OpenID Connect Provider Certification

Master's Thesis
University of Turku
Department of Future Technologies
Software Engineering
2019
Anssi Kivinen

UNIVERSITY OF TURKU
Department of Information Technology

ANSSI KIVINEN: OpenID Connect Provider Certification

Master's Thesis, 79 p., 0 app. p.
Software Engineering
February 2019

The thesis looks into authentication and authorization theory and reviews some protocols used for identity management. The most important protocols in the thesis are OAuth 2.0 and OpenID Connect.
The method of research used in the thesis is literature review, where a set of selected items are examined. Many of the items are technical documentation, which were then used to build an overview of the OpenID Connect authorization framework, as well as a set of requirements for the OpenID Connect Provider certification.
The thesis also provides a practical view of the OpenID Connect Provider certification process and an analysis of the OpenID Connect Provider implementation in the Trivore Identity Service platform in terms of the certification requirements. After analysing the implementation, recommendations on improvements to meet the certification requirements are given.
The implementation already conforms to the Config profile. However, the implementation has to be improved to properly conform to the Basic, Implicit, Hybrid, and Dynamic conformation profiles. For basic and implicit profiles, the session user session management should be improved. Additionally, support for the hybrid authorization flow and dynamic client creation should be added as well as.

Keywords: Identity and Access Management, OpenID Connect, Software Certification, OAuth 2.0

TURUN YLIOPISTO
Informaatioteknologian laitos

ANSSI KIVINEN: OpenID Connect Provider -toteutuksen sertifiointi

Diplomityö, 79 s., 0 liites.
Ohjelmistotekniikka
Helmikuu 2019

Diplomityössä tarkastellaan tunnistautumisen ja auktorisoinnin teoriaa sekä lisäksi joitain identiteetinhallinnassa käytettäviä protokollia. Lopputyössä esitellyistä protokollista tärkeimmät ovat OAuth 2.0 ja OpenID Connect.
Diplomityössä käytetään tutkimusmenetelmänä kirjallisuuskatsausta, jonka puitteessa käydään läpi muun muassa teknisiä dokumentaatiota. Dokumentaatioita käytetään apuna muodostaessa yleiskuvaa OpenID Connect -protokollasta ja etenkin sen sertifiointiin liittyvistä yksityiskohdista.
Diplomityössä käydään läpi myös käytännönläheisesti OpenID Connect Provider -toteutuksen sertifiointiprosessia. Lisäksi tarkastellaan Trivore Identity Servicen OpenID Connect Provider -toteutuksen tilannetta sertifioinnin näkökulmasta. Työn lopussa annetaan ehdotuksia toteutuksen parantamiseksi, jotta sen sertifiointi olisi mahdollista.
Työssä tarkasteltava toteutus noudattaa jo Config-profiilin mukaisia määrityksiä, mutta muiden määrityksien noudattaminen vaatii toteutukselta joitain parannuksia. Basic- ja Implicit-profiilien mukaisiin määrityksien toteuttaminen vaatii pieniä parannuksia käyttäjien sessionhallintaan. Lisäksi toteutukseen täytyy lisätä tuki Hybrid-auktorisointivuolle sekä Dynamic-profiilin mukaiselle asiakassovelluksen luonnille ja hallinnalle.

Asiasanat: Identiteentin- ja pääsynhallinta, OpenID Connect, Ohjelmiston sertifionti, OAuth 2.0

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Identity and access management (IAM) is among the key factors in the IT infrastructure of any organization. By implementing their IAM systems to be effective, enterprises can decrease the cost of the infrastructure and improve information security. IAM is also strongly related to information security. These are among the main reasons that make it worth it to look at the IAM more closely and implement new protocols and frameworks to manage different use cases in IAM.

This thesis looks into the certification process of the OpenID Connect authentication framework. The information was gathered by reviewing related literature and technical documentation. The acquired knowledge was then applied in practice to give recommendations on how the OpenID Connect Provider (OP) implemented for the Trivore Identity service may be certified.

The thesis is divided into five chapters. Chapters 2 to 4 are theoretical and Chapter 5 will apply the theory to practice. Chapter 2 presents some of the basic theory behind authentication and authorization. Chapter 3 looks into the technical details of OIDC and some of the technologies that influences it. Chapter 4 presents the OIDC certification process and the fundamental technical details of the certification requirements. Chapter 5 introduces the Trivore identity management platform and the framework for a web based application developed by Trivore Corp. This chapter also covers the test report, which covers the tests required to pass in order to gain the certification for the OP. At the end of Chapter 5, there are recommendations for gaining the OP certificate.

Chapter 2

Identity management

This chapter introduces some of the basic elements of identity, authentication and authorization. Many other concepts and aspects related to identity management like accountability, audit trails and reporting will not be reviewed in this chapter. Even though they are an essential part of both authorization and authentication, they are not directly related to the topic of this thesis, the OpenID Connect and the certification process discussed later.

At the core of identity management lie the concepts of authorization and authentication and, of course, the identity itself. The essential goal of identity management is to limit the available actions and access to some resources to a limited set of entities. Many important services, like online banking, for example, depend on identity management. Identity and access management are also used to enforce resource ownership.

Identity management is in direct relation with information security, organizational efficiency, and developing new operational and business models. Identity management is essential in regards to information security as one of the definitions of information security is about preventing and detecting unauthorized access. Additionally, well implemented identity management model helps end-users with password management and reporting of authorization usage. [2]

Among other benefits of identity management are increased organizational efficiency due to time savings and increased quality of service. Developing identity management may reveal new business operation models due to decreased identity management related costs. [2]

2.1 Identity in a computer system

In the context of computer systems, an *identity* is defined as a ”set of attributes related to an entity” [7]. An entity in its turn is then defined as an ”item inside or outside an information and communication technology system” [7] that could be, for example a person, a device or an organization. An *attribute* defined in ISO/IEC 24760 [7] is a ”characteristic or property of an entity that can be used to describe its state, appearance, or other aspects”. Attributes can be either persistent like a date of birth and eye color or temporary like an address [8].

An *identifier* is something that distinguishes entities within the context of a specific name space. It could, for example, be an account number or a social security number. An identifier is only meaningful in its name space and only when it corresponds to the entity that it identifies. [8]. An identifier has several properties. Ideally one identifier should only refer to a single entity for audit trails to work properly. Each entity with an identity must have a non-empty, non-null identifier. Identifiers should be chosen so that it is possible to assign an identifier to all identities. [2]

A person can have an identity in a computer system so that a record in a database has an identifier, for example a username, as well as a set of optional attributes like the person’s full name, email address and phone number. A person may have multiple digital identities across many computer systems. As an example, a person may have an email account as well as a banking account, each with their own identifier. In an email account the identifier could be the email address itself and in a banking system the user could have a user account number assigned by the bank.

The process of identity management refers to the way that entities and their digital identities are presented in a computer system. Furthermore, identity management covers the syntax and semantics of both data in transit and on disk. The identity management also covers organizational processes that are used to maintain the digital identities. [2]

2.2 Authentication

This section describes the concept of *authentication* as well as lists several forms of authentication. Authentication has multiple levels, which are based on the strength of the authentication.

Authentication is defined as the process of verifying an entity's identity. In other words, authentication links the digital identity to the corresponding entity. The strength of this said link along with the initial user identity proofing then defines the level of authentication. For example, the European eIDAS regulation defines three levels of authentication, which are `LOW`, `SUBSTANCIAL`, and `HIGH`. [2]

Authentication differs from *identification*, the initial process of confirming identity, in a sense that identification could be considered the first step in the authentication process. The second step of the authentication process is authorization, which verifies the user's credentials. [9]

A good example if identification is when an end user enters his/her username to a login prompt. The end user is successfully authenticated when he/she enters the correct password. Finally, the user may be authorized to access some resources, like his/her files on a Unix file system.

There are a number of different forms of authentication, the most common being passwords. Additional methods are biometric, smart cards, PINs and photo IDs. The types of authentication can be categorized roughly into four different types. These are ”what you know”, ”what you have”, ”what you are” and ”where you are”. [9]

The ”what you know” category consists of authentication types like passwords, passphrases, PINs and security codes. The main idea is that only the correct entity knows the secret. Passwords should be stored as a cryptographic hash by using a one-way hashing algorithm. [9] This is done to prevent anyone from gaining access to the plain passwords. In addition to hashing the passwords, there should be a salt value hashed along with the password to add a layer of obfuscation and increase the difficulty of off-line dictionary attacks.

The second category namely ”what you have”, also called ”authentication by ownership or in possession”. It includes things like photo IDs, smart cards and security dongles, which are electronic device-based systems used for authentication.

The ”what you are” types of authentication can be further categorized into behavioral attributes and physiological attributes. Behavior attributes relate to a person’s behavior pattern, for example, a person’s voice, the way a person walks or types on a keyboard. The physiological attributes, on the other hand, are the ones that are related to the human body and cannot be changed easily. The physiological attributes include, for example, authentication methods like fingerprint recognition, facial recognition or retina scanning. The physiological attributes are considered to be more reliable authentication method than the behavioral attributes. [9]

The fourth type, ”where you are” consists of location-aware authentication methods. The location estimation may use GPS, IP address, or cell tower ID. This authentication type can be used with other methods to verify the identity of the user.

2.3 Authorization

After authentication is performed, the authenticated entity may be authorized to perform a set of actions or access a set of resources. In a computer system the resources are often pieces of data, that only a specific set of entities (i.e. end-users or other systems) is supposed to access.

Formally defined, authorization is about a function $f : S, O, A, e \rightarrow \{yes, no\}$, where

- S, subject, is an authenticated entity,
- O, is an object that an action is to be performed on,
- A is the action to be performed (e.g. read or write) and
- e, environment, is a collection of properties related to the environment such as date and time or the network the request originated from.

The function returns *yes* or *true* if and only if the authorization was granted. As seen from the function parameters, successful subject authentication is a prerequisite for authorization. [2]

2.3.1 Access control matrix

The simplest way to determine if an authorized entity has access to a certain resource is to utilize an access control matrix. A column in the matrix is called an access control list, (ACL) and each column lists a protected item along with per user access rights. A row in the matrix represents user's capabilities telling what resources the user can access. [2]

The example shown in Table 2.1 represents a Unix file system along with two users with different access schemes. User with username *alice* is able to read and write to file located in her own home folder but is only allowed to read file */tmp/bar*. Another user

	/home/alice/foo	/tmp/bar	/etc/passwd
alice	read, write	read	-
bob	read	read, write	-

Table 2.1: An example of an access control matrix. Adapted from [2]

with username *bob* is able to only read the file */home/alice/foo* but can both read and write to file */tmp/bar*. However, neither of the users is able to read or write the */etc/passwd* file.

An issue with such a matrix is that as the number of users and protected resources grows, the matrix becomes more and more difficult to maintain.[2] Nevertheless, there are other, more maintainable approaches to authorization, which are described next.

2.3.2 Role-based access control

The role-based access control (RBAC) introduces an abstraction or roles that are assigned to the users. The roles are usually created to represent a set of tasks within the organization. There could be a set of roles for system administrators, system auditors, backup operators as well as roles for regular end users, who have to access the system to do their jobs. Each user may have a number of roles and each role has a certain set of permissions, which are referred as ”operations” in the RBAC paper [1]. The permissions authorize the user to access the resources associated with the permissions. In other words, whenever the system has to determine if a user has the right to access a resource, the system will inspect the user’s roles to see if any of the roles has the required permission to access the resource. [1]

The roles can be hierarchically organized to inherit the rights from other roles. Inheriting roles greatly reduces the administrative overhead of RBAC. [1] Figure 2.2 shows a simple example with role inheritance. The example is described a bit later.

The Figure 2.1 shows a simplified model of the RBAC authorization scheme with role composition. This model is a static view of the RBAC scheme, which also does not take the system environment into consideration.

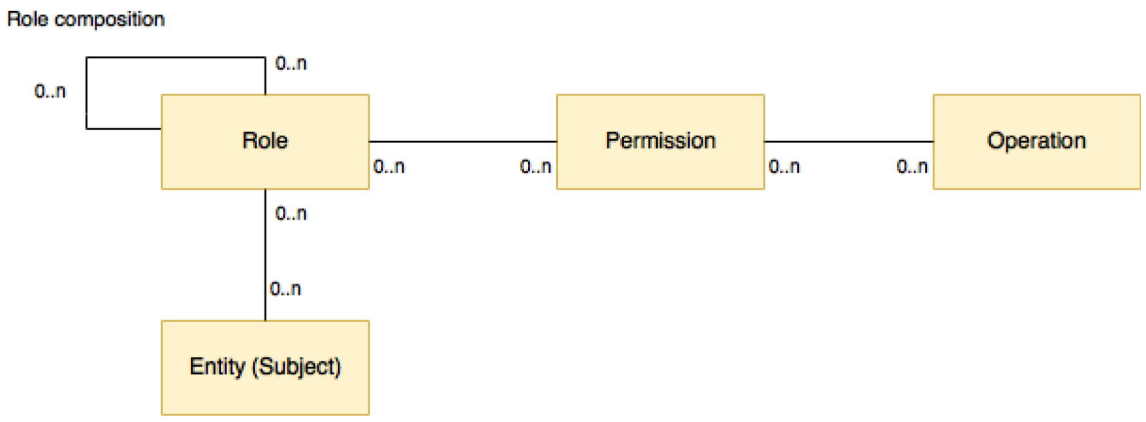

Figure 2.1: RBAC structure, figure is based on the RBAC description from [1].

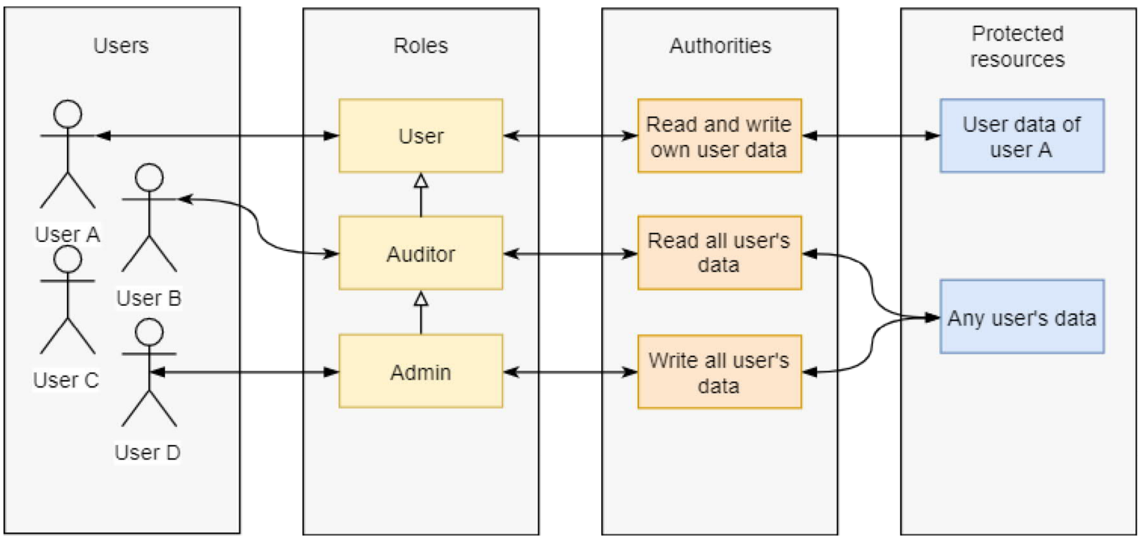

Figure 2.2: An example of RBAC. Adapted with modifications from [2]

Figure 2.2 shows an example of an RBAC based system, that has a database of user accounts containing data. Each user with the *User* role is allowed to read and modify their own data. Additionally, the *Auditor* role gives a user the permission to read any user's data and the *Admin* role gives permission to modify the data of any user.

In the example, the *Admin* role is inherited from the *Auditor* role, which in turn is inherited from the *User* role. The inheritance simply ensures that any user with the *Admin* role is also has the same read rights as the *Auditor* role. Both the *Admin* and *Auditor* roles have all the same permissions as the *User* role. This kind of approach makes it easy to add more permissions to all roles.

2.3.3 Attribute based access control

Another method of access control is the attribute based access control (ABAC), where the access to a protected resource is determined by the attributes of the accessing entity. [3] For example if access to the resource is restricted by user's current location or home town, the system will check the user's attributes when the user's agent is making the request. The request is authorized if the attributes match the predefined conditions on the server.

2.3.4 Mandatory Access Control

Mandatory Access Control (MAC) disallows the owner of the data to give permissions to access the data. The core idea is that the system itself enforces that data has the correct permissions. A system administrator can then define the security policy that will be enforced for the users. [2]

The MAC scheme is described by the following two rules [2]:

No read-up states that user will not be access data above their clearance level.

No write-down states that users will not be able to write data that is below their clearance level.

Together these rules define a system where users will only be able to write but not read data that has a higher clearance level than the user. This scheme results in write-only data that user cannot read after writing.

The MAC in its purest form is mostly a theoretical model. It is challenging to implement correctly and difficult to use. [2]

2.3.5 Least privilege principle

According to the *least privilege principle*, a user may only have the access rights he/she needs to complete his/her tasks. The least privilege principle could be considered one of the core principles of information security. [2]

In order for the least privilege principle to be valid, the organization has to constantly validate the access rights of the system users. Every time an employee's job description changes, his/her access rights have to be checked. The least privilege principle is related to the *segregation of duties* (SOD) principle so that it should be actively monitored that the access rights of any user will never break the SOD. [2]

2.3.6 Segregation of duties

The principle of *segregation of duties* (SOD) states that there are tasks that cannot be assigned to the same person to minimize the chance of human error and intentional malicious use of the system. [2] A practical example of SOD is that the firewall configuration may be forced to be done by a different person than the one who installs services to a system with administrator rights. The reasoning behind this is that a single person will not be able to open services to a public network.

It may not be practical in small organizations to segregate duties due to the small number of employees. In such cases the risk of not utilizing SOD should be noted and other means of risk control should be implemented. [2]

2.3.7 Delegation of Authorization

In the context of identity management within an organization, delegation of authorization means temporarily transferring access rights from one user to another due to, for example, someone having a vacation or other temporary leave. Authorization can also be delegated if someone in a relatively high position in the organization wants to give some of their tasks to be handled by someone else. [2]

Delegating authorization should not be done by handing system credentials (i.e username and password) to someone. The ideal situation is that the identity manager software can specifically handle the authority delegation use-case so that the audit trail is collected correctly. For practical reasons, the delegation is often done by giving the same access rights to the main user and delegated user. [2]

From the technology's point of view, authorization delegation can refer to the kind of delegation where, some entity, for example a client in OAuth 2.0, makes requests to protected resources on behalf of the resource owner. This kind of authorization delegation will be further inspected in the following chapters.

2.4 Authentication in web environment using HTTP

This section provides a short overview of authentication in web based applications using HTTP. In the context of this thesis, the most interesting technical detail is the use of bearer tokens described in RFC 6750 [10], which are used to access OAuth 2.0 protected resources.

In this thesis web based applications are viewed as distributed systems that consist of a server with multiple clients connecting to it using HTTP. HTTP is used to transmit information between the web server and the clients. The server may additionally connect to a database server that may reside in the same system as the web server itself or in another system. It is also common that the web application utilizes load balancing to achieve high availability, but this kind of setups are more advanced and this work will not look into them. The somewhat limited view of web applications considered in this thesis is due to the fact that this work focuses on OAuth 2.0 and OpenID Connect, both of which operate solely over HTTP.

The authentication process is used to reliably distinguish end users or systems. An authenticated end user or system can be shown personalized content that may not be accessible to others.An authenticated end user or client can also be authorized to access some protected resources. For example, when an end user is logged in to a banking application with his or her user-agent, he or she is able to see his or her own balance but not anyone else's.

HTTP itself has support for several different authentication methods such as 'Basic' and digest. The 'Basic' authentication scheme defined in RFC 7617 [11] simply sends credentials as username-password pairs. The 'Basic' authentication requires that the username and password are in format `username:password`, which is then encoded as a base64 string and sent along with HTTP requests in the HTTP header. Web applications may also implement authentication by using cookies, defined in RFC 6265 [12], which are small pieces of data containing both a key and an associated value. Cookies are set by the web server, saved by the user's client software and then retransmitted to the server on every HTTP request. Cookies are used to manage the user's session and remember stateful information in the otherwise stateless HTTP protocol. This state can be associated with a valid authentication as the client performs the authentication process, for example, by providing valid credentials.

2.5 Federated identity management

Federation refers to the association between service providers and identity providers. Both the service providers and identity providers will have to trust each other to a certain degree to exchange messages between each other. [13]

Federated identity management on the other hand refers to federation, where the messages exchanged between the service providers and the identity providers contain information on user authentication and authorization credentials. Additionally, user from one system should be able to access protected resources on another federated system. [13]

Federated identity management enables users to access several different services by authenticating themselves only once. It also enables service providers to consume the services of identity providers instead of having to manage user identities themselves. [13]

2.6 Single Sign-On

Single Sign-On (SSO) is a configuration that allows users to access all their resources and systems they have access permission by authenticating and authorizing at a single location. Many SSO systems use Lightweight Directory Access Protocol (LDAP). [9] From the end user's point of view the SSO works so that the user has to authenticate only once and the SSO transparently takes care of any further authentications. SSO benefits users and organizations by eliminating redundant usernames and passwords. This will also reduce the number of help desk calls and administrative costs. [14] SSO systems can be distinguished into two main types of systems, the *pseudo-SSO* and *true SSO*.

2.6.1 Pseudo-SSO

The first main type, called *pseudo-SSO* is a system where a server or some other pseudo-SSO component acts as a proxy component between the user and the *service providers* (SP). After the user authenticates him/herself to this proxy component, the proxy can perform the authentication to the SPs. The initial authentication step to the pseudo-SSO component is called *primary authentication*. The user authentication is done separately every time the user accesses an SP during an SSO-session. The pseudo-SSO scheme is illustrated in Figure 2.3. Each SSO identity is SP specific and a user can have many identities for a single SP, so the relation between SSO identities and SPs is $n : 1$. [3]

The pseudo-SSO systems can be implemented to work either locally on the end user's device or by using a proxy server that seamlessly handles authentication to SPs. [3]

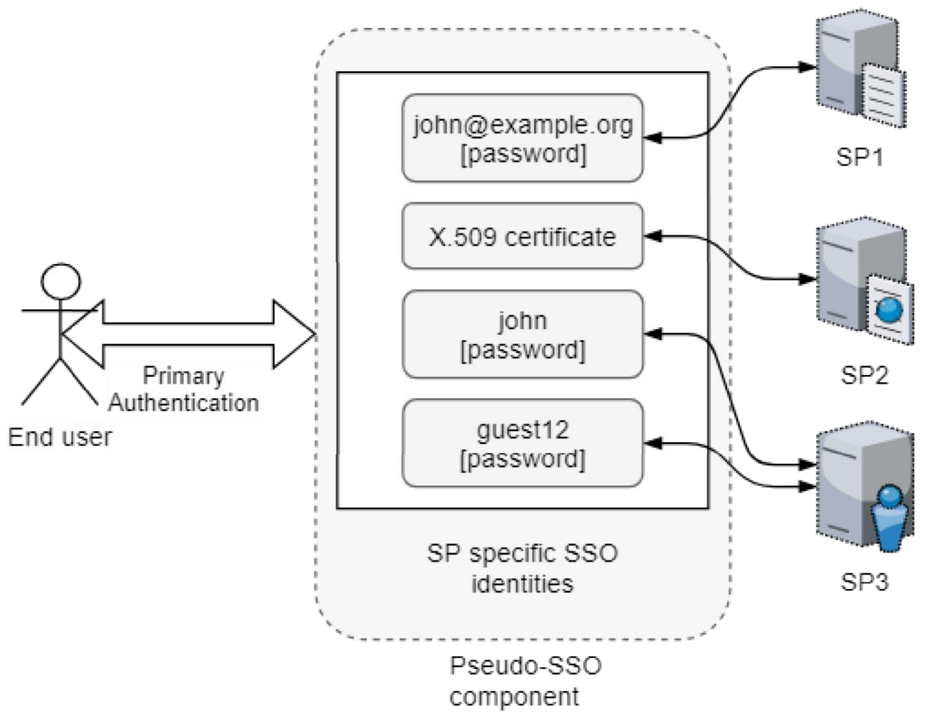

Figure 2.3: Pseudo-SSO. Adapted from [3]

2.6.2 True SSO

The second type of SSO, called *true SSO*, is essentially different from pseudo-SSO. In the true SSO scheme there is an entity called the *Authentication Service Provider* (ASP), which is responsible for the user authentication and the transfer of authentication assertions to SPs. The ASP and the SPs have to have an existing relationship for true SSO to work. In true SSO the relation between SSO identities and SPs is $n : m$. [3]

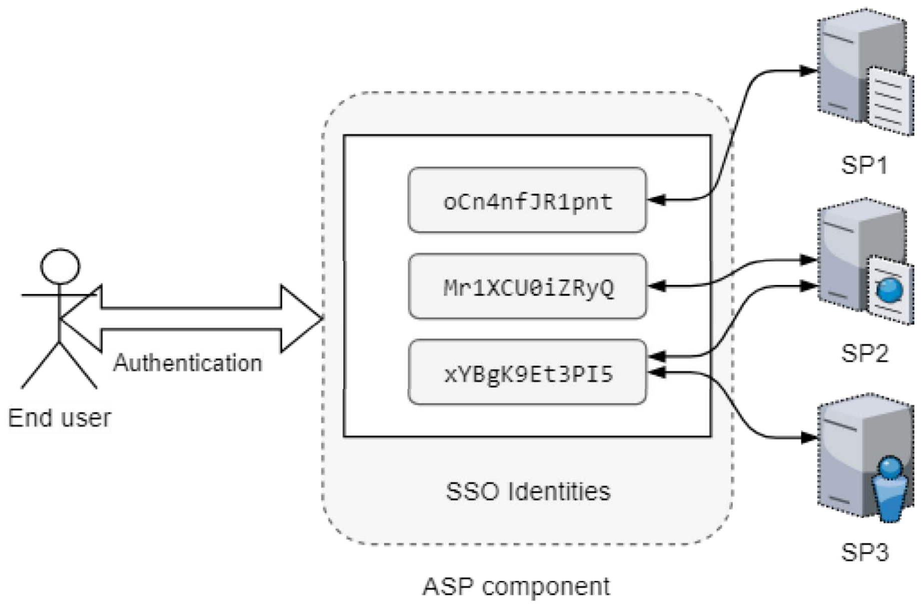

Figure 2.4: True SSO. Adapted from [3]

Like the pseudo-SSO, the true SSO can be either local or proxy-based. In the local true SSO architecture a component on the user's system is used as the ASP. The local ASP and the SPs must still have a trust relationship. When the true SSO is proxy-based, the ASP is an external server. [3]

For the true SSO architecture, the user identities can be tied to precise descriptions and policy regulations. Furthermore, a true SSO system can be closed in case the user credentials compromised. [3]

There are a number of setup, maintenance, security and usability aspects related to both local and proxy server based architectures for pseudo-SSO and true SSO. However, these aspects are not directly related to the subject of this thesis and will not be further discussed.

Chapter 3

OAuth and OpenID

This chapter introduces the main concepts of OAuth, OpenID and finally the OpenID Connect (OIDC). The reasoning behind covering both OAuth and OpenID even when the thesis is mainly about OpenID Connect is that the OIDC is based on OAuth 2.0 and is the successor of OpenID 2.0. For OAuth both the current and obsoleted versions, OAuth 2.0 and OAuth 1.0 respectively, are discussed. OpenID Connect and OAuth 2.0 protocols are described with the accuracy that is required to properly understand the certification requirements whereas the rest of the protocols are covered more briefly.

OAuth and OpenID protocols are fundamentally different. OAuth is an authorization protocol, whereas OpenID is an authentication protocol. OAuth may be used for authentication as well, but it is not the main use case for OAuth. The OpenID Connect attempts to capture the best parts of both OpenID and OAuth.

The previous chapter viewed authentication and authorization mainly from the end user's point of view. In this and the following chapters, the entity to be authenticated may be the end user but it might also be another computer system. For both OAuth and OpenID protocols the type of the entity to be authenticated is irrelevant.

It is also worth noting that none of these protocols specify any exact means of authentication. In the context of these protocols, the authentication process may thus be carried out in any way and it is up to the application to define the means of authentication. Quite often in practice, end users are authenticated with a username and password pair and computer systems are authenticated by utilizing asymmetric cryptography and cryptographic certificates. The exact details of the workings of these cryptographic concepts are out of the scope of this thesis, however.

3.1 OAuth 1.0

OAuth 1.0 authorization protocol, described in RFC 5849 [15], can be used to provide delegated access to third party websites and applications. OAuth provides a process that can be used to authorize a third party client to access protected resources without sharing the resource owner's credentials with the client.

Traditionally resource owners or end-users are authenticated by the server hosting the services by entering credentials, usually a username and password pair, which are then validated. This kind of model makes it difficult to securely authorize a third party application (*client* in OAuth's terms) to access the end-user's (or *resource owner*'s) resources. In this model, the client application would have to have access to the end-user's credentials in plain text, which would impose additional security risks. OAuth protocol aims to remedy the situation by introducing flows that can be used to delegate authorization.

OAuth 1.0 was later deprecated by its successor, OAuth 2.0, which is more flexible in terms of client types and simplifies the encryption scheme. Despite the naming, OAuth 1.0 is not compatible with OAuth 2.0 and they are to be considered different protocols.

3.1.1 OAuth terms

When discussing the OAuth protocol, some terminology has to be defined. The OAuth 1.0 specification [15] has definitions for the entities related to the protocol.

The client is simply defined to be ”an HTTP client capable of making OAuth-authenticated requests” [15]. One has to keep in mind that in OAuth the client is not the same thing as an end-user but rather the application working on behalf of the end-user.

The server in OAuth 1.0 is an HTTP server, which can accept OAuth-authenticated requests. Unlike in OAuth 2.0, in OAuth 1.0 the server is expected to host the protected resources as well as authenticate users. The OAuth 2.0 specification has definitions for two different servers, the *resource server* and the *authorization server*.

Resource Owner in OAuth is an entity that controls the resources that the client wishes to access. The resource owner may be the application end-user, but it can also be another system.

Credentials are defined as a ”unique identifier and a matching shared secret” [15]. Credentials can be classified to be either either client, temporary or token credentials. *Token* is a unique server to client issued identifier, which associates the authenticated requests with a resource owner.

For the authentication, the client has to gain permission from the resource owner. There is a token and a shared secret that together represent the permission. Tokens can have restricted scope and a limited lifetime for added security.

3.1.2 Basic Authorization Flow

The basic authorization flow has three steps which are shown in Figure 3.1. In the first step (steps (1) and (2) in the Figure 3.1) the client obtains a set of temporary credentials from the (authentication) server. The second step (steps (3) and (4) in the Figure 3.1) is the authentication of the resource owner at the server. The OAuth protocol itself does not specify how the authentication is done exactly and it is considered to be an implementation detail. After authentication the resource owner authorizes the client's request at the server. In the third step ((5) and (6) in the Figure 3.1) the client makes a request for a set of token credentials by using the temporary credentials. Finally, the client can use the token credentials to make requests on behalf of the resource owner. These steps are described in more detail in the following sections.

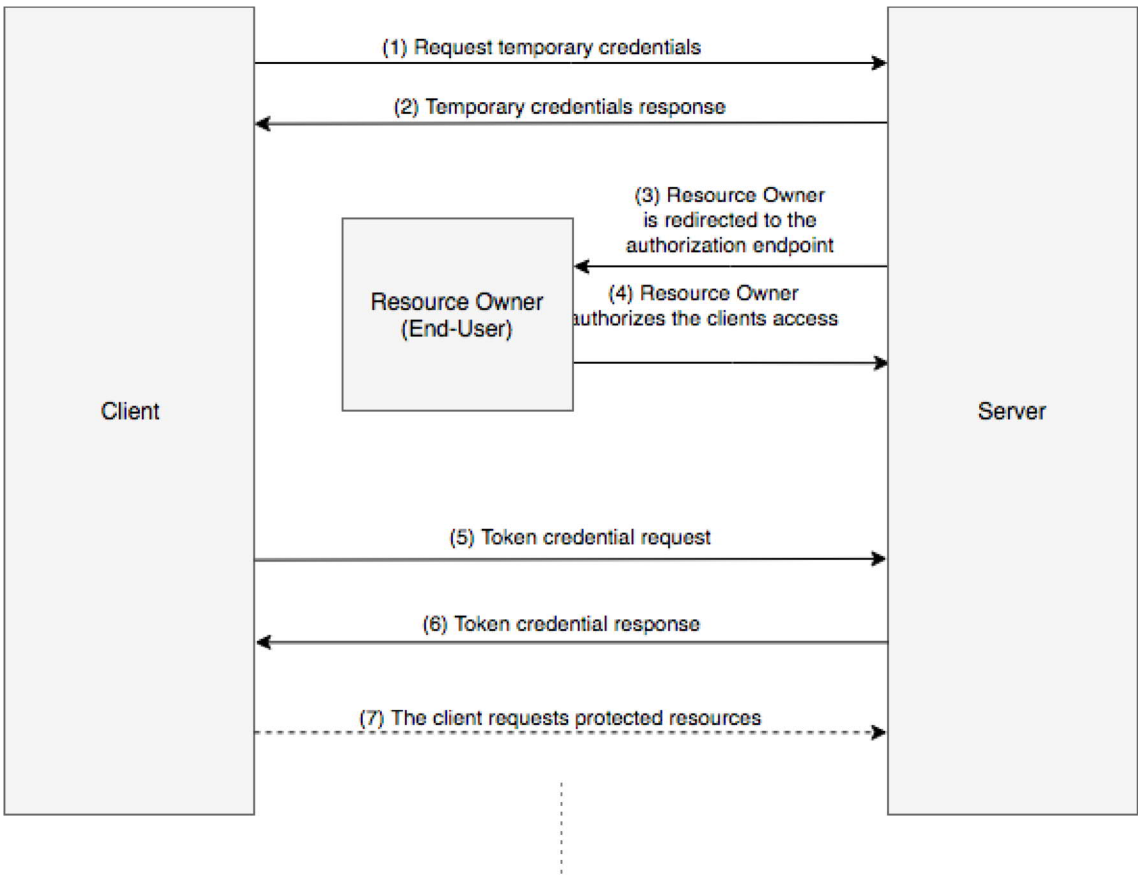
(1) Request temporary credentials
(2) Temporary credentials response
(3) Resource Owner is redirected to the authorization endpoint
Resource Owner (End-User)
(4) Resource Owner authorizes the clients access
Client
Server
(5) Token credential request
(6) Token credential response
(7) The client requests protected resources

Figure 3.1: The OAuth 1.0 basic authorization flow

3.1.3 End points

The OAuth 1.0 protocol defines three different endpoints that are used for requests. [16] The endpoints are:

- Temporary Credential Request
- Resource Owner Authorization
- Access Token Request

Temporary Credential Request end point is responsible for handling the requests for unauthorized Request tokens. The request tokens can then be used to get user approval and the access token.

Resource Owner Authorization is used to retrieve user authorization.

Access Token Request is used to gain a token that can be further used to access protected resources from the resource server.

3.1.4 Request signature

OAuth has support for signing requests for message authentication verification. The methods for generating the signature are `PLAINTEXT` `HMAC-SHA1` and `RSA-SHA1`. If the message has a signature attached to it, according to the specification it must be verified for the message to be valid.

The `PLAINTEXT` method is the simplest one but it provides no message authentication or integrity. When the `PLAINTEXT` method is used, the messages must be sent over a secure channel. The `HMAC-SHA1` uses a shared secret to generate a digest with the HMAC-SHA1 algorithm. The shared secret has to be known by both parties, the sender and the recipient. This requires that the secret has been previously shared between the server and the client during a registration process.

`RSA-SHA1` method utilizes public key cryptography to verify the message authentication and integrity. The `RSA-SHA1` uses `RSASSA-PKCS1-v1_5` with `SHA1` as the hash function. The client and the server are required to exchange public keys prior using this method.

3.1.5 The detailed authorization flow

The OAuth 1.0 protocol specification [15] describes a so-called three-legged authorization flow (three being the number of parties involved: the client, the server and the resource owner) as ”redirection-based authorization”.

Requesting the Temporary Credentials

The server provides a set of temporary credentials for the client at the Temporary Credential Request end point. The client will make an authenticated HTTP POST request at the server’s Temporary Credential Request end point. The request will contain the `oauth_consumer_key` which is used to identify the client, `oauth_signature_method`, which tells the OP the signature method used and `oauth_signature`, which is the signature itself. The client must add the `oauth_callback` parameter to the request when fetching the temporary credentials. The `oauth_callback` parameter defines the URI the server will redirect the resource owner after completing the authorization step. The server may optionally specify additional parameters that can be used for implementation specific purposes.

The Listing 3.1 shows an example of an HTTP POST request to the Temporary Credential Request end point. In the example listing, the signature method used is

PLAINTEXT. The signature will in this case consist of a client shared-secret with a value of jd83jd92dhsh93js, which has to be known by the server before the client makes the request. Additionally, since the signature method is PLAINTEXT, the request has to be made over a secure channel.

```
POST /request_temp_credentials HTTP/1.1
Host: server.example.com
Authorization: OAuth realm="Example",
 oauth_consumer_key="jd83jd92dhsh93js",
 oauth_signature_method="PLAINTEXT",
 oauth_callback="http%3A%2F%2Fclient.example.net%2Fcb%3Fx%3D1",
 oauth_signature="ja893SD9%26"
```

Listing 3.1: Temporary credential request example from [15]

The response will contain at least the oauth_token, oauth_token_secret and oauth_callback_confirmed parameters. The oauth_token is used to identify the temporary credentials, oauth_token_secret is the shared secret of the temporary credentials and oauth_callback_confirmed is set to true and is used to differentiate from previous protocol versions.

Resource Owner Authorization

The resource owner authorization is used to gain authorization from the end-user by re-directing the user to the Resource Owner Authorization endpoint of the server. The only required parameter for Resource Owner Authorization is the oauth_token, that represents the previously acquired temporary credentials. The client will then re-direct the resource owner to the server for authorization.

After verifying the resource owner's identity and receiving authorization, the server will re-direct the user to the URI given to the server as the oauth_callback parameter. The server will also generate a random verification code and send it to the client as the value of the oauth_verifier parameter along with the oauth_token received from the client.

Token Credentials Request

After the resource owner has authorized the client, the client will make an authenticated HTTP POST request to the server's Token Request endpoint to acquire a set of token credentials. The request will contain the verification code is the value of the `oauth_verifier` parameter, that the client received from the server during resource owner authorization.

The HTTP response will contain the `oauth_token` as well as the `oauth_token_secret`. The client will then store the credentials and be able to make authenticated request to protected resources on behalf of the resource owner. The requests will contain the client credentials and the received token credentials.

3.2 OAuth 2.0

This section covers the basic concepts and functionality of the OAuth 2.0 authorization framework. The section refers to [17] and [4] as its main sources of information. The OpenID Connect protocol is based on OAuth 2.0 and therefore understanding the basics of OAuth 2.0 is required to be able to implement OpenID Connect. The framework is covered in the detail, that is required to comprehend the test requirements in the OpenID Connect certification process.

OAuth 2.0 is not backwards compatible with OAuth 1.0 due to major protocol changes, but the three-legged authorization flow of OAuth 1.0 is comparable to the Authorization Code flow OAuth 2.0. OAuth 2.0 adds support for additional authorization flows on the framework specification level and also supports use-cases, where the client cannot be authenticated. Additionally, OAuth 2.0 does not require signatures to be used like in OAuth 1.0. Therefore OAuth 2.0 is may be considered easier to implement than OAuth 1.0.

3.2.1 Parties

The parties or ”roles” as described in [4] are among the key terms of OAuth 2.0. Even if OAuth 1.0 already defined some of these terms, the definitions may differ.

Resource owner refers to the entity that is able to grant access to a protected resource. If the entity is a real person, the terms end user and Resource owner can be used interchangeably.

The authorization server is the entity that asks the Resource owner's confirmation that the client can be authorized to access protected resources. The authorization server provides the client with an access token, which the client uses to specify which user is making the request.

Resource server is the entity that serves the protected resources. In many cases it is the same as the authorization server, but the OAuth 2.0 framework specification still separates these two entities. The authorization server and the resource server must have a way exchange information about the authorized clients, but this is left outside of the framework specification.

The Client is the application making the requests on behalf of the resource owner. OAuth 2.0 does not limit the type of the client, which could be, for example, a mobile application or a web application. The authorization flow is different for different types of clients.

3.2.2 Client types and general profiles

Unlike OAuth 1.0, its successor, OAuth 2.0 can support several different kinds of clients. OAuth 2.0 clients can be divided into two different types based on whether the client is capable of keeping the client specific credentials secret. The client specific credentials consist of a secret shared between the client and the authorization server.

Confidential clients are expected to be able to keep the client's credentials secure. These types of clients can be, for example, web applications that can keep the client's credentials on the server side.

Public clients are clients that are not able to keep the credentials secure. These can be, for example, web applications that are fully downloaded on to the user agent (web browser in this case) or applications that are downloaded and executed on the user agent.

The two types of clients are further categorized into three different general profiles based on the environment the application runs in.

Web applications are confidential types of applications that are considered confidential clients. The data is stored in on the server side and the client side will not be able to access it.

User agent based applications are public clients that are run in a user agent environment. These are, for example, pure JavaScript web applications that are run in a web browser.

Native application is a public client installed on a device. The native application could be an application downloaded on a mobile device.

3.2.3 OAuth 2.0 Endpoints

Endpoints are URIs that define a logical location that entities should use when making certain requests. Endpoints can be categorized into server endpoints and client endpoints.

The OAuth 2.0 framework defines the following end points:

- Authorization endpoint
- Token endpoint
- Redirection endpoint (client endpoint)

The Authorization endpoint is located at the authorization server. It is the endpoint where the resource owner will be authenticated and will then grant authorization. In the OAuth 2.0 authorization code grant flow the authorization endpoint will respond to a successful authorization grant with an authorization code. In the implicit flow the authorization endpoint will respond directly with an access token. Since the authorization resource owner may have to enter clear text credentials to the authorization endpoint, the OAuth 2.0 specification requires the connection to the endpoint to be secured with TLS.

The authorization endpoint has to support HTTP GET method and may additionally support HTTP POST method as well. Parameters without value must be omitted and unrecognized parameters must be ignored completely. Each parameter must be present in a request or response at most once.

The Token endpoint at the authorization server is where the client will make an HTTP POST request to acquire an access token. The token endpoint is used during all authorization grant type flows except the implicit flow. The OAuth 2.0 specification requires the endpoint to be secured by TLS.

The redirection endpoint is the endpoint that the client will be redirected after the resource owner grants authorization. It is defined at the authorization server during the client registration process or it may be given as a parameter when the client makes an authorization request.

3.2.4 Basic Authorization flow

The basic abstract authorization flow in OAuth 2.0 can be separated into the six steps shown in Figure 3.2 [17].

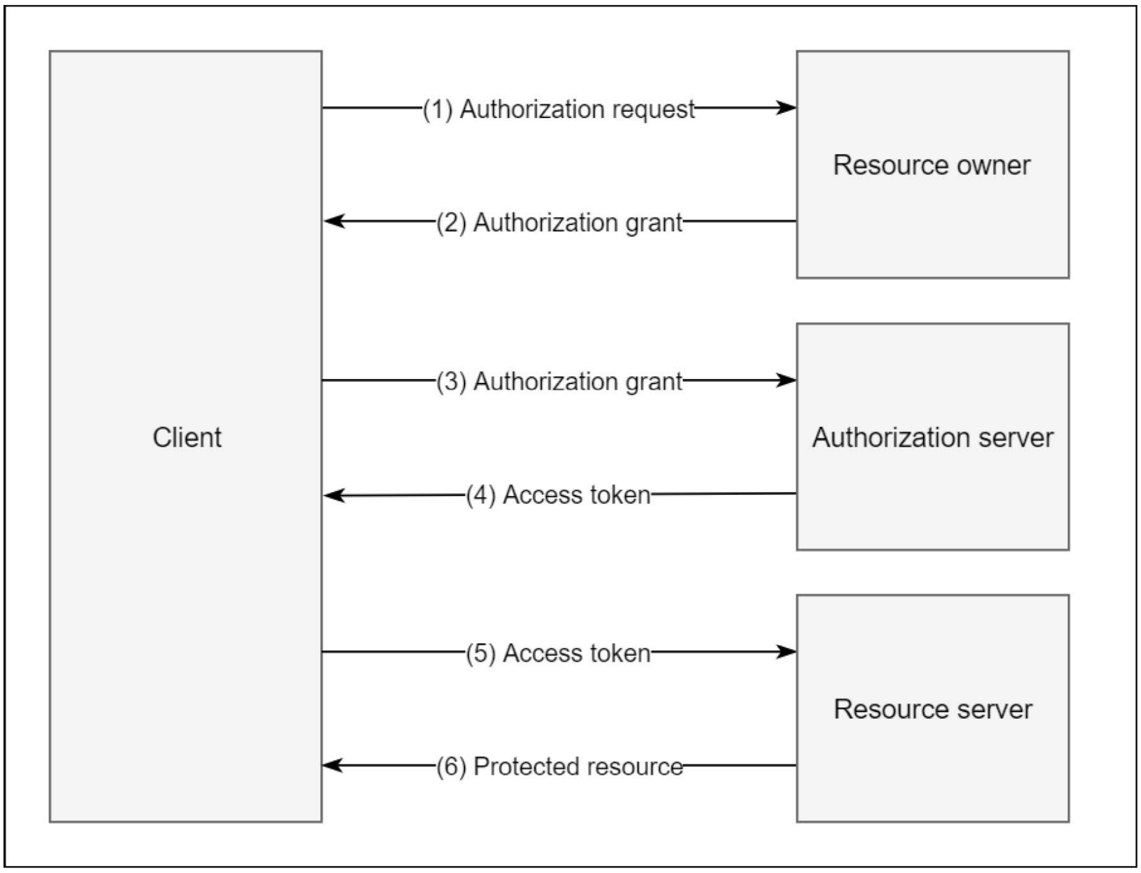

Figure 3.2: OAuth 2.0 abstract protocol flow. Adapted from [4]

.

- (1) The Client requests authorization from the resource owner.

- (2) The Resource owner authorizes the request and provides an *authorization grant* to the client.
- (3) The client sends the authorization grant to the authorization server
- (4) The authorization server validates the authorization grant and sends an access token to the client.
- (5) The client requests a protected resource from the server and provides the access token.
- (6) Finally, the server checks the validity of the access token and if it is valid, it responds with the requested resource.

3.2.5 OAuth 2.0 Tokens

Tokens in OAuth 2.0 are either *access tokens* or *refresh tokens*. Access tokens are used by the client when requesting the protected resources form the server. Refresh tokens are used to renew an expired access.

Access token

The *access token* is used to retrieve protected resources from the resource server. The access token is a formal proof of resource owner's authorization provided by the authorization server. [17]

Listing 3.2 shows an example of an access token. The expiration value has been set to 1 hour or 3600 seconds.

```
{
    "access_token": "JOnnhkJZxJG9s6H3WcfJwz6SlymUkqxo",
    "expires_in": 604799,
    "refresh_token": "wimDUQlOJPnydHsMEx4C7XSAZttrS5kV",
    "scope": "openid",
    "token_type": "Bearer"
}
```

Listing 3.2: An example of an access token in JSON format.

- `access_token` represents authorization granted by the resource owner.
- `expires_in` represents the number of seconds the issued token is valid. If protected resources are requested with an expired access token, the resource server will return an error.
- `scope` (Optional) defines the parts of the protected resources the client can access.
- `state` (Optional) The client may use the `state` parameter to hold a CSRF (Cross Site Request Forgery) token.
- `refresh_token` (Optional) A string containing the parameters when a new token is requested.

Refresh token

The *Refresh token* can be used by the client to renew an expired access token. The client exchanges the refresh token with the authorization server for a new access token.

The Figure 3.3 shows the flow of refreshing an expired access token. In steps (1) and (2) the client retrieves a refresh token along with an access token from the Authorization Server. In steps (3) and (4) the client successfully requests protected resource and the Resource Server responds. In step (5) the client attempts to request another protected resource but in step (6) the server responds with an invalid token error. In step (7) the

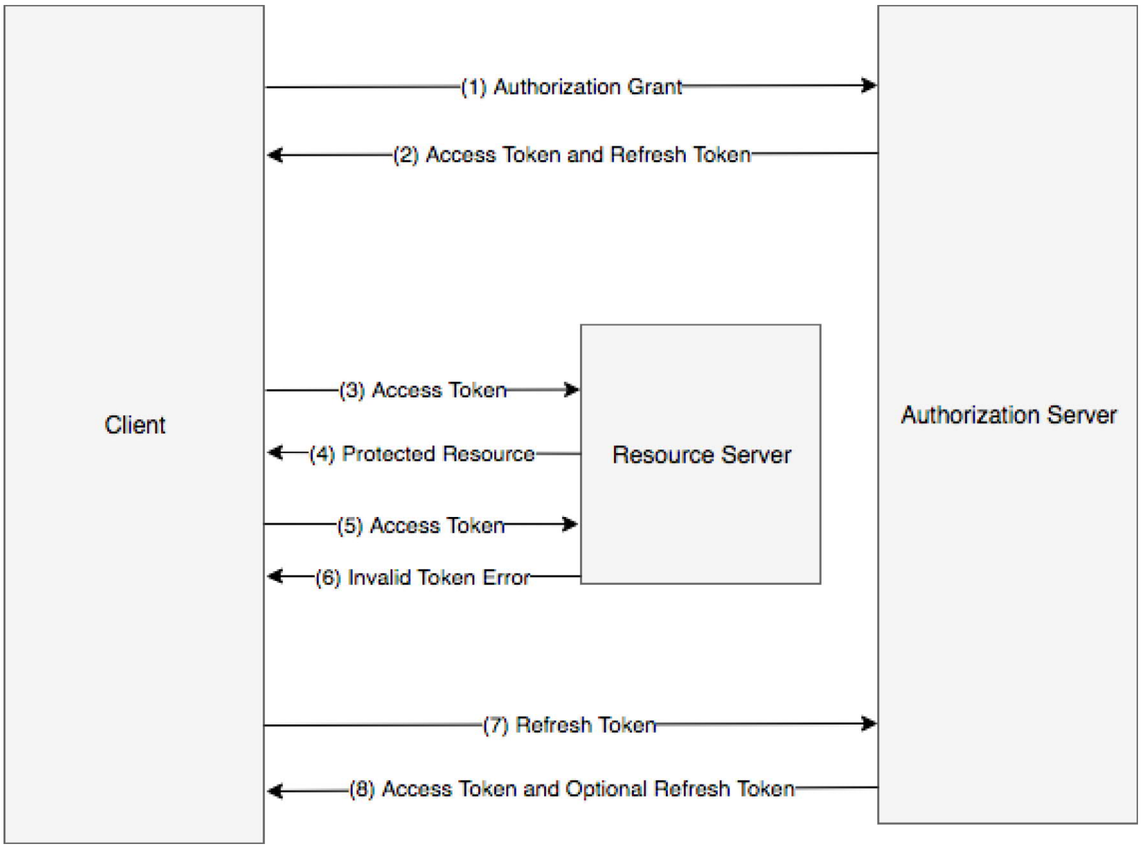

Figure 3.3: OAuth 2.0 refreshing an expired access token flow. Adapted from [4]

.

client then uses the refresh token make an authenticated request a new access token and in step (8) the Authorization server responds with a new freshly generated access token and optionally a new refresh token.

The refresh token request must contain the parameters `grant_type` with value `refresh_token` and a parameter `refresh_token` with the refresh token issued by the Authorization server. Optionally the request may also contain `scope` parameter, which may specify the scope of the new access token.

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

Listing 3.3: An example of an HTTP access token refresh request

3.2.6 OAuth 2.0 Grant Flows

OAuth 2.0 has various grant flows that support different use cases. This is an improvement over OAuth 1.0. In addition to the existing grant flows, the OAuth 2.0 framework allows defining additional custom types of grant flows. [17] [4]

The following sections will cover selected grant flows in the detail required to understand the testing process described in the next chapter. The most important grant flows in the context of the OpenID Connect certification process are the authorization code and the implicit grant flows. The resource owner password credential flow and the client credential flows are not tested by the OIDC testing tool and that is why they will not be described further in this thesis.

Authorization code grant flow is used for web applications and is the most commonly used grant flow. The authorization server works as an intermediary between the client and the resource owner or end user. The resource owner is redirected to an authorization server, which further redirects the resource owner back to the client after authorization is obtained form the resource owner.

Implicit grant flow is used in case a client application will not be able to keep the credentials secure. It is optimized for web applications implemented in a scripting language, for example JavaScript. The main difference between the implicit grant and authorization code grant is that in the implicit grant, the authorization server returns an access token instead of an authorization code.

Resource owner password credentials grant flow sends the actual resource owner's username and password pair to the authorization server to obtain an access token. This grant flow is used when mitigating existing solutions to OAuth 2.0. The password credential grant flow can only be used for well-trusted clients as they are required to handle the Resource owner's credentials.

Client credentials grant flow is used when a client applications is used to request protected resources on their own behalf, i.e the client is the resource owner.

OAuth 2.0 Authorization Code grant flow

This section describes the OAuth 2.0 authorization code grant flow in more detail. The authorization code flow is the most common flow used in OAuth 2.0. It is used by authenticated confidential clients (i.e. clients, that can keep the client credentials secret like web applications with a server side). It is roughly comparable with the "redirection-based authorization" in OAuth 1.0. In the Authorisation Code flow the `request_type` parameter will always have the value `code`.

Figure 3.4 shows the steps of the OAuth 2.0 authorization code grant flow. The flow steps are described below.

(1) The authorization code grant flow is initialized by the client. The client will redirect the Resource Owner's user agent to the server's authorization endpoint. The user agent will make an HTTP GET request to the authorization endpoint and sends at least the `response_type` with value `code`, and the `client_id`, which identifies the registered client to the server.

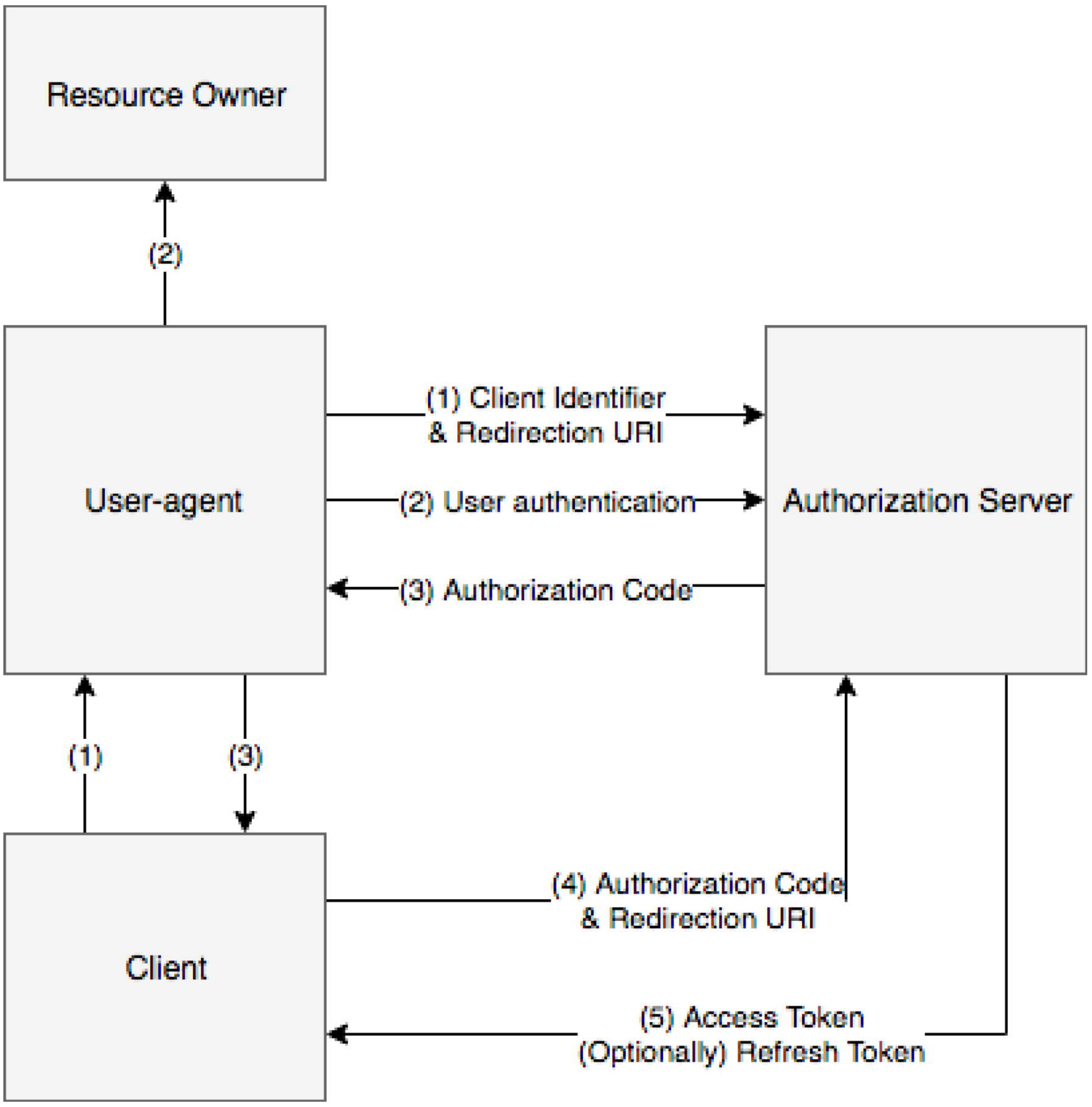
Resource Owner
(2)
User-agent
(1) Client Identifier & Redirection URI
(2) User authentication
(3) Authorization Code
Authorization Server
(1)
(3)
Client
(4) Authorization Code & Redirection URI
(5) Access Token (Optionally) Refresh Token

Figure 3.4: OAuth 2.0 authorization code grant flow. Adapted from [4]
.

Optionally, the client may also provide values for parameters `redirect_uri`, `scope` and `state`. The `redirect_uri` is optional in the Authorization Code flow and it specifies the URI that the server will redirect the Resource owner's user agent to after the Resource Owner authorizes the client.

The `scope` is also an optional parameter, which specifies a list of scopes of the client's access request. The authorization server may specify custom scopes to fit its requirements. The `state` is a recommended parameter which ties together the requests made and their responses during the grant flow. It is used for mitigating cross-site request forgery attacks.

(2) The authorization server authenticates the Resource owner. The Resource owner is presented the choice to authorize the request. The authentication process used is outside the scope of the OAuth 2.0 specification and in considered and implementation detail.

(3) If the resource owner grants access, the user-agent is redirected back to the client using the redirection URI. The authorization code is included in the redirection URI as the value of parameter `code`. The OAuth 2.0 specification recommends that the maximum lifetime of the authorization code is 10 minutes. Additionally, the code may only be used once. Any subsequent uses of the authorization code must result in the request being denied and all tokens issued for that authorization code should be revoked.

The response will also contain the `state` parameter with the exactly same value as it was in the request. If the request did not have `state` parameter, it is also not included in the response.

(4) The client uses the authorization code to request an access token from the server endpoint. The client also authenticates itself and provides the redirection URL. The request is an HTTP POST request to the token endpoint with the form encoded parameters `grant_type`, which always has the value set to `authorization_code`, `code`, `redirect_uri` and `client_id`. The `code` is required and it contains the authorization code that the client received in the previous step. The `redirect_uri` parameter is required, if the authorization request also contained the `redirect_uri` parameter. Their value must also be identical. The `client_id` is required, if the request does not include client authentication information. However, the specification requires that confidential clients should always authenticate themselves.

(5) Access Token Response The authorization server authenticates the client and validates the authorization code. The authorization server responds with an access token along with an optional refresh token. After acquiring the access token, the client can make requests to the resource server on behalf of the resource owner.

The response will contain at the very least the `access_token` and `token_type` parameters. Optionally, the response may also contain the `expires_in`, `refresh_token` and `scope`. The `token_type` defines the type of the token. The token may of type ”bearer” or ”mac”. The `expires_in` is recommended and informs the client on the lifetime of the access token in seconds. The `refresh_token` may be used by the client to obtain a new access token. The `scope` value is required only if the final scope of the access token differs from the client’s initial request. [4]

Implicit

The implicit grant flow is meant to be utilized by public clients, that have a well known redirection URI. The main difference between the implicit flow and the authorization code flow is that in the implicit flow the client is not authenticated and it receives the access token directly from the authorization endpoint after the resource owner authorizes the client.

The Figure 3.5 shows the detailed steps of the implicit flow. The steps of the implicit flow are as follows.

(1) The flow is initiated by the client, which will direct the resource owner's user agent to make an HTTP GET request to the authorization endpoint of the authorization server. The GET request will contain `response_type`, and `client_id`. Additionally, the client can include `redirect_uri`, `scope` and `state` parameters. In the implicit flow the value of the `response_type` parameter must always be `token`. Other parameters are as in the authorization code flow.

(2) The resource owner is authenticated and will be requested to authorize the client at the authorization server just like in the authorization code flow.

(3) If the resource owner authorizes the client access, the User-Agent is redirected back to the URI that was provided as the `redirect_uri` in step 1. The redirect URI will be appended with a fragment that will contain at least the values for parameters `access_token` and `token_type`. Optionally, the server may also return values for parameters `expires_in`, `scope` and `state`. All of these parameters are equivalent to the parameters in the authorization code flow. The main difference is that in the implicit flow the authorization server will not issue a refresh token.

(4) The user agent will be redirected to a web-hosted client resource, but, by definition, will not pass the URI fragment part containing the return parameters to the end point .

(5) The web-hosted client resource will return a web page, which usually contains a piece of JavaScript code capable of extracting the parameters in the URI fragment.

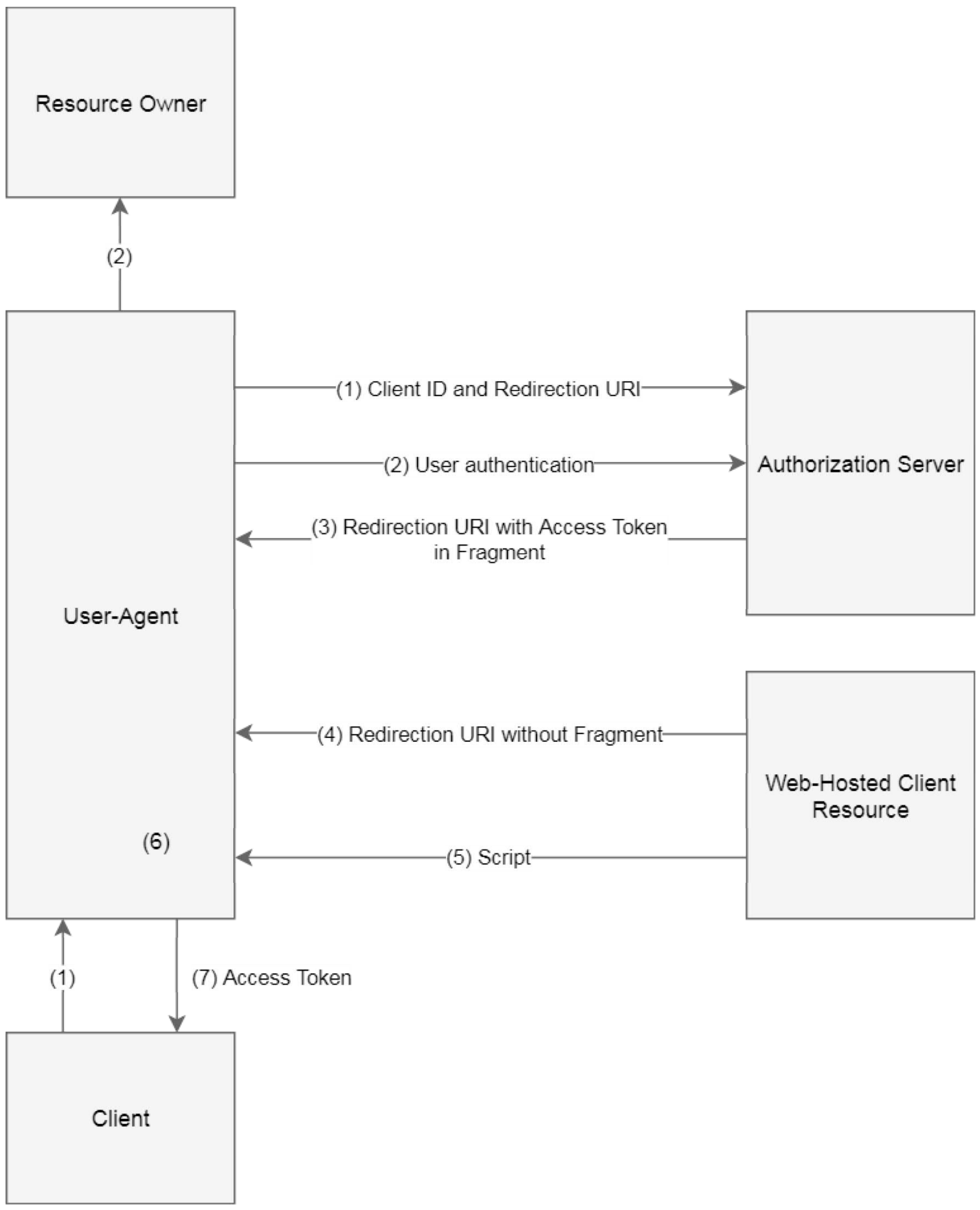
Resource Owner
(2)
User-Agent
(1) Client ID and Redirection URI
(2) User authentication
Authorization Server
(3) Redirection URI with Access Token in Fragment
(4) Redirection URI without Fragment
Web-Hosted Client Resource
(6)
(5) Script
(1)
(7) Access Token
Client

Figure 3.5: OAuth 2.0 implicit grant flow. Adapted from [4]

.

(6) The User-Agent will execute the script embedded in the web page returned by the web-hosted client resource. The script will extract the access token from the URI.

(7) The User-Agent will pass the extracted access token to the client. The detailed way that this is done, is left outside of the specification.

3.3 OpenID 2.0

This section provides a brief summary of the OpenID 2.0 protocol. The protocol can be considered the predecessor of OpenID Connect 1.0 even though they do not share a significant amount of technical details. The OpenID 2.0 is not technically compatible with the OpenID Connect framework. The OpenID 2.0 is, however, compatible with the OpenID 1.1 if certain implementation details described in the OpenID 2.0 specifications are met. [18] OpenID 2.0 is a protocol that enables end-users to authenticate to third party relaying parties without having to provide authentication credentials to the relaying party.

The protocol has some steps that have to be followed to successfully authenticate a user. These steps are in order of execution discovery, association establishment and finally, the user authentication. Each of these steps is described next in more detail.

3.3.1 Discovery

The initial step of the OpenID 2.0 protocol is the discovery, where the user supplies an identifier to the Relaying Party. The identifier could be for instance `https://example.com/user`. The RP then normalizes the user input and performs OpenID Provider Discovery based on this information.

3.3.2 Association

The association phase begins after the RP has discovered the OP. The protocol uses Diffie-Hellman key exchange (DH) protocol for the RP to exchange secrets with the OP. Alternatively, the use of may be omitted if the OP and the RP communicate on a secure TLS channel.

3.3.3 Authentication

After the RP has discovered and associated itself with the op, it can send the authentication request. The RP will redirect the user agent of the end-user to the OP along with an authentication request. The end-user authenticates to the OP and the OP will check whether the user is authorized to perform OpenID authentication.

The user agent is then redirected back to the RP along with information on whether the authentication was successful. The RP is required to validate the information received from the OP.

3.4 OpenID Connect

This section will explain some of the main concepts behind OpenID Connect (OIDC). The section will not and is not meant to cover all the details of the protocol, but rather focuses on the main things relevant to the OIDC certification process. An interested reader is advised to look into the official OIDC documentation [5] provided by the OpenID Foundation.

OIDC is an identity layer build on top of the OAuth 2.0 protocol. The OIDC uses RESTful HTTP APIs and JSON data format. Most of the specifications in OAuth 2.0 apply to also to OIDC. This also means that the OIDC specification has most of the

OAuth 2.0 capabilities integrated into the protocol. The focus of this section lies in some the capabilities that the OIDC adds on top of OAuth 2.0.

3.4.1 Abstract OIDC authorization flow

The abstract authorization flow of the OIDC is much like the one basic flow of OAuth 2.0. The OIDC adds the steps (5) and (6) to the OAuth 2.0 abstract flow.

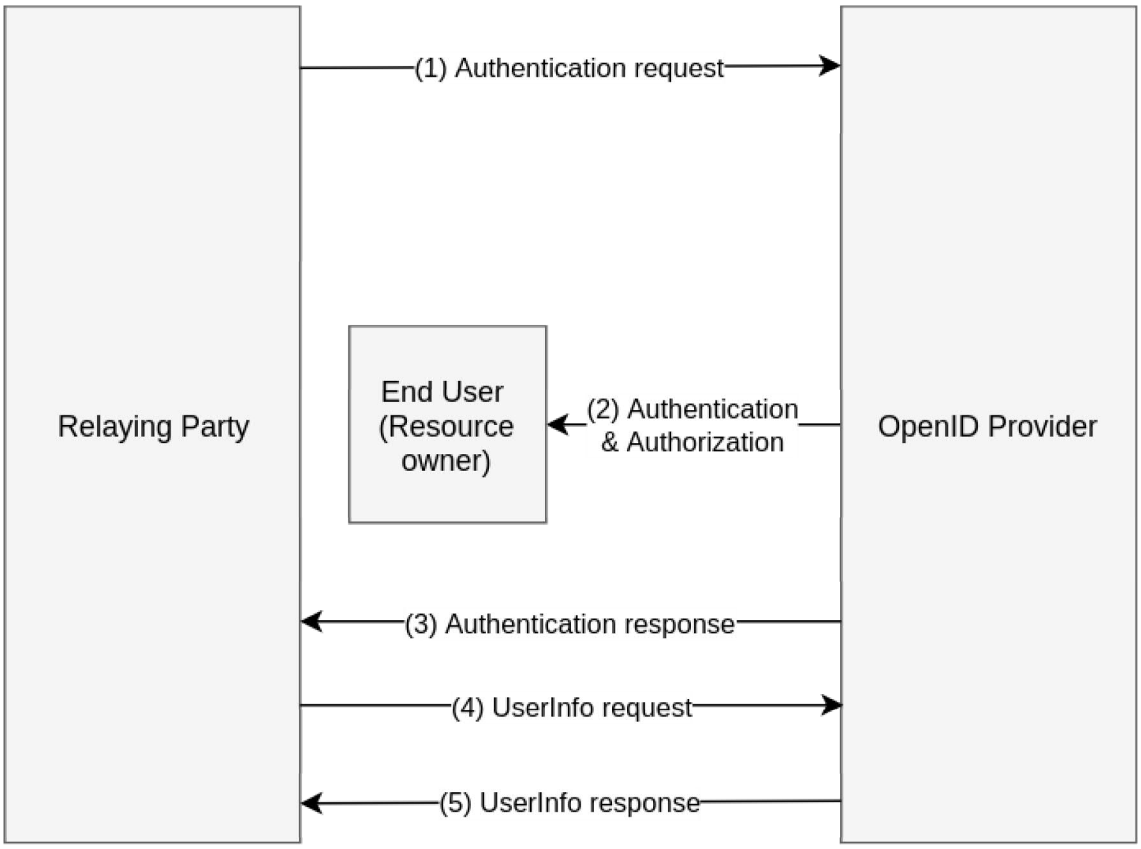
Relaying Party
(1) Authentication request
End User (Resource owner)
(2) Authentication & Authorization
OpenID Provider
(3) Authentication response
(4) UserInfo request
(5) UserInfo response

Figure 3.6: OpenID sequence diagram. Adapted from [5]

Figure 3.6 shows an abstract overview of the flow of the OIDC protocol. The flow steps are as follows.

- (1) The RP initializes the flow by sending a request to the OpenID Provider (OP).
- (2) The OP handles the authentication of the End-User and obtains authorization.

- (3) The OP responds to the relaying party by sending an ID Token along with an Access Token.
- (4) The Relaying Party (RP) makes a request with the Access Token to the `UserInfo` endpoint.
- (5) The `UserInfo` endpoint sends End-User Claims.

3.4.2 The protocol suite

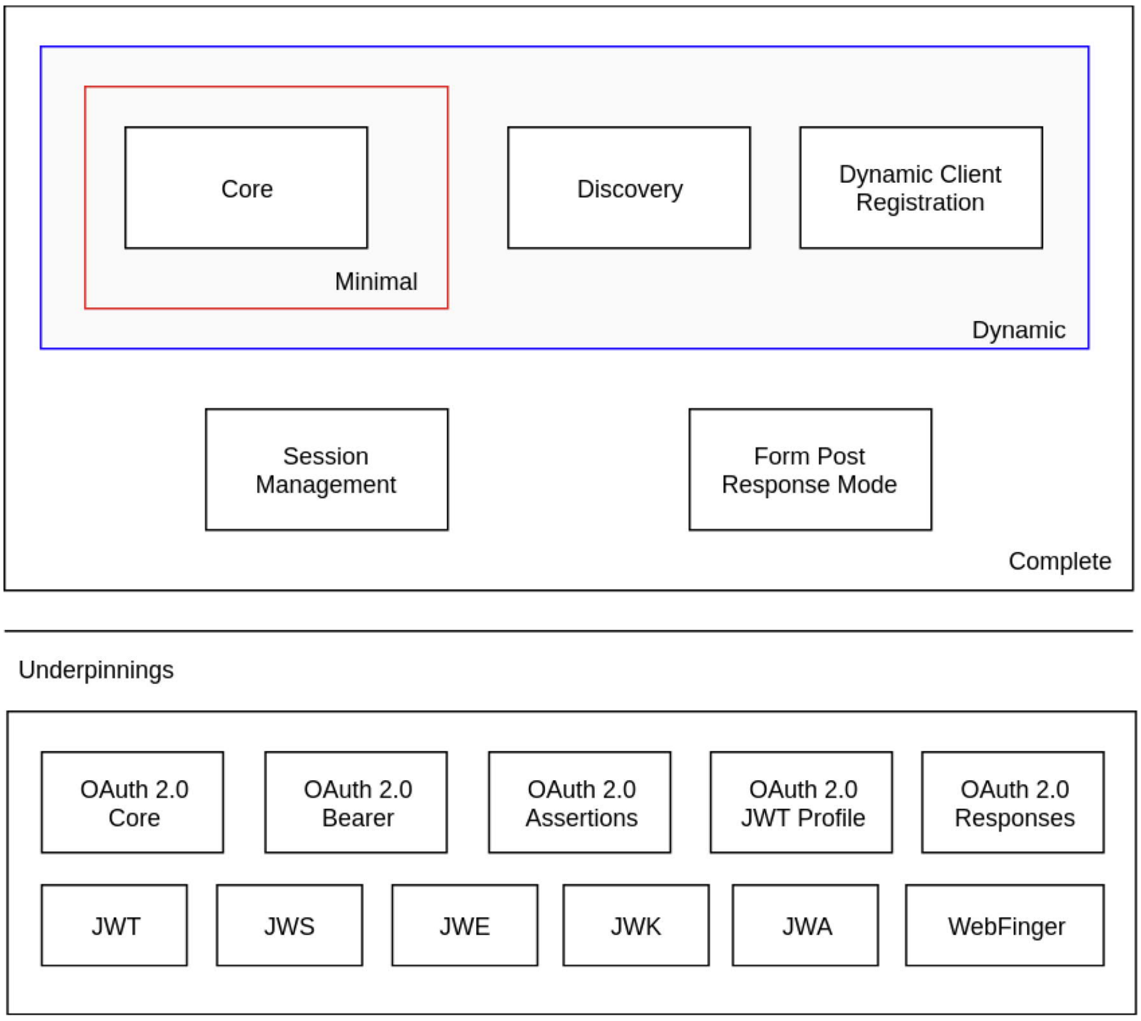

Figure 3.7: OpenID Connect protocol suite. Adapter from [6]

The structure of the OIDC protocol suite is shown in Figure 3.7. The OICD specification is separated into a number of different documents which are as follows:

- Core
- Discovery
- Dynamic Registration
- OAuth 2.0 Multiple Response Types
- OAuth 2.0 Form Post Response Mode
- Session Management
- Front-Channel Logout, and
- Back-Channel Logout

The core document has definitions for the main functionality of OpenID Connect. Discovery is optional and defines how the clients can find information about OIDC providers. The dynamic registration defines how the client can register with OIDC providers dynamically. For the sake of the certification process, the main parts of Core, Discovery and Dynamic Registration are covered in more detail in the in the following sections. The rest of the documents are not relevant and will not be discussed.

3.4.3 Core

The Core document [5] defines the most fundamental functionality for OpenID Connect. The Core contains specifications for *ID Token* structure, authentication flows and endpoints as well as the standard set of Claims that can be obtained about the End-User and the Authentication event. The Core specification also includes security and end-user privacy considerations.

JSON Web Token

JSON Web Token (JWT) is a data structure, which can be used to represent claims between two parties. It is described in RFC 7519 [19]. JWT has support for encryption and via JSON Web Encryption (JWE) and signing via JSON Web Signature (JWS). As the name suggests, the JWT uses JSON objects as its data structure. JWT Claims are key value pairs where the key (Claim Name) is always a string and the value (Claim Value) is an arbitrary JSON value. The JWTs are, however, always encoded using the JWS Compact Serialization or the JWE Compact Serialization.

ID Token

ID Token is a data structure built on top of OAuth 2.0 as an extension to enable end-user authentication. ID Token can be thought to be like an identity card. ID Token uses the JSON Web Token (JWT) format. The OIDC specification requires that the ID Tokens are signed by the OpenID Provider using JWS. Optionally, the ID Tokens may be signed using JWS and then encrypted using JWE. The signing provides authentication and integrity. Additionally, encryption provides confidentiality for the token contents.

The OIDC Core document defines the following Claims for the ID Token. ID Tokens may, however, contain additional application specific Claims. The presence of the Claims required by the OIDC Core specification is enforced by the certification testing tool.

- `iss` - An issuer, which identifies the principal that issued the token. In OIDC this is an URL with https scheme.
- `sub` - The locally unique subject, for which the ID Token belongs to.
- `aud` - The intended audience of the ID Token. The value is a single string or an array of strings. In OIDC one of the values must be the OAuth 2.0 client id.
- `exp` - The expiration time of the ID Token. The time is represented as the number of seconds after 1970-01-01T0:0:0Z.
- `iat` The time of issuing of the JWT. Uses the same time format as `exp`.

- `auth_time` - The time at which the End-User was authenticated. Uses the same time format as `exp`.
- `nonce` - Number used once, a string value, which links a client session to the ID Token. The `nonce` is used to mitigate replay attacks. This value must be present in the ID Token exactly like in the request, if the `nonce` parameter was present in the request.
- `acr` - String value, which tells the Authentication Context Class Reference. The use of this value is optional and is not required for the OP certification.
- `amr` - Authentication Methods References, optional value, which identifies the authentication methods used during the resource owner authentication.
- `azp` - Authorized party, in the context of OIDC this claim contains the OAuth 2.0 client id. The presence of this parameter is optional, unless the ID Token has only a single `aud` entity and it is not the same as the `azp` value. The presence of this Claim is not tested or required for the certification.

The `iss`, `sub`, `aud`, `exp` and `iat` Claims are always required. Additionally, the `auth_time` is required when `max_time` request is made or when an Essential Claim requests `auth_time`. Additionally the `nonce` Claim must be present when the client includes it in the request. The `nonce` parameter must be included in the request, when the `response_type` value includes either `token` or `id_token`.

```
{
   "iss": "https://server.example.com",
   "sub": "24400320",
   "aud": "s6BhdRkqt3",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "iat": 1311280970,
   "auth_time": 1311280969,
   "acr": "urn:mace:incommon:iap:silver"
}
```

Listing 3.4: An example of an ID Token containing a set of Claims

3.4.4 OIDC Endpoints

The endpoints of the OIDC are very similar to those of the OAuth 2.0 framework. The authentication

An addition to the OAuth 2.0 is the UserInfo endpoint, where the client can request the info of the end user in a form of selected claims. The set of claims to be included in the response is defined by the `scope` parameter value given during the authentication request.

The UserInfo endpoint is a protected resource, from which the client can request a set of Claims about the end user. The endpoint must be secured by TLS. The OIDC core specification states, that the endpoint has to support both HTTP GET and POST methods, and must additionally support the use of OAuth 2.0 Bearer token. The UserInfo endpoint will respond with either a JSON object or JWT in case the response was requested to be encrypted or signed during the client registration process.

The `scope` parameter value tells the OP, which claims should be returned in the UserInfo response and in the ID Token. However, the OP does not have to support other than the `openid` scope, which states that only the `sub` claims will be returned.

3.4.5 Authorization grant flows

The OIDC supports the OAuth 2.0 authorization grant flows, from which the most important ones are the Authorization Code and Implicit grant type flows. These two grant flows form the basis of the OIDC certification conformance profiles.

Authorization Code Flow

The OIDC requires the OAuth 2.0 Authorization Code flow to be used with some certain parameters. The `scope` parameter is required and it must at least contain the value `openid`. The request must include the `redirect_uri`, which must also match one of the redirect URIs registered by the client. The request may additionally include parameters `display`, `prompt`, `max_age`, `ui_locales`, `id_token_hint`, `login_hint`, and `acr_values`. The inclusion of the `prompt` parameter in the request is optional, but the OP must attempt to authenticate the end user, if the parameter value is `prompt`. Additionally, the OP must not interact with the user, if the value of the `prompt` parameter is `none`. The OP must return an error, if the end user cannot be authenticated.

The authorization code flow also requires that the ID Token optionally contains an `at_hash` Claim (Access Token hash). The value of the `at_hash` is a base64url encoded leftmost half of the hash of the `access_token`.

Implicit flow

In Implicit flow the difference in the request compared to the authorization code is that the `response_type` parameter must have a value of either `id_token` or `id_token token`. The `redirect_uri` must have an URI that is pre-registered at the OP for the client. The implicit flow requires the use of `nonce` parameter.

A successful authentication response will contain `access_token` parameter value, if the `response_type` is not `id_token`. Additional included parameters are `token_type`, `id_token`, `state` and optionally the response may also contain `expires_in` parameter.

For the ID Token the `nonce` Claim is required. The `at_hash` is required, if the ID Token was issued with an `access_token`.

3.4.6 Discovery

The OIDC Discovery specification defines a method, which clients can use to dynamically discover the configuration of the OP. The discovery protocol utilizes WebFinger to locate an OP. The Discovery specification can be thought to be divided into two parts. The first part describes the use of the WebFinger protocol in the context of OIDC and the second part describes the format of the OP metadata.

WebFinger

The WebFinger protocol, defined in RFC 7033 [20], is used to used to query information about static objects. The query path is always `/.well-known/webfinger`. The endpoint is issued an HTTP GET request with `resource` and optionally `rel` parameters. OIDC uses the `rel` value of `http://openid.net/specs/connect/1.0/issuer` to indicate that it requests a reference to the OIDC issuer. The `resource` is a user input in either email address of URL syntax. The RP will normalize the input and determine the location of the host.

OP metadata

The Discovery specification also specifies that OPs have to make a JSON document available at `/.well-known/openid-configuration`. The JSON document must by minimum contain the parameters in the following list. The OP may provide additional parameters, some of which are listed in the OIDC Discovery specification.

- `issuer` The issuer URL, must be identical to the URL provided by WebFinger
- `authorization_endpoint` OAuth 2.0 Authorisation Endpoint URL
- `token_endpoint` (optional, if only Implicit Flow is used) OAuth 2.0 Token Endpoint URL
- `userinfo_endpoint` (optional, but recommended) OP's UserInfo endpoint URL
- `jwks_uri` URL of the document containing the JSON Web Key Set, which are used by the RP to validate OP's signatures or encrypt requests to the OP
- `registration_endpoint` (optional, but recommended) OP's Dynamic Client Registration Endpoint URL
- `scopes_supported` (optional, but recommended) List of scopes supported by the OP. May not contain all supported scopes
- `response_types_supported` OAuth 2.0 Response Types supported by the OP. `code`, `id_token` and the `token id_token` Response Types must be supported.
- `subject_types_supported` list of Subject Identifier types supported by the OP
- `id_token_signing_alg_values_supported` list of JWS signing algorithms, which are supported by the OP
- `claims_supported` (optional, but recommended) list of Claim Names the OP may support. May not include all claims supported by the OP.

3.4.7 Dynamic client registration

The OIDC Dynamic Client Registration document [21] describes a method for RPs to use to register themselves to the OP. The specification defines endpoints for client registration and configuration.

A client can be registered to the OP by issuing a HTTP POST request to the Client Registration endpoint. The request must contain at least the `redirect_uris`, which is a list of redirect URIs used by the client. Additionally the clients may specify URIs for logo, privacy policy and terms of service as `logo_uri`, `policy_uri`, and `tos_uri` and, if given, the OP should show them to end-users. Also, if the client specifies `jwks` or `jwks_uri`, the OP must use the registered keys and support key rotation. All other metadata values are optional and as they are not required for the certification, they are not discussed any further.

The Client Registration endpoint may also require an Initial Access Token, which is an OAuth 2.0 Access Token can be used to restrict access to the client registration. The specification itself does not, however, describe its use any further. Its contents and methods of verification are application specific.

The Client Registration endpoint will return with a JSON document containing, at the minimum, the `client_id` parameter, which is a unique identifier of the client. All other parameters are optional.

Chapter 4

Certification

A software certificate represents the mapping of certified artifact to the property being asserted and the certification authority. A certifiable artifact can be software or non-software artifacts like documentation, for example. A certificate may additionally include technical evidence supporting the certification. [22]

4.1 OpenID Certification

The OpenID Connect Provider certification is done in a manner of self-certification. In self-certification the certification authority is the same as the author of the artifact being certified. In the case of OP certification, the artifact being certified is the OpenID Connect Provider implementation itself. In this context, there are no certificates for non-software artifacts.

The properties being certified are the implementation details of the OP implementation, which will be tested by using the official testing tool provided by the OpenID Foundation. The test results will be considered technical evidence supporting the certification.

The main purpose of the OpenID certification is to enable implementations of OpenID Connect to be certified to meet the certification requirements. The two components of the OpenID certification are technical evidence of conformance resulting from testing and legal statement of conformance. Most of this chapter describes the process of collecting the technical evidence for the certification.

The certified implementations are allowed to use the OpenID Certified logo. The OpenID Foundation maintains a list of certified OpenID providers and certified relaying parties.

4.1.1 Certification value

The value provided by the certification is fundamentally either technical or business related. From the technical point of view, the certification gives more certainty for the software and ensures that the integration of the implementation requires no custom code. However, even though to pass the testing process successfully, the OIDC OP implementation must follow some certain security related best practices, the OIDC OP certification does not ensure that the implementation is secure. The certification does not as such compete with or make security audits or security related certificates obsolete.

From the business point of view the certification may enhance the reputation of the organization and the implementation itself. The customers may think that the certified implementation is better than non-certified implementations and prefer to use the certified one.

4.1.2 Certification costs

The certification costs $200 for OpenID Foundation members and $999 for non-members. The cost for non-members wanting to certify a new deployment of an already certified implementation is $499. The cost covers all the profiles sent within a calendar year. Additionally, the certification is bound to have a running cost for the implementing company due to manual testing, the maintenance of any integration test as well as the OP implementation itself, and any additional legal fees that may occur. [23]

4.1.3 Conformance profiles

The OIDC certificate is divided into five different conformance profiles. The certificate for the OIDC implementation may include technical evidence for any combination of the profiles. For example, if the OP implementation supports the authorization code grant flow only, it may be certified to only conform to the Basic OP profile.

The five examined conformance profiles for OpenID providers are as follows: [24]

- Basic OpenID Provider
- Implicit OpenID Provider
- Hybrid OpenID provider
- OpenID Provider Publishing Configuration Information and
- Dynamic OpenID Provider.

For the sake of completeness, the five comparable conformance profiles for OpenID Relying Parties are listed below:

- Basic Relaying Party
- Implicit Relaying Party
- Hybrid Relaying Party
- Relaying Party Publishing Configuration Information and
- Dynamic Relaying Party.

The certification process for Relaying Parties is, however, not looked into any further in this thesis.

4.2 The OP Certification process overview

This section covers the OIDC certification process itself and the legal documentation related to it. The process consists of the following steps described in [25] and [26]. The technical details of the testing process are covered in later sections.

1. Setting up the testing environment
2. Running the selected tests on the test suite
3. Submitting the test results along with any required documents
4. Paying the certification fee described in subsection 4.1.2.

4.2.1 The testing procedure

The testing can be done by the party implementing the deployment or by a third party. The self-certification can be considered as trustworthy as the entity conducting the review. The entity conducting the review is legally fully responsible for the functioning of the implementation.

The testing is done against the OpenID specification with the web based testing application provided by the OpenID Foundation. The application can be found in `https://op.certification.openid.net:60000`.

After the tests have passed, the test result logs along with the signed legal declaration that implementation conforms to a profile are sent to the OpenID Foundation. The organization pays the certification fee and the implementation is verified and the certification is granted. The certification will be listed on OpenID Foundation certified implementations page at `http://openid.net/certification` and `http://www.oixnet.org/openid-certifications`. The certification includes logs of the tests in addition to the certification itself.

4.2.2 Testing, collecting the technical evidence

The collection of the technical evidence is vital for the sake of the certification process. As the certification is done as self-certification, the technical evidence is practically the only hard proof that the OP implementation truly conforms to the OP profiles.

This section lists the most relevant tests that the OIDC Provider (OP) implementation must pass in order for it to be certified. The tests are listed in two parts. First part covers the Basic OP, Implicit OP and Hybrid OP. The second part describes the tests that are performed for OPs publishing configuration and OPs supporting dynamic client registration. The conformance profiles for these types are Config and Dynamic, respectively.

The list of tests was acquired from the official OIDC Provider testing tool itself [27] and the OIDC conformance profile documentation [24]. Each of the test has to be passed for the implementation to be eligible for certification. Warnings returned by the test tool are acceptable, however.

4.2.3 Basic, Implicit and Hybrid OpenID Connect Providers

Essentially, the Basic OP profile requires testing for the `code` response type, the Implicit OP profile requires testing for `id_token` and `id_token+token`. Finally, the Hybrid OP profile requires testing for `code+id_token`, `code+token` and `code+id_token+token` response types.

response_type parameter

For the Basic, Implicit and Hybrid OP profiles, the `response_type` parameter must be present in the authorization request. If the parameter is not present, the request must be rejected.

For the Basic profile, the OP must support `response_type` with value `code`. The Implicit profile OP must support the `id_token` as well as `id_token`token+ as `response_type` parameters. According to the Hybrid OP conformance profile the OP implementation must support `code+id_token`, `code+token` and `code+id_token+token` response types.

ID Token

The implementations conforming to the Basic, Implicit and Hybrid profiles must support the following ID Token claims. The claims must be present in the Id_token returned by the OP.

- `iss` (issuer)
- `sub` (subject, the identifier of the end-user)
- `aud` (audience, which is the name of the OIDC client)
- `iat` (issued at, the time when the ID Token was issued)
- `kid` (key id, identifier for the key used to sign the id token)

All three conformance profiles mentioned before must support ID Token signing. Both Implicit and Hybrid profile OP implementations must always sign the ID Tokens and include an `at_hash` parameter when ID Token and Access token are returned from the authorization endpoint. Hybrid OP implementations must also provide a `c_hash` value when using `code+id_token` response_type.

The Basic OP implementations must support unsecured ID Token signatures if request parameter `signature` is `none`.

UserInfo Endpoint

OIDC Provider implementations conforming to Basic, Implicit or Hybrid profiles must have the UserInfo endpoint. The UserInfo endpoint must be accessible with HTTP POST method with bearer either in the HTTP request header or body. Additionally, the endpoint must be accessible with HTTP GET method with bearer in the HTTP request header. The UserInfo must also contain the `sub` claim.

Client Authentication

The Basic and Hybrid conformance profiles require the OP implementations to support client authentication. The client should be able to authenticate itself to the OP by making a HTTP POST request to Token endpoint and providing the credentials in the HTTP header as specified in the Basic schema. The credentials may also be provided by form encoding them in a HTTP POST request to the Token endpoint.

Nonce

The implementations conforming to Implicit and Hybrid OP profiles must enforce the presence of the `nonce` request parameter. Requests not containing `nonce` value must be rejected. The `nonce` request parameter is optional in the Basic OP conformance profile, but the OP implementation must still support it, if it is included in the request. If a `nonce` is included in the request, it must also be included in the ID Token returned by the OP.

Prompt and display request parameters

The testing tool also checks the OP's support for the `prompt` parameter. Basically, the `prompt` parameter tells the OP, whether it should show the prompt requesting reauthentication or user consent. The prompt parameter values can be either `login`, none, consent or select_account. At least the none and login should be supported by the OP. When the value is none, the OP must not show the authentication dialog at all and it must return an error if the user is not logged in.

The consent tells the OP, that it should ask for the end-user's consent before allowing the client to access certain information. The select_account, on the other hand, specifies, that the OP should ask the end-user with multiple accounts to select an account. Supporting consent and select_account prompt values is optional in practice and will not be tested.

The OP implementation conforming to either Basic, Implicit or Hybrid profile should also support display request parameter with values page and popup. The value page tells the OP to show the authentication UI in a full web browser page. This is also the default mode. The value popup tells the OP that it should show the authentication view in a popup such as a web browser popup window. At the minimum, the OP must not return an error when the display parameter is specified.

Scope request parameter

The scope parameter tells the OP, what information the client wants to request from the OP. The end-user is then informed which claims the client has requested. The testing tool verifies that the ID Token contains the claims that were requested. The scopes tested are all, address, email, phone and profile. However, since the OIDC providers are not required to support these scopes, not including them in the ID Token is does not fail the test. The testing tool will issue a warning, though.

Additional request parameters

Some additional testing is done to make sure that the OP functions when additional bogus parameters are given, as well as providing acr_values and claims_locales. The bogus parameter test makes sure that the OP continues to function correctly even when there is an additional not understood parameter.

The `login_hint` parameter should be supported. The string value provided should be written in the OP login prompt's username field when end-user's user agent is redirected to that endpoint.

Additionally, it is checked that the OP functions correctly if `prompt=none` and user hint via `id_token_hint` parameter is provided. The OP should return a positive response when a request is made with `prompt=none` and the `id_token_hint` has an ID Token of a user that is logged in.

Also, there are tests to see that authentication age is enforced via the `max_age` parameter and that UI locales can be provided with `ui_locales` parameter. It is enough that providing the `ui_locales` request parameter does not result in errors. For the `max_age` parameter, the OP should request the end-user to re-authenticate if the previous authentication is stale. Also, the OP should use the previous authentication if it is still valid.

Finally, the `redirect_uri` sent along with any request must match a registered `redirect_uri`.

OAuth behaviors

The testing tool checks some OAuth behaviors related to security. If the client attempts to use the authorization code twice, the OP should respond with an error result. The test is also repeated with the difference that there is a 30 second delay between uses. It must also result in an error. Also previously issued access tokens should be revoked if there is an attempt to use the authorization code twice.

4.2.4 Config and Dynamic OP conformance profiles

The Config and Dynamic OP conformance profiles are functionally essentially different from the Basic, Implicit and Hybrid profiles. The features required to implement by the Config and Dynamic profiles can be considered optional and are not directly related to OAuth 2.0 or the OIDC authorization flows. Instead they, define and enforce a set of specifications that the RPs can use to acquire OP configuration and a method for registering new clients. In practice, the Config profile is a subset of the Dynamic profile and there is probably little point in implementing just one of them. Therefore the section 4.2.4 will cover only the properties that are not required by the Config profile.

Config

The Config profile requires the implementation to share it's configuration according to the OP specifications.

The OP should publish OpenID configuration discovery information at `/.well-known/openid-configuration`, where must be an issuer that matches the prefix of the configuration endpoint URL. The issuer must match the ID Token `iss` value. The configuration must contain the URLs for `authorization_endpoint`, `token_endpoint`, `userinfo_endpoint` and `jwks_uri`. The keys in OP JWK must be well formed. The configuration must also publish parameters `scopes_supported`, `respose_types_supported`, `subject_types_supported`, `id_token_signing_alg_values_supported`and `claims_supported`. All of the OP endpoints must have the https scheme.

Dynamic

When the OP conforms to the Dynamic profile, if there are multiple registered redirect URIs the OP must reject all requests that do not have `redirect_uri` specified. The OP must also keep the `redirect_uri` parameter values in both registered and request parameter URIs and reject any requests with non-matching query parameter. The OP must also reject registration of `redirect_uris` with fragment. The OP should have well formed keys in JWKs. The OP should also support user identity discovery with WebFinger email and URI syntax.

The OP is required to have a `registration_endpoint` and that it can be used to register a new client. The login page on the OP should display URI for logo, policy and terms of service URI. It is not an error to not show these URIs and will not prevent the OP implementation from being certified.

The OP must support `request_uri` request parameter with both unsecured and signed requests. Additionally the OP must support the use of keys registered with the `jwks_uri` and `jwks`. Both OP and RP signing key rotation as described in the OIDC Core specification must also be supported.

Chapter 5

OnePortal and the OpenID Provider certification

This chapter describes the main features of onePortal developed by the Trivore Corp. The features of onePortal are described in the detail that is required for the reader to get an overview of the state of the OP implementation in onePortal.

5.1 OnePortal

OnePortal is a comprehensive and flexible web application platform with a cloud database and centralized identity management. The platform is multi-tenant by design, which means that multiple organisations are able to use is simultaneously. Figure 5.1 shows the platform's most important features.

OnePortal originally consisted of the Web UI, which is nowadays referred to as the Management UI as well as LDAP support. The REST API was developed mainly after the Management UI. OnePortal has since the beginning also had comprehensive messaging support for sending email and SMS messages. OnePortal later also gained support for

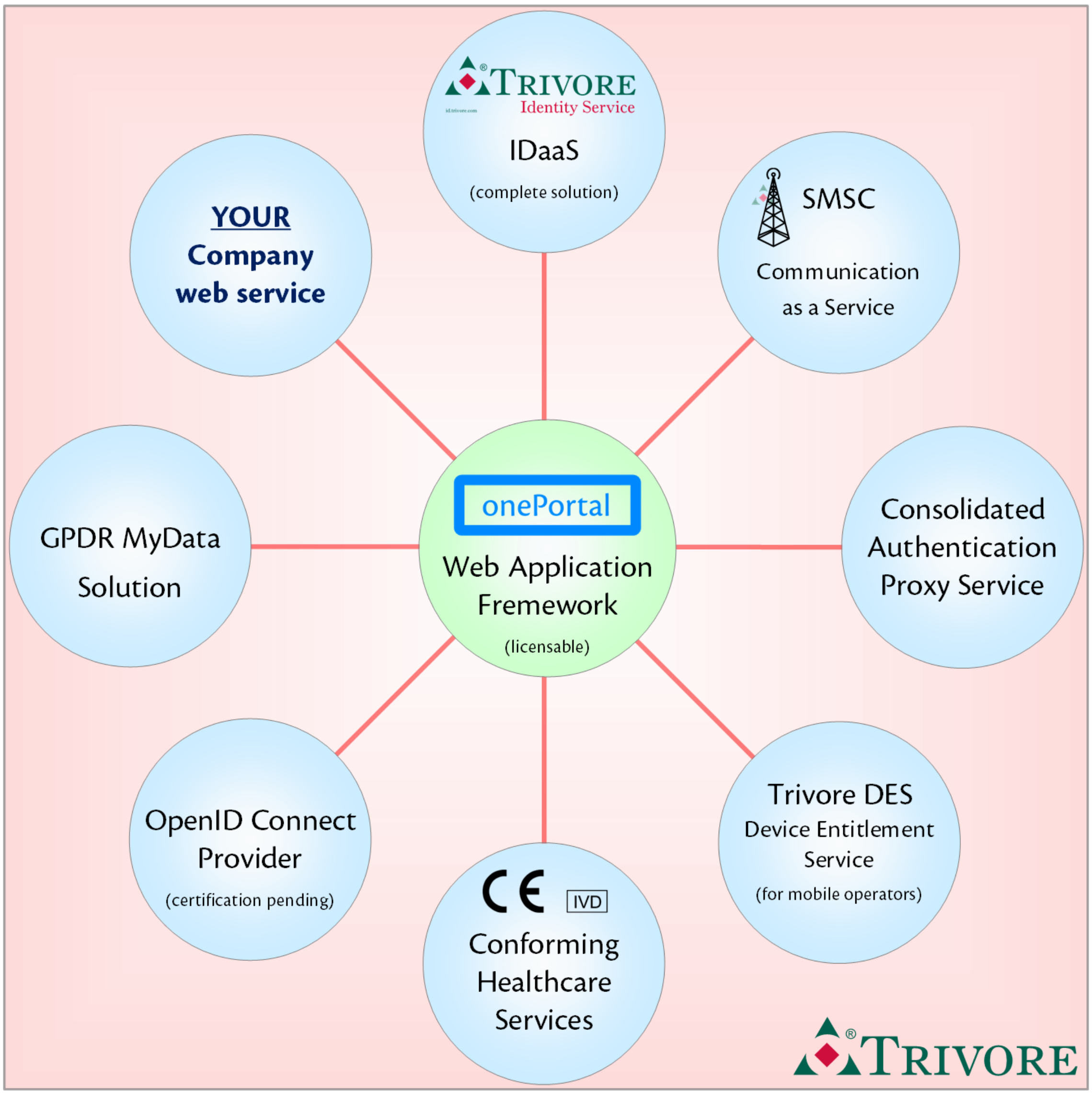

Figure 5.1: The onePortal framework overview. ©Trivore Corp.

OAuth 2.0 and OpenID Connect, and the REST API was then added support for OAuth 2.0 authorizations.

While onePortal is rich in features, this thesis will focus on the platform's identity management features, most notably its OP implementation. The details of the OP implementation are covered in the following sections.

5.1.1 Trivore Identity Service

The Trivore Identity Service works as a federated identity management platform. It is highly customizable to meet the requirements of customers. For example, the OAuth 2.0 sign-in screen can be themed to match the look and feel of the customer's website. Additionally, the group and namespace data structures are flexible and can be made to fit a wide range of various organization structures. Additionally, onePortal is fully GDPR compatible, it supports two factor authentication and has a Health Insurance Portability and Accountability Act (HIPAA) compatible secure audit trail.

5.1.2 OpenID Connect Provider Implementation

OnePortal has a custom OpenID Connect Provider implementation and its overview is shown in Figure 5.2. The following listings of supported properties were obtained from the onePortal's OpenID configuration endpoint.

So far, according to the OpenID configuration info published by the platform, onePortal's OP implementation supports the following authorization grant types:

- `authorization_code`
- `refresh_token`
- `implicit`
- `password`

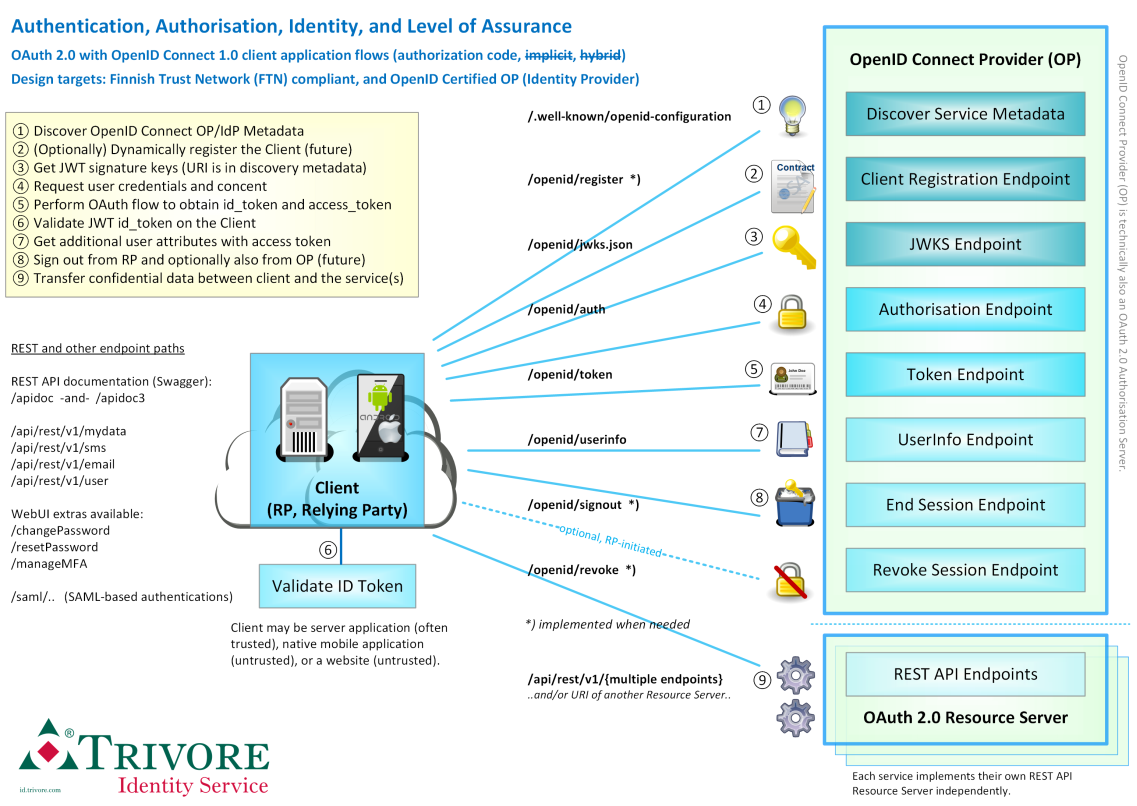
Authentication, Authorisation, Identity, and Level of Assurance
OAuth 2.0 with OpenID Connect 1.0 client application flows (authorization code, ~~implicit~~, ~~hybrid~~)
Design targets: Finnish Trust Network (FTN) compliant, and OpenID Certified OP (Identity Provider)
① Discover OpenID Connect OP/IdP Metadata
② (Optionally) Dynamically register the Client (future)
③ Get JWT signature keys (URI is in discovery metadata)
④ Request user credentials and concent
⑤ Perform OAuth flow to obtain id_token and access_token
⑥ Validate JWT id_token on the Client
⑦ Get additional user attributes with access token
⑧ Sign out from RP and optionally also from OP (future)
⑨ Transfer confidential data between client and the service(s)
REST and other endpoint paths
REST API documentation (Swagger):
/apidoc -and- /apidoc3
/api/rest/v1/mydata
/api/rest/v1/sms
/api/rest/v1/email
/api/rest/v1/user
WebUI extras available:
/changePassword
/resetPassword
/manageMFA
/saml/.. (SAML-based authentications)
Client
(RP, Relying Party)
⑥
Validate ID Token
Client may be server application (often trusted), native mobile application (untrusted), or a website (untrusted).
/.well-known/openid-configuration ①
/openid/register *) ②
Contract
/openid/jwks.json ③
/openid/auth ④
/openid/token ⑤
/openid/userinfo ⑦
/openid/signout *) ⑧
optional, RP-initiated
/openid/revoke *)
*) implemented when needed
/api/rest/v1/{multiple endpoints} ⑨
..and/or URI of another Resource Server..
OpenID Connect Provider (OP)
Discover Service Metadata
Client Registration Endpoint
JWKS Endpoint
Authorisation Endpoint
Token Endpoint
UserInfo Endpoint
End Session Endpoint
Revoke Session Endpoint
OpenID Connect Provider (OP) is technically also an OAuth 2.0 Authorisation Server.
REST API Endpoints
OAuth 2.0 Resource Server
Each service implements their own REST API Resource Server independently.
TRIVORE
Identity Service
id.trivore.com

Figure 5.2: The onePortal OpenID Connect flow. ©Trivore Corp.

The OIDC related endpoints utilized by the OP in onePortal are

- `/openid/auth` (The Authorization endpoint)
- `/openid/token` (The Token endpoint)
- `/openid/userinfo` (The UserInfo endpoint)
- `/.well-known/openid-configuration` (The configuration endpoint for discovery)
- `/openid/jwks.json` (The jwks endpoint where the public key is stored)

The onePortal OP supports the response types listed below. Note, that the response types required for the Hybrid conformance profile are not yet supported.

- `code` (Used for the authorization code grant flow)
- `id_token token` (Used for the Implicit grant flow)
- `id_token` (Also used for the Implicit grant flow)

The following default scopes are supported by the OP in onePortal. For the sake of certification, only the `openid` must be supported. The rest is optional but recommended.

- `openid`
- `profile`
- `email`
- `address`
- `phone`

OnePortal also supports a fair number of custom scopes, used to authorize access to various REST API endpoints. The number of these claims increases over time as more REST API functionality is implemented.

The following standard claims are supported by the OP in onePortal:

- `sub`
- `name`
- `given_name`
- `family_name`
- `middle_name`
- `nickname`
- `preferred_username`
- `email`
- `email_verified`
- `birthdate`
- `zoneinfo`
- `locale`
- `phone_number`
- `phone_number_verified`
- `address`

- `updated_at`

Additionally, onePortal supports a number of custom application specific claims, which are not relevant for the sake of certification.

The onePortal's OP implementation has support for some discovery features and the most current configuration can always be retrieved from any onePortal instance's `/.well-known/openid-configuration` endpoint as specified by the OIDC discovery document.

5.2 Testing the OpenID Connect Provider Implementation

For onePortal to gain the OIDC Provider certification, the OIDC Provider implementation has to be tested with the OIDC test suite provided by the OpenID Connect Foundation. Initial testing can be done using a locally hosted instance of the test suite, but when applying for the certificate, the testing has to be conducted with the OIDC foundations own test suite instance.

The tests in the suite were executed for each of the conformance profiles. The order that the conformance profiled tested was Basic, Implicit, Hybrid, Config and finally, the Dynamic profile. The following sections list the failures and inconsistencies that occurred during the testing procedures. The results of each of the conformance profile tested are listed in their own section. The focus of the descriptions is in the failed test cases so that the OP can be improved to fully conform to the profile in question.

Basic

The test cases for the Basic conformance profile described in section 4.2.3 were mostly run successfully. However, there were some test cases, that failed and others that resulted in warnings or inconclusive results. The rest of this section will cover the test cases in more detail.

The `response_type` related tests were run successfully. If the `response_type` is missing the server correctly returns an error and when the `response_type=code` the OP returns JSON encoded authentication message after authorization was granted.

The `UserInfo` endpoint related tests were also run successfully. The endpoint can be accessed with an HTTP POST request with Bearer token in the request body. The endpoint can also be successfully accessed with HTTP POST and GET methods with Bearer token in the request header.

The `claims` request parameter was tested with `name` claim with additional attribute `essential` set to true. The OP returns the requested claim from the `UserInfo` endpoint and the test is passed.

The `display` request parameter as described in the section 4.2.3 was tested with values `page` and `popup`. The OP does not return any errors when these values are used and therefore passes the tests.

The `prompt` request parameter, as described in section 4.2.3 was tested with values `login` and `none` with user logged in and not logged in. The tests were passed apart from the `prompt` parameter with value `none` when user was logged in. The OP returned an error with code `interaction_required` even when it should have determined that the user was already logged in.

With Basic profile with `request_type code`, the `nonce` parameter as described in section 4.2.3 may be given as part of the request. It was tested that when the nonce is not given, the OP does not return any errors. Also, when the `nonce` was given, the OP successfully returned it as part of the `access_token`.

The OP should also check that the `redirect_uri` request parameter is one of the registered URIs. The OP implementation in onePortal passes the test as it returns an error, when the given `redirect_uri` does not match a registered one.

Some additional request parameters described in section 4.2.3 were also tested. It was tested, that extra query parameters, `acr_values` or `claims_locales`, do not cause errors. Additionally, the `login_hint` parameter can be provided and its value is shown in the username field as expected. A request for a specific UI locale can also be provided with the `ui_locales` parameter without any errors being returned.

When the RP sends a request with the parameter `prompt` with value `none` along with an `id_token_hint`, the Authorization server should be able to determine from the ID Token of the hint, whether the user is logged in. In the conducted test case, the authorization server should have returned a positive response, but instead an error `interaction_required` was returned. This should be fixed before applying for the certification.

The OP correctly asked the end-user to re-authenticate when the `max_age` request parameter was given and the previous authentication was too old. However, when the authentication was still valid and the OP should have used the same authentication, it requested another unnecessary end-user authentication. This resulted in a failed test.

The OAuth 2.0 requires that, an attempt to use the same authorization code twice has to result in an error and that any previously issued access tokens has to be revoked. The OP correctly returned an error when the authentication code was used twice, but it still allowed data to be retrieved from the `UserInfo` endpoint with a revoked access token. This means that the access token was not fully revoked when the RP made a second request using the same authorization code.

Implicit

This section covers the test results for the implicit conformance profile. Most of the test cases are exactly the same in the Implicit and Basic profile. Thus, this section focuses only on the test cases not present in the Basic profile test set.

The main differences between the Basic and Implicit conformance profiles is that in the Implicit profile the access token request is made with the `response_type` parameter value `id_token` or `id_token token`. Additionally the `nonce` parameter is required and it must also be contained in the returned ID Token. When `response_type=id_token token` the ID Token must also contain the `at_hash` Claim.

The OP implementation passed all the Implicit profile related test cases. After improving the features that fail the test cases in the Basic profile, the OP should be certifiable for the Implicit conformance profile as well.

Hybrid

The OP does not support the request types required for the hybrid flow. The only supported response types are `code`, `id_token` and `id_token+token`. For the OP to be certified for the Hybrid conformance profile it should support the `code+id_token`, `code+id_token+token`, and `code+token` response types. Therefore the testing for the Hybrid conformance profile was not be conducted any further.

Config

The Config conformance profile tests require the OP to publish configuration information in a well known location. More specific requirements were discussed in section 4.2.4. It was tested, that the required set of information could be found from the OP's `.well-`

`known/openid-configuration` endpoint. The Config conformance profile related tests run successfully and therefore the OP is ready to be certified for Config conformance profile.

Dynamic

The dynamic client registration end points and WebFinger support have not yet been implemented in the OP. The client registration end point would have to be implemented before as discussed in section 4.2.4 the OP is eligible for proper testing.

5.3 Recommendations

After performing testing, some recommendations were given so that the OP implementation could be improved to fully conform to all of the OIDC conformance profiles.

Apart form Config, the OP implementation in onePortal still does not fully conform to any of the OIDC conformance profiles. The most significant limit in the implementation is the fact that it does not yet support dynamic client configuration or the OIDC hybrid flow.

The certification process can be completed in steps one or more conformance profiles at the time without additional certification fees. It is up to the company management, whether to initially acquire certification for Basic, Implicit and Config conformance profiles and later for the Hybrid and Dynamic profiles. Another option is to add the features required by the Hybrid and Dynamic profiles and do the certification for all profiles at the same time. Between these two options the step by step certification would seem better as the implementation could receive the certificate sooner and it does not imply additional costs.

Even if the dynamic client registration process should remain closed in a way that only selected Relaying Parties would be able to be registered, the registration endpoint can be protected by requiring an additional token called the `initial_token`. The `initial_tokens` can be generated and given to selected RP authors. In onePortal this kind of feature set would be feasible to be implemented by utilizing the existing management REST API to generate the initial token and share it to authorized parties. After acquiring the initial token the clients could register themselves dynamically to the OP by using the registration endpoint as described in the OIDC specification.

The following list contains the recommended improvements. The items listed should be fixed before applying for the full certificate.

1. Session management should be improved.
   (a) The correct functionality of the `prompt` parameter with value `none` should be ensured for cases with or without the `id_token_hint`.
   (b) Improve the handling of `max_age` parameter. The OP should allow the reuse of existing and still valid authentication.
2. Correct OAuth 2.0 behavior should be ensured.
3. Support for some of the standard claims should be added.
4. Support for the hybrid flow should be added by implementing support for the following `response_types`:
   (a) `code id_token`
   (b) `code token`
   (c) `code id_token token`
5. An endpoint for dynamic client creation and management should be implemented.
6. Optionally, support for the rotation of signing and encryption keys should be added.

Chapter 6

Conclusion

The aim of this thesis was to find out what has to be done in order to gain the certificate for the OP implemented as part of the Trivore Identity Management system. The results of this thesis show that some improvements have to be made for the implementation to conform to the certification requirements.

The first chapter worked as an introduction for this thesis explaining the subject and the structure of the thesis. The second chapter explained some of the key concepts of identity management, including authentication and authorization. The third chapter explained the main parts of OAuth 1.0, OAuth 2.0, OpenID 2.0 and finally, OpenID Connect.

The fourth chapter discussed the certification requirements for each of the OpenID Connect conformance profiles. Finally, the fifth chapter introduced the OP implementation and explained the test results for each of the conformance profiles. Also, recommendations to improve the implementation were given.

The OpenID Foundation published another conformance profile called ”Form Post”. This profile was left out of this thesis. The OP could also be tested and later certified for this new profile. Additionally, this thesis did not cover the maintenance aspects for the conformance testing. The implementation should be tested regularly after changes are introduced to the system. It should be figured out how the tests can be included in the existing testing process.

References

[1] David Ferraiolo, Janet Cugini, and D Richard Kuhn. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–48, 1995.

[2] Mikael Linden. *Identiteetin- ja pääsynhallinta*. Tampere University of Technology, 2015. ISBN 978-952-15-3568-0. URL `https://tutcris.tut.fi/portal/en/publications/identiteetin-ja-paasynhallinta(a62fef21-2c83-44c4-b771-94bc52105a69).html`.

[3] Andrea. Pashalidis and Chris J. Mitchell. A taxonomy of single sign-on systems. Springer, 2003.

[4] D. Hardt. The oauth 2.0 authorization framework. Rfc, RFC Editor, Oct 2012. URL `https://tools.ietf.org/html/rfc6749`.

[5] N. Sakimura et al. Openid connect core 1.0 incorporating errata set 1. Technical report, Nov 2014. URL `http://openid.net/specs/openid-connect-core-1_0.html`.

[6] OpenID Foundation. Openid connect, . URL `http://openid.net/connect/`.

[7] BS ISO/IEC 24760-1:2011: Information technology. Security techniques. A framework for identity management. Terminology and concepts, Jan 31,

2012. URL https://bsol.bsigroup.com/en/Bsol-Item-Detail-Page/?pid=000000000030143799.

[8] J. L. Camp. Digital identity. *IEEE Technology and Society Magazine*, 23(3):34–41, 2004. doi: 10.1109/MTAS.2004.1337889. URL http://ieeexplore.ieee.org/document/1337889.

[9] Dipankar Dasgupta, Arunava Roy, and Abhijit Nag. *Advances in User Authentication*. Springer, Cham, Aug 22, 2017. ISBN 9783319588063. doi: 10.1007/978-3-319-58808-7.

[10] M. Jones and D. Hardt. The oauth 2.0 authorization framework: Bearer token usage. Rfc, RFC Editor, Oct 2012. URL https://tools.ietf.org/html/rfc6750.

[11] J. Reschke. The 'basic' http authentication scheme. Rfc, RFC Editor, Sep 2015. URL https://tools.ietf.org/html/rfc7617.

[12] A. Barth. Http state management mechanism. Rfc, RFC Editor, Apr 2011. URL https://tools.ietf.org/html/rfc6265.

[13] David W. Chadwick. *Federated Identity Management*, pages 96–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03829-7. doi: 10.1007/978-3-642-03829-7_3. URL https://doi.org/10.1007/978-3-642-03829-7_3.

[14] Kelly D. Lewis and James E. Lewis. Web single sign-on authentication using SAML. *CoRR*, abs/0909.2368, 2009. URL http://arxiv.org/abs/0909.2368.

[15] Eran Hammer-Lahav. The oauth 1.0 protocol. Rfc, RFC Editor, Apr 2010. URL https://tools.ietf.org/html/rfc5849.

[16] OAuth Core Workgroup. OAuth Core 1.0 Revision A, June 24, 2009. URL https://oauth.net/core/1.0a.

[17] Martin Spasovski. *OAuth 2.0 Identity and Access Management Patterns*. Packt Publishing, Birmingham, 2013. ISBN 9781783285600. URL http://ebookcentral.proquest.com/lib/kutu/detail.action?docID=1572907. ID: 1572907.

[18] OpenID Foundation. Openid authentication 2.0 - final. Technical report, Dec 2007. URL https://openid.net/specs/openid-authentication-2_0.html.

[19] M. Jones et al. Json web token (jwt). Rfc, RFC Editor, May 2015. URL https://tools.ietf.org/html/rfc7519.

[20] P. Jones et al. Webfinger. Rfc, RFC Editor, Sep 2013. URL https://tools.ietf.org/html/rfc7033.

[21] N. Sakimura et al. Openid connect dynamic client registration 1.0 incorporating errata set 1. Technical report, Nov 2014. URL https://openid.net/specs/openid-connect-registration-1_0.html.

[22] Ewen Denney and Bernd Fischer. Software certification and software certificate management systems. 2005.

[23] OpenID Foundation. Openid certification frequently asked questions (faq) — openid, . URL https://openid.net/certification/faq/. Accessed: May. 4, 2018.

[24] OpenID Connect Working Group and OpenID Foundation. Openid connect conformance profiles v3.0. Technical report, Jun 2018. URL http://openid.net/wordpress-content/uploads/2018/06/OpenID-Connect-Conformance-Profiles.pdf.

[25] OpenID Foundation. Submission of results for ops — openid, . URL `https://openid.net/certification/submission/`. Accessed: May. 4, 2018.

[26] OpenID Foundation. Conformance testing for ops — openid, . URL `https://openid.net/certification/testing/`. Accessed: May. 4, 2018.

[27] OpenID Foundation. oidctest, . URL `https://github.com/openid-certification/oidctest/tree/v1.1.5`. Accessed: Jul. 2, 2018.