication succeeds then a credential based on all the user's attributes is constructed and returned to the user.

3.4.6 IdP 2 IdP

Add Partial Server Signature (Username, PartialSignature)

Receive a Partial server Signature on a Username from another server. This is used during user account creation or user password change to ensure that each server knows that all servers agree that they have received a request from a given user.

Add Partial MFA Secret (Username, Secret, TokenType)

Receive a partial secret for an MFA of TokenType for the user with a specific Username. This is used during MFA request by a user to construct a shared MFA secret among the servers.

Add Session (Cookie, Authorization)

Receive a Cookie and an Authorization. The Authorization specifies an ID, a set of roles (user, administrator or IdP) and an expiration time. A caller can the supply this Cookie to REST calls to prove that they are authorized to make these calls. This is used to setup the IdPs and to do key-share refresh.

Add Mastershare (Nonce share, Key share)

Receive a share of a nonce and the server's master key. This is used during key refreshing to receive nonce and master key shares from the other partial IdP servers allowing the given server to update its signing and OPRF keys by restoring its master key share.

Add Key Share (ID, Key share)

Receive a sharing of an other partial IdP's master key share. This is used during key share refresh to allow each server to keep a secret shared backup of their master key share on the other servers.

3.4.7 Password JWT and Distributed RSA IdPs

The following methods are specific for the Password JWT and Distributed RSA IdP setting.

CreateUser (Username, Password)

Creates a new user account based on a Username and Password. Fails if an account already exists with the given Username.

CreateUserAndAddAttributes (Username, Password, idProof)

Creates a new user account based on a Username and Password and associates certain attributes with this user's identity based on the data contained in idProof. Fails if an account already exists with the given Username or if the proof of the user's attributes is not accepted.

StartSession (Username, Password, Token, TokenType)

Start a new session of an already existing user based on their Username, Password and optionally a MFA token of a certain type (required if an MFA of TokenType has been added for that user). If the authentication succeeds then a time-constrained session Cookie is returned to the user which they use to authenticate themselves towards the server for the rest of the session.

ValidateSession (Cookie)

Verifies that a session Cookie is still valid. If it is, then its lifetime is renewed.

Authenticate (Username, Cookie, Policy)

Authenticates an already existing user based on their Username and a session Cookie. If the authentication succeeds then a token is constructed and returned to the user, based on the user's stored attributed in accordance with the Policy supplied. That is, the token may or may not contain information based on the user's attributes as specified by the Policy.

Add Attributes (Username, Cookie, idProof)

Authenticates an already existing user based on their Username and a session Cookie. If the authentication succeeds then the attributes in idProof is added to the user's account, assuming that these attributes can be verified.

GetAllAttributes (Username, Cookie)

Authenticates an already existing user based on their Username and a session Cookie and returns all the verified attributes a user has stored with their account.

DeleteAttributes (Username, Cookie, Attributes)

Authenticates an already existing user based on their Username and a session Cookie and deletes the Attributes specified from the user's account.

DeleteAccount (Username, Password, Cookie)

Authenticates an already existing user based on their Username, Password and a session Cookie and deletes the user's account completely.

ChangePassword (Username, OldPassword, NewPassword, Cookie)

Authenticates an already existing user based on their Username, OldPassword and a session Cookie and replaces the user's OldPassword with a NewPassword.

RequestMFA (Username, Password, Cookie, TokenType)

Authenticates an already existing user based on their Username, Password and a session Cookie and returns an MFA challenge for an MFA authenticator of a certain type.

ConfirmMFA (Username, Password, Cookie, Token, TokenType)

Authenticates an already existing user based on their Username, Password, a session Cookie and an MFA token constructed based on the a MFA challenge received by a call to RequestMFA. If the token is verified then the MFA of the TokenType is added to the user's account such that it will be **required** to use to authenticate in the future.

RemoveMFA (Username, Password, Cookie, Token, TokenType)

Authenticates an already existing user based on their Username and Password, a session Cookie and an MFA token of TokenType. If the verification is successful then the MFA of TokenType is removed from the user's account such that it is not needed in the future.

3.5 Verifier API

The verifiers in the project are mostly minimally viable implementations that needs to be extended by any RP that wishes to integrate with Olympus. However, tokens created by Password JWT, Distributed RSA or Pesto client/IdP are fully JWT compatible using RSA 2048 and thus can be verified using any library supporting this format. The content of the token must still be processed appropriately though, e.g. as an Open ID Connect token. The proof-of-concept verifier for JWT can be found in the package *eu.olympus.verifier* in the class JWTVerifier, which takes a Token as input and verifies the signature on this against a provided public key.

For dp-ABC tokens a class PSPABCVerifier is also provided in the package *eu.olympus.verifier* which must be setup using Olympus specific parameters and key material, that can be obtained through the IdP REST handles. This verifier can verify a presentation token (constructed by the dp-ABC client) according to a specific, public Policy. This policy specifies which of the user's attributes should be leaked to the verifier. However, the policy also allows the user to specify revealing a range membership of one of its attributes, instead of the exact value. This can for example be used with simple integer ranges, but also dates and thus allows increased privacy for the user towards the verifier while still allowing the verifier to assert that the needed constraints are met. This can e.g. be used to verify a user is above the age of 18 (assuming that the virtual IdP holds the user's birthday).

CHAPTER

FOUR

GETTING STARTED

You have read basic information of the OLYMPUS components, architecture, APIs and how the code is organized. Now, you are ready to get started with trying the OLYMPUS code. This section describes how to install and build the elements in the repository, as well as the steps necessary to configure and deploy your own virtual Identity Provider and Client.

4.1 Installation and build

The code for the Olympus distributed Single Sign-On service is freely available here.

To build the whole project (note that executing the commands in module subdirectories allows building only those modules – and their dependencies), use the commands:

```
mvn clean
mvn install
```

Note that the 'mvn clean' command is needed to install the MIRACL jar dependency into the local m2 repository in order to build the project.

Note that 'mvn install' builds a Docker test setup, so Docker must be running while running this command.

DO NOT skip the tests in 'mvn install'. The tests are necessary for constructing tests keys that is compiled into the Docker Images.

Two demonstration cases are included in this project: an example deployment of a virtual IdP for a specific use case (for both PESTO and dp-ABC approaches) and a demonstrator of the application of OLYMPUS as an identity provider in an OIDC flow. Once the installation process is finished, you can try them following the instructions in the repository. These demonstrators can serve as a base example for the configuration and deployment steps explained in following sections.

4.2 Deploying a vIdP

An OLYMPUS virtual IdP (vIdP) consists of a number of components working together. Some of these are delivered as part of the OLYMPUS framework, whereas others are specific for a concrete application and must therefore be implemented by the application developer. The main components are the following:

- Configuration handling
- Identity proofing
- Cryptomodule and storage
- Multifactor authentication

- REST interface deployment

This document introduces the various components, along with small samples. A full example can be found in the sourcecode as TODO and in the end of this page.

4.2.1 Configuration handling

The vIdP needs a number of parameters, such as key material, lifetime of issued tokens and information on the other servers, as part of the setup procedure. For full functionality of the distributed IdP, a PABCConfiguration should be used. This can either be provided as a custom implementation or using the standard PABCConfigurationImpl class provided by the OLYMPUS framework. The standard PABCConfigurationImpl can be provided using JSON (a full working set of configuration files can be found in test/resources):

```
       {"keyStorePath":"src/test/resources/keystore.jks",
        "keyStorePassword":"server1",
        "trustStorePath":"src/test/resources/keystore.jks",
        "trustStorePassword":"server1",
        "port":9080,
        "myAuthorizationCookie":
↪"hk5NQgrO9BJlBedHaD8Qro8rc6SGL65H9W90Wt2snpr2cKwnSQ+C2bK55F2sKZRa0gxAkKz6P5mu3Nk9WKLGSg==
↪",
        "authorizationCookies":{
        "5tabBLiLXuD1eWTTliSHvkfVLYr9+lhs0WhPpET3NGBCp3r4onpzRUh2/
↪tyZvdW6fDp1yr2uAePpGPvXnhkwCw==":{"id":"server1", "roles":["SERVER"]},
        "WS66b+TWK3XHHLJW3YHJ3KGJk0wEfeTmxouRFVc+/w9hps+OcL2JwxVW5kK6wH1tD3jAmD18yv/
↪6fs3308vYDw==":{"id":"server2", "roles":["SERVER"]},
        "8Y9mocwbGZbU0YSNQR46kb6DHYuniHqpXmOjM2uUQ+iEmlX/ka4ZPzBjgrWz9Zw/
↪zeNA4Neq9LSLAaPa6+B0Vg==":{"id":"administrator", "roles":["ADMIN"]}},
        "servers":["http://127.0.0.1:9081","http://127.0.0.1:9082"],
        "tlsPort":9933,
        "keyMaterial": {"modulus":209145188754454650...
↪475716953100736172823863436754298814161 7, "privateKey":1462249078153...0513422592,
↪"publicExponent":65537},
        "oprfBlindings":{"1":162683617192684354721534228919969509586,"2
↪":201103837588655148085398197875718785305},
        "rsaBlindings":{"1":116250903442045051614123370266866563879,"2
↪":139820269263465496115751481575796568221},
        "localKeyShare":"56tnFNMZfBX/...
↪fqJHkH4y4eVEcvadnSv3iqmS6v7plckHPasX5kpxPqbmg07Zkb/WO0hrWWif08o/
↪ceGRW9m9yadC5UG0l4aQuA==",
        "remoteShares":{"1":"iHOmSPB1DxgImGYJTv1rKk2wPZ4pMRa3...
↪3FpFdBSuRLxtslThgNqn5Bkhhx8y6NXkWL/NXmC2F7yGxgn75+Q==", "2":
↪"nhVAQ6gKlD8XxfjPMLLM+rbW39Iex3...UU5m8FPeYygiYkEJJCchTHn33dA=="},
        "oprfKey
↪":22726316559010596523768535026535856125560373627562633952896888222674336938455099155145599187801 6
↪
        "refreshKey":"AAAAAA==",
        "id":0,
        "waitTime":1000,
        "attrDefinitions":[ {"id":"url:PurchaseTime", "shortName":"Time of Purchase",
↪"minDate":"2000-01-01T00:00:00", "maxDate":"2021-02-01T00:00:00", "type":"Date"},
          {"id":"url:DateOfBirth", "shortName":"Date of Birth", "minDate":"1950-01-
↪01T00:00:00", "maxDate":"2021-09-01T00:00:00", "granularity":"DAYS", "type":"Date"}
          {"id":"url:IntegerWithNegatives", "shortName":"Int", "minimumValue":-20,
↪"maximumValue":300, "type":"Integer"},
          {"id":"url:HasDrivingPermit", "shortName":"Has Driving Permit", "type":
↪"Boolean"},
```

(continues on next page)

(continued from previous page)

```
          {"id":"url:FamilyName", "shortName":"Family Name", "minLength":2, "maxLength
↪":16, "type":"String"} ],
        "seed":
↪"cmFuZG9tIHZhbHVlIHJhbmRvbSB2YWx1ZSByYW5kb20gdmFsdWUgcmFuZG9tIHZhbHVlIHJhbmRvbQ==",
        "lifetime":72000000,
        "allowedTimeDifference":10000}
```

4.2.2 Identity proofing

Identity proofing is the process of getting user attributes into the vIdP. After a user account has been created, a user may send an IdentityProof to the vIdP. If the vIdP can validate the proof, the attributes in the IdentityProof is then stored as part of the user's account. The IdentityProof is essentially just a key-value mapping of attributes along with some kind of signature. This can be implemented done in many different ways, using JWT, signed XML or many other schemes, and is left up to the application developer to implement.

For each identity proofing scheme the application should support, the application develop must implement two components, an implementation of the IdentityProver interface and a matching implementation of the IdentityProof interface. The following is a very basic example, using a key-value map of attributes along with a signature string as an IdentityProof and a matching IdentityProver, that verifies that the signature is "SIGNATURE" and stores the attributes.

- IdentityProof:

```
      @JsonTypeInfo(use=Id.CLASS, include=As.PROPERTY, property="@class")
      public class DemoIdentityProof extends IdentityProof {
         private String signature;
         @JsonTypeInfo(use=Id.CLASS, include=As.PROPERTY, property="class")
         private Map<String, Attribute> attributes;

         public DemoIdentityProof() {
         }

         public DemoIdentityProof(String signature, Map<String, Attribute>␣
↪attributes) {
            super();
            this.signature = signature;
            this.attributes = attributes;
         }

         public Object getSignature() {
            return signature;
         }

         public void setSignature(String signature) {
            this.signature = signature;
         }

         public Map<String, Attribute> getAttributes() {
            return attributes;
         }

         public void setAttributes(Map<String, Attribute> attributes) {
            this.attributes = attributes;
         }
      }
```

- IdentityProver:

```
public class DemoIdentityProver implements IdentityProver {

   private Storage storage;

   public DemoIdentityProver(Storage storage) {
      this.storage = storage;
   }

   //Only validates that the proof is a TokenIdentityProof
   @Override
   public boolean isValid(String input, String username) {
      ObjectMapper mapper = new ObjectMapper();
         try {
            mapper.readValue(input, DemoIdentityProof.class);
         } catch (IOException e) {
            return false;
         }
         return true;
   }

   @Override
   public void addAttributes(String input, String username) {
      ObjectMapper mapper = new ObjectMapper();
      DemoIdentityProof proof;
      try {
         proof = mapper.readValue(input, DemoIdentityProof.class);
         storage.addAttributes(username, proof.getAttributes());
      } catch (IOException e) {
      }
   }
}
```

4.2.3 Cryptomodule and storage

In some applications there may be requirements to storing sensitive information in special hardware. In order to support this, the application developer should make an implementation of the relevant storage interface (in order to persist data) and may implement a custom implementation of the ServerCryptoModule interface.

The storage interface allows the application developer to use their choice of database. The OLYMPUS framework does contain an InMemory implementation of the relevant interface, however as the name implies, this does not persist data after restarts.

The ServerCryptoModule allows integration with HSM modules or other strong cryptographic solutions. The OLYMPUS framework contains a software implementation of the interface and may be used if HSM support is not required.

The following code sample will create an InMemoryDatabase and a SoftwareServerCryptoModule:

```
PestoDatabase db = new InMemoryPestoDatabase();
ServerCryptoModule cryptoModule = new SoftwareServerCryptoModule(new␣
↪SecureRandom());
```

4.2.4 Multifactor authentication

Some applications may require the use of multifactor authentication to provide an additional layer of security. Similar to the processes around identity proofing, this can be done in a number of ways and is therefore left for the application developer to implement. Each authenticator implementation must implement the MFAAuthenticator interface, which specifies 4 main methods:

- generateTOTP(String secret) - Given some private keymaterial, generate a one-time key.
- isValid(String token, String secret) - Validate if the provided token is valid with regard to the key material.
- generateSecret() - Generate private key material.
- combineSecrets(List secrets) - Given a list of partial key material, combine it into a single key.

Note that since the vIdP is distributed, each partial IdP need to generate "trusted" key material, which is then combined to the key material that is actually used. In some cases, this may not be necessary, in which case the combineSecrets method should be implemented accordingly.

The following sample code shows how a simple TOTP based Authenticators could be implemented. Note that each type of authenticator should have some unique identifier, so the the clients can specify which type of authenticator is used:

```
        public class TOTPAuthenticator implements MFAAuthenticator {

            public static final String TYPE = "TOTP_AUTHENTICATOR";
            private static final int BYTES_IN_SECRET = 20;
            private final CommonCrypto crypto;
            private final Base32 base32 = new Base32();

            public TOTPAuthenticator(CommonCrypto crypto) {
                this.crypto = crypto;
            }

            @Override
            public boolean isValid(String token, String secret) {
                String totp = generateTOTP(secret);
                return totp.equals(token);
            }

            @Override
            public String generateTOTP(String secret) {
                byte[] bytes = base32.decode(secret);
                String hexKey = Hex.encodeHexString(bytes);
                return TOTP.getOTP(hexKey);
            }

            @Override
            public String generateSecret() {
                return base32.encodeToString(crypto.getBytes(BYTES_IN_SECRET));
            }

            @Override
            public String combineSecrets(List<String> secrets) {
                byte[] combinedSecret = base32.decode(secrets.remove(0));
                while(secrets.size() > 0) {
                    combinedSecret = Util.xorArray(combinedSecret, base32.decode(secrets.
↪remove(0)));
                }
```

(continues on next page)

(continued from previous page)

```
            Base32 base32 = new Base32();
            return base32.encodeToString(combinedSecret);
        }
    }
```

4.2.5 REST based deployment

The OLYMPUS framework offers a basic REST server (RESTIdPServer), allowing for the vIdP functionality to be exposed as a series of REST endpoints. The RESTIdPServer takes an VirtualIdP as argument and is afterwards configured with relevant ports for HTTP/HTTPS communication as well as a trust- and keystores for TLS connectivity.

```
    RESTIdPServer restServer = new RESTIdPServer();
    restServer.setIdP(idp);
    List<String> servlets = new ArrayList<>(1);
    servlets.add(PestoIdPServlet.class.getCanonicalName());
    String keyStorePath = "some/path";
    restServer.start(8080, servlets, 443, keyStorePath, "keyStorePassword",
↪"trustStorePassword");
```

4.2.6 Complete example

The following tutorial code shows how a partial IdP may be started using components included in the framework. Note that it is uses the previously described DemoIdentityProvider and TOTPAuthenticator. While the TOTPAuthenticator is "secure", the DemoIdentityProvider does no validation and should only be used for testing. A slight modified version of the example code may also be found in the source code as the RunOIDCServer or RunCFPServer classes in the oidc-demo-idp and cfp-usecase projects.

```
public class RunTestServer {

        /**
         * Main method to start a server. Takes the path to a configuration file as␣
↪a paramenter. If no paramenters
         * are given, it looks for the location of the configuration file in the␣
↪ENV variable CONFIG_FILE
         */
        public static void main(String[] args) throws Exception {
            ObjectMapper mapper = new ObjectMapper();
            PABCConfigurationImpl configuration = null;
            if (args.length == 0) {
                String configFile = System.getenv("CONFIG_FILE");
                configuration = mapper.readValue(new File(configFile),␣
↪PABCConfigurationImpl.class);
            } else {
                configuration = mapper.readValue(new File(args[0]),␣
↪PABCConfigurationImpl.class);
            }

            List<PestoIdP> others = new ArrayList<PestoIdP>();
            for (String s: configuration.getServers()) {
                others.add(new PestoIdP2IdPRESTConnection(s, configuration.getId(),␣
↪configuration.getKeyStorePath(), configuration.getKeyStorePassword(),
                    configuration.getTrustStorePath(), configuration.
↪getTrustStorePassword(), configuration.getMyAuthorizationCookie()));
```

(continues on next page)

(continued from previous page)

```
            }

            //Setup databases
            PestoDatabase db = new InMemoryPestoDatabase();

            //Setup identity provers
            List<IdentityProver> identityProvers = new LinkedList<IdentityProver>();
            identityProvers.add(new DemoIdentityProver(db));

            ServerCryptoModule cryptoModule = new SoftwareServerCryptoModule(new␣
↪SecureRandom());

            List<String> types = new ArrayList<>(1);
            types.add(PestoIdPServlet.class.getCanonicalName());

            //Setup the IdP.
            OIDCPestoIdPImpl idp = null;
            Map<String, MFAAuthenticator> authenticators = new HashMap<>();
            authenticators.put(TOTPAuthenticator.TYPE, new TOTPAuthenticator());

            idp = new OIDCPestoIdPImpl(db, identityProvers, authenticators,␣
↪cryptoModule);
            idp.setup("ssid", configuration, others);
            //And also an in memory database for authorization of servers and admins
            for(String cookie: configuration.getAuthorizationCookies().keySet()) {
               Authorization authorization = configuration.getAuthorizationCookies().
↪get(cookie);
               authorization.setExpiration(System.currentTimeMillis()+604800000); //
↪valid for one week
               idp.addSession(cookie, authorization);
            }

            RESTIdPServer restServer = new RESTIdPServer();
            restServer.setIdP(idp);
            try {
               restServer.start(configuration.getPort(), types,
                  configuration.getTlsPort(),
                  configuration.getKeyStorePath(),
                  configuration.getKeyStorePassword(),
                  configuration.getTrustStorePassword());
            }catch(Exception e) {
         }
      }
```

4.3 Client configuration and use:

4.3.1 Configuration

Configuring an OLYMPUS client is fairly simple. When the application client instantiates the relevant type of OLYMPUS client, eg. PestoClient or PabcClient, it must provide a) a list of IdP servers, b) a client cryptomodule and c) a CredentialManagement implementation in the case of the PabcClient.

- a) The list of servers should be provided as a list of PestoIdPRestConnection objects, each taking a URL, an access token and an id parameter. Note that the access token is only used in connection with starting the

key-refreshing protocol, and should be null for non-admin users/applications. The PestoIdPRestConnect is essentially just a REST wrapper for the IdP functionality. An application developer may therefore implement a different wrapper is necessarry.

- b) A SoftwareClientCryptoModule may be used as the cryptomodule. As the name suggests, the SoftwareClientCryptoModule offers a software implementation of various necessary cryptographic operations. If desired, a custom implementation of ClientCryptoModule may be used, allowing the application to utilize hardware based cryptographic operations.
- c) As the PabcClient operates with credentials in order to support offline usage, the PabcClient needs an CredentialManagement interface to access these credentials. The PSCredentiaManagement class may be used for this purpose. Note that the PSCredentialManagement class requires a storage component. The InMemoryCredentialStorage can be used for this, however this storage is *not* secure and should not be used outside of testing. Instead a custom implementation should be used, allowing the credentials to be stored in a secure manner, ie. in hardware.

Sample code of configuring a PestoClient and PabcClient, to use a vIdP located at https://127.0.0.1:8090, https://127.0.0.1:8091 and https://127.0.0.1:8092 may be found below:

```
        List<PestoIdP> idpList = new ArrayList<>();
        idpList.add(new PestoIdPRESTConnection("http://127.0.0.1:8090", null, 0));
        idpList.add(new PestoIdPRESTConnection("http://127.0.0.1:8091", null, 1));
        idpList.add(new PestoIdPRESTConnection("http://127.0.0.1:8092", null, 2));
        ClientCryptoModule cryptoModule = new SoftwareClientCryptoModule(new Random(0),
↪ modulus);
        //PestoClient:
        UserClient client = new PestoClient(idpList, cryptoModule);
        //PabcClient:
        PSCredentialManagement credentialManagement = new PSCredentialManagement(true,␣
↪new InMemoryCredentialStorage());
        credentialManagement.setup(publicParam, publicKeys, seed);
        UserClient client = new PabcClient(restIdps, credentialManagement,␣
↪cryptoModule);
```

4.3.2 Usage

After the appropriate OLYMPUS client has been initialized, it is possible to access the methods of the UserClient interface, e.g. createUser(...), proveIdentity(...), authenticate(...), etc. The calls have a number of parameters in common:

- Username - The user's account name
- Password - The user's password
- Token - In case MFA is enabled, this is the OTP code or response to a challenge. If MFA is not enabled for the account, a null value can be used.
- Type - In case MFA is enabled, this should is the MFA scheme name. The concrete value is dependant on the vIdP deployment and is further described in the IdP section. In case MFA is not used, the string "NONE" must be used.
- IdentityProof - In order to attach attributes to an account an IdentityProof must be supplied. The concrete instantiation of the IdentityProof to use, is dependant on the vIdP deployment and is further described in the IdP section.
- Policy - The policy specifies what the output of the authenticate method should contain. The policy contains an identifier and a list of predicates. A Predicate consists of an attribute name, an operation and a value used with comparing operations.

The following sample code will create a new user account (named “Bob” with the password “Secret”). After that, it will enable MFA, with a Google Authenticator app and finally it will attempt to authenticate, with a policy comparing the user’s age with 18 and revealing the name of the user.

```
        client.createUser("Bob", "Secret");
        String challenge = client.requestMFAChallenge("Bob", "Secret", "GOOGLE_
↪AUTHENTICATOR");
        String mfaToken = ... // Use the challenge and a Google Authenticator app to␣
↪generate the mfaToken out of band.
        client.confirmMFA("Bob", "Secret", mfaToken, "GOOGLE_AUTHENTICATOR");
        List<Predicate> predicates = new ArrayList<>();
        predicates.add(new Predicate("Name", Operation.REVEAL, null));
        predicates.add(new Predicate("Age", Operation.GREATERTHAN, new Attribute(18)));
        Policy policy = new Policy(predicates, "policy-identifier");
        String token = client.authenticate("Bob", "Secret", policy, mfaToken, "GOOGLE_
↪AUTHENTICATOR");
```