



CCV & Radboud University Nijmegen
Master thesis project



Secure Single Sign-On

A comparison of protocols



Author:

Nick Heijmink
nheijmink@gmail.com
S4250559

Supervisor:

E. Poll
e.poll@cs.ru.nl

Supervisor CCV:

Rogier Hofboer

Version: July 27, 2015

Abstract

In the age of internet and cloud computing, usability and security are extremely important. Swift interaction on the internet is, however, often compromised by the need of frequently supplying usernames and passwords. To prevent users from having to authenticate frequently, services can be provided with Single Sign-On. In this research, we focus on the requirements of CCV. This thesis discusses a use case that is of interest to many companies that are planning to centralise their authentication. CCV is a company that is specialised in offering payment solutions and various online services. These services include "CCV shop", which is a webshop service, and "my transactions" which is a service that gives an overview of all financial transactions in a certain period. By the use of Single Sign-On CCV wants to make its services more user friendly. There are, however, several SSO protocols available, all of which have different properties. This thesis provides a comparison of these protocols based on their suitability and security. The comparison should make it easier for CCV to decide which of the SSO protocols fits its needs best.

This study started by gathering information on seven different SSO protocols. We looked into their properties and compared each of them. At first we filtered the protocols based on functionality. This resulted in three protocols remaining: OpenID Connect, SAML and LDAP. Since this thesis focusses on a secure implementation, we performed a security analysis on the three remaining protocols. This security analysis is an analysis of publicly available information on the web. In this analysis, we found various security issues, none of which was critical. Because there were no critical security issues we investigated how often those protocols are used and whether there are still companies that implement the protocol. It turned out that SAML is used most often, but OpenID Connect is the fastest growing protocol. The LDAP protocol was built for server SSO in a local network, not for web applications and it is for this reason of less interest for this study. Both LDAP and SAML are becoming outdated [8], whereas OpenID Connect is new, web oriented and already used by many leading companies such as Google, Yahoo and Facebook. OpenID Connect offers authentication and authorization, uses modern standards and has a growing community. Because of this OpenID Connect is the best protocol for CCV.

After choosing OpenID Connect we had to choose which of the OpenID Connect flows fits best with the requirements of CCV. OpenID connect has three kinds of flows. After a small comparison we discovered that the so called "Hybrid Flow " is the best flow for CCV. It offers refresh tokens and does not require the client to directly contact the authorization and authentication server to get an access token and identity token. Refresh tokens are useful, they reduce the number of times a user has to re-authenticate which is good from a usability point of view. The fact that the clients do not have to contact the authorization and authentication servers means that they do not require any knowledge of them, and are therefore independent of the servers used.

CCV is planning to use verticals. These are a kind of layers that contain services which use the same resources and are independent of other services. Since each vertical is independent, it can be edited or removed without disrupting other services or functionality. All verticals have their own authorization server. This means that the SSO implementation should be able to handle multiple access tokens. We came up with a solution that can handle multiple tokens and uses OpenID Connect with the Hybrid Flow. This solution can be found in Section 8.11.

Keywords. SSO, OAuth2, OpenID Connect, SAML, CAS, LDAP, Cosign, OZ protocol, Single Sign-On, Single Sign-Off, JWT, Authentication, Authorization

Contents

1	Introduction	5
1.1	Requirements SSO solution	6
2	Materials & Methods	8
3	Background	9
3.1	Basic definitions	9
3.1.1	SSO terminology	9
3.1.2	Protocol and Flow	10
3.1.3	Security	10
3.1.4	Authentication or Authorization?	11
3.1.5	Proof of authentication/authorization	12
3.2	What is SSO?	14
3.3	Why use SSO?	16
3.4	Enterprise SSO vs Web SSO	16
3.5	Permission to share information	17
4	Background on the protocols	18
4.1	Cookies and Tokens	18
4.1.1	Cookies	19
4.1.2	JSON Web Tokens	20
4.2	OAuth2	21
4.2.1	The protocol	22
4.2.2	Evaluation of OAuth2	24
4.3	OpenId Connect	25
4.3.1	The protocol	25
4.3.2	Evaluation of OpenID Connect	26
4.4	SAML	27
4.4.1	The protocol	28
4.4.2	Evaluation of SAML	28
4.5	LDAP	29
4.5.1	The protocol	29
4.5.2	Evaluation of LDAP	31
4.6	CAS	31
4.6.1	The protocol	32

4.6.2	Evaluation of CAS	33
4.7	CoSign	33
4.7.1	The protocol	33
4.7.2	Evaluation of Cosign	34
4.8	OZ	34
5	Functional comparison	36
5.1	OAuth2 vs OpenID Connect	36
5.2	OpenID Connect vs SAML	37
5.3	OpenID Connect vs LDAP	37
5.4	OpenID Connect vs CAS vs Cosign	38
5.5	Conclusion	38
6	Security of the proposed solutions	40
6.1	Attacker model	40
6.2	General security issues	41
6.2.1	Eavesdropping and stealing cookies and tokens	41
6.2.2	Spoofing of tokens	44
6.2.3	DNS Spoofing	46
6.2.4	Forgotten log out	46
6.3	Known security issues in OAuth2 and OpenID Connect	47
6.3.1	CSRF Attack	47
6.3.2	Open redirect attack	48
6.4	Known security issues in SAML 2.0	51
6.4.1	XML signature wrapping attack	51
6.4.2	Man in the middle attacks on bindings and profiles	53
6.4.3	User tracking	53
6.4.4	Replay	53
6.5	Known security issues in LDAP	54
6.5.1	LDAP basic security settings	54
6.5.2	LDAP injection	55
7	Final comparison	56
7.1	Comparison of adoption	56
7.2	Choosing the protocol	57
8	The solution for CCV	58

8.1	Which flow?	58
8.2	Centralized or decentralized authorization	59
8.3	By value or by reference tokens?	60
8.4	Possible flows	62
8.4.1	Flow 1	62
8.4.2	Flow 2	63
8.4.3	Flows after authentication & authorization	64
8.5	Two-factor authentication	65
8.6	Role based access control	66
8.7	Migration process	66
8.8	Signing and encrypting	67
8.9	Verticals	68
8.10	Managing authorization	69
8.11	The current and new system	71
8.12	Security management	71
9	Existing SSO implementations	77
9.1	ForgeRock	77
9.1.1	OpenAM	77
9.2	WSO2	78
9.2.1	External database connection	78
9.3	Passport and Express Server	79
9.4	WSO2 and passport	82
9.5	Conclusion of the implementations	85
10	Future work	86
11	Conclusion	87
12	Glossary	89
	Appendix A Federated Identity Management	91
	Appendix B OAuth2 alternative flows	93
	Appendix C Code Passport application	94
	Appendix D WSO2 and Passport application	97

1 Introduction

This master thesis consist of a study of Single Sign-On, that was executed on behalf of CCV. CCV is an international organization that is specialised in electronic payments. They have offices in the Netherlands, Belgium, Germany and Switzerland. This study was executed at the CCV headquarters in Arnhem, because this made it easier to consult the stakeholders. Since 2011, CCV is an official payment institution under the supervision of the Dutch Central Bank.

CCV has a large number of customers (more than 137,000). These customers are mainly small and medium enterprises, but they also includes some big organizations. They deliver: payment terminals (more than half a million sold), offer services for these terminals and handle the transaction data. Most of the payment terminals in the Netherlands are CCV terminals.

CCV offers its customers services like the following:

- *MyCCV*, which allows users to see their transaction data and
- *CCV shop*, in which customers can set up their own webshops.

Customers currently log in to the CCV services through a Drupal login system. This system gives customers access to several CCV services. CCV has already updated its system to a token based authentication system which handles the login for the user.

CCV is looking for a user friendly way to connect their services to each other so they look like one seamless system. This can be achieved with the use of Single Sign-On (SSO). However, many SSO protocols are available. In this study we determine which SSO protocol meets the requirements of CCV best. To answer this question CCV requested an overview and comparison of various SSO protocols. CCV had made a list of requirements which the SSO protocols should meet, these can be found in Section 1.1.

This research begins in Chapter 4 by considering the functionality of Single Sign-On protocols and determining which of them is secure. It is possible that some protocols require precautionary measures, such as the use of HTTPS and HTTP-only cookies.

In Chapter 7 we determine whether the protocol has a large community and compare the cons and pros of each SSO protocol. The proposed protocols are analysed through the study of online information and discussing with faculty members of the Radboud and consulting security experts within CCV.

For the final implementation, existing SSO implementations can be used or one can be built. Existing access management solutions are available in both open and closed source. We will look into two different open source SSO solutions and provide an overview of how they work, what functionality they have and the difficulty of implementing them.

Problem context: Although research has been done into various SSO protocols, none of them have the same requirements as CCV. In addition most research is focused on comparing the properties of two specific protocols. In this study we will compare multiple protocols. The focus of this study is to provide insight into various SSO protocols that meet the requirements (or that can be extended as to meet the requirements) and to compare them. Based on the pros and cons of each protocol, one protocol will be chosen for which an implementation plan should be made.

Figure 1 shows the context diagram. This diagram provides a simple overview of what a SSO protocol could look like. This protocol defines multiple users that use computers, tablets, mobile phones etc. (which we call clients). These users want access to a service which can be data, an application, or a part of an application. Using SSO the client retrieves a proof of identity and access, which is used to access the services.

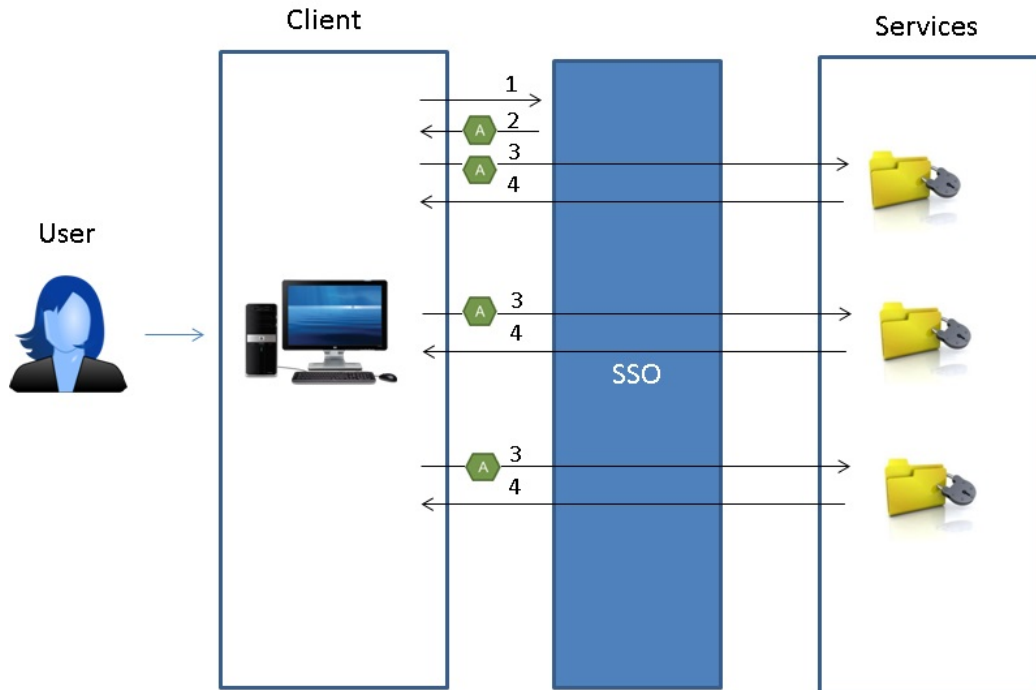


Figure 1: Context diagram

Structure: This thesis begins by defining the requirements of the final solution. We will then explain how information was collected for this study and how we translated it into the final result. Chapter 3 provides background information on the terms and definitions used in the thesis. The SSO protocols we compared can be found in Chapter 4. We conclude with an implementation plan for the protocol that best fits the requirements of this thesis. The best protocol will be selected on the basis of the comparisons of the SSO protocols conducted in Chapters 5 and 7. Afterwards we compare some applications and libraries that implement the best matching SSO protocol. We conclude this thesis with a short overview of our goal, the issues that we faced and the solutions that we have found.

1.1 Requirements SSO solution

This study will look at the properties of multiple SSO protocols that are available. CCV has some requirements that need to be taken into account when choosing an SSO protocol.

1. The protocol should provide both authentication and authorization (as defined in Section 3.1.4).
2. The protocol should be web-oriented, as all services that need to be connected using SSO are web based.
3. The SSO protocol must be able to handle services from multiple domains (e.g. ccvshops.nl and myccv.nl).

4. It is preferred that the protocol is mature (i.e. used by a large community), since this makes it easier to find support if problems occur during the research or implementation phase. It is also likely that more people have performed a security analysis on these protocols, which makes it more likely that security issues have been found.
5. CCV prefers that the solution is built in Java, NodeJS or C#, since in-house knowledge is available for these programming languages.
6. The CCV services consist of different applications that are written in multiple programming languages. Because the services are built in multiple programming languages, we need to communicate using a technique that can be used by all of them. All of the programming languages used have support for JSON and XML, therefore this research aims at an SSO solution that communicates in these two formats.
7. The use of two-factor authentication should be possible when users try to access certain confidential information. The idea is that the first authentication factor is used to log into the system and see most information. When a user wants to access a certain page with confidential information, then the second authentication factor is required (e.g. an SMS code).
8. Some SSO protocols require users to give permission before personal data is shared with a service. It would be nice if this permission notice can be avoided when authenticating to websites that are trusted by the SSO solution, since permission request are not necessary in this context and disturb the seamlessness.
9. The advised solution should have one sign on page that grants access to multiple websites, such as www.ccv.nl, www.ccv.nl/myccv and other domains like "*ccv-shops.nl*".
10. It would be nice if minimal changes have to be made to existing services.
11. CCV plans to implement verticals (see Section 8.9). The final SSO implementation should cope with these verticals.

2 Materials & Methods

We started this study by speaking with the stakeholders and collecting the requirements of the project. The goal of this study is to come up with a solution that makes it possible for users to log in once and afterwards have access to all services. CCV has already indicated that their problem could be solved with Single Sign-On (SSO). However, multiple SSO protocols exist, so CCV wants to know which SSO protocol fits their requirements best. To answer this question we collected information on seven different SSO protocols. We collected published articles, manuals, books and online sources. On the new protocols fewer published articles and books are found and for that reason online sources are used in general. To ensure that the information is correct we used multiple sources and compared the information we collected from them.

After collecting information we tried to find common information on all protocols. During this process we determined that information was still missing for some protocols for making a good (thorough) comparison. We collected the missing data and made a functional comparison of the protocols. Afterwards it turned out that more research was required to make a well informed choice. Since we are looking for a secure solution we looked into the security issues of the protocols that remained after the comparisons. However, this analysis still did not leave us with a clear indication of which protocol to use. We therefore made our final decision on the basis of the number of recent implementations, the adoption and the release date.

After choosing a protocol we had to define which configuration should be used for CCV. We started by choosing the right flow. We knew that the flow has great influence on the final solution, because each flow has different properties. Based on this choice we tried to make a sequence diagram of the new solution. During the process we discovered that it might be interesting to use decentralized authorization. Without decentralized authorization access tokens may become huge and give access to many critical services. Also the idea of decentralized authorization corresponds with the idea behind the use of verticals. During this study we came across "reference tokens" and "value tokens" and their pros and cons. Based on the results from the token types and the choice for decentralized authorization we identified two possible solutions.

During the study we encountered issues that need to be addressed, but are independent of the chosen solution. These are issues CCV will come across when they implement SSO, such as migration and signing and encrypting. We documented these issues to make CCV aware of them so that they may solve them before they start to implement SSO.

We combined all information gathered during this research into this thesis. Based on this information we answered the research question. We offered advice that explains how the new solution should work and that shows the current solution along with it.

3 Background

This chapter provides some background information about the topics that are handled in this thesis. We discuss what we mean by SSO, show its relation with Federated Identity Management and provide an overview of common SSO structures and definitions.

3.1 Basic definitions

Articles on SSO include a mixture confusing terminology. To prevent misinterpretations, we therefore start this thesis by providing definitions.

3.1.1 SSO terminology

For SSO protocols various abbreviations and terms are used for the systems involved. In this section we explain what these abbreviations and terms mean and what they are used for. These definitions are based on definitions from various papers [30, 82, 85, 91].

Definition 1. **IdP** stands for Identity Provider. An Identity Provider (IdP) is responsible for the authentication process and handles the storing of user information as attributes in an identity token. When a user authenticates, the IdP creates an object that contains the information of the user, which can be used when a service provider requests user attributes.

An IdP can be compared with the government. They have certain personal data about their citizens. When you want to travel to another country, you need a passport. Using the government you can get a passport, which constitutes proof of your identity.

Definition 2. **AS** stands for Authorization Server. The authorization server issues access tokens to the clients after successful authentication. When an IdP returns access tokens, then the IdP also operates as an authorization server.

The AS can be compared with a travel agency that provides flight tickets to a customer. This ticket proves that you have access to a certain flight.

Definition 3. **SP** stands for Service Provider ¹. The service provider offers a certain service and is in general protected by a kind of security guard. When a user wants to use the service the guard requests identifying information from the user.

The SP can be compared with an airport. Before you may catch your plane you must prove to a guard that you should be on that flight by using a flight ticket and your passport to prove that the ticket is yours. Based on your passport and ticket, the guard decides whether you may enter the flight.

Definition 4. When we speak of a **client**, we do not mean a person or company, but the device (pc/laptop/tablet/mobile phone/etc.) through which a user accesses a service.

Definition 5. A **user agent** is software on the client, that is used to communicate with the servers in the protocol. Since we are looking for a solution that connects web services with each other the user agent is typically a web browser.

Figure 2 shows a simple example of an SSO implementation. The figure illustrates a user that wants to go on a flight to a tropical destination. To get access to the flight the service provider wants to know who is boarding the flight and whether this person is authorized to do so. The SP asks for a passport and a flight ticket. To get a passport the passenger

¹Some use the term "resource server", which has the same meaning.

goes to the town hall(IdP) and requests a passport. An employee verifies the identity and provides a passport. The user goes to a flight agency(AS) and buys a flight ticket. When the user arrives at the airport he hands over his ticket and passport to the service provider, who will grant or deny access to board the plane.

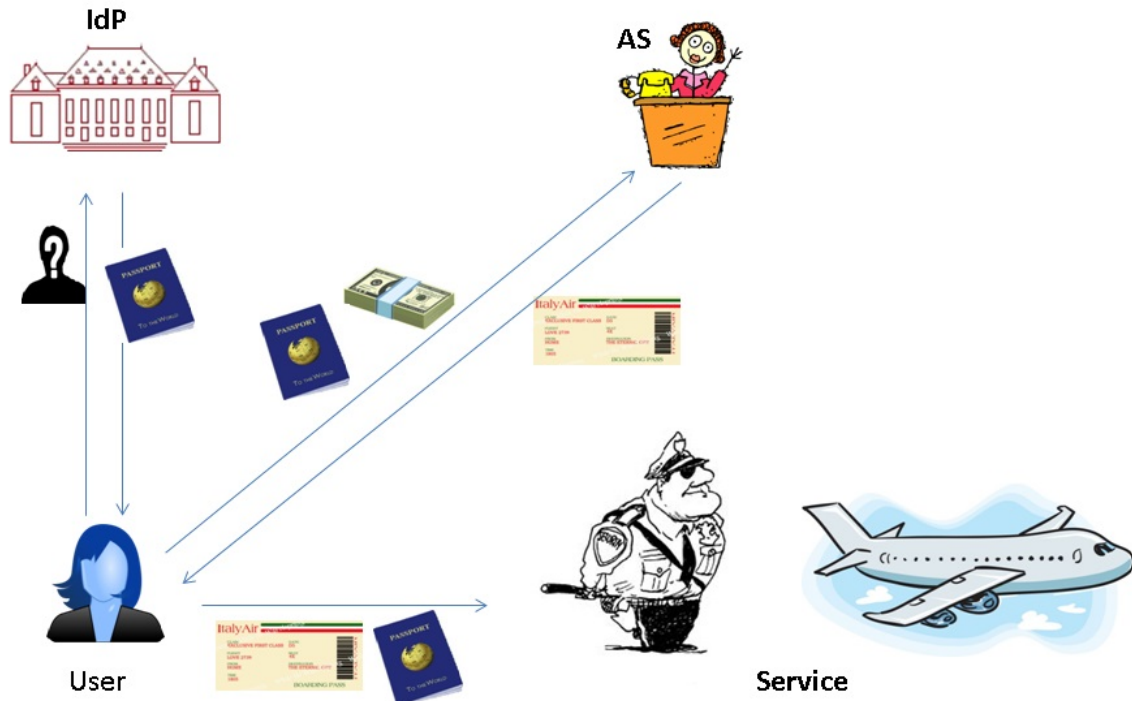


Figure 2: SSO actors

3.1.2 Protocol and Flow

Protocol and flow are different words that can be used to mean the same thing. Therefore we explain which term will be used and for what. In this thesis the term "protocol" is used to define a complete solution, for example "the OAuth2 protocol". Some of the SSO solutions consist of multiple sub-protocols. We use the term "flow" for a sub-protocol in a protocol, for example the "Implicit Flow" is a flow in the OAuth2 protocol. We use flow, as this term is used by many sources and is a common term inside CCV [45, 64, 69, 75, 82, 91]. OAuth2 has its own name for flows: "Grant", another word for the same thing.

3.1.3 Security

Security is a term that is used for many different things. Security can refer to a nightclub bouncer, a national army, a safe place to store valuables or for example "the state of feeling safe". Since we have specified that we are looking for a secure SSO solution it is important to have a tight definition of security.

Definition 6. For this thesis we use the definition of **information security** defined in ISO 27000[21]: "the preservation of confidentiality, integrity and availability of information".

Other properties, such as authenticity, accountability, non-repudiation, and reliability are also relevant .

3.1.4 Authentication or Authorization?

Many people confuse the terms authentication and authorization, therefore a small recap of the definitions of these terms.

Definition 7. By **Authentication** we mean: the ability of a party to determine the identity of the other party in a transaction. This authentication may be one way or mutual [14, 55] .

Definition 8. By **Authorization** we mean: allowing or preventing the use of a particular resource or service by an authenticated party [14, 55] .

Based on these definitions we can say that authorization requires authentication. Some also take advantage of the fact that authorization requires authentication [13]. Some abuse authorization by turning it into a kind of pseudo-authentication, on the basis that if the client obtains an access key from the authorization server through OAuth2, and shows it to service S, then S assumes that the client authenticated the authorization server before granted the access key. OpenID Connect uses this principle.

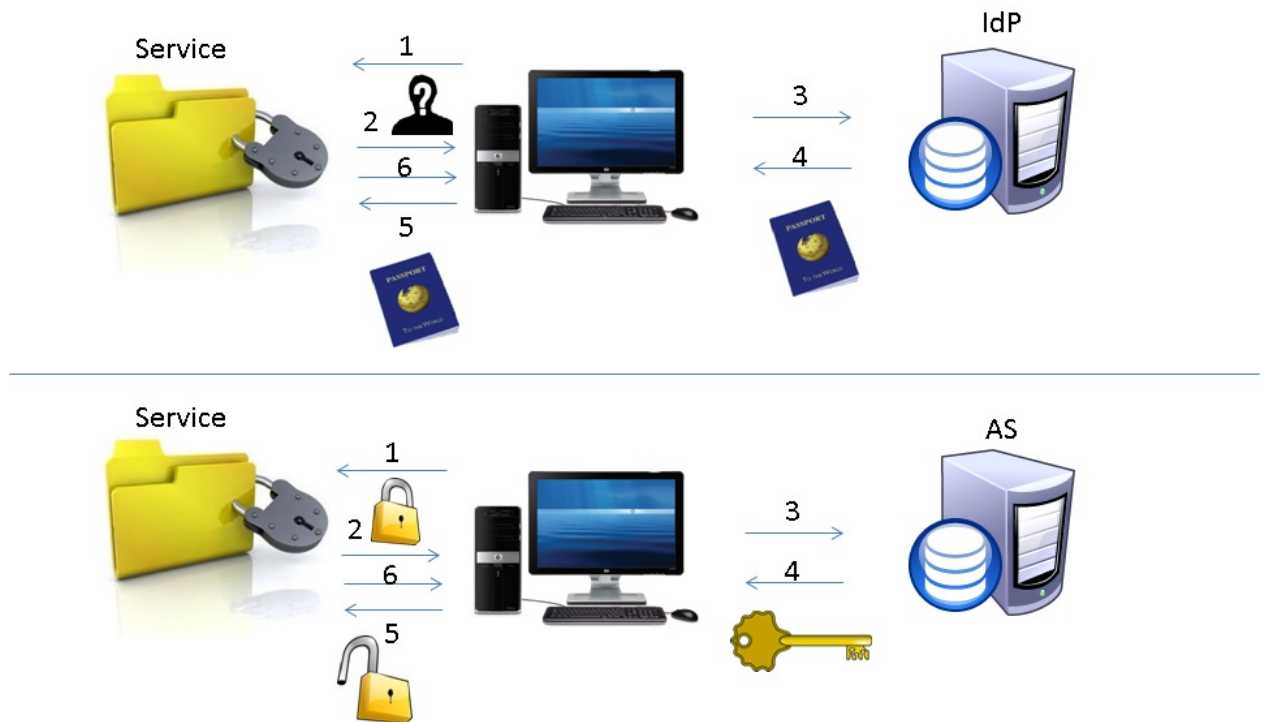


Figure 3: Pseudo Authentication

Figure 3 shows the difference between authentication and pseudo authentication. The protocol at the top asks the IdP for the identity of the client. The IdP verifies the identity based on user credentials and gives a proof of identity (which can be seen as a sort of passport). In this way the client proves to the service that he is who he says that he is. The service then determines whether the user is allowed to see the source. In the bottom protocol the client wants to access a service. To prove the user has access to the service, the service asks the user to open a lock. Only people that have the key to open this lock are allowed to enter the service. The client asks an AS to give him the access right of the user. The AS looks at the access rights of the user and creates an access token. This token can be seen as a key which gives access to open certain locks. If the client can use this key

to open the lock provided by the service, then the service believes that the user is allowed to enter, though it does not know who this user is.

Consider the following hotel analogy: a cleaner opens the door of my hotel room using a key she owns. Then because she has a card to my room (and presumably some hotel related clothing) I assume that she is a real cleaner. I trust that the key master would not have given her a key that gives access to my room if she was not a real cleaner.

OpenID Connect operates in an interoperable way that makes it possible for clients to verify the identity of the end user based on the authentication performed by an authorization server and to obtain basic profile information about the end-user.

3.1.5 Proof of authentication/authorization

In addition to the various types of flows, systems and entities there is also variation in the types of proof of authentication and authorization. These proofs can be represented in the form of access tokens and data in cookies.

Definition 9. To create an authentication cookie or token a user provides login **credentials**. In most cases these consist of a username and password. These credentials are verified by the IdP which returns a cookie or token if the credentials are correct.

Definition 10. **Tokens** can be seen as objects that contain a set of claims and are stored in local storage or in a cookie.

Definition 11. **Claims** are the predefined attributes and their values.

None of the claims defined below are mandatory, they are a starting point for a set of useful, interoperable claims [42, 89].

***iss:** Issuer of the token.*

***exp:** The timestamp after which the token expires.*

***iat:** The time the token was issued. This time is used to determine the age of the token.*

***nbf:** "not before" identifies a moment in time after which the token will be active.*

***jti:** Unique identifier for the token. Used to prevent replay attacks and to detect manipulation of tokens.*

***sub:** Subject of the token.*

***aud:** Audience of the token.*

Below an example of a claim Set:




```
{
  "iss":"joe",
  "exp":1300819380,
  "http://example.com/is_root":true
}
```

A SSO provider and service may agree to any claim name that is not one of the predefined attributes. Unlike public names, these private names are liable to collision and should therefore be used with caution[42].

Definition 12. An SSO flow uses so called **identity tokens** (or **ID tokens**) to prove authentication. Some SSO protocols also use **access tokens** for the authorization. The identity tokens are sent to the client after authentication and stored in a cookie or in local storage. If the client wants to access the service, then the token is sent to the service which validates the token. If an access token is also sent it can also be used to determine whether access will be granted to the service.

Definition 13. Typically an access token has an expiration time after which the token has to be refreshed or replaced by a new one. **Refresh tokens** can be used to refresh expired tokens. These are sent to the AS or IdP depending on the implementation used.

This thesis uses various images to represent these tokens. For the tokens the following symbols have been defined:

-  Access Token
-  Identity Token
-  Refresh Token

Definition 14. Not every web application or web source uses tokens. Many of them store data (e.g. an identifier) in a **cookie**. When trying to access a service the client sends the cookie containing a reference to the authentication session. When the client tries to access a service, the cookie is sent to the server which determines if the reference inside the cookie exists and checks to see if it is expired.

3.2 What is SSO?

Single Sign-On is a term that was used already in the 90s and is still in use today. In the beginning it was an enterprise solution that was used to log users into applications automatically, but over time the aim changed from enterprise towards the web [18].

Definition 15. People have several views on the meaning of **Single Sign-On** (SSO). We define SSO as a solution that allows users to log in using a single login page and afterwards to have access to multiple services [4, 18, 39] as shown in Figure 4.

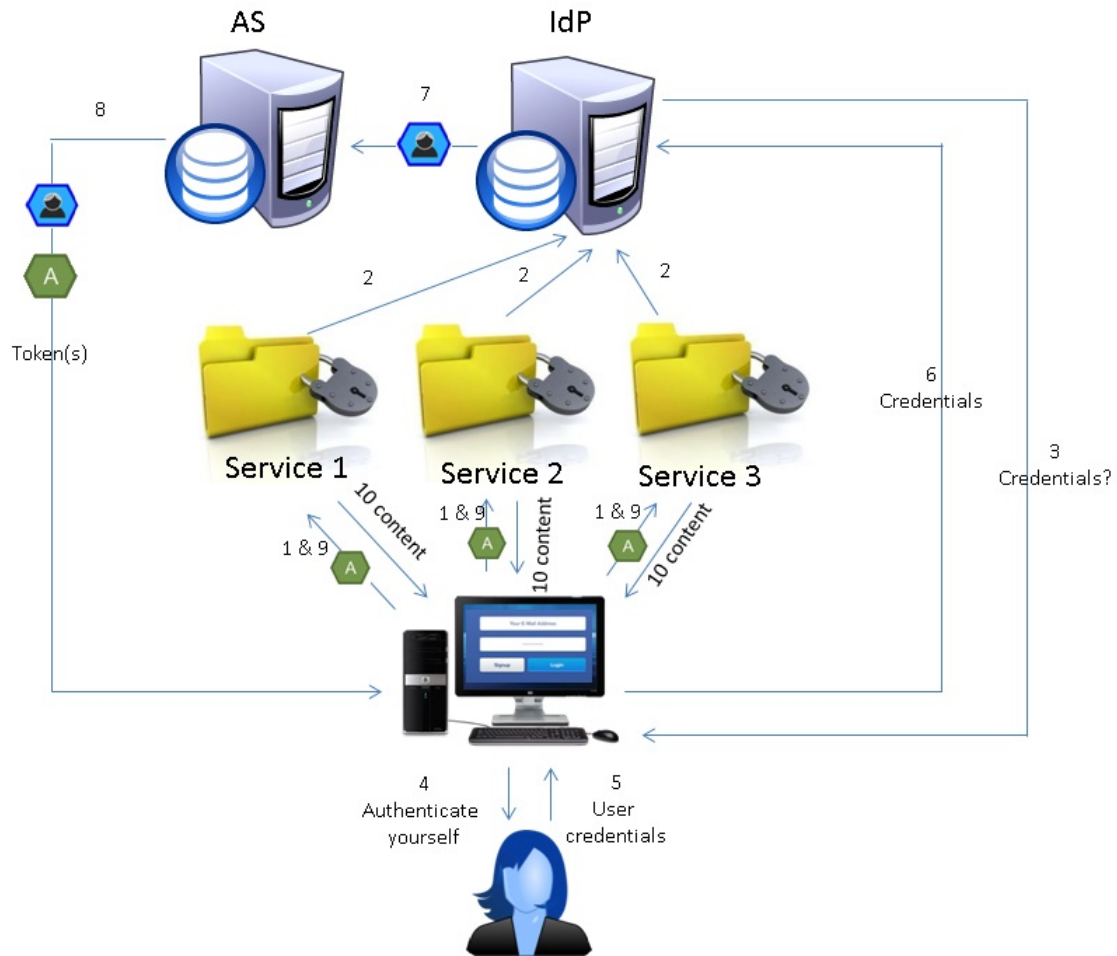


Figure 4: SSO authentication

Using this interpretation, SSO has the following steps:

- 1 The client tries to access a service. If the client already has an access token for this service, then the token is added to the request. Afterwards, go to step 10.
- 2 The service calls the IdP to handle the authentication.
- 3 The IdP asks the client for login credentials.
- 4 The client asks the user to give login credentials.
- 5 The user hands over the login credentials.
- 6 The client sends these credentials to the IdP that validates the credentials.

- 7 If the credentials are correct an ID token is send to the AS; otherwise it returns to step 3.
- 8 The AS collects the rights that are assigned to the user and creates an access token. The access token and ID token are sent to the client.
- 9 The client tries to access a service using the access token.
- 10 The service grants access to the service.

This process can be seen as getting a key from the manager to get access to all the rooms in the building. The first time you want to enter a room you go to the manager to ask for a key. Once you have the key you can enter each room in the building using this single key.

Some say that SSO is a technique that allows users to log in to a website using an external IdP [16]. So you need one single login account to sign on to multiple websites as shown in Figure 5. This is what we call Federated Identity Management.

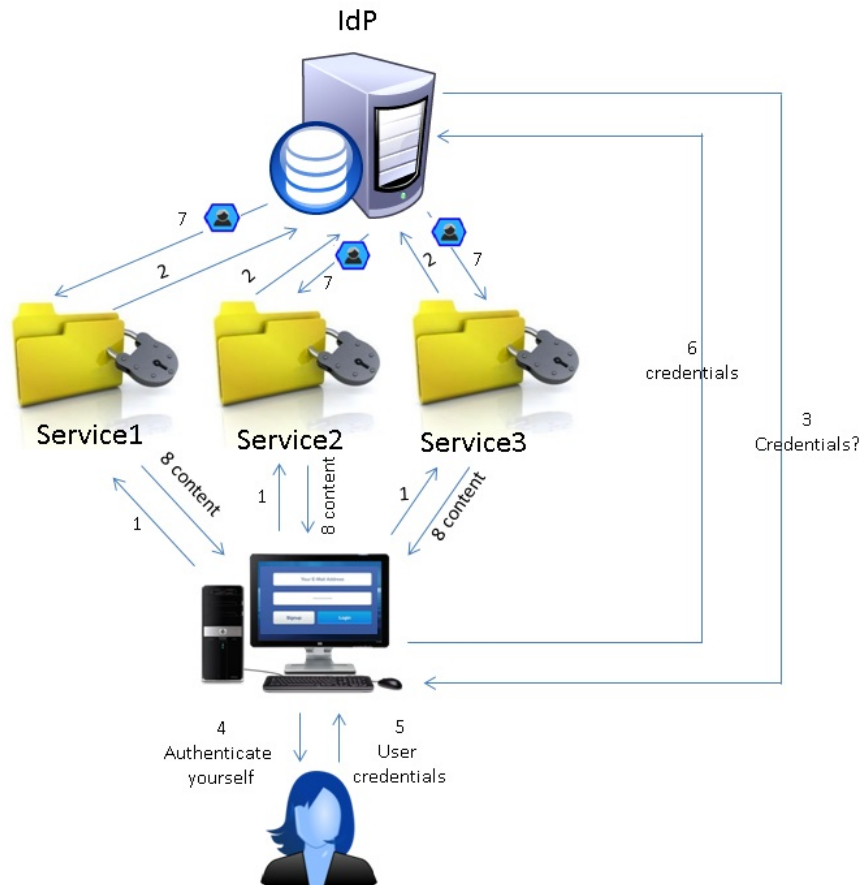


Figure 5: Federated Identity Management

The steps for Federated Identity Management are almost the same as for SSO, it does not however handle authorization (optional in SSO) and re-authentication for each of the services is required. When the user wants to access a second service then the authentication process starts all over again. There is no step from 1 to 8 because there is no token that can be shared between the services. For additional information on Federated Identity management, see Appendix A.

Federated Identity Management can be thought of as a user getting a key from a manager to gain access to a specific room in a building. For every room the user wants to enter he has to visit the manager to ask for the key. The difference with "basic" non-federated identity management is this: in this solution, there is only one person to speak instead of many key holders all of whom having one key for one specific door.

3.3 Why use SSO?

To show the motivation behind SSO, consider the following example taken from Hursti [39]. Consider a company that has several departments. Over the years, the company has purchased several kinds of computing systems, some of which are legacy mainframes and some of which are state-of-the-art workstations. Each system has services that need to be made available in a secure manner to a group of employees. To achieve this, systems which provide authentication and authorization must be in place. Typically, users have an account for each service. To get access to a service, a users logs in by entering their username and password. With a heterogeneous system all users need to have passwords for each service and log-in separately. This is not an ideal situation. The practice of handling authentication and authorization is called "identity management".

Having to enter several passwords to get into all the systems every time you start working is annoying and decreases productivity. Users have to remember many passwords and, even worse, they have to change them frequently. Because users have many passwords they tend to forget them after they changed them in something completely different. This results in passwords being written down or modified only slightly (e.g. update from pwd1 to pwd2). Because of the many login procedures that have to be performed each time after logging out, the users will avoid this issue by not logging out. This makes the systems vulnerable to internal attacks.

Also it takes a long time to create an account and password, every time a new employee enters the company. Even worse is the situation in which someone leaves a company. Nobody knows how many systems the employee had access to, so the system administrator has to go through all systems and delete all user accounts related to the employee who is leaving.

The solution to this problem is Single Sign-On. With Single Sign-On the user has only to log in once and the electronic identity (token) is shared with all computing resources. The rights associated with the users and resources are administered in a central manner, so there is no need for IT managers to worry about several databases when the privileges of a user account have to be set or updated.

3.4 Enterprise SSO vs Web SSO

When diving deeper into SSO you will find that two types of SSO exist: enterprise SSO and web SSO. This study aims at Web SSO solutions because most services that need to be authenticated using SSO are web services. Nevertheless, it is good to know that these two variants exist and what their differences are..

Enterprise SSO is designed to provide Single Sign-On to almost all the applications a user needs, including, Windows executables, Java applications, terminal-emulator applications and in some cases web applications. Typically, enterprise SSO is implemented as a desktop client that manages user credentials on behalf of a single user [22]. It captures the user-ID

and password for the application when the user logs in. The next time the application is launched, the solution will detect this and automatically enter the credentials on the user's behalf [32]. It can also be programmed to handle password changes for example, password expiration. For Enterprise SSO, an executable is installed on the user's desktop and profiles are created to recognize the login/password change screens so that the agent can respond to them. An example of such an SSO solution is a password manager that automatically logs a user in when a certain website is visited. Since no changes have to be made to the applications, setting up this kind of SSO is in most cases a relatively easy way to provide SSO [32].

Web SSO focuses on web-based applications [22, 32]. An authorization server is used to determine who can have access to which service. An IdP will authenticate the user and checks the access rules in the AS to see if the request should be passed on to the web server. This provides a more integrated approach, as access control is added in addition to SSO. This SSO type is useful when many web applications are involved. In Web SSO, the user has only one password to authenticate. Once the user has a session with the Web SSO product, they can access multiple sites without having to authenticate again [32].

3.5 Permission to share information

Some of the solutions discussed in this thesis require the user to give permission to share certain information of the user with the service. For example, service S wants to know the email address of the user. S then contacts the identity provider for example, Facebook and asks for the data of the user. Before providing this information, the identity provider asks the user to give permission to share his email address.

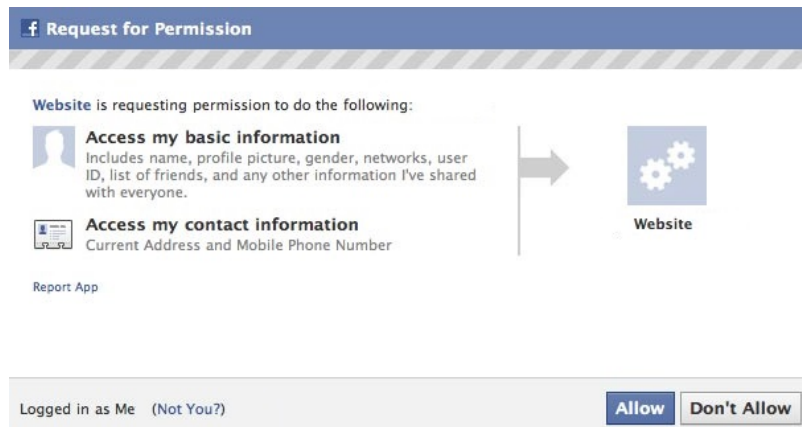


Figure 6: Permission to share

Figure 6 gives an example of a "permission to share information" screen. When the user accepts this request, then Facebook shares his information with the website, which in turn logs in the user based on the information retrieved from Facebook.

4 Background on the protocols

We start this chapter by providing information on tokens and cookies. In 3.1.5 we already gave a short introduction about tokens and cookies and defined the differences between them. Section 4.1 focusses on the pros and cons of both types. This section is included, since these techniques are used by the protocols to keep users authenticated and authorized.

The other sections provide information about various SSO protocols; they contain descriptions of how the protocols work and what the properties of each protocol are. A simple version of SSO can be achieved using cookies. The downside of a cookie-based approach is that they work only for services in the same domain.

In this chapter we look at the following protocols:

1. OAuth2 (Section 4.2)
2. OpenID Connect (Section 4.3)
3. SAML (Section 4.4)
4. LDAP (Section 4.5)
5. CAS (Section 4.6)
6. CoSign (Section 4.7)
7. OZ (Section 4.8)

We do not look at OpenID (which is different from OpenID Connect), since OpenID Connect is the successor of OpenID and because OpenID also requires re-authentication for every service, which makes it Federated Identity Management instead of SSO.

According to Wikipedia ² OAuth2 and OpenID Connect are also no SSO solutions. Wikipedia states that SSO is typically accomplished using LDAP. We do not agree with this claim. Based on our definition of SSO (Section 3.2) OpenID Connect is an SSO solution, since one-time authentication is required to get access to multiple services. We may even say that OAuth2 is an SSO solution because authorization implies authentication and because the access token can be used to grant permissions to several services.

This chapter discusses three kinds of solutions:

1. solutions offering authorization,
2. solutions offering authentication and,
3. solutions offering authentication and authorization.

To prove authentication and authorization, most protocols use tokens. In general, two types of tokens exist: XML and JSON.

4.1 Cookies and Tokens

In this section we will explain the difference between cookies and tokens and why one should be used over the other. For the tokens we will also explain the difference between XML tokens, as used by SAML, and JSON tokens, as used by OAuth2 and OpenID Connect.

²http://en.wikipedia.org/wiki/Single_sign-on version: 24 march 2015

4.1.1 Cookies

A basic authentication technique for a login page is a page containing a login form. After submitting the login form, data is sent to a server that validates the credentials and returns a (session) cookie. The idea behind cookies is that they help to maintain continuity and states on the web. They consist of strings that encode relevant information about a certain user. Cookies are stored on the computer of the user using the web browser. When the user visits the same website again, the browser sends the cookie to the web server which uses this cookie to identify the user. The purpose of a cookie is to provide information for the subsequent server-browser communications. Without cookies, the web page has to ask the user for the same information over and over again. For instance, a merchant website could use a cookie that contains the user's name and credit card number, and this information can be used to make future payment transactions easier [62].

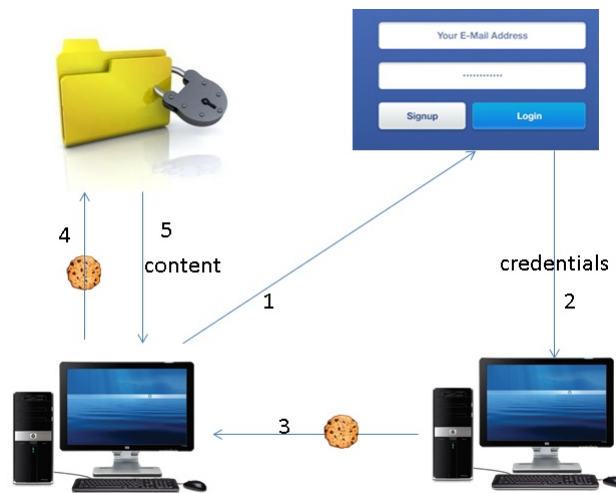


Figure 7: cookie authentication

A typical login process is described in Figure 7. The first step consists of a client logging in to a web page using the login-credentials of the user. If the credentials are correct, the system continues to step two, which returns a cookie back to the system. The client then sends this cookie to the service. Based on this cookie, it decides whether the user is granted access to the data. In some cases, the service contacts the authentication server to verify whether the cookie is still valid.

The problem with cookies is that they can only be used for a specific domain. For this reason, they only allow single domain Single Sign-On [52]. This study seeks a solution that works on multiple domains, so a single-cookie solution does not meet the requirements of this research.

A token-based approach (e.g., OAuth2, OpenID Connect, SAML, etc.) has multi-domain support. Tokens can be stored on local storage which can be accessed by all domains. Alternatively, multiple cookies can be used, one for each domain.

4.1.2 JSON Web Tokens

Just as there are many different SSO solutions, there are also different types of access tokens. They differ in the way they can/should be secured and in the notation of the attributes (JSON/XML/String). Some SSO solutions make it possible to use various types of tokens. In the beginning, many saw this as a negative thing. In practice, it turned out to be a good feature, since it gives immense flexibility [82]. Various types of tokens can now be used, and the freedom to switch them around enables interoperability. The JSON web token (JWT) is increasingly used nowadays.

JWTs are very flexible security tokens that make it possible to add custom claims (i.e., name/value pairs). A JWT consists of three different parts (header, payload and signature), and is base-64 encoded [64, 88, 51].

JWTs provide a way to represent claims that have to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed with a JSON Web Signature (JWS).

JWTs are part of the JSON Identity Suite, which also offers JWE for encryption and JWS for signatures. JSON Identity Suite is used by both OAuth2 and OpenID Connect [51].

These JWTs can be stored in cookies or in local storage. The advantage of storing tokens in local storage instead of cookies is that storing locally makes it possible to use the data on multiple domains [88].

4.1.2.1 Format

JSON Web Tokens are a string of "url safe" characters that encode information. Tokens have three components separated by periods, as shown below [56, 88]:

```
{"typ":"JWT","alg":"HS256"} // header
.{ "key":"val","iat":123456789} // payload
.eUiabuiKv-8PYk2AkGY4Fb5KMZeorYBLw261JPQD5lM // signature
```

Header

The first part of a JWT is an encoded string that represents a simple JavaScript object which describes the token and the hashing algorithm used. The member names within the JWT header are referred to as "header parameter names". It is important that these names are unique. The values corresponding to these names are referred to as header [56, 88].

Services that receive these tokens must understand all content in the header, otherwise they must reject the JWT.

Payload

The payload is the body of the JWT and is the core of the token. Its function is to store the information. This information is stored in so-called claims.

The length of the payload depends on the amount of data that is stored in the token.

Signature

The third part of the JWT is a signature which is generated based on the header, the payload and private key. The signature is used by the services to verify that the JWT is valid.

4.1.2.2 JSON vs XML

For a long time, XML was the only choice for open data communication [89]. In the last couple of years, however, there has been a shift in open-data sharing. The lightweight JSON is nowadays a popular alternative to XML [64, 89]. Reasons for the shift to JSON include the following:

- JSON-encoded data is more compact. XML uses more words than necessary (e.g., open and close tags) [89].
- Parsing XML software is slow and cumbersome. It is faster to read JSON data, especially when working with JavaScript (e.g., AngularJS, which is used for MyCCV). Parsing large XML files costs large amounts of memory due to its verbosity [89].

Figure 8 gives an example of a XML presentation. This XML structure is not intuitive, which makes it hard to represent in code. The same attributes can also be represented in JSON, as shown in Figure 9:

```
<?xml version="1.0" encoding="UTF-8"?>
<response uri="http://fake-url.com">
  <result>
    <Accounts>
      <row no="1">
        <FL val="id">124216000000072038 </FL>
        <FL val="phone">3 </FL>
        <FL val="website">3 </FL>
        <FL val="employees">3 </FL>
        <FL val="billingStreet">3 </FL>
        <FL val="shippingStreet">3 </FL>
        <FL val="billingCity">3 </FL>
        <FL val="shippingCity">3 </FL>
        <FL val="billingState">3 </FL>
        <FL val="shippingState">3 </FL>
        <FL val="billingCode">3 </FL>
        <FL val="shippingCode">3 </FL>
        <FL val="billingCountry">3 </FL>
        <FL val="shippingCountry">3 </FL>
        <FL val="description">3 </FL>
      </row>
    </Accounts>
  </result>
</response>
```

Figure 8: XML data example [89]

```
{
  "id": "124216000000072038",
  "description": "3",
  "website": "3",
  "numberOfEmployees": "3",
  "phone": "3",
  "name": "account3",
  "shippingAddress": {
    "country": "3",
    "stateOrProvince": "3",
    "city": "3",
    "postalCode": "3",
    "street1": "3"
  },
  "billingAddress": {
    "country": "3",
    "stateOrProvince": "3",
    "city": "3",
    "postalCode": "3",
    "street1": "3"
  }
}
```

Figure 9: JSON data example [89]

This JSON is intuitive, which makes it easy to read (which can be useful for testing) and to map directly to domain objects in whatever programming language is being used.

4.2 OAuth2

Some sources[18, 27] say that OAuth2 is not an SSO protocol, as it does not provide authentication. One can disagree with this statement, however, as OAuth2 handles authorization [10, 51] and authentication is required for authorization. Since the access token can be used by all services, authentication to these services can be automated because an access token implies that the user is authenticated. The main difference between an ID token and an access token is that an access token should only contain access information and does not contain any identifying information (see Section 3.1.5).

In OAuth2, authorization is delegated over several entities. A client asks for the AS a message, saying, in effect, "this client has access to this source". The client gives this message to server S, which trusts the AS and will grant access. So the AS can deliver temporary access tokens to the client without giving the client too much power.

This system can be thought of as a hotel. The Hotel has a key master (OAuth2 server) named AS. He gives all cleaners (clients) keys that open the door of the rooms they have to clean. They can only open the doors assigned to them and no others. All of these keys self-destruct after a few hours, after which time the cleaners cannot open any door.

OAuth2 has several authorisation flows. These all obtain access tokens. All flows and steps are specified in [25] and match with the flows described at the official OAuth2 specification [30]. Each flow describes a possible OAuth2 solution.

4.2.1 The protocol

In this section, we describe only the Implicit Flow and the Code Authorization Flow. The Resource Owner Flow and Client Credential Flow are of less interest to this research because they aim at solutions that do not involve a user and because we are looking for a solution that automates the login process for CCV customer services. The Resource Owner Flow and Client Credential Flow can be found in Appendix B.

Implicit Flow

The Implicit Flow (or grant as they call it in the OAuth2 specification) is used to obtain access tokens and is optimized for clients that are known to operate on a particular redirection URI. The Implicit Flow does not support the distribution of refresh tokens.

The Implicit Flow should be used when clients are not able to keep their credentials secret. These applications cannot keep the confidentiality of user credentials because user credentials are distributed with a copy of the application, which makes them vulnerable for theft [10, 51].

Figure 10 shows how the Implicit Flow authorizes a client [30].

- 1 The client initiates the flow by directing the user agent to the authorization server. The client includes its client identifier and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).
- 2 The authorization server authenticates the user with the user-agent and determines whether the user has given permission to share personal information (as explained in Section 3.5).
- 3 Assuming that the user grants access, the authorization server redirects the user agent to the service provider with the redirection URI provided earlier. The returned data contains the redirection URI and the access token.
- 4 The user agent follows the redirection instructions by making a request to the service provider. The user agent stores the information locally.
- 5 The service provider returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the information stored by the user agent and to extract the access token (and other parameters).
- 6 The user agent executes the script provided by the service provider locally, which extracts the access token and passes the access token to the client.

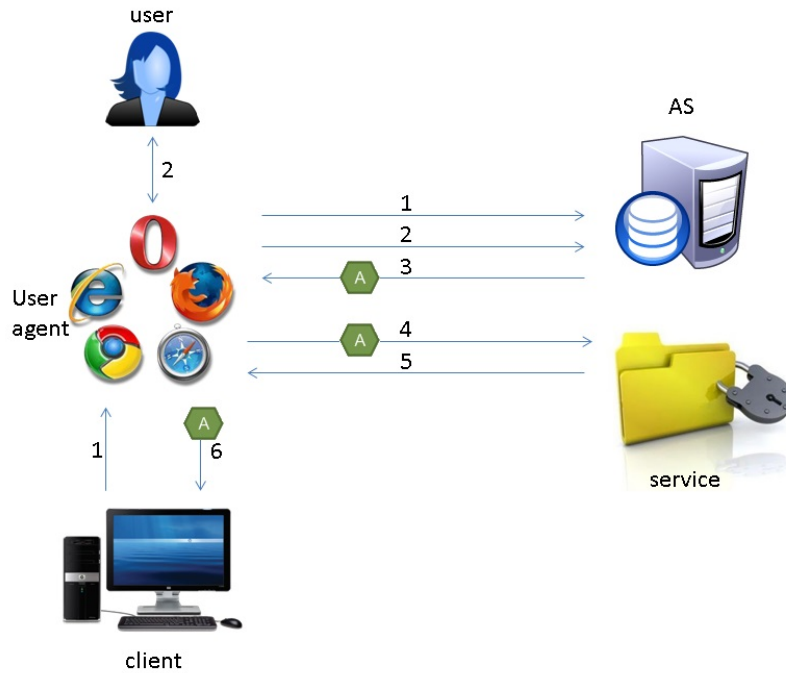


Figure 10: OAuth2 Implicit Flow

Code Authorization Flow

The Code Authorization Flow is slightly different from the Implicit Flow. In step 3, the Code Authorization Flow returns authorization codes instead of access tokens. These authorization codes are sent to the client. The client will use them to get an access token and optionally a refresh token from the AS. In this process, the user agent never sees the access token; the user agent sees only an authorization code. The access token is kept private from the user.

This flow should be used when a user is using a client that interacts with a website. Examples are a client being a web-application and a desktop applications like Spotify that uses an embedded browser [10].

The first two steps of the Code Authorization Flow are equal to the steps of the Implicit Flow; afterwards, the process changes. Whereas the Implicit Flow returns an access token, the Code Authorization Flow returns an authorization code which the user agent returns to the client. The client sends the authentication code to the authorization server and gets an access token and perhaps a refresh token back. This refresh token makes it possible to use short-lived access tokens that will be refreshed after the access token is expired. The authorization server should check to see if the refresh token belongs to the client that is requesting a new access token. For the refresh process, authentication is required. If authentication of the client is not possible, then the authorization server should use another solution to detect refresh-token abuse. A solution could be to use a refresh-token rotation in which a new refresh token is issued to the client after every access token refresh response. The authentication server stores the previous refresh tokens and marks them as invalid. When an invalid refresh token is received, it is likely that someone or something with malicious intentions is trying to get access to the account. The server will detect this and take an appropriate action such as logging the user out and requiring re-authentication.

Refresh tokens may only be communicated using protected connections. When a refresh token gets compromised by an attacker and both the attacker and the legitimate client try to get a new token, then one of them will send an invalidated refresh token. The authorization server must generate secure refresh tokens that cannot be generated, modified, or guessed in order to prevent the production of valid refresh tokens by unauthorized parties [30].

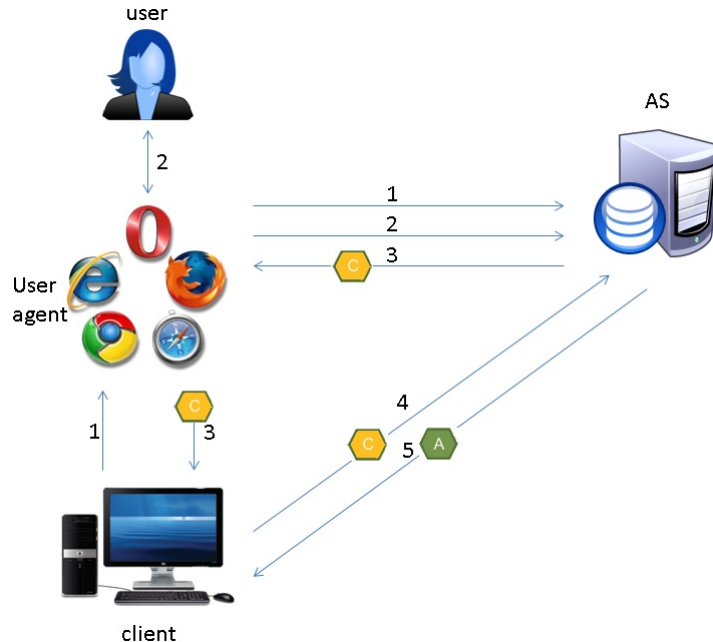


Figure 11: OAuth2 Code Authorization Flow

4.2.2 Evaluation of OAuth2

Eran Hammer who worked on the OAuth2 framework decided to resign from his role as lead author and editor at the final steps of the OAuth2 project [27]. In Eran's opinion OAuth 2.0 became a bad protocol, so bad that he no longer wanted to be associated with it any more. He called the project "his biggest professional disappointment of his career and the road to Hell" [27]. This remarkable event will be the guide for this section.

In Eran's opinion, too many compromises were made in the OAuth2 project which resulted in a specification that fails to deliver its two main goals: security and interoperability. One of the compromises they had to make was to rename OAuth2 from a protocol to a framework. Compared with OAuth 1.0, the 2.0 specification is more complex, less interoperable, less useful, less complete, and most importantly, less secure [27].

Other sources disagree with Eran and say that OAuth2 is not that bad. GLUU [74] for example, expects that OAuth2 is getting used more often and in a couple of years will be the most-used SSO standard. Bray [12] says that the protocol is not perfect, but big companies like Google and Facebook are integrating this protocol. This suggest that the protocol is not as bad as Eran describes, since, presumably, many well-trained security experts have looked into it. Homakov [37] agrees that there are some issues with OAuth2 but argues that they are fixable. He says that we must not keep on complaining and must rather work on a solution.

According to Eran, the core of the problem is the unbridgeable conflict between the web and the enterprise worlds: "The OAuth working group at the IETF started with strong web presence. But as the work dragged on past its first year, those web folks left along with every member of the original 1.0 community. The group that was left was largely all enterprise, and me" [27]. In an early meeting, the web folks wanted a flow optimized for in-browser clients whereas the enterprise folks wanted a flow using SAML claims.

Another issue mentioned by Eran and Manso[51] is the lack of standardisation. The specification of OAuth2 includes more than 50 instances of the word should, which in most cases should be changed into must [27, 51].

Eran concludes by saying that he hopes that someone will "take OAuth2 and produce a 10 page profile that is useful for the vast majority of web providers, independent of the enterprise".

In our opinion, Eran's accusation is exaggerated. Because of the lack of the ability to sign the tokens Eran says that these so called "bearer tokens" are insecure [28]. These bearer tokens are tokens that are not signed and encrypted and rely fully on the security of TLS. We do not totally agree with this, since the attributes in the tokens can still be encrypted and/or signed. Also JSON webtokens can be used instead of the bearer tokens discussed in Eran's paper [28]. The benefit of these tokens is that they can be signed.

Eran argues that the core problem of OAuth2 is that it focusses too much on enterprise [27]. This is a personal opinion that we do not agree with. Big web companies like Google and Facebook have implemented OAuth2, and other web-oriented companies will follow. There are nice schemes with clear descriptions available of the OAuth2 flows, and these simplify the implementation of a flow. Implementing the services defined in the protocol should be feasible for most programmers, especially since examples are available.

Nowadays, solutions are offered for the potential security issues present in the OAuth2 framework. Many of these can be found in Chapter 6 and on websites like that of Homakov [37]. So to conclude, if OAuth2 fulfils your needs, use it but be aware of the security issues.

4.3 OpenId Connect

OpenID Connect (OIDC) can be described as an identity layer on top of OAuth2 that offers various extensions, of which the most import is authentication. So this protocol offers authorization as well as authentication. The flows in OpenID Connect are very similar to those of OAuth2; the main difference is that IdP and ID tokens are added.

4.3.1 The protocol

For OpenID Connect, three flows are available: the Implicit Flow, the Code Authorization Flow, and the Hybrid Flow. The Hybrid Flow is very similar to the Code Authorization Flow. The Hybrid Flow makes it possible to send the tokens directly to the client without the interference of an authorization code. The Hybrid Flow has the possibility to send one or more additional parameters in step 9 [75].

1. Client prepares an authentication request containing the desired request parameters and sends the request to the IdP.
2. Then IdP asks for user credentials, which the user provides.
3. If the user credentials are correct, the IdP returns an ID token.

4. To retrieve access rights, the User agent contacts the AS sending his by reference ID token.
5. The AS returns an access token based on the identity provided by the ID token.
6. The User agents sends a request having an access token to the service to get access.
7. Based on the access token, the service returns a set of data which is displayed on the user agent.

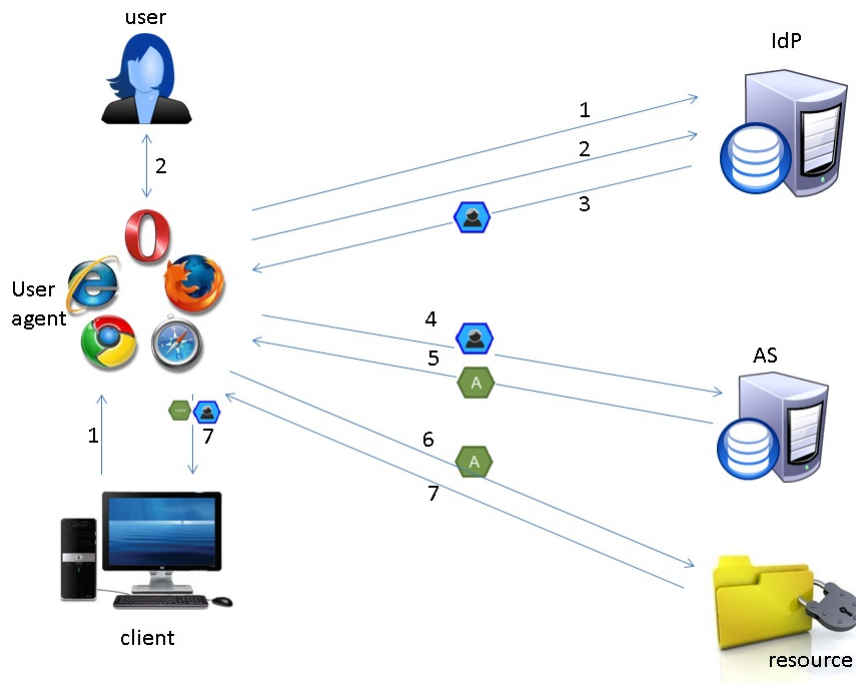


Figure 12: OpenID Connect hybrid flow

4.3.2 Evaluation of OpenID Connect

Identity management experts predict that OpenID Connect will become the leading SSO solution [74, 75]. This seems to be a reasonable conclusion, judging from the following indicators [74]:

- Supported by large consumer IdPs: Google, Microsoft, Yahoo and Facebook,
- Consolidation of several protocol communities such as OpenID, OAuth2 and a subset of the SAML community,
- Move in consumer market to JSON/REST Authentication API's,
- Explosion of mobile applications requiring better authentication API's for non-web interactions,
- Expanded role of a "client" acting as an agent of the person to access Web APIs,
- New standards that are building on OpenID Connect authentication, such as UMA and the new OpenID Connect Native SSO working group.

Since OpenID Connect is based on the OAuth2 protocol, many of the things said about OAuth2 also apply to OpenID Connect. Eran [27] also mentions OpenID Connect in his statement about "OAuth 2.0 and the road to Hell". He said that OpenID Connect, like OAuth2, was a simple proposal turned into almost a dozen complex specifications.

Spencer [81] and GLUU[73, 74] disagree and say that it has become easier to implement OpenID than its previous versions. GLUU even expects that it is going to replace SAML in the future, which is the current leading standard in SSO [48, 74]. Like OAuth2, OpenID Connect is used by many big companies like Facebook and Google. This suggests that trained security experts have looked into the protocol.

We found only one negative review, which was more a complaint about SSO in general. Nicholas Piasecki [65] complains that the identity providers are a single point of failure. When Google is not available, for example, then users cannot login to stackoverflow.com with their Google login account. It is possible to create a second "backup" SSO account at another provider, but this is contrary to the idea of SSO having a central point of authority to handle online identity. On the other hand, how often are these services unavailable? According to a study done by the International Working Group on Cloud Computing Resiliency, most services are unavailable less than 0.4% of the time (which is a total of 35 hours spread over a year) [24]. In the case of CCV, they are the IdP themselves, so the services are managed by the same party that offers the IdP.

KuppingerCole, a company that focusses on information security and identity and access management, recognized OpenID Connect as the best new standard in the category of information security. They said that it provides all of the consumerization of SAML but is making this much simpler [44].

Because of its acceptance by leading web organizations and our own positive findings concerning this protocol, we would recommend OpenID Connect as an authentication and authorization solution.

4.4 SAML

Security Assertion Markup Language (SAML) is a protocol that offers authentication and authorization. Using SAML, a service can contact an IdP to authenticate users who want to access secure content. Version 1.1 of the SAML specification provided support for web SSO only[70]. Since then, SAML has evolved into a more extensive framework [70]. In this section we focus on SAML version 2.0.

SAML 2.0 is the current leading standard for inter-domain authentication [48, 74]. It is supported by off-the-shelf software, and major service providers such as Google, Salesforce, WorkDay, Box, Amazon, and many others [74]. SAML is a standard for extensive business-to-business, government and educational networks around the globe.

SAML supports bindings which are the means by which the Identity Provider redirects the user back to the Service Provider. There are two relevant types of bindings: the HTTP Redirect and the HTTP POST binding. The HTTP Redirect will use, as the name already suggest, a HTTP Redirect to send the user back to the Service Provider. The HTTP Redirect binding is mainly useful for short SAML messages. Longer messages (e.g. messages that contain signed SAML claims) should be transmitted using a HTTP POST Binding [11, 70].

The recommended way to use an HTTP POST is somewhat unusual. For example you have to decide whether the user has to click an extra button to submit a form or you utilize JavaScript to automate the submitting of the form. You may ask: why do we need a form that needs to be submitted? The main reason for this is that SAML is getting outdated. In 2005, when SAML 2.0 was released, it was necessary to use this solution to perform a HTTP POST to send the SAML token back to the Service Provider [11, 74].

This workaround for sending a HTTP POST request is however not possible for a native application like a mobile app. Since a native application only has access to the URL to launch the application, the app does not have access to the HTTP POST body. This means that we cannot read the SAML token [18]. Tokens in SAML are in XML format, unlike the JSON web tokens supported by OAuth2 and OpenID Connect.

4.4.1 The protocol

Unlike OpenID Connect and OAuth2, SAML has only one flow which is displayed in Figure 13.

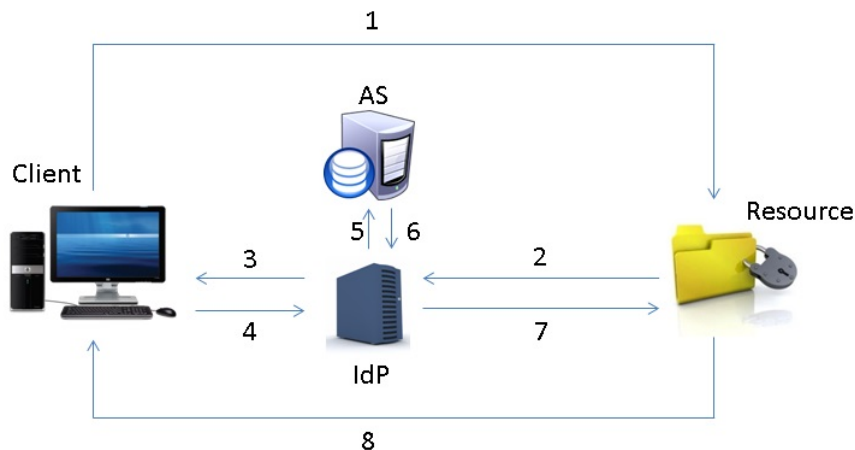


Figure 13: SAML Protocol [18]

The protocol starts with a client contacting the service (1). To access the service, the client sends a message to the IdP and AS to ask permission to access the application (2). The IdP/AS server asks the client to provide login credentials (3). The client returns the credentials to the IdP/AS server which in turn validates them (4). If the credentials are valid, the IdP contacts the AS and asks for authorization data (5). The AS returns the access token to the IdP (6), which in turn informs the service that the user is authenticated and sends an access token (7). The service logs the client in and grants access to the requested resource (8).

4.4.2 Evaluation of SAML

GLUU[74] predicts that SAML will be critical for domains for some years to come. In the next fifteen years or so, organizations will more and more look into OAuth2 implementations, and will de-commission SAML relationships.

GLUU [74] expects a takeover of the leading position currently occupied by SAML. OAuth2 and OpenId Connect will take over in a couple of years. This claim is supported by other sources [8, 44] who say that SAML is outdated. It is also supported by a study done by Bjorne [11] in 2014. The latest version of SAML (V2.0) was released in March,

2005 [54]. Baier [8] Baier [8] refers to SAML as the Windows XP of identity: "no funding, no innovation. People still use it", but according to Baier it has no future.

Kearns [44] refers to SAML as being "the king of the first decade of the 21st century, a king that now is dead". Although we are halfway the second decade, SAML still is the most used SSO solution and it will probably be so for some time to come.

We are not sure if the growth of OAuth2 and OpenID Connect will be as rapid as GLUU assumes. Only time can tell. The fact that OAuth2 and OpenId Connect have been adopted by Google, Facebook, Yahoo and even Microsoft [74, 91]], however, is a good sign that these protocols will become popular in the near future.

In our opinion, SAML is a good solution that matches all functional requirements. As others have noted, however, it is a bit outdated. The fact that SAML is not compatible with JSON web tokens makes it less suitable for future (e.g., mobile [51]) purposes, as explained in Section 4.1.2.

4.5 LDAP

The Lightweight Directory Access Protocol (LDAP) is a protocol for accessing distributed directory services, which can be used for authentication and authorization [38, 90]. LDAP is often used in combination with an active directory.

Definition 16. A **directory** is a kind of specialized database. A simple example of a directory is a phone book [38].

LDAP is based on DAP which is a protocol defined by the X.500 computer network standards. Early adopters of DAP thought that it was fairly complex and not well suited for the desktop computers of that day. DAP is large, complex, and difficult to implement, and most implementations perform poorly [38]. Around 1990, two independent groups devised similar protocols that were simpler and easier for desktop computers to implement [38]. The two lightweight protocols for desktop computers were called: Directory Assistance Service (DAS) and Directory Interface to X.500 Implemented Efficiently (DIXIE).

After DIXIE and DAS showed that a lighter-weight access protocol could be produced for X.500, the members of the OSI-DS Working Group (of the IETF) decided to join forces and produce a new full-featured, lightweight directory access protocol for X.500 directories. This work led to the birth of LDAP in 1993.

If the vast majority of your actions are simple "selects", and if you want anyone to be able to perform such selects over the network, then use LDAP. This is the kind of performance at which LDAP excels.

4.5.1 The protocol

The LDAP protocol consist of LDAP clients and an LDAP server. The clients create an LDAP message that contains a certain request and sends it to the server. The server processes this request and sends the results back to the client as one or more LDAP messages. For example, when an LDAP client searches for a specific entry, then it sends an LDAP search request message to the server. Each message contains a unique message-ID generated by the client. The server receives a message from its directory and sends it to the client followed by a separate message that contains the result code. All of the communication between the server and the client is identified by the message-ID provided in the request of the client [38, 90].

If the server searches the directory and finds multiple entries, those entries are sent to the client in a series of LDAP messages, one for each entry, as shown in Figure 14. The search results are terminated with a result code, which contains the overall result of the search operation.

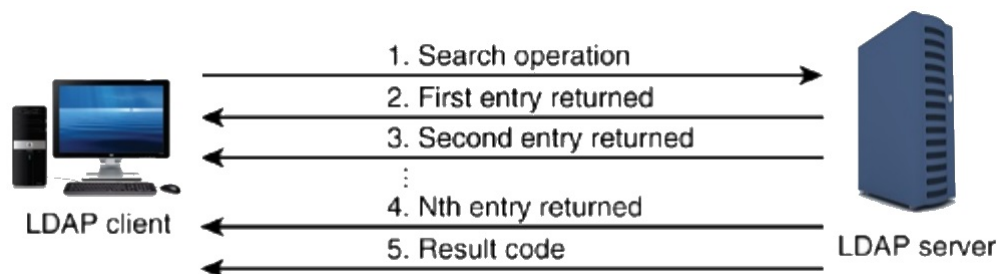


Figure 14: LDAP communication [38]

But LDAP can do more than just search for certain results. We can divide its functionality into three categories:

- **Interrogation** operations: search, compare.
These two operations allow you to ask questions to the directory server.
- **Update** operations: add, delete, modify, modify DN (rename).
These operations allow you to update information in the directory.
- **Authentication** and **control** operations: bind, unbind, abandon.
The bind operation allows a client to identify itself to the directory by providing identity and authentication credentials; the unbind operation allows the client to terminate a session; and the abandon operation allows a client to indicate that it is no longer interested in the results of an operation it had previously submitted.

For distributed authentication, two scenarios in LDAP can be applied:

- Server 1 might tell Server 2 who the authenticated user is, and Server 2 might simply choose to believe Server 1. This approach requires that Server 2 trusts Server 1 to verify authentication credentials correctly.

OR

- Server 1 passes the users identity and authentication credentials to Server 2. Server 2 could then independently verify the credentials. This approach requires that Server 1 trusts Server 2 not to misuse the authentication credentials (for example, if the credentials consist of a plaintext password, then Server 2 must not reveal it to a third party).

The second scenario can also be executed by using certificates instead of login credentials. If the client has authenticated to Server1 using certificate-based client authentication, Server1 can pass along the digitally signed request and allow Server2 to verify the client's identity directly, since the identity of the client is proven by the digital signature.

Step one in Figure 15 is the same in both scenarios: a user logs in using his login credentials. The second step differs, this can be the message "User X has Logged in" or "User X has logged in and here are the credentials to prove this".

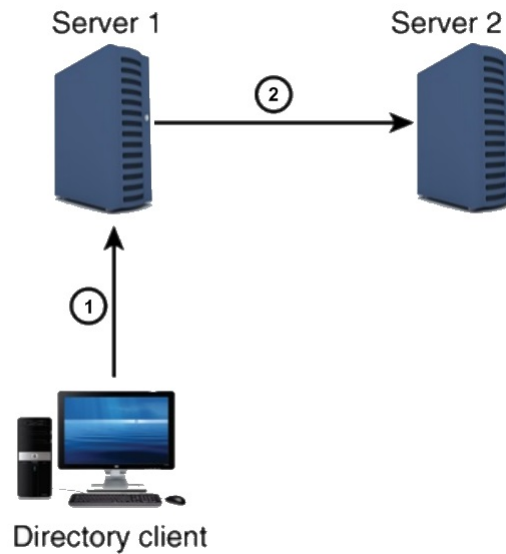


Figure 15: LDAP authentication

4.5.2 Evaluation of LDAP

LDAP is a broad solution that can be used for many purposes including authentication and authorization. LDAP is a good example of an Enterprise SSO solution. It is robust and supports many features many more than we need for this study. Because of this, the protocol is harder to understand than SAML or OpenID Connect. LDAP is a suitable solution if you are looking for an SSO solution that can work with directories or when an enterprise solution needs to be implemented. If this solution is chosen, then we recommend reading a manual that provides many details on LDAP and gives tips on how to implement the protocol properly: namely, *Understanding and Deploying LDAP Directory Services?* [38].

4.6 CAS

Central Authentication Server (CAS) 1.0 was developed by Yale University as an easy-to-use Single Sign-On solution for the web. It consisted of servlets and web pages [6, 40]. In December of 2004, CAS became a Jasig (now Apereo) project. Together with Yale and Rutgers, Jasig produced the open-source CAS 3.0. The goal was to make CAS a flexible and extendible protocol able to meet the varying requirements of other institutions.

CAS is an authentication SSO solution, as its name, "Central Authentication Server", already suggests. CAS is an open protocol that consists of a Java server component that communicates with clients written in:

- Java
- .Net
- PHP
- Perl
- Apache
- uPortal
- etc. [6, 40].

4.6.1 The protocol

The CAS protocol consist of two parts: the CAS server and the CAS Client.

CAS server

A CAS server is a single machine used for authentication. This server is the only system that knows the user's passwords and has a double role [6]:

- Authenticate users;
- Transmit and certify the identities of authenticated users to the CAS clients.

An SSO session starts when the server issues a granting ticket to the user upon a successful login. A service ticket is issued to the service after a request is received from the user. The issuing of service tickets is handled via browser redirects that use a ticket-granting ticket which is a kind of token. The client validates the service ticket at the CAS server via back-channel communication [40].

CAS clients

A CAS client is any service provider that is CAS-enabled and can communicate with the server. A CAS client is a software package that can be integrated with several software platforms and applications in order to communicate with the CAS server [6, 40].

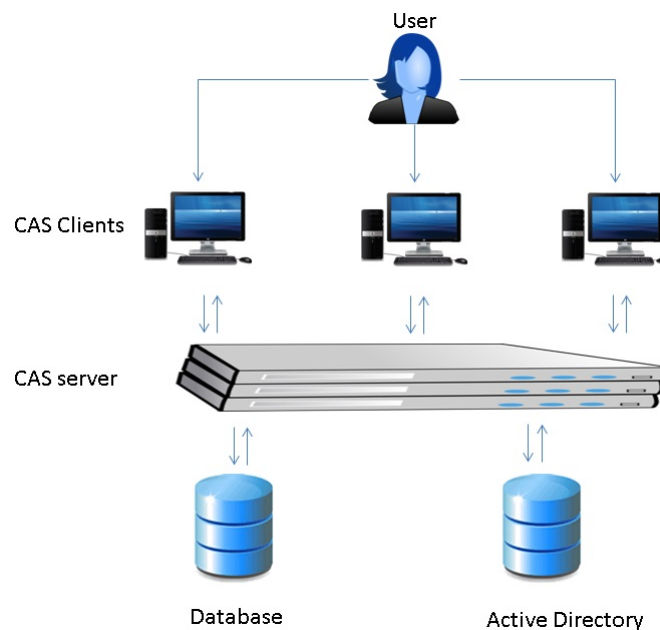


Figure 16: CAS protocol

4.6.2 Evaluation of CAS

Backed by LDAP, CAS is a good choice for organizations that are planning to centralize authentication. CAS also offers access control based on the network address of services. In this way, it limits the services that can use the centralized authentication service. For this reason, CAS is implemented by many higher education institutions [6, 40]. According to GLUU [74] CAS should no longer be implemented, as more functional authentication standards like SAML and OpenID Connect are available. This opinion is supported by leading companies like Facebook, Google and Yahoo, who do not implement CAS but choose SAML and OpenID Connect instead [6, 40, 74].

4.7 CoSign

Cosign is yet another authentication protocol. According to the official Cosign wiki [1], Cosign is an open-source project that was originally designed to provide the University of Michigan with a secure, Single Sign-On, web-authentication system [1, 58]. Cosign was revised in 2012 and updated to version 3.2.0 [1].

4.7.1 The protocol

The Cosign protocol depends on two primary components: the weblogin server (IdP) and the Cosign-protected web service(s) (service provider). The weblogin server consists of a so called Cosign CGI (Common Gateway Interface) component which logs users in and out, a Cosign daemon that manages information about the state of the session for each authenticated user, and a number of administrative scripts. Each Cosign service has a filter which communicates with the Cosign daemon to assure that the users who want to access a protected area are authenticated. Users must authenticate themselves to the CGI component on the weblogin server before they are permitted access to a protected service.

The protocol that the CGI component uses to communicate with the Cosign daemon is a straightforward text-based, newline-terminated protocol [57].

Daemons

Cosign utilizes two kinds of daemons: Cosignd and Monster. Cosignd handles the implementation of the server-side Cosign protocol and maintains the Cosign cookie database. If required, it also replicates the login, register, and logout commands of all other Cosignd daemons in the server pool.

Monster handles the deleting of old cookies (used for login or service) and can reproduce the logged in/out state and last update time for all of the login cookies in its database.

CGIs

There are also two CGIs defined in Cosign: the login and logout CGI. The "login" CGI handles the log in of users into the central Cosign server. It also handles registration for each service a user logs into.

The logout CGI is responsible for removing the user session from the central Cosign server. Once the log out is verified, the CGI disables access to all services by writing a "null" valued cookie for all services and setting their expiration date to a time in the past. Because the login state is centralized, the user is directly logged out of every application that he/she has visited. When the service uses data caching, the application at which the user was logged in may still report the user as logged in for the cache time (default is 60 seconds).

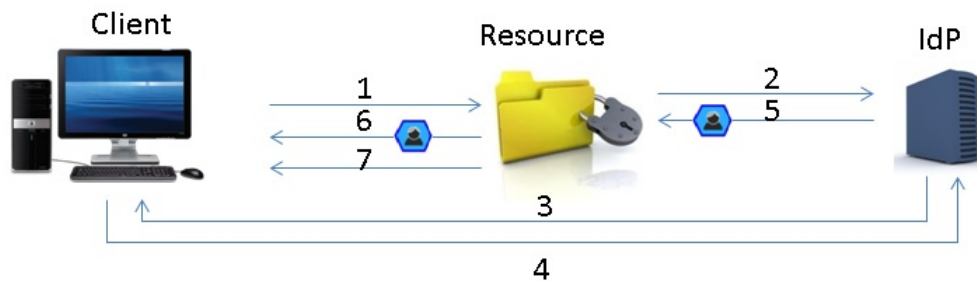


Figure 17: Cosign Protocol

Like CAS and SAML, Cosign has only one defined flow. It is depicted in 17. The flow starts when a client contacts a service with a request for access to the information (1). The service requests that the user logs in and asks the IdP to authenticate the user (2). The IdP asks the client to provide login credentials (3). The user fills in the credentials and the client returns these to the IdP (4). The IdP verifies the credentials and sends an ID token to the service (5). The service sends the token to the Client (6), and sends the resource the client requested (7).

4.7.2 Evaluation of Cosign

We discovered that it is hard to find useful information about Cosign. There are not many sources available, and those we found are based mainly on the official Cosign webpage. Some quick research in the usage of Cosign revealed that only nine questions about Cosign were posted on stackoverflow in the month January of 2015, which is few compared to those posted for OAuth2 (491), SAML (178) or OpenID Connect (49).

4.8 OZ

Eran Hammer, who worked on the OAuth2 framework and decided to resign, came up with an authentication protocol of his own named OZ. OZ has no wiki-like webpage, only a github page. On this page, OZ is described as, "a web authorization protocol based on industry best practices, putting mobile and native apps first. A redo of the original ideas behind OAuth, providing true interoperability and security" [29].

It is not described how OZ manages authorization and what makes it different from OAuth2. The github page shows no documentation on the OZ protocol, only a JavaScript implementation which is still under construction.

In addition, little information can be found about the OZ protocol. Most hits on Google mention only the existence of OZ and do not add any additional information. The system consists of five elements:

- a client
- a server
- tickets
- endpoints
- a scope object

Based on the code, we know that the server is used to authenticate the user. The endpoint is the server that verifies the credentials and issues a ticket. The server also reissues tickets when necessary. Data is sent back to the clients using a callback. As far as we can see, the scope object is not (yet) used. OAuth2 uses a scope object to let the user specify what type of access the clients need [30, 78]; We presume the scope for OZ will have a similar goal. Because of the lack of information, and because it is an incomplete implementation, we consider this solution as not yet ready for implementation and for this reason not a suitable solution for this research.

5 Functional comparison

This chapter compares the functionality of the SSO solutions considered here based on the information in Chapter 4. We start by comparing all solutions to OpenID Connect. We chose OpenID Connect because it is one of the protocols that offers authentication and authorization and because much information is available about the protocol.

Section 5.5 concludes our findings with a table that provides an overview of the functionalities offered by the SSO protocols.

5.1 OAuth2 vs OpenID Connect

Before explaining the differences between OpenID Connect and OAuth2, we start by explaining what OAuth2 and OpenID Connect have in common.

First, they both focus on security, identity, and authorisation. They are both open-web standards that use REST for communication. In both solutions, the user can actively manage the rights that are given to the service provider. Figure 18 shows an example of how Facebook has implemented rights management.

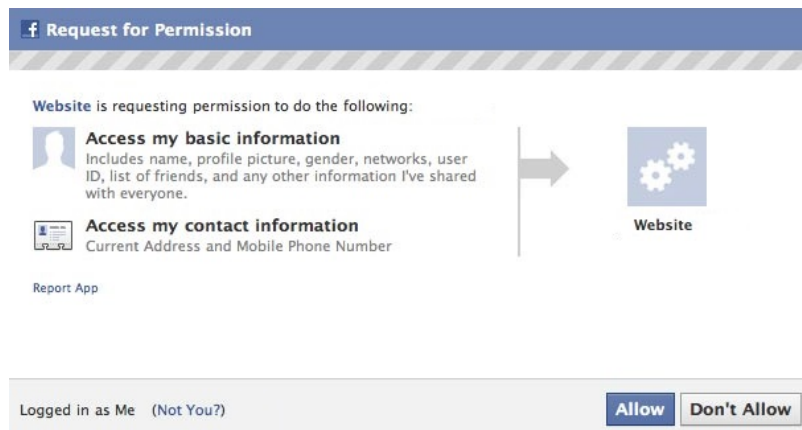


Figure 18: Authorization Facebook

The big difference between OAuth2 and OpenID Connect is that OAuth2 offers authorization only and OpenID Connect also offers authentication [51, 91]. For this reason, OpenID connect also has additional authentication tokens that are called "ID tokens".

However, OpenID Connect has no flow to share data between two applications. With OAuth2, any information a user holds on any website can be shared with another website. A user could, for example, copy his GMail address book into Facebook by allowing Facebook to read his GMail account. Without OAuth2, the only way to solve this is to give Facebook the GMail user name and password. This is clearly not a smart thing to do, and this is exactly what OAuth2 is set up to prevent.

On the other hand, OpenID Connect has an additional flow called Hybrid Flow that is another user-interactive flow. As in Implicit Flow, tokens are returned to the client, but there is no redirect URI. The client has to handle the communication with the service as in the Code Authorization Flow.

5.2 OpenID Connect vs SAML

Both OpenID Connect and SAML offer authentication and authorization [91], so both can be used for the same purposes with only slightly different technical means. But why pick one over the other?

OpenID Connect is a recently published protocol that uses new techniques, like JSON and REST (vs XML and SOAP for SAML). It was designed to support web apps, native apps and mobile applications, whereas SAML was designed only for web-based apps.

Because XML requires a heavy library to parse content, JSON and REST are easier to implement in the implementation languages commonly used on the web.

According to SURF [91], there is also a difference in the focus of the protocols: OpenID Connect is user-centric, whereas SAML 2.0 is organization-centric. User consent is an essential part of the OpenID Connect protocol, but it is also possible with SAML. Some analysts point out that the pre-defined set of user attributes offered by OpenID Connect is more geared toward consumer-to-web service provider scenarios than toward enterprise scenarios (where one would expect roles, entitlements, etc.) [91].

In SAML, security is implemented at the message level and uses XML-based technologies for signing and encrypting messages. Additionally, transport-level security can be implemented with TLS, which is a smart thing to do (defence in depth).

OpenID Connect has it the other way around: transport-level security is mandatory, and message-level security is optional [91]. The IdP will typically demand that clients implement additional message-level security in user-not-present situations, for instance (for browser-based clients the user is always present). Message-level security is made possible through encryption and by signing JWT objects.

While SAML uses a static trust-configuration setup through meta-data, OpenID Connect has some dynamic aspects. OpenID Connect allows so-called "Dynamic Client Registration", in which new clients are provided with access tokens and are redirected to a client-registration endpoint to be registered with an IdP and to receive further client credentials [91].

To conclude, both protocols solve the same problem, but in a different way [91]. Because OpenID uses modern techniques, we expect that the market share of OpenID Connect will grow and take over the leading position of SAML.

5.3 OpenID Connect vs LDAP

Both OpenID Connect and LDAP offer authentication and authorization; the main difference is that LDAP is an enterprise-oriented solution [38] whereas OpenID Connect is more web-oriented [75]. LDAP is used for sharing of information about users, systems, networks, and applications in the (internal) network [38]. With LDAP, user data is stored in a directory that contains user accounts. OpenID Connect does not talk directly to a directory or database, but requires a separate service that authenticates a user, which can be an LDAP implementation.

According to the literature [38, 90], it seems that LDAP is used mainly to search for specific data (like users) and that SSO is an additional feature.

OpenID uses the open standards REST and JSON, whereas LDAP uses XML. This is not unusual because LDAP was released in 1993 when Javascript (1995) [19] the basics of JSON and the REST protocol (2000) [20] did not even exist.

In short, LDAP may be used when an Enterprise SSO solution is required. If you are looking for a web SSO solution, use OpenID Connect. It is built with the web in mind, it is a smaller, and, in our opinion, it is easier to understand. OpenID Connect also uses the new standards in web programming and is therefore easier and faster to implement for web solutions.

5.4 OpenID Connect vs CAS vs Cosign

We combined three solutions in this section because Cosign and CAS are very similar when compared to OpenID Connect. Cosign with OpenID Connect have fewer features for authentication [74]]. They are also strictly limited to user authentication; they deal neither with authorizations nor with the propagation of user attributes [6].

Unlike CAS and Cosign, OpenID Connect is a new standard that is adopted by many leading companies. Because OpenID Connect is implemented by leading companies, it is more likely that well-trained security experts have looked into the protocol.

The main difference between CAS and Cosign is that more information can be found on CAS and that CAS is implemented more often. For that reason, we advise that CAS be used instead of Cosign.

CAS and Cosign should be used for SSO authentication of local users. They do not support federational authentication, which is possible with SAML and OpenID Connect [31].

5.5 Conclusion

Table 1 shows an overview of the functional differences between the protocols.

	Authenticate	Authorize	Token format	User consent	Token expiry
OAuth2 [30]	X*	V	XML or JSON	V	O
OpenID C [75]	V	V	JSON	V	V
SAML 2.0 [76]	V	V	XML	X	V [79]
LDAP [38]	V	V	?	X	X
CAS [38]	V	X	XML	X	?
CoSign 3 [63]	V	X	XML	X	?
OZ [29]	?	V	JSON	?	?

Table 1: functional comparison

X = Does not offer this property

V = Offers this property

O = Optional

? = We do not know the value for this property of this protocol.

* Authentication is not supposed to be handled by OAuth2, but many developers do use the protocol for this purpose on the bases that authorization implies authentication [13].

Based on this comparison, we know that OAuth2, CAS and CoSign do not meet the requirement of offering authentication and authorization. We do not know if OZ offers authentication; but because it is far from being a complete operational protocol, and because very little information is available about it, this solution should not yet be implemented. Thus, three potential solutions remain: OpenID Connect, SAML 2, and LDAP. Because OAuth2 is the basis of OpenID Connect, we decided to continue looking into this protocol.

6 Security of the proposed solutions

One of the main requirements of this study is that the solution is secure. As with all distributed systems, designing a secure SSO solution requires careful thought and planning. It is always useful to think as a "bad guy" and try to come up with ways to compromise the security of the system that you are designing.

The goal of this chapter is to find out if there are major flaws in the protocols that cannot be fixed and that result in an insecure solution. Since none of the remaining SSO protocols contain a major security flaw?or at least none that has been found some protocol-specific security issues are mentioned to give an idea of what to look for when implementing a solution. This chapter gives an overview of some of the vulnerabilities that are known. The issues mentioned are probably not all that exist; these are merely the issues we found. Vulnerabilities present in multiple solutions are mentioned in Section 6.2.

The mere fact that fewer security issues are known for a certain protocol does not mean that it is more secure. It may be that fewer issues have been found because fewer people have looked into the protocol. A popular protocol probably has had more people looking into its security, and this increases the chance that a security flaw will be found. However, absolute security is impossible [15].

For this study we only look at potentially interesting solutions based on the functional comparison in Chapter 5:

- OAuth2
- OpenID Connect
- SAML
- LDAP

Known security issues are discussed for each of these solutions and if possible countermeasures are provided.

6.1 Attacker model

To get an idea of what security issues to look for, an attacker model is created. This section defines which skills attackers have and why they want to target CCV.

CCV is an interesting target for attackers because payment data is involved. Even when the data contains only statistics about transactions, leakage of these data can be interesting for attackers. Such a data breach brings discredit to CCV, which may result in loss of customers.

There is always some risk affecting the information system. It should not be a goal to eliminate all risk; this is impossible for financial and technical reasons. Also, it does not make sense to apply a countermeasure that is more expensive than the cost of the asset. The level of security that should be implemented should be based on the potential risk and the cost of fixing it.

To make a good risk-and-cost estimation, it is important to know which skills and tools attackers have. In this thesis we assume that,

- The attacker has the ability to perform passive attacks, such as monitoring unprotected communications, performing traffic analysis and decrypting weakly encrypted traffic.

- The attacker has the ability to perform active attacks such as man-in-the-middle, replay and relay attacks.
- The attacker knows the protocol used.
- The services, IdP and AS are running on servers that cannot be physically accessed by an attacker.
- The services, IdP and AS are running on servers that cannot be remotely controlled by an attacker.
- The encryption and the signatures used to secure the messages cannot be broken or generated by an attacker.
- The communication is secured using TLS.

6.2 General security issues

In this section, we discuss vulnerabilities that can be found in all of the SSO solutions discussed. These are some basic issues that need attention when building an implementation that uses SAML, OpenID Connect or LDAP.

6.2.1 Eavesdropping and stealing cookies and tokens

Tokens are stored in local/session storage or in a client-side cookie. When a cookie is created, it may be eavesdropped on the network, or hijacked by malicious XSS scripts that are injected on any page under the service domain. To address this issue, Facebook revised its SDK to use a signed authorization code instead of an access token stored in a cookie [83]. However, it turned out after the revision many services that use the Facebook SDK store the token in a cookie themselves.

Many of the token-based SSO examples found use so-called "bearer tokens" to communicate with their actors. These bearer tokens are unencrypted. Protection should for this reason be offered by the use of TLS encryption on the channels. Bearer tokens have no internal protections. If for any reason the TLS layer is broken, the entire security architecture of bearer tokens falls apart [27, 51]. The person who holds the token is regarded as the rightful owner, as it is impractical to prove otherwise [86].

Relying on only one layer of protection is a mistake, because defence in depth is important for good security [2, 51]. If the one layer turns out to have a vulnerability, then the entire protocol can be compromised; a multi-layer security implementation, on the other hand, remains protected by the other layers. Since CCV handles sensitive data, defence in depth is a must.

Some AS have a so-called "automatic authorization granting" feature, which returns an access token (without the user's intervention) for an authorization request. This feature is activated when the requested permissions denoted in the request have been granted by the user previously and when the user has already logged into the IdP in the same browser session. This automatic authorization mechanism allows an attacker to steal an access token by injecting a malicious script into any page of a service to initiate a client-side login flow and subsequently obtain the responded token. Two possible attacks are the following:

- Use the current page as the redirect URI to extract the access token from the fragment identifier.

- The exploit dynamically loads the SDK and uses a special SDK function named `getLoginStatus` to obtain the access token.

Both attacks (so-called "clickjackings") can be executed by creating a hidden `iframe` element to transport a forged authorization request to the AS and obtaining an access token in return. Once the token is obtained, the exploit script sends it back to `attacker.com` with dynamically created image element [83].

Another widely exploited web vulnerability is Cross-Site Request Forgery (CSRF) [59]. This is used to make a user load a webpage that contains a malicious request which in turn disrupts the integrity of the victim's session data on the website. There are many ways to exploit a CSRF attack. The malicious HTML construct could be embedded in an email, hosted on a malicious website, or planted on websites through an XSS or SQL-injection attack [59, 83]. It is common that they are embedded in the attack URL of an HTML construct for example, in an image element. This causes the browser to automatically execute the malicious request when the infected element is loaded. There is no detectable difference between the attack request and that received from a legitimate user, since the malicious request originates from the victim's browser (which is a legitimate source) and the session cookies that were set previously by the website are sent along with the request automatically.

A typical CSRF attack requires that the victim already has an authenticated session with the website. If the user is not yet authenticated, then a force-login CSRF attack can be carried out by an attacker to achieve this prerequisite. This attack takes advantage of the "automatic authorization granting" feature. CSRF logs the victim into the service automatically by making him or her view an exploit page that sends a forged login request or authorization request to the browser of the victim. If this exploit is carried out successfully, the attacker is able to carry out subsequent CSRF attacks without passively waiting for the victim user to log into her website. Some services sign users in automatically if the user has already logged into Facebook; but this "auto login" feature enables an attacker to launch CSRF attacks actively [83].

Countermeasures

To prevent a cookie-stealing attack, it is possible to send a hidden parameter in the http-requests. The receiving service checks to see if the hidden parameter is also included when the cookie is sent. If the parameter is missing, the cookie might be stolen. When the service detects that the parameter is missing, access should be prohibited and re-authentication should be required. It would be nice to keep track of these events; if they occur multiple times in a certain timeframe, the user should be informed of the possible attacks.

Another way to prevent cookie stealing is to use HTTP-only attributes. This attribute determines that the token can only be used by HTTP request. Script languages like JavaScript do not have access to this token [68, 83]. This means that this solution cannot be used in combination with services that are built in JavaScript.

However, even when the cookies are secured with HTTP-only attributes, tokens can still be eavesdropped on or stolen. By sniffing the unencrypted communication between the browser and service, access tokens can be eavesdropped. HTTPS provides end-to-end protection, and it is therefore recommended as a way to protect against attacks that manipulate network traffic. By setting the secure bit in the cookie, it is possible to force the browser to transmit the cookie only when HTTPS is in place [68].

HTTPS imposes management-and-performance overhead; it makes web contents non-cacheable and may introduce undesired side-effects such as browser warnings when secure (HTTPS) and insecure (HTTP) content is mixed. Because of these unwanted complications, many websites use HTTPS only for their login pages [72, 83]. In Sun's research [83], 49% of services use HTTPS to protect their login forms, but only 21% use HTTPS for their IdPs. Because all the request to a services require an access token, all the request should be protected using HTTPS otherwise the tokens can still be eavesdropped on. For this reason, our advice is to protect all pages.

When all pages are protected with HSTS it is possible to make the browser remember that the webpage may only be accessed using HTTPS. HSTS (HTTP Strict Transport Security) is used to protect websites from downgrade attacks. For example, when yourwebsite.com is replaced by evilwebsite.com by way of a hacked DNS, then because evilwebsite.com lacks the certificate for yourwebsite.com it will try to make you use their website using http. Because the original website used HSTS which says that the webaddress may only be accessed using HTTPS the browser detects that the certificate is missing and prevents access to the webpage e [34, 67].

A countermeasure against the clickjackings attacks is to use the so-called "X-Frame-Option" setting in the webserver. This header can take the values DENY, SAMEORIGIN, or ALLOW-FROM origin, which will prevent all framing, prevent framing by external sites, or allow framing only by the specified websites, respectively [71, 72]. Since CCV does not use iframes of their own websites, the DENY value should be chosen.

Another countermeasure against iframes is called frame busting. In framebusting, the website checks to see if it is used in an iframe. If this is the case, it tries to bust the frame. Typically, this busting is achieved with JavaScript [66].

The use of bearer tokens should be avoided. Instead of bearer tokens, the more secure JSON-web tokens should be used. These tokens can be signed and encrypted, which results in a solution that is protected by multiple layers of security and privacy [88].

Cross-Site Request Forgery can be prevented by checking the referrer header. It is possible to spoof the referrer header by using your own browser; this is, however, impossible with a CSRF attack [60]. Checking the referrer is a method that is commonly used to prevent CSRF.

Checking the referrer header is not the best way to prevent CSRF, however, as there are some implementation issues with the referrer checks. For example, when the CSRF attack originates from an HTTPS domain, the referrer will be omitted. This should be considered as an attack, since the request is performing a state change [43, 60].

Another way to prevent CSRF attacks is to use origin headers. Origin headers are introduced as a defence against CSRF and other cross-domain attacks. Unlike the referrer-header approach, the origin is present in a HTTP request that originates from an HTTPS URL [60]. To determine whether the origin is valid, the following steps should be followed [9]:

1. Let A be the first origin and let B be a second origin.
2. When either A or B is not a scheme, host or port, then return a no origin message.
3. When the scheme components of A and B are not identical, return false.
4. When the host components of A and B are not identical, return false.

5. When the port components of A and B are not identical, return false.
6. Otherwise, return true [9].

6.2.2 Spoofing of tokens

Spoofing comes in various forms. In this section we describe two common attacks: an impersonation attack and a session swapping attack. Spoofing attacks are exploited by the lack of contextual binding. By contextual binding we mean that the service does not provide a state parameter during the authorization request to set the state between the request and response. This state parameter is typically a value that is bound to the browser session, for example, a hash of the session, which will be appended to the response by the IdP when the user gets redirected back to the service.

An impersonation attack works by sending a stolen or guessed SSO credential to an IdP through an attacker-controlled user-agent. To successfully carry out an impersonation attack, these things are required:

- The attacker should be able to obtain a copy of or guess an SSO credential.
- The SSO credential must not be protected by one-time use.
- The service does not check to see if the response is sent by the same browser that issued the authorization request (i.e., because of a lack of "contextual binding" validation).

Sun [83] studied on impersonation attacks on implementations of the Facebook SSO SDK and found that 9% of the services use the user's identity provider profile as an SSO credential. For these services an attacker can simply log in as the victim by using the victim's Facebook account identifier, which is publicly accessible.

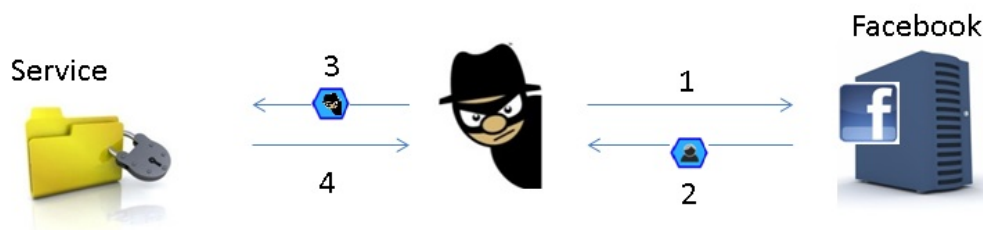


Figure 19: Token spoofing

Because of this users can log into a service with the Facebook ID of someone else as shown in Figure 19. A malicious user can claim access to the account of another user by using the following steps.

1. The application starts with a malicious user who uses Facebook as SSO provider.
2. Facebook verifies the user credentials and returns an access token.
3. Before sending this token to the service, the user changes the Facebook ID to that of another user.
4. The service uses this changed Facebook-userID and logs the user in using the manipulated ID.

"Session swapping" is another way to exploit a lack of contextual binding. To launch a session-swapping attack, the attacker signs into a service with the attacker's identity from the IdP, intercepts the SSO credential on the user-agent, and completes the attack by embedding the intercepted SSO credential in an HTML construct (e.g., img, iframe). This causes the browser to send the intercepted SSO credential to the IdP when the exploit page is viewed by a victim user.

When the intercepted SSO credential is bound to the attacker's account on the service, a successful session swapping exploit allows the attacker to stealthily log the victim into her service as the attacker to spoof the victim's personal data or mount an XSS attack.

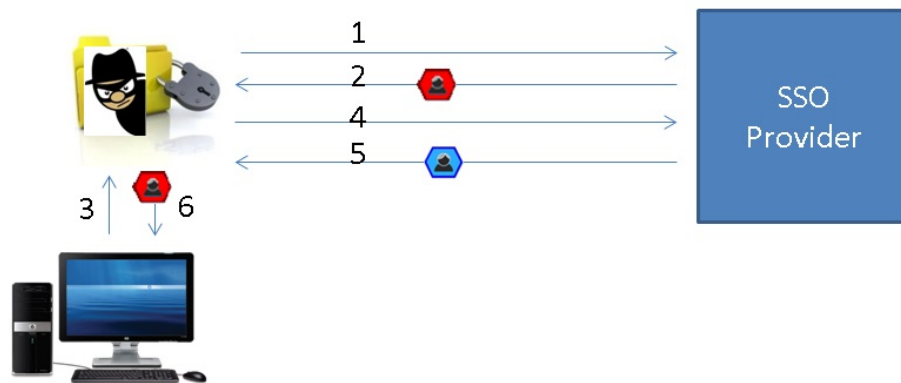


Figure 20: session swapping

As depicted in Figure 20, the following steps are used to exploit this attack:

1. A malicious user M wants access to a service logs in at an SSO provider.
2. The SSO provider verifies the user credentials and returns an access token.
3. User X wants to access the same service as M.
4. To access the service X logs in at a SSO provider.
5. The SSO provider verifies the user credentials and returns an access token.
6. User M who is acting as a man in the middle changes the response of the SSO provider with the response provided to him. The token of user M is now returned to user X instead of his own token.

Countermeasures

Both attacks are caused by the same issue and can for that reason be solved with the same countermeasure: the use of nonces in the communication. In that case, the client will detect that the nonce returned by the service/SSO provider does not match the nonce that he sent when he logged in at the SSO provider. The communication should also be signed; otherwise the man-in-the-middle can replace the nonce to make the response seem valid when it is not. TLS offers both measures: it secures the communication and uses nonces to prevent replay attacks. Thus, these attacks can be prevented with TLS.

6.2.3 DNS Spoofing

Attackers could spoof a DNS. Thus, `ldap.example.com` could be redirected to the addresses of `evil.com`. This will cause the data of LDAP services to be redirected to the attacker's server. This data could, for example, contain email addresses and even user credentials.

Countermeasure

One solution to detect an invalid redirect is to monitor the DNS for (unexpected) changes. When unexpected changes are made then you possibly have a security breach.

The Domain Name System Security Extensions (DNSSEC, for short) provides a solution to this problem. The primary goal of DNSSEC is to provide authentication and integrity for data received from the DNS database [5, 87]. DNSSEC is an extension of the DNS, and is introduced to prevent attackers from redirecting users to other domains. DNSSEC enables the identification of DNS sources and ensures that content cannot be modified during transfer. It does so by cryptographically signing the resource records and adding asymmetric cryptography in the DNS hierarchy. These signature schemes are based on public-key cryptography. Currently, DSA and RSA are the digital-signature algorithms supported by DNSSEC [5].

The idea behind DNSSEC is that each node in the DNS tree is associated with some sort of public key. All messages from the DNS server must be signed under a corresponding private key [5]. It is assumed that the public keys of the DNS servers are publicly known. Since CCV hosts its own services, it should be aware of the possibility of DNS spoofing and implement DNSSEC.

6.2.4 Forgotten log out

This attack starts with a user that is prompted by the server to enter the login credentials to grant access to the service. If the credentials are valid, the user gets redirected back to the service. Serious security concerns can result if the implementation specifies that the user remains logged in on the server even after leaving the client application—as twitter does, for example.

For example, Mr. X registers himself, signs into his twitterfeed and selects twitter as his IdP and AS to prove permission to post an item on his blog. Mr. X gets directed to the Twitter authorization endpoint where he signs in using his Twitter account. Based on his Twitter username and password an access token is granted which is sent as prove of authorization to the twitterfeed. Mr. X completes the feed, closes his browser and walks away. A malicious user with access to the unattended computer on which Mr. X has worked can now compromise Mr. X's Twitter account [45].

Countermeasures

To limit forgotten log out attacks we advise a combination of two countermeasures:

1. One solution is to make users aware of the risk of not logging out in the hope that they will log out in the future. To make it easier for them to log out, a clearly visible button that gets the users attention should be used.

2. To limit the chance of an attacker abusing an active session, the lifetime of an access token must be set. This is, however, hard to set right. When the lifetime is too short, a user has to log in often, which is not good for the usability. If the lifetime is too long, an attacker has time to sniff through the user's data. Perhaps a short token lifetime that is reset after a new request will solve the issue. In that case, the attacker has to take over the system at which the user immediately afterward.

6.3 Known security issues in OAuth2 and OpenID Connect

This section discusses the security issues that we found for OAuth2 and OpenID Connect. We have combined these protocols, because the issues we found have a huge overlap. We conclude this section with specific security issues for both solutions.

6.3.1 CSRF Attack

Kiani [45] shows that both OAuth2 and OpenID Connect are vulnerable to CSRF attacks. This attack can be executed using the four steps explained below. The term OAuth2 account, login and provider could also be OpenID Connect account, login and provider.

1. First we choose a client which suits the attack "condition", for example website.com. Then we start the authentication process and add an OAuth2 login. It is important that we get a callback from the IdP. We should not visit it. Since all modern browsers redirect their users automatically this is difficult, but it can be solved using the NoRedirect extension for Firefox, for example.

2. For step two we copy the following link, for example:

```
"http://pinterest.com/connect/facebook/?code=someCode#_=_"
```

and put it into an image tag (), iframe (<iframe>) or anything else that can be used to send requests.

3. Afterwards, the user of website.com must send an HTTP request on the callback URL. We can force the browser to visit example.com/somepage.html by placing for example an <iframe src=URL>, post on his wall or by sending him an email/tweet. It is important that the user is logged on to website.com when he sends the request. Now the OAuth2 account is attached to the user's account on website.com.

4. From this moment on it is possible to press "Log In" using OAuth2 Provider X and we are logged in directly to the account of the victim.

Since we are now logged in we can read private messages, post comments, change payment details, etc. When we are ready, we can disconnect from the OAuth2 Provider. No one will have an idea what has happened; so long as no identifying information is posted, no fingerprints are left [36, 45].

This attack can be executed at each implementation that does not send a ?state? param if the redirect_uri param is static [30, 36, 49].

Countermeasures

To solve this problem, a special optional param `?state?` must be sent. It may be any random hash you get back by the provider in the users callback: `?code=123&state=HASH`. Before adding an OAuth2 account, the session state must be equal to the params `[state]`.

6.3.2 Open redirect attack

An open redirect is an application that takes a parameter (for example from an URL) and redirects a user to the value associated with that parameter without any validation (e.g., `test.com?redirect=evilwebsite.com`). This vulnerability is used in phishing attacks to get users to visit malicious sites without realizing it [49, 61].

In this section, Facebook is used to give an example of how the attack can be executed. Facebook's SSO system was vulnerable because the authentication of parameter `"&redirect_uri"` in their SSO system was insufficient [41, 47]. It was possible to abuse this parameter to carry out open redirect attacks. Facebook has already solved this issue.

This attack could be used to collect sensitive information of both the third-party and users by setting the following parameters:

```
"&response_type"=sensitive_info,token
```

```
"&scope"=email,user_birthday,user_likes
```

The vulnerabilities occurred at facebookpage `"/dialog/oauth?"` with parameter `"&redirect_uri"`, for example:

```
www.facebook.com%2Fdialog%2Foauth%3Fclient_id%3D152973104736490%26redirect_uri%3Dhttp%253A%252F%252Fwzus.ask.com%252F%253F%253Dt%253Dp%2526u%253Dhttp%253A%252F%252Fwww.evil.com%252Fp1%252Fp2%252Fdistance.html%253F%26response_type%3Dcode%26scope%3Demail%252cuser_location%252cuser_birthday%26display%3Dpopup&redirect_token=orfdD9b_r0QrWs9uwpTC82X5rjh8MTQyNDkzOTMwMEAxNDI0ODUyOTAw
```

When a logged-in Facebook user clicks the URL above, he/she will be asked for consent to allow a third-party website to receive personal information. If the user clicks OK, the service redirects the user data the URL assigned to the parameter, `"&redirect_uri"`. If the user is not logged in to Facebook and clicks the URL, the same thing will happen after the login procedure.

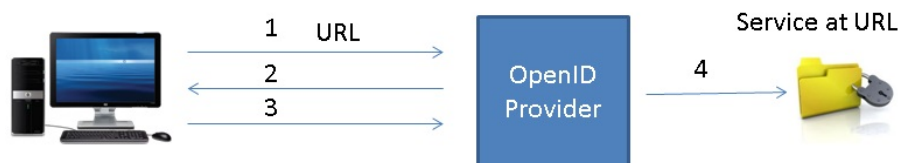


Figure 21: Open redirect

To summarize, these are the steps we described until now:

1. A user wants to log in using Facebook. The client creates a request to Facebook's OpenID provider and hands over a URL to which Facebook should redirect after the user gives consent to Facebook to send data to the client.
2. Asks the user to give consent.
3. Gives consent.
4. Creates tokens and redirects the user to the URL provided in step 1.

Facebook used to allow all URLs that belong to an authorized third-party domain. However, these URLs could be manipulated using the "&redirect_uri" parameter in the URLs. This parameter is supposed to be set by the third-party service, but an attacker could change this value to carry out an attack (e.g. a user could be redirected from Facebook to a vulnerable third party URL unwillingly) [41, 47].

Once the user accepts the authorization request, the attacker could completely bypass Facebook's authentication system and attack more easily. We use the attack scenario explained by Wang [41] and use the following redirect URL: "http://www.evil.com/p1/p2/distance.html". For this example we suppose that it is malicious and contains code that collects sensitive information of both third-party apps and users.

Using the ask domain we can create the following vulnerable URL:

```
http://wzap.ask.com/r?t=v&d=im&u=http%3A%2F%2Fevil.com
```

Using this URL we can create a malicious Facebook login request:

```
https://www.facebook.com/dialog/oauth?client_id=152973104736490&redirect_uri=https%3a%2f%2fsocial.ask.com%2fGS%2fGSLogin.aspx%3fst%3dzNQz0TjIZd42P_zI5MUVw5WtCHw7EDMc1YEjBVuz3bU.&response_type=code&scope=email%2cuser_location%2cuser_birthday&display=popup
```

And change it into:

```
https://www.facebook.com/dialog/oauth?client_id=152973104736490&redirect_uri=http%3A%2F%2Fwzus.ask.com%2Fr%3Ft%3Dp%26u%3Dhttp%3A%2F%2Fwww.evil.com%2Fp1%2Fp2%2Fdistance.html%3F&response_type=code&scope=email%2cuser_location%2cuser_birthday&display=popup.
```

This results in a response being sent to our domain with the access token.

```
http://www.evil.com/p1/p2/distance.html?code=AQBcJWmTvaW__V0klHw0-MEVI_aD2dsqQqzABILHTJvo6wG_gMz5AWEcTQpS2-lQId1wiExUamtVoeHvg2DATliVyH6f04lGtZaYy jEh- .....
```

The four steps in Figure 21 show that during step 1 the URL is changed to a slightly different URL in which the domain stays the same (ask.com) but the parameters are changed. A redirect URI is added, which means that the access data is returned to ask.com which directly sends all of its data to evil.com. So a fifth (or sixth when you consider the changing of the URL) step is added in which the client is redirected to another domain. If evil.com

is a malicious domain, it can abuse the access token that is redirected to it [41, 47].

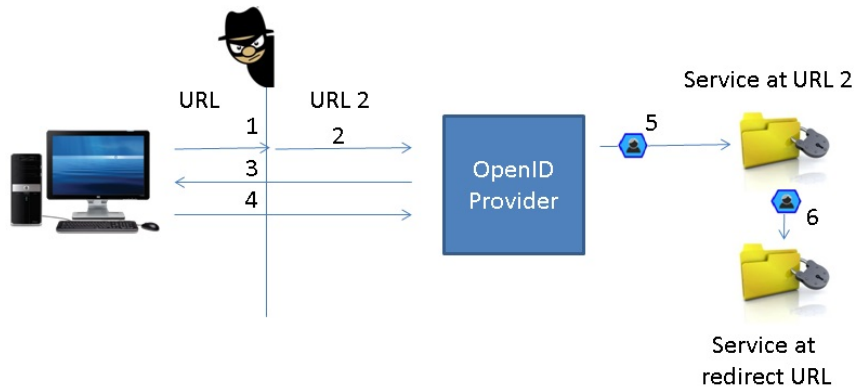


Figure 22: Open redirect attack

Figure 22 shows an example of how the attack can be executed :

1. A user wants to log in using an SSO account. The client creates a request to Facebook's OpenID provider, for example. A URL is handed over to which Facebook should redirect after the user gives consent to Facebook to share data with the client.
2. A man-in-the-middle changes the URL to a URL with a redirect.
3. The SSO provider asks the user to give consent.
4. The user consents to share information.
5. Creates tokens and redirects the user to the URL provided in step 1.
6. The data gets redirected to malicious client application B.

Instead of the man in the middle this attack could also be exploited by a malicious client B that looks and acts like client A.

Facebook has blocked this attack (probably by using a whitelist of redirect URLs). The ask.com redirect still works today (25-02-2015) [27, 41].

Open redirectors can also be abused by sending users to malicious sites. If that happened, the site would have to trick the user into allowing their third-party credentials, from Facebook or Google for example to log into the site. This process involves social engineering [77].

Countermeasures

The first countermeasure that should be taken by the SSO provider is: check if the access code received came from the domain to which the access token was issued. Using this fix, the theft of a token is less useful, as the access code can only be used on the official website (ask.com in the example above).

After this fix, it is still possible to attack using XSS. When a website is attacked using an XSS request, it can be sent from that domain. XSS then sends the stolen token with the valid domain, which matches the issued domain, and steals, for example, the personal data of a user.

To prevent this kind of attacks, a whitelist can be used that contains all allowed redirect URLs. This solution requires that third parties register their redirect URL (which must be static) before they can use the SSO provider. This redirect URL should be a specific redirect URL (e.g., `http://app.com/account`), and only redirections to this specific URL should be allowed. If no specific URL is used, an attacker can simply register the domain `ask.com` and use a malicious redirect URL: for example, `http://wzap.ask.com/r?t=v&d=im&u=http://evil.com`. A whitelist even helps to prevent the attacker from sending an access code to an invalid URL, and this decreases the chance of tokens getting stolen.

6.4 Known security issues in SAML 2.0

This section discusses four known security issues in SAML 2.0. For each, we will explain what the attack does and how it can be prevented. The Open redirect attack explained in Section 6.3 also works for SAML, and since the attack is executed like that for OAuth2 and OpenId Connect, we will not make any further mention of it in this section.

6.4.1 XML signature wrapping attack

In SAML, claims are represented in XML and used as security tokens. The SAML language is used to create these authorisation claims, whose authenticity is protected by XML digital signatures. SAML messages contain a security header with a signature element, that refers to one or more message parts that have been signed. The XML structure represented in Figure 23 could contain a statement such as "Alice is authorised to use this secret service". Typically, parts of the message are referenced by an ID. The recipient must find the element in the request that has the corresponding ID to validate the signature. The claim has an identifier (1 in this example), which should match with the digital signature for the validation. [33, 46, 80].

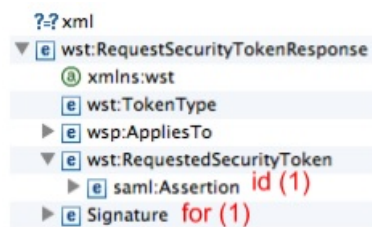


Figure 23: SAML signature [46]

Typically, an XML signature wrapping attack exploits the fact that the signature does not convey any information as to where the referenced elements are located in the document tree [46, 53, 80].

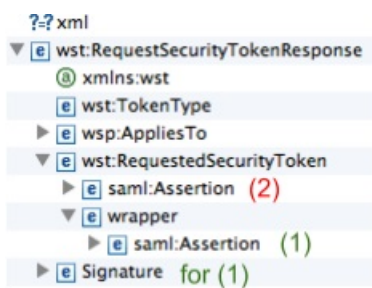


Figure 24: SAML signature attack [46]

In the scenario in which the credential is protected by a signature in the header of the request, the receiver verifies the signature and indicated that the signed credential can be trusted. The validation is completed after the receiver compares the wsu:Id of the claim to the reference URI in the signature.

An XML signature wrapping attack could be exploited as follows. A man-in-the-middle could use a valid request, copy the original claim (1), move it to a wrapper node and add a new malicious claim (2).

The message receiver searches for the referenceID as part of the signature validation process, and hits the copied claim element first. The message passes the validation because it has not been changed. To complete the process the recipient validates the Id of the claim by checking to see if it is actually referenced in the signature. Because the original body of the claim was not modified, the document will still pass the digital signature validation [46, 53, 80].

Countermeasure

To prevent an XML signature wrapping attack the structure of the XML received should be checked for consistency. To do this properly, it is important to define how the XML should be formatted. Also third-party schemas should be validated prior to using them, even if they come from a standard. Existing schemas should be updated and if possible wildcard statements should be removed [46].

The schema validation checks to see if the structure of the SAML token is correct. It does now, however, validate the authenticity and integrity of the claim. For this reason, the signature needs to be compared with the trusted public key of the signer [46].

Technically, digital signature validation is a complex process that should involve the following steps:

1. Establish trust for the public key of the signer with a trusted anchor (CA certificate), validation of the path, and revocation checks.
2. Use the public key of the signer to cryptographically validate the document signature.
3. Cryptographically validate the signed hash of the document against normalized hash computed from a received document to confirm the document's integrity.

The first of these steps is the most complex of the three because it requires validation of the features of all certificates in the trust path, including keyUsage, time constraints and revocation status [46].

The final check of this step is to validate the digital signature against the trusted public key of the signer. The signed document will frequently contain public keys. Since the purpose of signature validation is to validate the document received from an untrusted source, any key embedded in the document must not be trusted. The public keys must not be used for validation until they are themselves validated [46, 53, 80].

In case of SAML, we have only to deal with a single-identity provider authenticated by a single certificate, which simplifies the validation process. We can validate the public key directly against this certificate that we trust explicitly [46].

6.4.2 Man in the middle attacks on bindings and profiles

SAML uses SOAP over HTTP(S) bindings to communicate between services. Because a SOAP binding does not require authentication and has no explicit requirements for in-transit confidentiality or message integrity, it is vulnerable to a wide variety of common attacks: for example, eavesdropping and man-in-the-middle attacks [35].

A MITM attack can interfere and forward the communication between B and S. The two honest parties cannot distinguish the adversary from the intended communication partner. Based on this information, users can be tracked. This is undesirable from a privacy point of view. To solve this problem the same countermeasure can be used as discussed in [26, 35].

Countermeasure

In order to prevent man-in-the-middle attacks, some sort of two-party verified conversation should be used. The system should allow both parties to determine that the message they received came from the other party. Since SAML depends on asynchronous messaging, the authentication can be combined with information from the transport layer to confirm that the sender and author are the same. This can, for example, be solved by using SOAP over HTTPS with both server- and client-side certificates.

6.4.3 User tracking

When service S has a connection to the browser, it is possible to use this connection for user tracking. This connection does however not provide authentication, therefore tracked information cannot be bound to a specific identity. We assume that browser B stores it by the hostname of source site S.

- (a) S → B: request for prove of identity of user
- (b) B → S: knowledge of identity

A MITM attack can interfere and forward the communication between B and S. The two honest parties cannot distinguish the adversary from the intended communication partner. Based on this information users can be tracked which is an undesired situation from a privacy point of view. To solve this the same countermeasure can be used as discussed in Section 6.4.2.

6.4.4 Replay

Prerequisites: An adversary A can intercept a connection and can observe the connection from browser B to the receiver URL <URL> D.

1. B → D: Redirect the client to <URL>D
E: Eve E intercepts this message and closes the connection from B to D.
2. E_B → D: replay of redirect client to <URL>D
Eve replays the message she intercepted, impersonating B. D cannot distinguish between B and Eve because of the lack of authentication. D therefore proceeds as specified in the protocol.
3. D → S: SAML request for permissions.
4. S → D: SAML response containing permissions.
This response contains the permissions for B since D thinks he is talking to B.

5. $D \rightarrow E_B$: Response to Step 2
D will grant E_B the permissions of B.

Countermeasure

This problem can be solved by using the standard replay-attack countermeasure, a nonce. When it is required for B to send a nonce, then D can check to see if the nonce received is fresh. This nonce could be a UNIX timestamp, which then also can be used to check if the granted permissions are expired. Since we sign the message, the possibility of the nonce being guessed is not an issue, since only a person having the private key can generate the signed message.

When B signs the message before sending it to D, then D knows that he is talking to B. Because of the nonce which D can return to B is encrypted using B's public key, he also knows he is talking to D and B knows the message is not a replay [26].

Because of the integrity and confidentiality property of step 3 in the example above, adversary E cannot see or modify the content of the message p. But these properties do not prevent a replay attack. Adversary E can impersonate browser B to destination site D. Because of the lack of authentication in this step and because of the missing identifiers in the redirect, D cannot distinguish between the parties B and E_B . Apart from the IP address of the communication partner, the view of D is the same in the communication with B and E_B .

Source site S can input the IP address of browser B as query-string parameter into the redirect. The destination site D checks the IP address in the message received to see if it is the same as the IP address of browser B. If the IP addresses do not match, then the check fails and D aborts the protocol run. This measure might interfere with the IP rollover of certain ISPs, because it produces a false positive if B's IP address is changed between steps 2 and 3 of the example at the beginning of this subsection. A second way to prevent the described attack is to enhance the integrity property claimed for steps 1-3 so that it includes binding to the sending party or the underlying channel. One can also choose to use a secure channel, which provides freshness and replay prevention [26, 35].

6.5 Known security issues in LDAP

Like the other protocols LDAP also has its weaknesses. This section introduces issues that we have found online and explains how each of them can be prevented.

6.5.1 LDAP basic security settings

If a bind (authenticate user) is used, then TLS protection should be used to prevent exposure of the passwords on the network. Servers that contact an LDAP service must disallow the use of passwords when TLS is not in use. Sadly, very few servers use this setting, thereby allowing password theft via unprotected LDAP services.

The X.500 computer network standard, to which LDAP belongs, has a so-called "get back exactly what you put in" principle. If this principle is followed, then servers store passwords in plain text or in a form that can be converted back to plain text by default. It is, however, recommended that passwords are stored using a non-reversible cryptographic hash that includes a unique salt. This combination provides good protection against recovery of passwords from stolen disks or backup tapes [26, 38].

To ensure the availability of a service, servers should apply a limit on the size of search results. Large sets of results can consume a lot of memory, and these cost a lot time to transfer the big message to the client. The appropriate value for the limit depends on the application. When we consider a server that only supports authentication, then a limit as low as two entries might be enough; but a server that supports a browsable service might require a limit of 100 entries or more.

6.5.2 LDAP injection

LDAP injection is based on the same idea as SQL-injection: i.e., not validating the user input. Take for example some code that generates and executes an LDAP query that returns all records of employees who are managed by a given manager. Let us presume that the manager's name is returned via an untrusted connection (e.g., HTTP). The result will look something like this:

```
...
DirectorySearch src = new DirectorySearch("(manager=" + managerFirstName.Text + ")");
src.Root = de;
src.Scope = Scope.Subtree;

foreach(Result res in src.FindAll()) {
...
}
```

When we for example look for employees who are managed by John Doe, the filter that will be executed is this: (manager=Doe, John).

Since the filter is generated by concatenating a string and the not-validated-user input string, the query may be abused by using LDAP meta characters. If a user enters the string Eve, Evil)(|(objectclass=*) for the manager name, the query will be affected and will become (manager=Eve, Evil)(|(objectclass=*)))[3].

Countermeasure

It is good to assume that all user inputs are potentially malicious. For this reason, all user inputs must be sanitized before they can be used as parameters in the LDAP interpreter. Regular expressions can be used for detecting injection attacks by analysing the parameter by and looking for known attack patterns. This technique has, however, the disadvantage that false positives may occur. This approach might exclude some valid user inputs such as parentheses, asterisks, logical (AND "&", OR "|" and NOT "!") and relational (=, >=, <=, =) that might be valid in some cases.

7 Final comparison

The security issues discussed in Chapter 6 do not help to exclude SSO protocols; all protocols have security issues, but all of the issues discussed can be solved. However, it is likely that not all security issues have been found. For this reason, it is important that the chosen protocol is used by a large community. Security issues will in that case be detected earlier because more people are looking into the protocol.

7.1 Comparison of adoption

We have measured the community using several aspects:

- The number of issues reported at stackoverflow.com in the months January, March and May of 2015 and the number of them that are solved.
- The total number of reported issues.
- How many results Google returns when searching for a certain solution and the release date of a protocol.

The number of questions is found by adding `is:question` to the name of the protocol, sorting the results by date and counting the number of questions in the month of January.

	Questions	Answered	Total questions	Hits Google	Release date
OAuth2	217	144	12,332	851.000	October 2012 [30]
OpenID C	21	25	918	827.000	February 2014 [75]
SAML 2.0	19	9	684	377.000	March 2005 [70]
LDAP	192	143	23.264	15.200.000	1993 [38]

Table 2: Adoption January and information

We also measured the number of questions and answers for the months of March and May, which resulted in the following numbers:

	Questions Mar	Answered Mar	Questions May	Answered May
OAuth2	360	259	352	147
OpenID C	36	38	60	43
SAML 2.0	22	14	27	8
LDAP	189	140	188	118

Table 3: Adoption March and May

These tables show that OAuth2 is the most active solution of them all. The low total number of SAML 2.0 issues relates to the fact that some issues are not reported as a "SAML 2.0" issue but as "SAML". In percentage, the number of OpenID Connect questions has grown by almost 300%. In actual numbers, OAuth2 has the biggest growth, with an increase of 145 questions in May. Of the protocols discussed here, LDAP is the only protocol for which the number of questions has not increased in the specified time domain. LDAP does, however, have the highest number of hits on Google, which can be explained by the fact that it is also the oldest of the protocols.

7.2 Choosing the protocol

One of the main requirements of CCV is that the solution should offer authentication and authorization. Based on this requirement, only three solutions remain: SAML, OpenID Connect and LDAP.

In our opinion, OpenID Connect is the best solution for the problem proposed in this research. The comparison between SAML, OpenID Connect and LDAP in 5 show that OpenID Connect is a new standard that is being implemented by leading companies and uses the new web-standards. Also, the LDAP protocol was built for server SSO in a local network and not for web applications, and it is for this reason also of less interest. OpenId Connect offers authentication and authorization and is a fast growing protocol, as explained in Section 7.1. The growth in use might be linked to the new standards it uses and to its adoption by Google, Facebook, etc. Based on this information, we think that OpenID Connect is the best fitting solution for CCV.

8 The solution for CCV

This chapter contains various decisions about how to implement OpenID Connect. The first section contains a description of the OpenID Connect flow that best meets the wishes of CCV. Before an implementation plan is made, we must decide whether CCV should use centralized or decentralized authorization. Afterwards, we compare by reference tokens and by value tokens and make a decision based on the pros and cons of both types. Based on the combination of the previous choices, two possible flows remain. We discuss these in Section 8.4. We then discuss some additional issues, such as migration, sign and encrypt, and verticals. We conclude this chapter with description of the current authentication system and a discussion of how the new solutions should be implemented.

8.1 Which flow?

In this section, we determine which OpenID Connect flow should be used and which of the roles should be fulfilled by which of the servers. For the flow, we have three options:

1. The Implicit Flow
2. The Code Authorization Flow
3. The Hybrid Flow

From the description of the protocols in Section 4.3.1, we know that the Code Authorization Flow is very similar to the Hybrid Flow. The main difference is that the Hybrid Flow returns the tokens directly to the client and that the Code Authorization Flow returns an authorization code which the client uses to receive the access token from the AS [75].

When we compare the Implicit and the Code Authorization Flow, we see more differences. The Code Authorization Flow offers refresh tokens and client authentication and gives the application the option to have its own login form; the Implicit Flow requires re-authentication and requires the user to log in at the IdP directly.

One of the requirements of this study is to prevent users from having to authenticate every time they want to access a service. Since the Implicit Flow does not offer refresh tokens, it does not meet this requirement. This leaves us with the Code Authorization Flow and the Hybrid Flow.

Section 8.2 explains that it is better to use decentralized authorization with security as its main reason. When using the Code Authorization Flow, the client receives the authorization codes of various authorization servers. The client has to know which AS belongs to which service; otherwise he cannot retrieve an access token. This issue does not exist in the Hybrid Flow, because the tokens are returned directly to the client. The client already knows to which service the access token belongs based on the request sent earlier.

To summarize, both the Code Authorization Flow and the Hybrid Flow can be used, but the Hybrid Flow is the easiest to implement and requires less communication. For this reason, the Hybrid Flow is the preferred solution for CCV.

8.2 Centralized or decentralized authorization

After deciding which flow to use, we started to make a sequence diagram of the solution. During this process, we had to decide where to put the authorization layer. We could use centralized or decentralized authorization. Figure 25 presents a schematic overview of how authorization can be implemented.

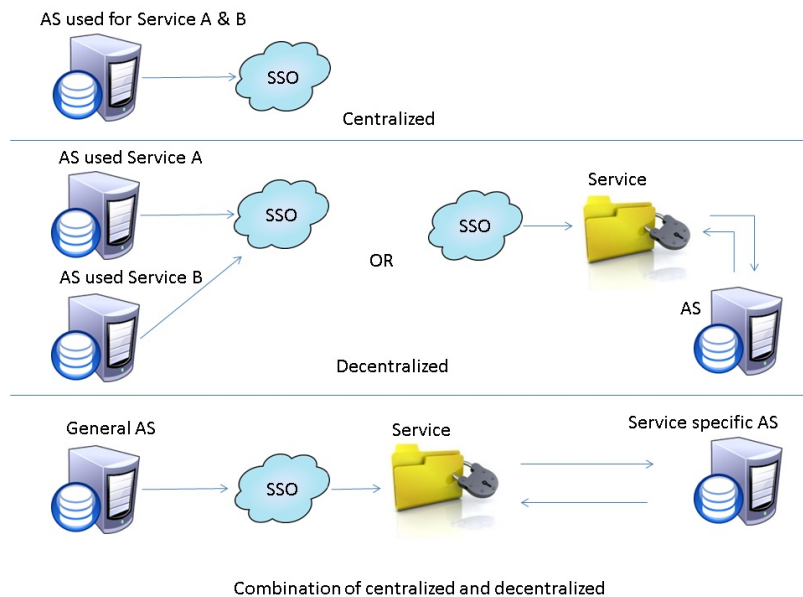


Figure 25: Authorization central, decentral or a combination

There are various ways to implement authorization. The authorization server may contain the authorization data of all SSO services (centralized), or multiple authorization servers may be used, one for each service (decentralized). Another option is to use a combination of these implementations. This means that we have one authorization server containing only the information that states whether a service may be accessed as a whole (may I visit service A?), and we have some additional authorization servers that manage the access rights for a specific service (which parts of service A may I visit?).

Decentralization makes it harder for an administrator to maintain these systems, which results in increased maintenance costs and increases the chance that a mistake is made. The resignation of an employee could, for example, lead to the re-provisioning of this user's access across many systems. To which services did he have access? To answer this question, all services need to be checked.

In the case of resignation, disabling centralised authentication will be enough to block the user from all services. This solution does not work when it is required that the user continues to have access to a part of the system for example, when access to invoices is still required. In that case, an application should be build that talks to each of these authorization servers and manages their rights. Because new services will be created, it should be easy to append these new services to the authorization management application. To achieve this, a principle like SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) should be used.

Because the data offered by several services contains sensitive data, we advise a decentralized authorization solution. If one access token is stolen, access to only one service or to a

small group of services is granted. This is also a solution that works well with verticals, as discussed in Section 8.9. We do not use the combination solution because we see no clear advantage in this solution and because it requires extra implementation work and additional resources. If access is denied for a service, this can be stored in the (decentralized) access token.

CCV is also already building a centralized authorization-management application. This application should, however, take into account that multiple authorization servers should be contacted.

8.3 By value or by reference tokens?

There are two distinctive ways in which tokens can be passed: by value and by reference.

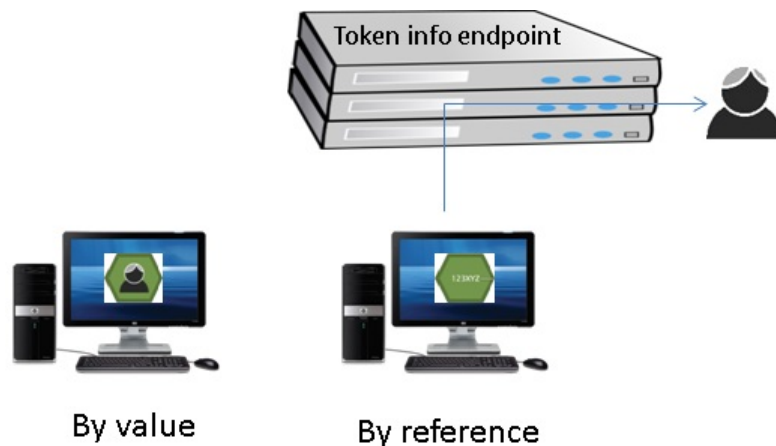


Figure 26: By value/reference tokens [82]

Tokens can be seen like the way programming languages handle data identified by variables. At run-time, the programming language will either copy the value of the variable onto the stack as it invokes the function being called (by value), or a pointer is created to the location where the data is stored (by reference). In a similar way, tokens will either contain all of the data of the identity, or they will be a reference to that data. (Figure 26) [82]. Both types of tokens have their pros and cons:

1. When using by-value tokens, the service can get all required access data from the access token. This means that, unlike by-reference tokens, no communication with a so-called "Token info endpoint" is required. A Token-info endpoint is a service that translates a reference token into a by-value token. This means that less communication is required, which makes this solution faster.
2. If a by-value token is stolen, all personal data is owned by the attacker. When a by-reference token is stolen, the attacker knows only the location where the information is stored, which will expire after some time. On the other hand, if the reference location referred to in the reference token is edited by someone with malicious intentions, this could lead to security issues. To prevent reference locations from being changed in the tokens, they should be signed.
3. When reference tokens are used for single sign off, the SSO server will send a message to the Token info endpoint saying, "user x has logged out". From this moment, the server stops translating all references and prevents further access to all services. When

a by-value token is used, all access tokens should be deleted or revoked. This process has a higher error risk when multiple access tokens are used, since all access tokens should be invalidated. There is also the risk that a single token is not removed, such that user X can still access the service.

4. One advantage of reference tokens is that they are smaller by virtue of containing only an identifier. This is an advantage for mobile devices, because less storage is required (remember that we are probably not the only cookie stored) and the token can be downloaded faster. On the other side, value tokens do not have to be translated into values, they can be used directly. Depending on the network speed, by-value tokens are a little faster, because reference tokens require translation to values and value tokens do not.

Based on these pros and cons, it is better to use by-reference tokens. The main reason for this is that they are more secure. This is important, because they provide access to sensitive information. The fact that these tokens are somewhat slower is of less concern. CCV has access to high-performance servers, which makes the difference in speed between the token types negligible.

Refresh tokens

The tokens can have various goals. the Code Authorization Flow defines the ID token, access token and refresh tokens. The refresh token principle is like a box which is locked by a different lock every 5 minutes. To retain access to this box, new keys are needed. The refresh keys give us access to a locker that contains the new key to the box and a new refresh key. Refresh tokens thus make it possible to use short-lived ID tokens, which will be refreshed after the old token expires. When a service goes unused for a predefined amount of time, the user is logged out and an attacker has less chance to take over the session.

To detect a stolen refresh token, token rotation should be used. This means that a new refresh token is issued to the client after every ID token-refresh response. The authentication server should store the previous refresh token and mark it as invalid. When an invalid refresh token is received, it is likely that someone or something with malicious intentions is trying to gain or already has access to the account. The server can detect this and require re-authentication before new tokens are provided.

The Implicit Flow does not support refresh tokens, so they must have a long expiry time or the users of the system will be redirected to the login page every time the token expires, which is not good for the user experience. This long expiry time is not a big issue if all users always logout when they are ready, but it is not likely that this will happen. Failure to log out leaves sessions vulnerable to attack. For read-only solutions that return no sensitive information (for example Google maps), this is no problem. A well-known organization that uses this approach is Foursquare [23]; they even say that their tokens do not expire at all. However, the information provided by Foursquare is limited; the only potentially sensitive information is the list of locations the user has checked into. By not expiring the token, however, it is easier for them to track their users and gain personal information. So for privacy reasons, it is recommended to log out of your Foursquare and Google account.

When long lived access tokens are used then we must store the token along with a flag defining if the token is still valid or revoked. This means that every time a user wants to access the system a request has to be sent to the IdP to check if the token is still valid.

To revoke access in a token-based system requires that, after logging out, the refresh token is marked as invalid. This, however, means that the access token is still valid for a short period of time after logging out. To solve this problem the access token should also be marked as invalid.

A better solution is to revoke the ID token, as this requires only that one token is checked and updated after logging out. The downside of this procedure is that the service should no longer check to see if the access token proves rights to access the service; it should also check to see whether or not the ID token is revoked.

Conclusion

Because the data in the tokens needs to be secure, CCV should use reference tokens and a Token info endpoint that translates all these reference tokens to value tokens and sends them to the services [7]. This has the advantage that value tokens live only in a trusted environment?since the sources are also part of CCV trusted area?because all sources that should be linked are also owned by CCV. This also solves the issue of all sources having to check to see if the refresh tokens or ID tokens are still valid. With this solution, we have to build this check once. The Token info endpoint should only send an access token to the source if the user is still logged in; otherwise it should send a not-authenticated/authorized message back to the client. The use of refresh tokens is no longer needed if the access token is kept in the Token info endpoint. Only if an attacker can get into this endpoint is it possible to steal a value token.

8.4 Possible flows

Based on the conclusion of the previous section, a couple of flows can be made. During flow 1, the client receives an ID token and an access token. During flow 2, the client receives only an ID token. However, knowledge of the request and the appropriate access token is required at the Token info endpoint.

8.4.1 Flow 1

- 1 The client contacts the identity provider and authenticates using the user's login credentials
- 2 The IdP checks the credentials and creates an ID token. This token is sent back to the Token info endpoint and the client.
- 3 The Token info endpoint stores the ID token.
- 4 The client sends a request for a service containing his ID token.
- 5 The Token info endpoint receives the message, sees that the client does not have an access token and sends the ID token to the authorization server.
- 6 The authorization server looks for the access rights belonging to the identity in the ID token and returns an access token.

- 7 The Token info endpoint stores the values of the access token, creates a reference to these values, creates a reference token and returns this token to the client.
- 8 The client repeats the request of a service but this time sends his access token.
- 9 The Token info endpoint reads the identifier out of the access token and looks up the corresponding values.
- 10 The corresponding values are gathered and stored in a value token.
- 11 The access token is sent to the requested service.
- 12 The service returns data based on the rights in the access token.
- 13 The result is returned to the client.

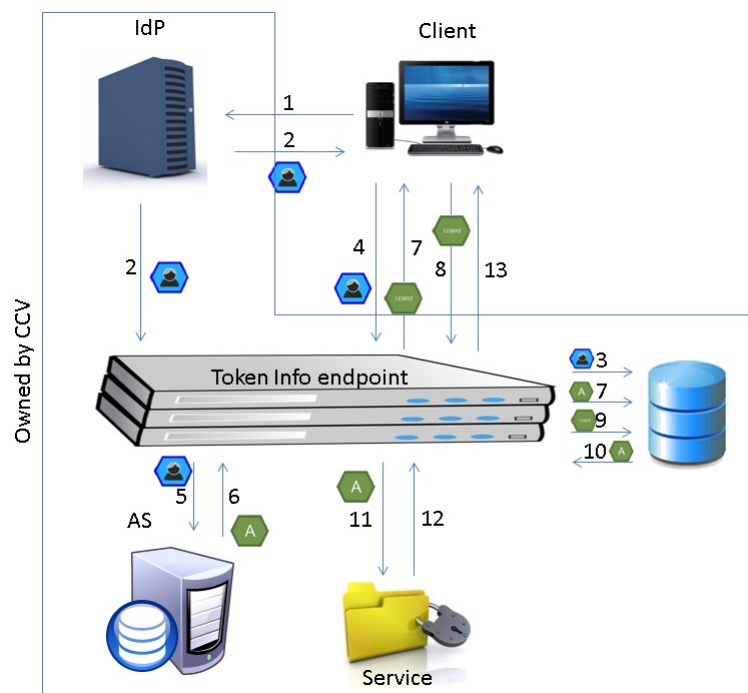


Figure 27: possible flow 1

Note that steps 8, 9 and 10 can be skipped if the Token info endpoint automatically tries to access the service after step 6.

8.4.2 Flow 2

Another solution is to send only the ID token to the client such that the Token info endpoint changes the ID token into the authorization token. The disadvantage of this solution is that the Token info endpoint must know which token should be used for which service when a multiple access token solution is used.

- 7 The Token info endpoint stores the access token and links this tokenit to the ID token and a source address or vertical.
- 8 The access token is sent to the service.

9 The service returns data based on the rights in the access token.

10 The result is returned to the client.

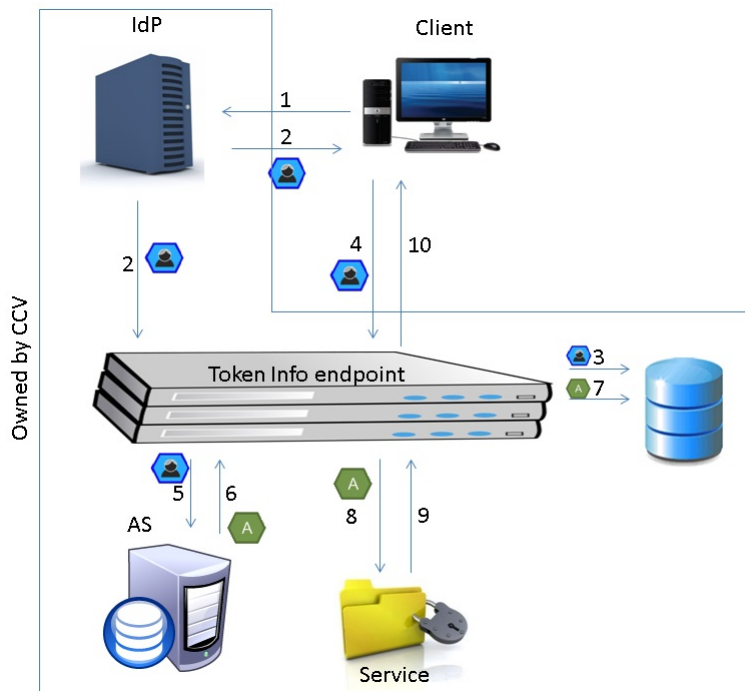


Figure 28: possible flow 2

8.4.3 Flows after authentication & authorization

After authentication and authorization, the process becomes very simple and contains only a small number of steps.

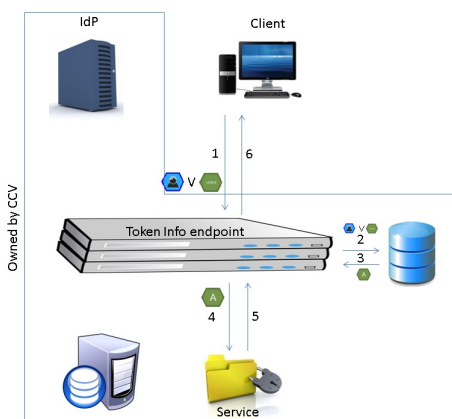


Figure 29: Flow after authorization

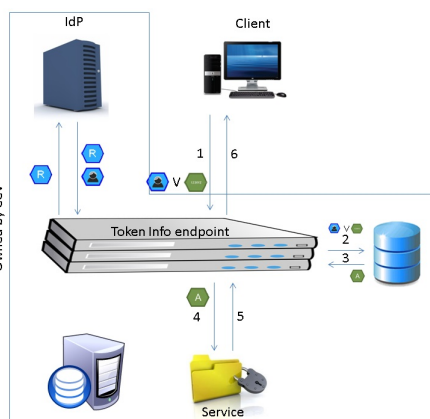


Figure 30: Protocol with refresh tokens

1 The client sends his access token (flow 1) or ID token (flow 2) to the Token info endpoint.

2 The Token info endpoint looks into the database for the access rights that are linked to the access token or the ID token.

- 3 It then generates a value token based on the reference token (flow 1) or gets the value token based on the ID token (flow 2).
- 4 The access token is sent to the service.
- 5 The service returns data based on the rights in the access token.
- 6 The result is returned to the client.

If refresh tokens are used, they are sent together with the ID token from the IdP to the Token info endpoint. When the ID token is nearly expired, the Token info endpoint sends the refresh token to the IdP which in turn will return a new ID token and a refresh token (Figure 30). To add extra security, the old ID token can be sent to the IdP. In that case, when a refresh token is stolen, the attacker still cannot request a new token.

8.5 Two-factor authentication

Independent of the protocol that is chosen, CCV requires that the final solution should implement two-factor authentication. They want to use a stepped approach. The idea is that the first authentication factor is used to log into the system and see most information. When a user wants to access a certain page with confidential information, the second authentication factor is required. The second factor can, for example, be an emailed code or an SMS. This is, however, not a common way to implement two-factor authentication; most solutions require both authentication factors at once. The solution CCV wants is like the systems used by banks. Only information on your bank account is provided the first time you authenticate. When you want to make a change or send some money, for example, then an extra authentication is required that typically uses some kind of code as shown in 31.

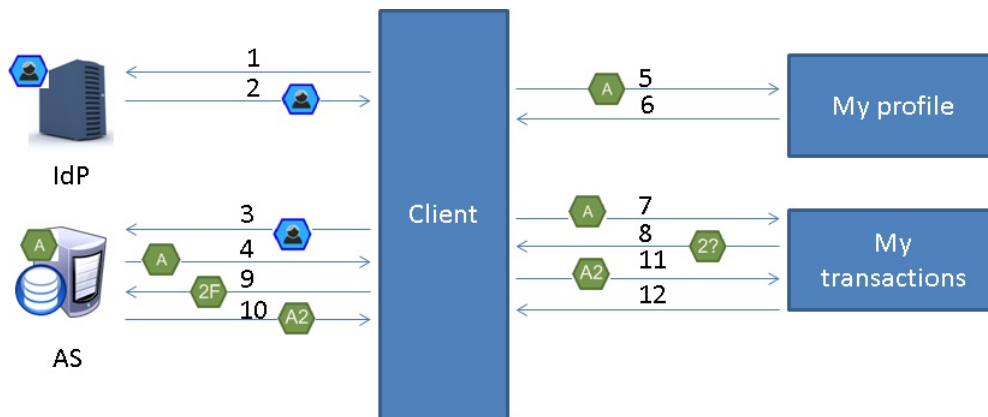


Figure 31: Two-factor authentication

When choosing for an existing SSO solution that implements OpenID Connect, it is likely that the two-factor authentication system should be built by CCV. Both ForgeRock and WSO2 offer two-factor authentication. However, this is a flat solution that uses both authentication factors at the same time solution; it is not a not a stepped one. This means that the existing SSO solutions should be edited. This is not a preferred solution, as these fixes might break when the solution is updated. When building an implementation, this issue does not occur; in that case, however, all functionality has to be built which the existing implementations already have. This is, however, a one-time process and gives CCV the option to create an application into its own corporate identity. There are also

some building blocks available (like passport) that will increase the building speed of the application. Finding out how an existing tool works and configuring it can also take a lot of time if it is done well.

8.6 Role based access control

We already discussed which techniques CCV should use to communicate access rights, we did not however discuss which kind of access control approach we want to use. Access to a system can be granted in various ways; one can use a role-based solution or a solution that is based on policies.

Role-based authorisation is a technique that is often used because it makes the administration of user rights easier. The administrator adds a role to the user, and a user can only access functionality that is linked to his role. However, it is a bad idea to define in code that access to functionality is granted to certain roles. If an additional role (e.g., "super admin") is created that should also have access to this source, then the role should be added to all locations inside the code to grant access to users with this role. To solve this issue, one should not check for a role, but instead should check to see if a certain policy is set. These policies are definitions of access rights to certain parts of a system. It is recommended that a policy be defined for all parts of the systems that require authentication. To use this solution, a set of policies should be communicated to the code instead of a role. This requires, however, that the system administrator links policies to a role. When a user logs in based on his role, the appropriate set of policies is retrieved and returned.

It is also possible to define policies only and link all of these policies to a user. This system will work for applications that require only a couple of policies. For a big system, this requires a lot of work, so a role-based solution should be used instead.

8.7 Migration process

At this moment, many services already have their own user and permission databases. To reduce the work involved in making the service communicate with the new SSO solution, it would be nice if these services were to stay intact. This can, for example, be accomplished by using an extra translation layer that translates SSO IDs to local IDs.

Another issue that we will face during the implementation phase is that some customers already have an account at multiple services. When a user logs into a service for the first time with an SSO ID, then in most cases a new account is created for that user. Personal data and settings are linked to the SSO and stored in the database of the user. This technique works if the user is new to the system; but in our case, services already exist and customers may already have an account at the service. If many accounts should be linked, then a tool or extension might be written that automatically links the users of the service to an SSO user or gives a hint to the user that these accounts can be linked. This can, for example, be executed by offering a "link existing account" button in the service. Then a script can be built that links the service account to the SSO account.

Building this feature will, however, cost a lot of time and is not worth the effort if the number of users that need to be linked is small especially since it is a one-time process. Another approach is to link the service user accounts to the SSO account by using stored identifying data in the services such as a name, address and mobile-phone number combination. This does not work, however, if this information is not available or is different from

an account that is, for example, still registered with an old mobile number. Then it still requires some user input to link the two accounts.

8.8 Signing and encrypting

A common solution is to protect a message by signing and encrypting. This approach has pitfalls, however. A sign-and-encrypt solution has a subtly different semantic than a symmetric-key-encryption solution. For the sender, the sign-and-encrypt solution guarantees the same properties as symmetric-key cryptography [17]. The sender knows for sure that the recipient knows who wrote the message and that the message can only be decrypted by the recipient. The difference between the sign-and-encrypt and symmetric-key cryptography is at the recipient's site. The recipient of a symmetric-key message knows who sent the message, but a sign-and-encrypt recipient knows only who wrote the message and has no assurance about who encrypted it. C could, for example, use a signed message from A and encrypt and send it to B. This makes sign-and-encrypt vulnerable to surreptitious forwarding.

Definition 17. Surreptitious forwarding is an attack in which a message, which is signed and encrypted by A, is sent to B. B decrypts the message, unsigns (encrypts) the message with the public key of C and then sends the message to C. C now thinks that A has sent him a message and does not know that it was originally meant for B [17].

An example of surreptitious forwarding attack on a confidential sales strategy, that gets exposed:

$$\begin{aligned} A &\rightarrow B\{\{\text{"sales strategy"}\}^a\}^B \\ B &\rightarrow C\{\{\text{"sales strategy"}\}^a\}^C \end{aligned}$$

If C is a competitor, then B can use this trick to safely sell the new sales strategy. This can be of great value. If the firm finds out that the sales strategy is leaked to competitor C, then using this trick A will be blamed for B's exposure.

The opposite order first encrypt and then sign the message is no better. The signature does not in this case prove that the sender was aware of the context of the plain-text. It is possible that the user blindly signed a message. Also, authorship of a message can be claimed, as in the following example:

$$\begin{aligned} A &\rightarrow B\{\{\text{"new invention"}\}^B\}^a \\ B &\rightarrow C\{\{\text{"new invention"}\}^B\}^c \end{aligned}$$

To summarize, sign-and-encrypt and encrypt-and-sign each have their own issues. There are, however, solutions that make both solutions secure. Davis [17] has proposed five different solutions.

1. Sign the recipient's name into the plaintext.
2. Encrypt the sender's name into the plaintext.
3. Incorporate both names.
4. Sign the signed and encrypted message again.
5. Encrypt the signed ciphertext again.

In each of these solutions, the signing layer and the encryption layer bind the sender and the receiver of the message. All of these solutions prove that A authored both the plaintext and the ciphertext. It is required not only that the author provides one of these five solutions; the receiver should also verify the proofs. Without proof that the signer and encryptor are the same person, the receiver should notice that the message's authenticity is suspect.

Since this study does not aim at secure communication between users but at secure communication between machines, the names mentioned above can be seen as IP or MAC addresses. When we decide to sign the system's name into a message, we change the message from,

$$A \rightarrow B\{\{ "message" \}^a \}^B$$

to

$$A \rightarrow B\{\{ IPaddressB, "message" \}^a \}^B$$

We can also encrypt the sender IP into the message

$$A \rightarrow B\{\{ IPaddressA, "message" \}^B, \# "message" \}^a$$

Or we define the transition from IP to IP instead of a single IP address

$$A \rightarrow B\{\{ IPaddressAtoIPaddressB, "message" \}^a \}^B$$

$$A \rightarrow B\{\{ IPaddressAtoIPaddressB, "message" \}^B, \# "message" \}^a$$

A solution that does not include the IP address is the sign, encrypt, sign solution.

$$A \rightarrow B\{\{ "message" \}^a \}^B$$

to

$$A \rightarrow B\{\{\{ "message" \}^a \}^B \}^a$$

The inner signature proves that A wrote the plaintext, the encryption proves that only B can see the plaintext, and the outer signature proves that A has encrypted the message and that no surreptitious forwarding has occurred. The same security can be achieved by using the encrypt-sign-and-encrypt approach. In this case, the first encryption proves that only B can see the plaintext, the signature proves that Alice wrote the plaintext, and the second encryption proves that only B can see that Alice wrote the plaintext and ciphertext.

The extra signing or the extra encrypting solutions have the disadvantage of requiring an extra sign or encrypt operation. These solutions therefore cost more computational power than the IP-address solution. On the other hand, when two addresses change, the extra signing or the extra encrypting solutions still work, whereas the IP addresses solution will fail. Since both solutions offer the same level of security, we advise that the IP-address approach be used [17].

8.9 Verticals

In the short term, CCV plans to implement so-called "verticals". This is a group of applications which use the same resources: for example, transactions, merchants or terminals. The goal is that these verticals will be independent of the applications and sources in other verticals. All that is required to make the applications work is available in the vertical. This means that, in an ideal solution, each vertical has its own authorization server. At this moment, all authorization data is stored in a single database. So when we look at Figure 32, we can see the current approach on the left and the ideal solution on the right.

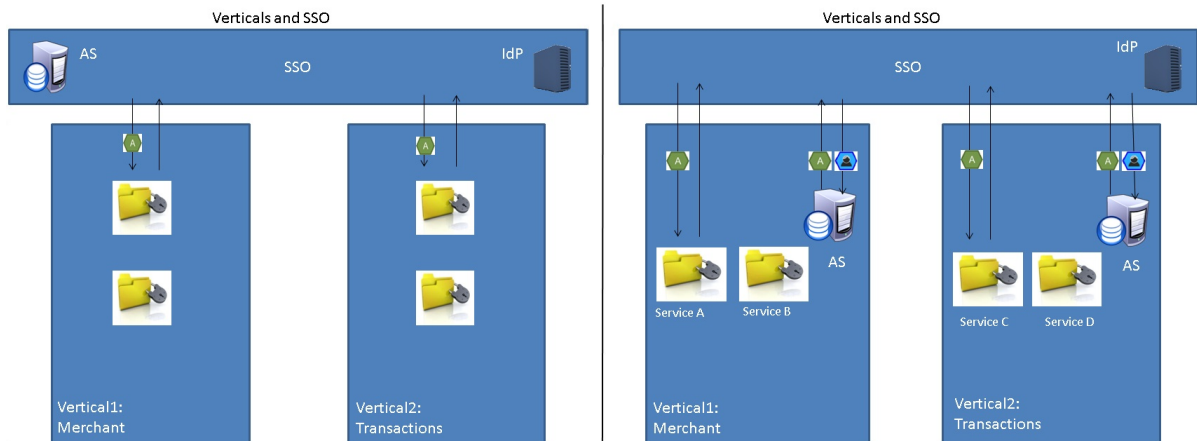


Figure 32: Verticals centralized and decentralized authorization

By creating these verticals, the authorization has become semi-decentralized. We already decided that decentralisation is a preferred solution for CCV based on the comparison of centralized and decentralized authorization in Section 8.2. So in short centralize the authentication and decentralize the authorization. So we still have to log in once and retrieve our ID token; but for every vertical an access token should be obtained.

It is preferred that the structure of the response be the same for all the verticals, as this makes it easier to link the verticals to the SSO. So, for example, a JSON object that contains a list (e.g., of name of access right, type of the access right and if the access right is granted) may contain only certain values. The last case may be interesting when access is being provided to transaction data for multiple store locations, for example. In this scenario, it may be interesting to see who can access information about which location (e.g., user X has rights to see transaction data of location Arnhem and Amsterdam, and user Y may only see transaction data of location Arnhem). The disadvantage of this solution is that, if an extra field is required (for example "permission required"), then all verticals should be edited by making them return the new JSON structure.

8.10 Managing authorization

This section is independent of SSO; it discusses ways to manage authorization.

To make it easy to manage the access rights of all verticals, it would be user friendly to have one generic authorization-management application. Because all verticals have their own authorization servers, the application should contact all of them and should be able to get, set and in some cases even delete their values. To make it easy to add a new vertical to the authorization-management application, a generic response type could be used that the authorization-management application can understand without further explanation. So every vertical has its own interface that handles the get, set and delete commands and that communicates with the authorization-management application. An example of such an implementation is the following:

GetAuth: returns [*Identifier*, *Name*, *Description*, *Format*, *Value*]

SetAuth: expects [*Identifier*, *Value*]

DelAuth: expects [*Identifier*]

When all verticals implement an interface that offers these three functions, the authorization management application can build a user interface based on the data provided by GetAuth and edit the authorization data using the SetAuth and DelAuth functions. This makes fast integration in the authorization management application possible. Also if the authorization rules of the vertical change then only changes in the vertical have to be made; the authorization management application will be updated automatically.

There are also some disadvantages to this approach. More work has to be done at the vertical to create a generic response that has the same structure for all verticals. Also, a user-interface builder based on the generic response has to be built for the authorization-management application. So the effort required to set up the authorization management will be increased, which means more time has to be spent building the authorization-management application in the first place. It is probably also harder to return a generic response than a response specific for a vertical, as information about which fields exist (and their types, etc.) must also be determined and sent. It gets worse when it turns out that the generic type is insufficient for the new vertical. A vertical may, for example, require that the value of a field can have only certain values (e.g., only 1, 2 and 4 are allowed). Changing the required response format means that all verticals and the authorization-management application must be updated. To limit the damage, one can choose to write a module that extends the existing generic response. The vertical returns this new extended format, and the authorization-management application should be informed that a different kind of response is used. This can be achieved by using a pre-request to the service by the management application that returns which format is used. When a new extension of a format is made, the management application should still be updated to make it aware of its existence. This raises the question whether it is worth the effort to use a generic response.

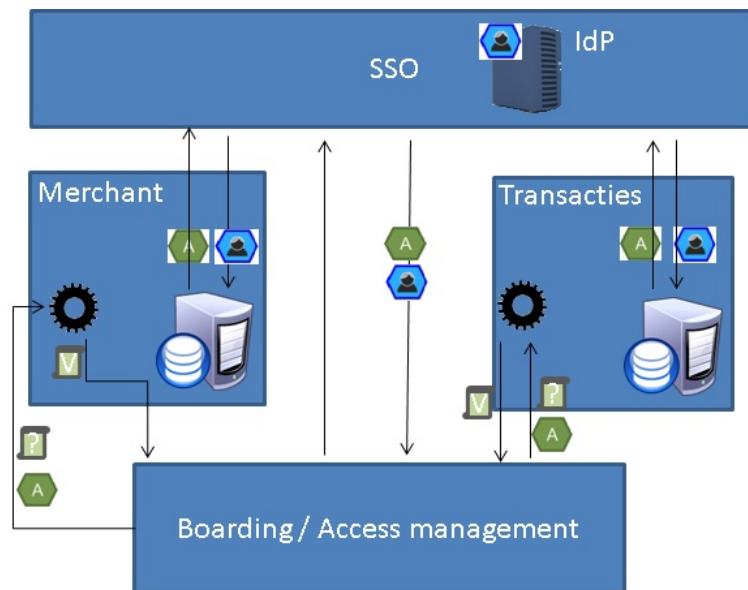


Figure 33: Verticals access management

Upon comparing the work that is required to create a generic response architecture and to manually add a new management page, we can see that, in many cases, manual building will require less work. The generic approach should be used when it is expected that many new verticals will be added in the future from which access rights should be managed or when fields in the verticals are likely to change. In these cases, you should be sure that the format that is implemented is sufficient for all verticals.

Since CCV still does not know which verticals they will have in the (near) future, and since they are also uncertain whether the structure above will be sufficient, it is likely that it will be faster and easier to build the management pages manually. If CCV still decides to use the generic approach for example, because this matches the idea of the verticals then we advise that some time be taken to determine whether the fields above are sufficient for all expected future verticals. Choosing a good structure will pay off in the end.

8.11 The current and new system

In the current system a Drupal login system is used. This system is already a kind of SSO, because after logging into the Drupal system access to various systems can be gained. Figure 34 shows a sequence diagram of the current login solution. This solution works with a single authorization server which creates a token that can be used on all subdomains.

The new solution is the OpenID Connect solution proposed in the sequence diagram in Figure 35. This solution is vertical based which means that multiple access tokens are used. Also the registration principle have to be changed. Where in the current registration process the authorization rules are set when a user is registered (Figure 36). In the new solution, access will be set when a user wants to access a service for the first time (Figure 37 and Figure 38). In some cases extra user information or rights should be added by using a boarding application. This application should be used by employees of CCV to set and confirm certain access right for the users of the CCV services.

8.12 Security management

The advice and protocols are based on the currently existing security issues and protocols. Every day new vulnerabilities are found in various systems and protocols, so it is possible that some day new critical security issues will be discovered for OpenID Connect or the chosen implementation. For this reason, we advise that someone is held responsible to check on a daily basis for new security issues for the chosen implementation. This can, for example, be done by using CVE and CCE vulnerability databases. Examples of these databases are the following:

- <https://www.exploit-db.com/>
- <http://www.cvedetails.com/>
- <https://web.nvd.nist.gov/view/vuln/search>

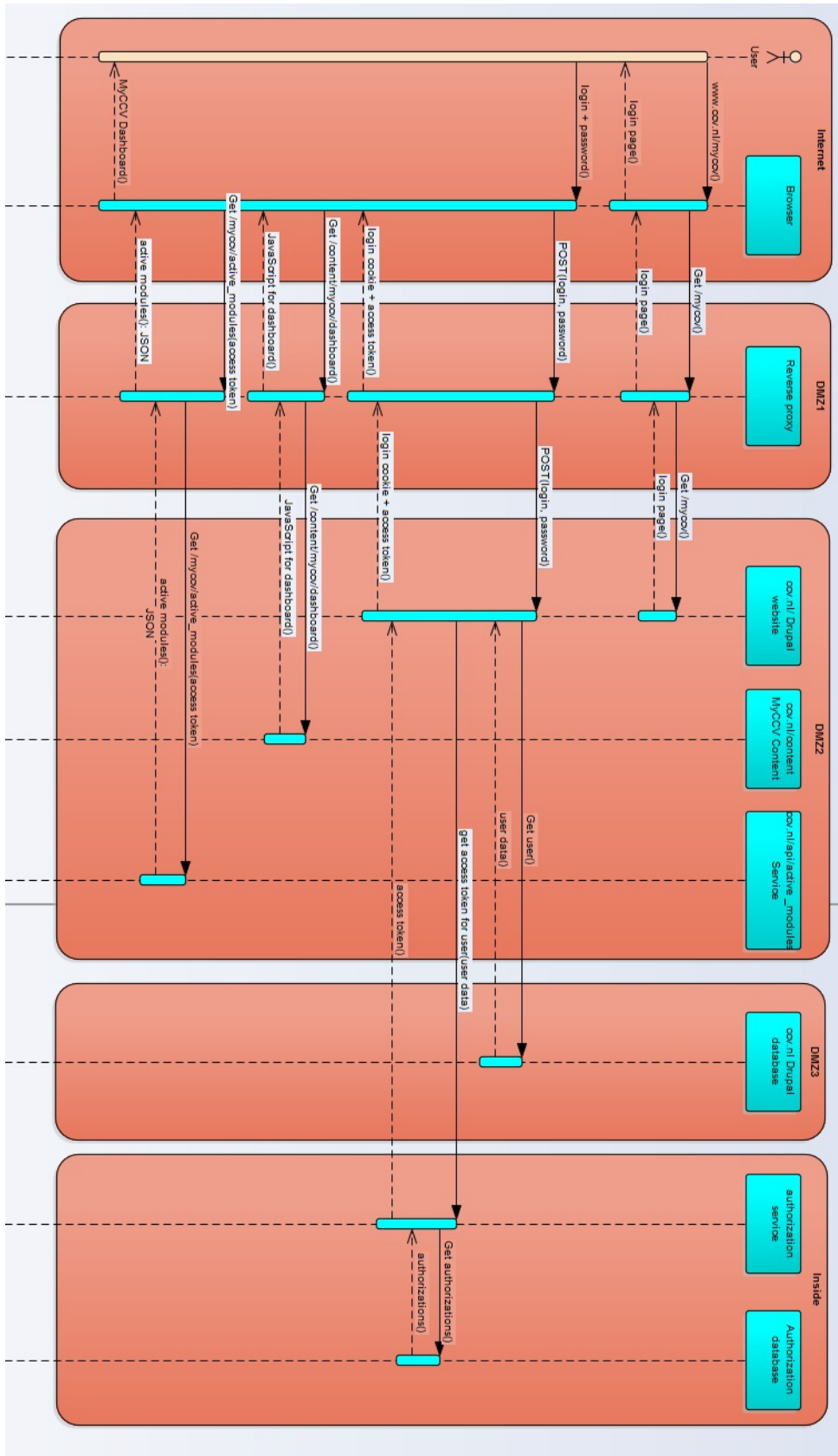


Figure 34: Current Drupal login

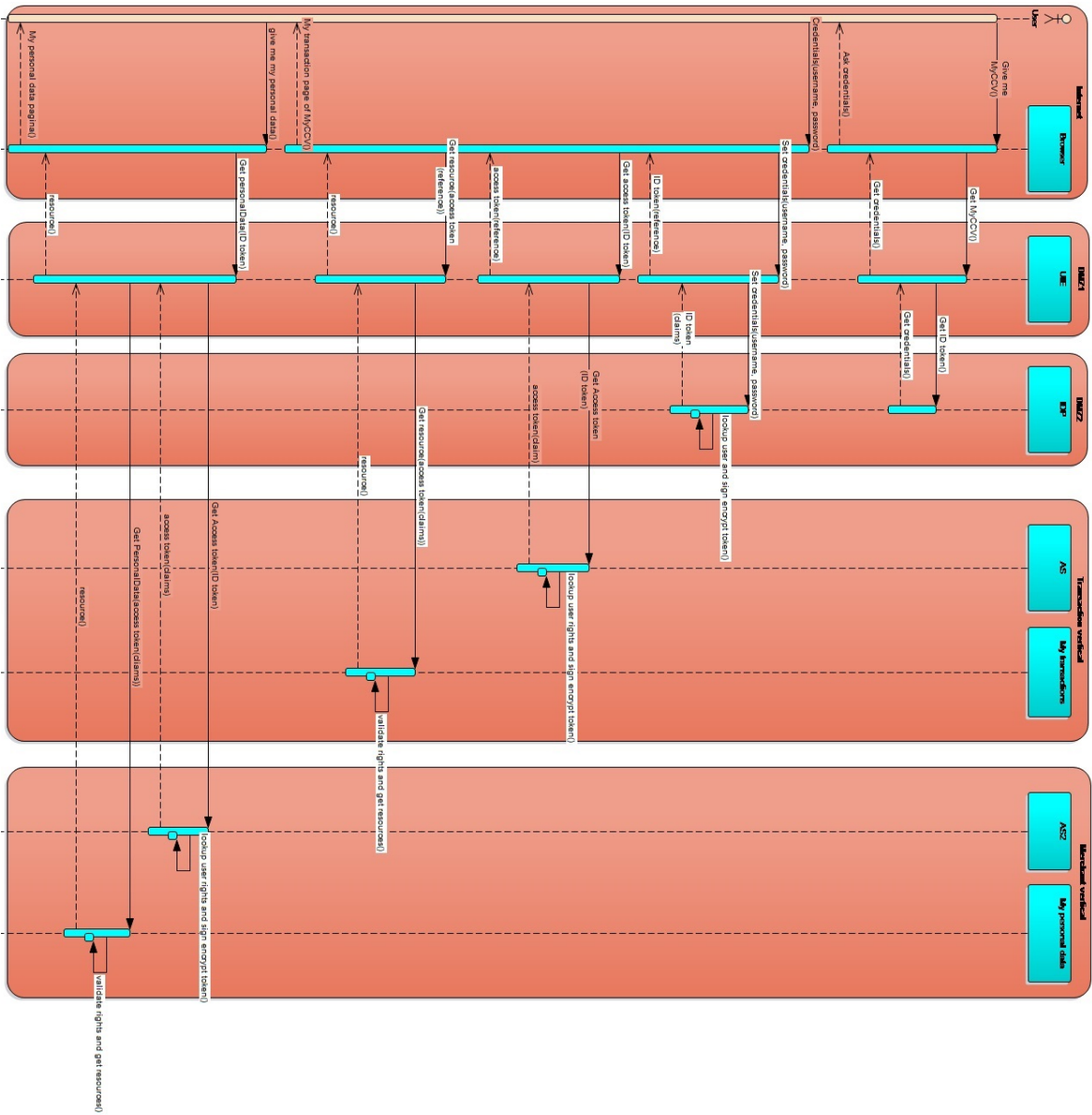


Figure 35: New OpenID Connect login

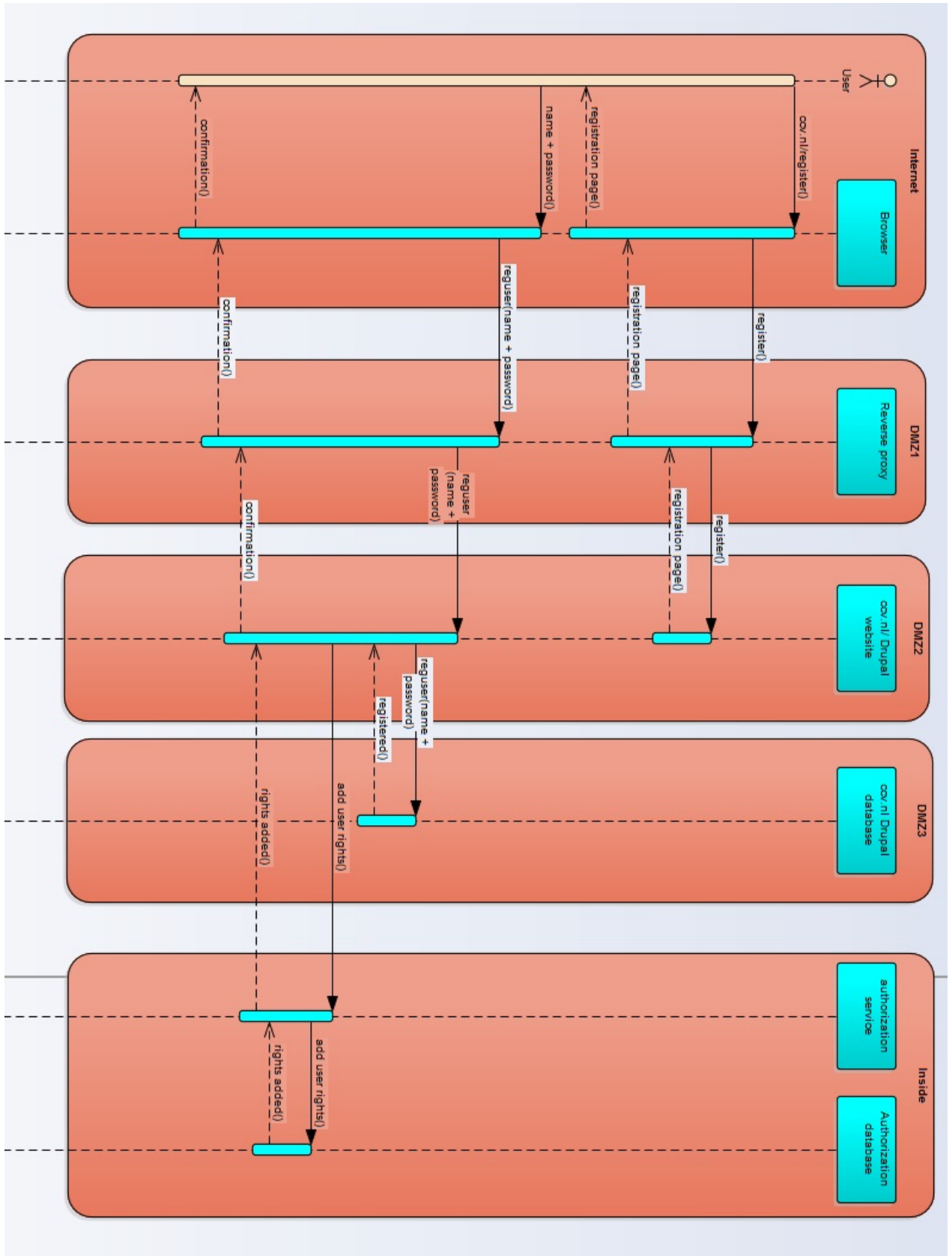


Figure 36: Current register process

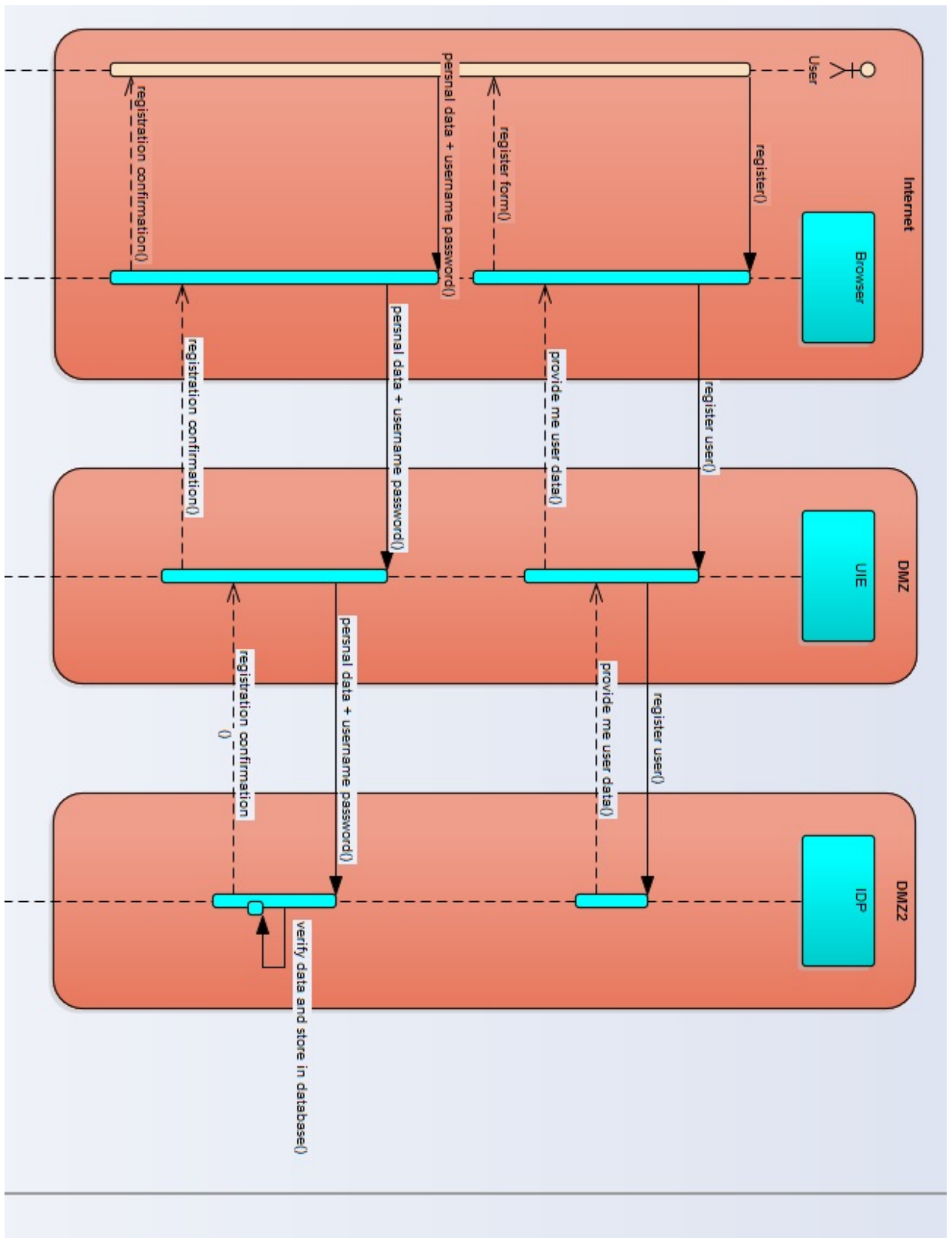


Figure 37: New register process

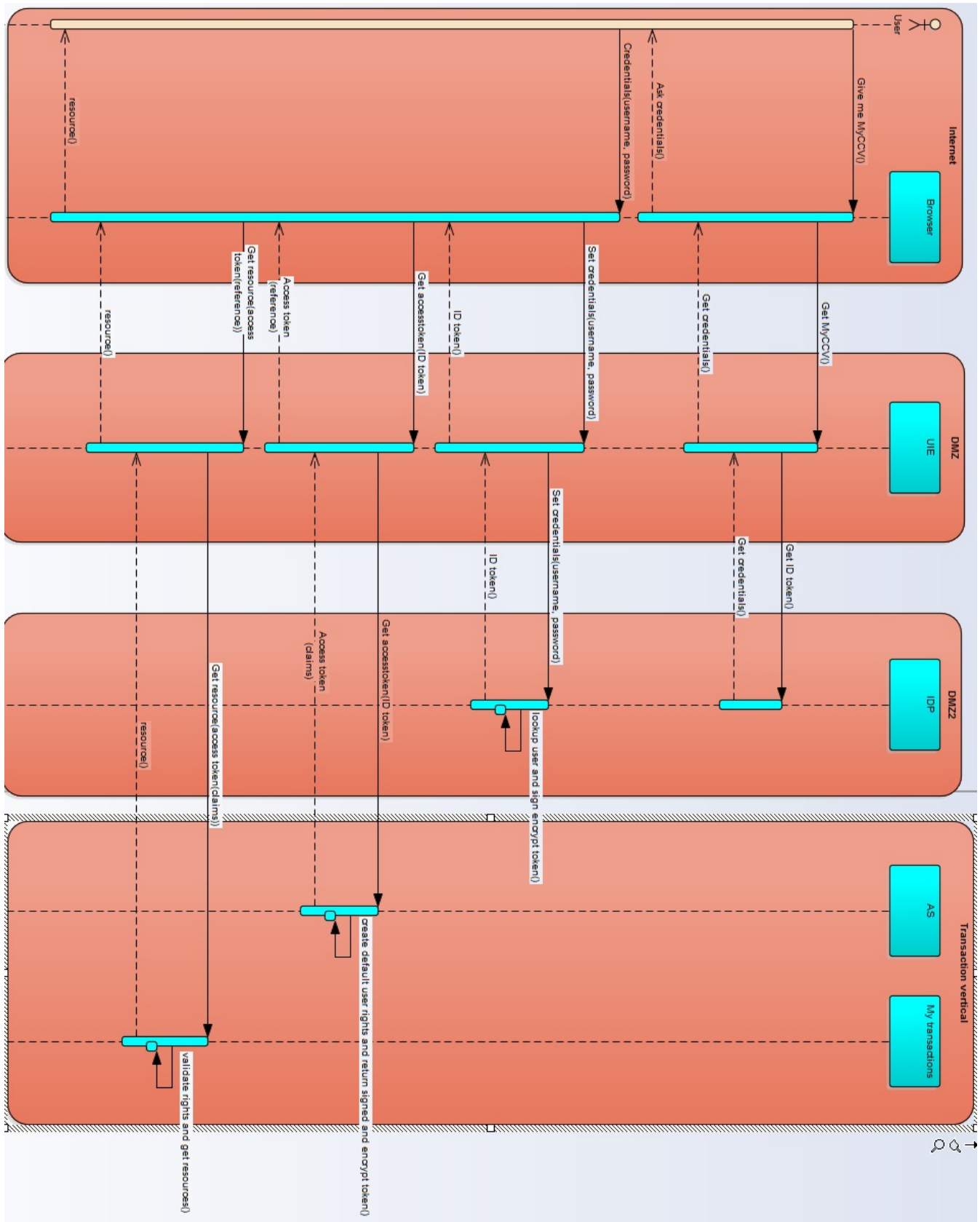


Figure 38: First time use of a service

9 Existing SSO implementations

In this chapter, we look into existing SSO implementations that can help with the implementation of the chosen solution. We discuss existing open-source solutions and the option of developing a solution. The first solution we look into is ForgeRock, which is an open-source solution that supports the OpenID Connect protocol. The current solution works with Drupal, which also has its own SSO module. The option to build the Drupal SSO solution does not meet the requirement that the solution is written in Java, NodeJS or .NET. Because Drupal is a PHP solution, and because CCV does not plan to hire PHP experts, this is not a preferred solution. The current Drupal solution is managed by an external company, which makes CCV dependent on them for changes. This is one of the reasons they want to change their current login system.

9.1 ForgeRock

ForgeRock is a project group that offers four products: OpenAM, OpenIDM, OpenIG and OpenDS. For an SSO implementation OpenAM should be used.

9.1.1 OpenAM

OpenAM is a product built in Java. It offers the following features: authentication, authorization, privilege/policy management, identity linking between systems, and SSO [84]. These are all features that we need for a working OpenID Connect solution.

OpenAM consist of two versions: a free version and a paid one. The free version contains only major releases; the paid version also contains maintenance fixes and support. According to the terms of use, a paid version should be used when the program is used in a production environment; the free version may be used only for testing and development environments. The paid version is an annual subscription for usage of the product and support. Before use, the product needs to be configured, so additional costs will be required to adapt the application to the wishes of CCV. Some coding is probably also required to build connectors to communicate with the existing database and services.

System requirements:

- **OS:** Linux, Solaris, or Windows system
- **Minimum RAM:** 1 GB
- **Free disk space:** a few hundred MB
- **Other:** a web browser and internet connection to download software.

Software requirements:

- Java Development Kit
- Apache HTTP Server
- Apache Tomcat
- OpenAM core server with OpenAM Console
- OpenAM Apache Policy Agent

9.2 WSO2

This section explains what WSO2 does and how it should be configured. Like ForgeRock WSO2 offers multiple products. For a SSO provider the WSO2 Identity Server should be used. WSO2 Identity Server offers support for OpenID Connect, OAuth2 and SAML2.0. In comparison to ForgeRock, WSO2 is much easier to install. After only 30 minutes and with no knowledge of the system a service provider was running. In this report we use WSO2 version 5.0.0. When installing this version of WSO2, we noticed that it was not compliant with the latest version of Java. We started with Java version 1.8.x but had to install Java 1.7.x to get WSO2 up and running.

System requirements:

- **OS:** Windows, Linux, Solaris, Ubuntu, Fedora, Mac OS X, Gentoo, SUSE or Debian
- **Minimum RAM:** 1 GB
- **Free disk space:** 180 MB
- **Other:** a web browser and internet connection to download software.

Software requirements:

- Java Development Kit
- Apache HTTP Server
- Apache Ant
- MySQL, MS SQL Server, Oracle, H2, DB2, Derby or PostgreSQL
- WSO Identity server

9.2.1 External database connection

Standard WSO2 uses a H2 database that is linked to it. It is not necessary to change the database type; however, WSO2 advises that another database type be used with WSO2 in a production environment. In this section, we describe how to change the database type and link the program to MySQL. The approach described is based on how it should be implemented in Windows; for other operating systems, some of the steps might be different.

The first thing that should be arranged is that MySQL is installed on the machine. MySQL offers an installer ³ that installs MySQL and additionally offers a database management tool named "MySQL Workbench".

When installing MySQL select the "MySQL for development purposes" option to get all the required packages and the database management tool. If some required packages are missing the installer will detect this. By pressing the execute button at the bottom of the page the additional packages are installed.

³<http://dev.mysql.com/downloads/windows/installer/>

After installation the WSO2 database structure and a database user having the required permissions should be created. Start by creating a database named "regdb" by executing the following command: "create database regdb;". Go to the settings of this database and set the character set to Latin1. Afterwards the database should be filled with the right tables and columns. WSO has a script ⁴ available for generating all these tables and columns. For Windows it is required that the character set of the um_description column of the um_claim table is set to UTF-8 otherwise some symbols used by WSO2 cannot be stored which will lead to issues when starting the application.

After these steps the database is complete, only a user remains to be created. Create a user for example "regadmin" and grant him access to the "regdb" database.

```
GRANT ALL ON regdb.* TO regadmin@localhost IDENTIFIED BY "regadmin";
```

We recommend to not use the "grant all" setting, but update the account to the proper settings, typically read, write and update of data in the database.

To connect WSO2 to the MySQL database an adapter is required. Since WSO is a tool written in Java the adapter "Connector/J" ⁵ can be used. The final step required is to update the WSO2 datasource settings. These settings are stored in the following folder: "WSO2Folder/repository/conf/datasources/master-datasources.xml". In this file the data source settings should be changed to the following:

```
<datasource>
  <name>WSO2 CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/MySQL</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://localhost:3306/regdb</url>
      <username>regadmin</username>
      <password>regadmin</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>80</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>
```

Do not forget to make a back-up of the original file, in case something goes wrong.

9.3 Passport and Express Server

After considering two SSO providers, we also wanted to build a client to determine whether the SSO providers are configured correctly. This client should be an SSO portal built on existing frameworks and middleware. In this section, we consider Passport, which is a middleware that is built with Node.js: a platform written on JavaScript. We chose this middleware because in-house knowledge is available for JavaScript and Node.js.

⁴WSOfolder/dbscripts/mysql.sql

⁵<http://dev.mysql.com/downloads/connector/j/>

To get Passport working, Express Server is required to set up a webserver that makes the Passport application reachable from the network. To install Express Server, an installation of Node.js and NPM (which stands for node.js package manager) is required. In this section, we explain how to get an OpenID client set up for Windows; many steps will also apply to other platforms, however. For Windows, an installer (node.msi) is available on the Node.js website which also installs the NPM.

After installing node.js and NPM, we can start by installing passport. First, create a directory in which we store our project. Next, open the Command Prompt and go to the created folder. To install Passport, Express Server must first be installed. This can be achieved by using the following command:

```
npm install express
```

To get an easy and fast server setup, install the express generator afterwards.

```
npm install express-generator
```

Now generate an Express application using the following command:

```
express myapp
```

This generates an express project with the name myapp. If the generation succeeds the console prints something like this:

```
create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.css
create : myapp/views
create : myapp/views/index.jade
create : myapp/views/layout.jade
create : myapp/views/error.jade
create : myapp/bin
create : myapp/bin/www
```

To install all required dependencies go to the newly created myapp folder and execute the following command:

```
npm install
```

To test the webserver, start the application

```
npm start
```

Now the webservice should be available on <http://localhost:3000/> (Figure 39). //

When the "Welcome to home" page shows it means that the installation was successful and it is time to start installing passport:

```
npm install passport
```



Home

Welcome to Home

Figure 39: Successful installation express

Passport offers a number of examples. Most of the examples are for OpenID Connect or OAuth2. When we use the OAuth2 approach existing IdPs such as: Google, Facebook and Paypal return an ID token (instead of an access token which you would expect from an authorization protocol). To implement a Google implementation of OpenID Connect, install the OpenID Connect Google package:

```
npm install passport-google-openidconnect
```

Currently the newest version (0.0.4) is still under construction. This version is however outdated and does not work with the current version of express server (4.12.4). Based on some online examples we managed to fix the issues. To create a working passport implementation open the myapp folder and change the content to the code in Appendix C. Also create two files named login.ejs and index.ejs in the view folder and assign the corresponding content of Appendix C to them. After changing these pages try to start the passport application using the following command:

```
node app.js
```

You will receive a message about missing dependencies. Install the following packages to retrieve a working passport application.

```
npm install express-session  
npm install method-override  
npm install ejs
```

When we start the application by using the "node app.js" command, the passport application will be started and a welcome page will be displayed. By clicking "login", a page will be shown that offers a list of identity providers. We only have Google in our example; but more can be added. Figures 40, 41, 42, 43 and 44 show a working example of a login process.



Figure 40: Passport home page

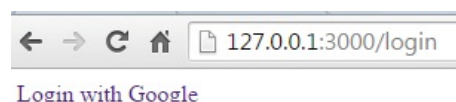


Figure 41: Passport login page

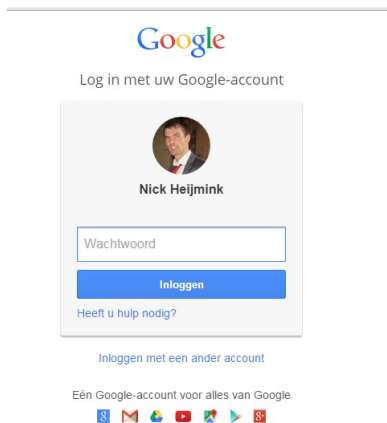


Figure 42: Authenticate to Google



Figure 43: Grant permissions



Figure 44: Successful log in

9.4 WSO2 and passport

Because WSO2 is an OpenID Connect provider, and because passport is an OpenID client, it should be possible to make these two systems communicate with each other. To achieve this goal, we used the WSO2 installation from Section 9.2. To link the passport client to the WSO2 server, the service has to be registered. When registering a service, a clientID and a clientcode have to be defined. These are used to identify the service and to check if the service is a valid service. Since WSO2 supports multiple protocols, we have to specify that the service uses OpenID Connect. For security reasons, we must also specify which return address should be used. This address should be identical to the one requested by the service. See Figures 45, 46 and 47 to get an idea of the configuration.

When WSO2 is set up, a passport client has to be build. We already have an implementation which works that uses Google and Facebook as IdP, which can be used for the basics. The first step required to make the existing solution communicate with the WSO2 server is to update the URL that is used for the authentication process. The URLs that need to be defined are the following:

1. AuthorizationURL
2. TokenURL
3. UserInfoURL
4. OidcIssuer

Application Settings	
OAuth Version	OAuth-2.0
Callback Url*	<input type="text" value="http://localhost:3000/auth/WSO/callback"/>
Allowed Grant Types	<input checked="" type="checkbox"/> Code <input checked="" type="checkbox"/> Implicit <input checked="" type="checkbox"/> Password <input checked="" type="checkbox"/> Client Credential <input checked="" type="checkbox"/> Refresh Token <input type="checkbox"/> SAML <input checked="" type="checkbox"/> IWA-NTLM
Access Token Url	https://localhost:9443/oauth2/token
Authorize Url	https://localhost:9443/oauth2/authorize
<input type="button" value="Update"/> <input type="button" value="Cancel"/>	

Figure 45: WSO2 redirect configuration

Basic Information					
Service Provider Name:*	<input type="text" value="passport test"/> <small>ⓘ A unique name for the service provider</small>				
Description:	<input type="text" value="passport application"/> <small>ⓘ A meaningful description about the service provider</small>				
<input type="checkbox"/> SaaS Application					
<input checked="" type="checkbox"/> Claim Configuration					
<input checked="" type="checkbox"/> Role/Permission Configuration					
<input checked="" type="checkbox"/> Inbound Authentication Configuration					
<input checked="" type="checkbox"/> SAML2 Web SSO Configuration					
<input checked="" type="checkbox"/> OAuth/OpenID Connect Configuration					
OAuth Client Key	<table border="1"> <thead> <tr> <th>OAuth Client Secret</th> <th>Actions</th> </tr> </thead> <tbody> <tr> <td>nVgHvCizknYI32qtlMOCQ4c0D7sa</td> <td> <input type="button" value="Show"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/> </td> </tr> </tbody> </table>	OAuth Client Secret	Actions	nVgHvCizknYI32qtlMOCQ4c0D7sa	<input type="button" value="Show"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/>
OAuth Client Secret	Actions				
nVgHvCizknYI32qtlMOCQ4c0D7sa	<input type="button" value="Show"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/>				

Figure 46: WSO2 service configuration

The values for these URLs can be found in the WSO2 server as shown in Figure 47. They need to be set in the strategy.js file. For a localhost implementation of WSO2 server, this results in the following:

```
options.authorizationURL = options.authorizationURL || 'https://localhost:9443/oauth2/authorize/';
options.tokenURL = options.tokenURL || 'https://localhost:9443/oauth2/token/';
options.userInfoURL = options.userInfoURL || 'https://localhost:9443/oauth2/userinfo';
options.oidcIssuer = options.oidcIssuer || 'https://localhost:9443/oauth2endpoints/token';
```

The app.js should also be updated to make it operate with WSO2. This is similar to the code of Facebook and Google and is appended in Appendix D.

When trying to authenticate using these settings, you will receive a "failed to obtain access token" message. After some research, it turned out that this was caused by the fact that WSO2 does not have a valid certificate. To fix this, a certificate must be purchased (for test purposes, it can also be generated using OpenSSL). When the certificate is added, you have a successful authentication solution that uses WSO2 and passport.

Identity Provider Name:* Enter a unique name for this identity provider

Display Name: Specify the identity provider's display name

Description: A meaningful description about the identity provider

Federation Hub Identity Provider: Check if this points to a federation hub identity provider

Home Realm Identifier: Enter the home realm identifier for this identity provider

Identity Provider Public Certificate: Geen bestand gekozen
Upload identity provider's public certificate in PEM format

Alias: If the resident identity provider is known by an alias at the federated identity provider specify it

Claim Configuration
 Role Configuration
 Federated Authenticators

OpenID Configuration
 SAML2 Web SSO Configuration
 OAuth2/OpenID Connect Configuration

Enable OAuth2/OpenIDConnect Specifies if OAuth2/OpenID Connect is enabled for this identity provider

Default Specifies if OpenID Connect is the default

Authorization Endpoint URL:* Enter OAuth2/OpenID Connect authorization endpoint URL value

Token Endpoint URL:* Enter OAuth2/OpenID Connect token endpoint URL value

Client Id:* Enter OAuth2/OpenID Connect client identifier value

Client Secret:* Enter OAuth2/OpenID Connect client secret value

Figure 47: WSO2 IdP configuration

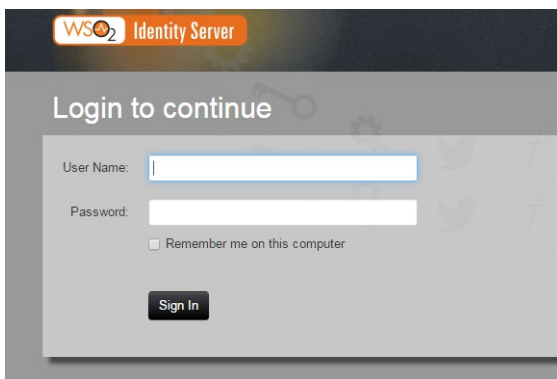


Figure 48: Authenticate to WSO2

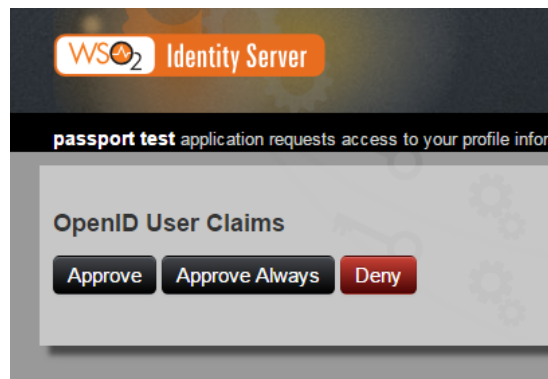


Figure 49: WSO2 grant permissions

9.5 Conclusion of the implementations

We tried only two SSO providers and one client. Since many more solutions are available, we cannot be sure which is best. However, we do know that it is much easier to set up a WSO2 solution than a ForgeRock. Installing these solutions is easy, but configuring them to work as an SSO provider is not. WSO2 was the easiest SSO provider to configure; finding out how to configure it took about a day. This is much faster than building a solution from scratch. The existing solutions do not support stepped two-factor authentication, however. This means that the existing solutions should be extended. It is possible to write modules for WSO2 and ForgeRock, but building a module for an existing application is likely to be harder than building the product for a solution from scratch. Building an module requires that you determine how modules can be added, which code styles and rules should be used, how it can communicate with the existing solutions and how to build the functionality itself. Also, when the solution is updated, the module can break, which may result in users being unable to access certain services.

10 Future work

Although we have compared various SSO solutions, we did not have enough time to compare all of them. Protocols such as Kerberos, Sesame and Kryptoknight were skipped because they are old and appeared to be barely used. During this study, we realized that this is not the case for Kerberos. It would be nice if someone would compare the functionality, security, community (etc.) of all these solutions. We already looked into a big selection of protocols, but a complete comparison of all available solutions would be useful.

OZ, one of the protocols we analysed, is still under construction [29], but it may become of interest in the future. At this moment in time the protocol is still in an early stage, but a future analysis when the protocol is completed could be useful.

This thesis has focused mainly on functionality and security. But what about privacy? How privacy friendly are these SSO protocols? And what information do the existing SSO providers such as Google offer applications that authenticate their users with their IdP? Is it possible to use the IRMA card ⁶ in combination with SSO, for example?

Privacy of the protocols is not an issue for CCV, because CCV uses only itself as its SSO provider and because the provider can be contacted only by the services of CCV. Privacy can, however, be an issue for other companies that offer their SSO provider capabilities to other services.

Table 1 of the functional comparison of the different SSO protocols in Section 5.5 contains some fields with question marks. These means that we do not know the value. It would be nice if someone would determine these missing values.

This thesis focussed on SSO protocols and how they should be implemented. We looked at two implementations: WSO2 and ForgeRock. It would be nice, however, if further research was done into other SSO implementations that compared them to building an own implementation. Knowing the existing implementations, the ease of using them and the functionalities they offer will make it easier for companies to decide which implementation they should use and in which cases they should decide to build their own implementation.

⁶Information about the IRMA card can be found at: <https://www.irmacard.org/>

11 Conclusion

This study started by determining the definition of SSO and what it is used for. We quickly discovered that there is not one definition of SSO, but at least two. One of the definitions is, however, what we call Federated Identity Management. By SSO we mean a solution that allows users to log in on a single login page and afterwards have access to multiple services with the same ID token.

If we follow this definition strictly, then OAuth2 is not an SSO solution. However, many call it an SSO solution because authorization can be used for authentication, as authorization requires authentication. During the passport implementation, we discovered that Google and Facebook can be used as identity providers by using OAuth2. They do not return an access token but an ID token (see Section 3.1.5 for a discussion of the differences).

During the study we also discovered that two versions of SSO exist: Enterprise SSO and Web SSO. Whereas Enterprise SSO focusses on SSO in internal networks and systems of an Enterprise, Web SSO is focused on connecting web services using SSO. In this study, we focussed on Web SSO.

After we defined what we meant by SSO and determined which type of SSO we were looking for, we investigated the functionality of various SSO protocols. We compared the functionality of the protocols and concluded that only three of the seven protocols match the functional requirement for providing authentication and authorization. These three protocols achieve authentication and authorization, but they do this by using different techniques.

During the study, new requirements were added to the research. We started with four and ended up with eleven. Even the authorization and authentication requirement was added afterwards. At the beginning of this study, we were not aware that two forms of SSO exist: authentication and authorization and authentication only. During the project, CCV came up with the idea of verticals, as described in 8.9. These are just two of the seven requirements that were added, all of which influenced the project. However, at the moment of making the final decision for a protocol, the requirements were complete and they have not been changed since.

To determine which solution we should use, we looked not only at the functionality of the protocols but also at their activity and security. By activity we mean the appearance of questions posted on forums and the availability of information. The activity check taught us that most information is available for LDAP, but that OAUTH2 is the most popular protocol. OpenID Connect is, however, the protocol for which we saw a huge growth in popularity. Specific numbers on the protocols can be found in 7. The security analysis revealed that all protocols have some security risks; all of them can be solved, however, as explained in Section 6.

OpenID Connect is the newest protocol (2014), and LDAP is the oldest (1993). Because LDAP was released when the internet was still in its infancy, its focus is not on web applications but on communication between servers of an enterprise. SAML and OpenID Connect both offer authentication and authorization, and both focus on the web. This means that they use protocols and notations in their communications that are supported by the web, that they are lightweight and that they also support SSO on multiple domains. At the moment of this writing, SAML is the protocol that is used most often, but OpenID Connect is being adopted rapidly. OpenID Connect can be used for the same things as

SAML, but it uses new standards like JSON and REST. Also, identity providers that already offer SAML, such as Google, have started implementing OpenID Connect. This together with the rapid growth of the protocol make OpenID Connect the best protocol for CCV.

The OpenID Connect protocol offers multiple sub-protocols that we call "flows". So after selecting the protocol, we also had to select a flow. After some research, it turned out that there are two flows that are applicable for CCV: the Hybrid Flow is, however, the best choice, as less communication between actors is required and it is easier to implement.

Since verticals were required, it would be nice if all verticals had their own authorization server. This makes them completely independent of all other verticals and servers. This does, however, mean that the implementation should be able to handle multiple access tokens. From a security point of view, this is a good thing, because the theft of one token does not lead to an attacker having access to all services. The disadvantage is that managing the access rights becomes more complicated. Based on the cons and pros, we decided that decentralised authorization in which all verticals have their own authorization server as explained in Section 8.2 and 8.9 is the best option.

We conclude this thesis by considering two existing SSO implementations: ForgeRock and WSO2. Installing and configuring WSO2 is a much than doing so for ForgeRock. We also built an SSO client based on passport, which is a middleware build in Node.js. The examples online are, however, outdated and do not function without modification. We updated the examples by making it possible to communicate with Google as an identity provider, and we afterwards changed it into a version that could also talk to WSO2 server. Based on the comparison of these two SSO providers, we recommend WSO2; but more existing implementations should be considered before choosing to implement WSO2.

12 Glossary

This glossary is based on that offered by the OpenID standard [75], which contains most of the important definitions. These definitions are extended with definitions that are used in this thesis and are missing in OpenID glossary.

AS: Authorization Server. The authorization server issues access tokens to the client after successfully authenticating a user and gives authorization.

Authentication Request: Request towards an identity provider asking to verify the identity of the user.

Authorization Code Flow: OAuth2 or OpenID Connect flow in which an authorization code is returned from the authorization server, where all tokens are returned from the Token Endpoint.

Authorization Request: Request towards an Authorization server asking for the rights of the authenticated user.

Bearer token: Tokens without a secret or other verification mechanism.

By-reference token: Token that contains a reference to the token values.

By-value token: Token that contains the actual values, for example access rights.

Claim: Piece of information asserted about an Entity.

Client: A device owned by a user, such as a PC, tablet or smartphone.

Credential: Data presented as evidence of being a certain identity.

User: Human participant.

Entity: Something that has a separate and distinct existence and that can be identified in a context. A user is one example of an Entity.

Federation: An association containing a number of service providers and identity providers.

Hybrid Flow: OpenID Connect flow in which an authorization token is returned from the authorization server and the ID token from the Token Endpoint.

ID Token: JSON Web Token (JWT) that contains claims about the authentication event.

Identifier: Value that uniquely characterizes an entity in a specific context.

Identity: Set of attributes related to an Entity.

IdP: An Identity Provider (IdP) is responsible for the authentication process and handles the storing of user information as attributes in an ID token. When a user authenticates, the IdP creates an object that contains the information of the user, which can be used when a service provider request user attributes.

Implicit Flow: OAuth2 or OpenID Connect flow in which all tokens are returned from the Authorization Endpoint and neither the Token Endpoint nor an authorization code are used.

Issuer: Entity that issues a set of Claims.

JWT: JSON web token.

Message: Request or a response between an Relying Party and an SSO provider.

OIDC: Abbreviation of OpenID Connect

Personally Identifiable Information (PII): Information that can be used to identify the natural person to whom such information relates, or is or might be directly or indirectly linked to a natural person to whom such information relates.

Protocol: A complete solution, for example "the OAuth2 protocol".

Relying Party (RP): Client application requiring user authentication and Claims from an SSO Provider.

Request Object: JWT that contains a set of request parameters as its Claims.

Request URI: URL that references a resource containing a Request Object. The Request URI contents MUST be retrievable by the Authorization Server.

Sector Identifier: Host component of a URL used by the Relying Party's organization that is an input to the computation of pairwise Subject Identifiers for that Relying Party.

Self-Issued OpenID Provider: Personal, self-hosted OpenID Provider that issues self-signed ID Tokens.

Service: An application or a resource to which a user wants to gain access.

SSO: A solution that allows users to log in on a single login page. Afterwards, access is granted to multiple services. SSO requires an IdP and in some cases also handles authorization using an AS.

Subject Identifier: Locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client.

Token Info Endpoint: Protected Resource that, when presented with a by-reference token returns a by-value token and when presented with a by value token returns a by reference token.

User Info Endpoint: Protected Resource that, when presented with an access token by the Client, returns authorized information about the user. Communication towards the UserInfo Endpoint must be protected using https.

Validation: Process intended to establish the soundness or correctness of a construct.

Verification: Process intended to test or prove the truth or accuracy of a fact or value.

Appendix A Federated Identity Management

This section gives some additional information on Federated Identity Management, this info is of no further use for this master thesis.

Definition 18. By **Identity management** we mean a set of functions and capabilities (e.g. administration, management and maintenance, discovery, communication exchanges, correlation and binding, policy enforcement, authentication and claims) used for enabling business and security applications and assuring identity information and identity of an entity, organizations, network and service providers, network elements and objects, and virtual objects [16, 50].

Definition 19. By federation we mean: "an association containing a number of service providers and identity providers"[16]. The fact that the various providers have formed an association between themselves means that they must have a certain level of trust in each other, sufficient to be willing to exchange messages. When these messages contain the authentication and authorisation credentials of users and they are used to allow users from one system to access services in a federated system, than we speak of **Federated Identity Management** [16].

An example of Federated Identity Management is the SURF network. Many universities from all over the world are connected to this network [91]. Using this network students and employees of universities can also use services at all the other SURF connected universities. So for example a student from the Radboud university can log in to the Technical University of Eindhoven and visa versa [50, 91].

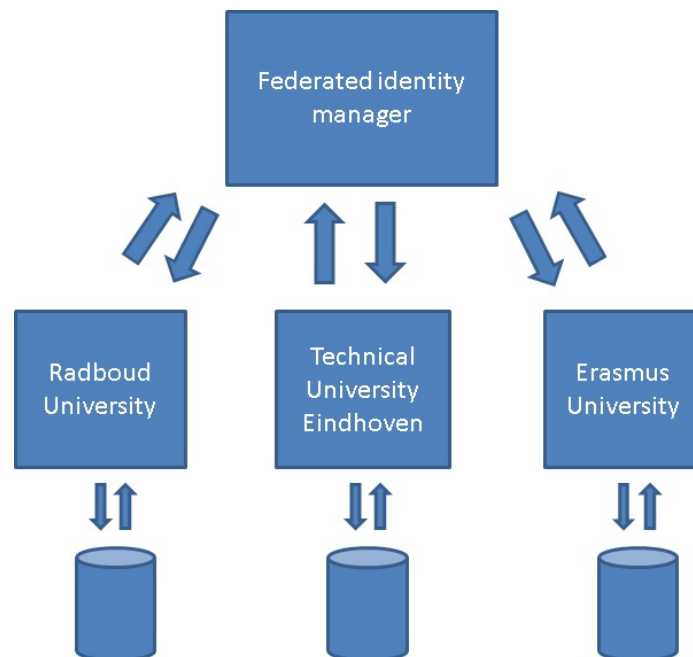


Figure 50: Federated Identity Management

Figure 50 illustrates how a federated identity system could look like. So for example a student from Radboud University visits the Erasmus University. At this university he wants to access the internet. To get access to the internet he should be authenticated, which is possible using his Radboud login credentials. The Erasmus does not recognise the login and sends the authentication request to the Federated Identity Manager. Based on information in the authentication request the Federated Identity Manager knows that

this is a login from the Radboud and redirects the authentication request to the Radboud. The Radboud checks its database for the corresponding user and sends its conclusion back to the Federated Identity Manager. The Federated Identity Manager in turn sends this conclusion to the Erasmus University. Based on this conclusion the Erasmus University will grant access to the user or show a message that the credentials are invalid.

Appendix B OAuth2 alternative flows

B.0.1 Resource Owner Flow

The Resource Owner Flow is suitable in cases where you only work with trusted clients, for example your own mobile client or a highly privileged application. This flow could also be used in cases where it is hard to implement the authorisation code in the software.

This flow looks like the basic username password login system, but instead of an access cookie or session the authentication server returns an access token. Like the Code Authorization Flow, the Resource Owner Flow supports refresh tokens. According to the OAuth2 specification the authorization server should take special care when using this flow and it should only be used when no other flow is viable [30, 10].

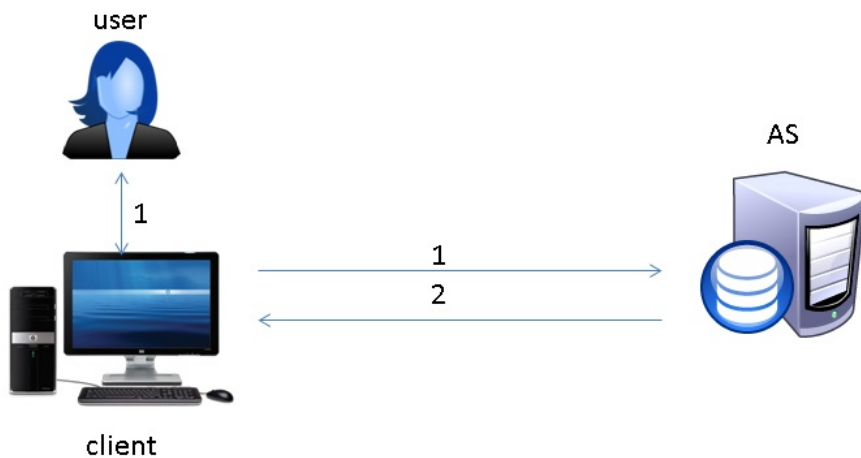


Figure 51: OAuth2 Resource Owner Flow

B.0.2 Client Credential Flow

The Client Credential Flow is even simpler than the Resource Owner Flow. In this flow the user is not part of the authentication process. Based on some information stored or generated on the client an access token is received from the authorization server.

This flow is suitable for machine-to-machine authentication, for example for a cron job that has to execute some scheduled task on the machine. It is not likely that this flow is useful for the problem we try to solve in this research [30, 10].

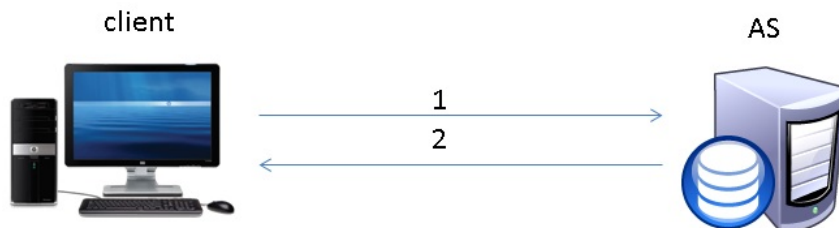


Figure 52: OAuth2 Client Credential Flow

Appendix C Code Passport application

App.js

```
var express = require('express')
, passport = require('passport')
, path = require('path')
, util = require('util')
, StrategyGoogle = require('passport-google-openidconnect').Strategy
, logger = require('morgan')
, session = require('express-session')
, bodyParser = require("body-parser")
, cookieParser = require("cookie-parser")
, methodOverride = require('method-override');

var GOOGLE_CLIENT_ID = "appcode";
var GOOGLE_CLIENT_SECRET = "secret";

// Passport session setup.
// To support persistent login sessions, Passport needs to be able to
// serialize users into and deserialize users out of the session. Typically,
// this will be as simple as storing the user ID when serializing, and finding
// the user by ID when deserializing. However, since this example does not
// have a database of user records, the complete Google profile is serialized
// and deserialized.
passport.serializeUser(function(user, done) {
done(null, user);
});

passport.deserializeUser(function(obj, done) {
done(null, obj);
});

// Use the StrategyGoogle within Passport.
// Strategies in Passport require a 'verify' function, which accept
// credentials (in this case, an accessToken, refreshToken, and Google
// profile), and invoke a callback with a user object.
passport.use(new StrategyGoogle({
clientID: GOOGLE_CLIENT_ID,
clientSecret: GOOGLE_CLIENT_SECRET,
callbackURL: "http://127.0.0.1:3000/auth/google/callback",
userInfoURL: "https://www.googleapis.com/plus/v1/people/me"
}),
function(accessToken, refreshToken, profile, done) {
// asynchronous verification, for effect...
process.nextTick(function () {

// returns a Google profile. In a typical application, you would want
```

```

// to associate the Google account with a user record in your database,
// and return that user instead.
return done(null, profile);
});
}
));

var app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(logger());
app.use(cookieParser());
app.use(bodyParser());
app.use(methodOverride());
app.use(session({ secret: 'keyboard cat' }));
// Initialize Passport! Also use passport.session() middleware, to support
// persistent login sessions (recommended).
app.use(passport.initialize());
app.use(passport.session());
app.use(express.static(__dirname + '/public'));

app.get('/', function(req, res){
  console.log(req.user);
  res.render('index', { user: req.user });
});

app.get('/account', ensureAuthenticated, function(req, res){
  res.render('account', { user: req.user });
});

app.get('/login', function(req, res){
  res.render('login', { user: req.user });
});

app.get('/auth/google',
  passport.authenticate('google-openidconnect'));

app.get('/auth/google',
  passport.authenticate('google-openidconnect',{ scope: ['https://www.googleapis.com/auth/pl
function(req, res){

});

app.get('/auth/google/callback',
  passport.authenticate('google-openidconnect', { failureRedirect: '/login' }),
  function(req, res) {
// Successful authentication, redirect home.

```



```

res.redirect('/');
});

// Logout functionality
app.get('/logout', function(req, res){
req.logout();
res.redirect('/');
});

app.listen(3000);

// Simple route middleware to ensure user is authenticated.
// Use this route middleware on any resource that needs to be protected. If
// the request is authenticated (typically via a persistent login session),
// the request will proceed. Otherwise, the user will be redirected to the
// login page.
function ensureAuthenticated(req, res, next) {
if (req.isAuthenticated()) { return next(); }
res.redirect('/login')
}

```

Index

```

<% if (!user) { %>
<h2>Welcome! Please log in.</h2> <a href="/login" >login</a>
<% } else { %>
<h2>Hello, <%= user._json.displayName %>. </h2>
<img src='<%= user._json.image.url %>' alt='profile' \>
<% } %>

```

Login

```

<a href="/auth/google">Login with Google</a>

```

Appendix D WSO2 and Passport application

App.js

```
var express = require('express')
, passport = require('passport')
, path = require('path')
, util = require('util')
, StrategyWSO = require('passport-wso-openidconnect').Strategy
, logger = require('morgan')
, session = require('express-session')
, bodyParser = require("body-parser")
, cookieParser = require("cookie-parser")
, methodOverride = require('method-override');

var GOOGLE_CLIENT_ID = "nVgHvCizknYl32qt1MOCQ4cOD7sa";
var GOOGLE_CLIENT_SECRET = "1efvGN1V8YZC0n51FLCRalbfba8a";

// Passport session setup.
// To support persistent login sessions, Passport needs to be able to
// serialize users into and deserialize users out of the session. Typically,
// this will be as simple as storing the user ID when serializing, and finding
// the user by ID when deserializing. However, since this example does not
// have a database of user records, the complete WSO profile is serialized
// and deserialized.
passport.serializeUser(function(user, done) {
done(null, user);
});

passport.deserializeUser(function(obj, done) {
done(null, obj);
});

// Use the WSOStrategy we build for Passport.
// Strategies in Passport require a 'verify' function, which accept
// credentials (in this case, an accessToken, refreshToken, and WSO
// profile), and invoke a callback with a user object.
passport.use(new StrategyWSO({
clientID: GOOGLE_CLIENT_ID,
clientSecret: GOOGLE_CLIENT_SECRET,
callbackURL: "http://localhost:3000/auth/WSO/callback"
},
function(accessToken, refreshToken, profile, done) {
process.nextTick(function () {

return done(null, profile);
});
});
}
```

```

));

var app = express();

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(logger());
app.use(cookieParser());
app.use(bodyParser());
app.use(methodOverride());
app.use(session({ secret: 'keyboard cat' }));

app.use(passport.initialize());
app.use(passport.session());
app.use(express.static(__dirname + '/public'));

app.get('/', function(req, res){
  console.log(req.user);
  res.render('index', { user: req.user });
});

app.get('/account', ensureAuthenticated, function(req, res){
  res.render('account', { user: req.user });
});

app.get('/login', function(req, res){
  res.render('login', { user: req.user });
});

// GET /auth/WSO2
//   Use passport.authenticate() as route middleware to authenticate the
//   request. The first step in WSO2 authentication will involve
//   redirecting the user to the WSO server. After authorization, WSO2 will
//   redirect the user back to this application at /auth/wso/callback

app.get('/auth/wso',
  passport.authenticate('WSO-openidconnect'),
  function(req, res){

});

// GET /auth/WSO/callback
app.get('/auth/wso/callback',
  passport.authenticate('WSO-openidconnect', { failureRedirect: '/login' }),
  function(req, res) {
    // Successful authentication, redirect home.
    res.redirect('/');
  });

```

```
app.get('/logout', function(req, res){
  req.logout();
  res.redirect('/');
});
```

```
app.listen(3000);
```

```
function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/login')
}
```

Index

```
<% if (!user) { %>
<h2>Welcome! Please log in.</h2>
<% } else { %>
<h2>Hello, <%= user._json.name %> <%= user._json.family_name %>.</h2>
<% } %>
```

Login

```
<a href="/auth/wso">Login with WSO</a>
```

References

- [1] B Adida. Cosign: Secure, intra-institutional web authentication. <http://weblogin.org/> (2012, August).
- [2] B Adida. defending against your own stupidity. <http://benlog.com/2010/09/07/defending-against-your-own-stupidity/> (2010, September).
- [3] Jose Maria Alonso, Rodolfo Bordon, Marta Beltran, and Antonio Guzmán. Ldap injection techniques. In *Communication Systems, 2008. ICCS 2008. 11th IEEE Singapore International Conference on*, pages 980–986. IEEE, 2008.
- [4] A. Armando. Formal analysis of saml 2.0 web browser single sign-on. *Proceedings of the 6th ACM workshop on Formal methods in security engineering*. 2008.
- [5] Giuseppe Ateniese and Stefan Mangard. A new approach to dns security (dnssec). In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 86–95. ACM, 2001.
- [6] Pascal Aubry, Vincent Mathieu, and Julien Marchal. Esup-portal: open source single sign-on with CAS (central authentication service). *Proc. of EUNIS04-IT Innovation in a Changing World*, pages 172–178, 2004.
- [7] D Baier, editor. *Single Sign-on & Authentication for Mobile, Web & Desktop Applications*. 2014, November.
- [8] D Baier, editor. *Unifying Authentication & delegated API Access with OpenID Connect & OAuth2*. 2014, December.
- [9] Adam Barth. The web origin concept. 2011.
- [10] Alex Bilbie. A guide to OAuth 2.0 grants. <http://alexbilbie.com/2013/02/a-guide-to-oauth-2-grants/> (2013, Februari) seen 2015-02-20.
- [11] Arne Dahl Bjune. OpenID Connect. 2014.
- [12] Tim Bray. On the deadness of OAuth2. <https://www.tbray.org/ongoing/When/201x/2012/07/28/Oauth2-dead> (2012, August) seen 2015-02-20.
- [13] Brian Campbell. authorization implies authentization. <http://oauth.net/articles/authentication/> (2014, November) seen (2015-05-13).
- [14] Scott Cantor, Internet2 John Kemp, Nokia Rob Philpott, and E Maler. Assertions and protocols for the oasis security assertion markup language. *OASIS Standard (March 2005)*, 2005.
- [15] Fernando Duarte Carvalho and E Mateus Da Silva. *CyberWar-Netwar: Security in the information age*, volume 4. IOS Press, 2006.
- [16] D. W. Chadwick. Federated identity management. University of Kent (2009).
- [17] Don Davis. Defective sign & encrypt in s/mime, pkcs# 7, moss, pem, pgp, and xml. In *USENIX Annual Technical Conference, General Track*, pages 65–78, 2001.
- [18] Z. Dennis. Choosing an SSO strategy. <http://www.mutuallyhuman.com/blog/2013/05/09/choosing-an-sso-strategy-saml-vs-oauth2/> (2013, May).

- [19] Brendan Eich and Rand Mckinney. Javascript language specification.
- [20] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
- [21] Andrew Findlay. Best practices in ldap security. 2011.
- [22] Matthew Flynn. Single Sign-On; multiple confusion points. <http://360tek.blogspot.nl/2006/11/single-sign-on-multiple-confusion.html> (2006, November) seen 2015-02-20.
- [23] Foursquare. Connect to foursquare. <https://developer.foursquare.com/overview/auth> (2014, January) seen (2015-05-7).
- [24] Maurice Gagnaire, Felipe Diaz, Camille Coti, Christophe Cerin, Kazuhiko Shiozaki, Yingjie Xu, Pierre Delort, Jean-Paul Smets, Jonathan Le Lous, Stephen Lubiarez, et al. Downtime statistics of current cloud solutions. *International Working Group on Cloud Computing Resiliency, Tech. Rep., June*, pages 176–189, 2012.
- [25] Idan Gazit. Grant types. <http://oauthlib.readthedocs.org/en/latest/oauth2/grants/password.html> (2012, October) seen 2015-02-20.
- [26] Thomas Groß. Security analysis of the SAML single sign-on browser/artifact profile. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 298–307. IEEE, 2003.
- [27] E. Hammer. OAuth 2.0 and the road to hell. <http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/> (2012, July).
- [28] E. Hammer. OAuth Bearer Tokens are a terrible idea. <http://hueniverse.com/2010/09/29/oauth-bearer-tokens-are-a-terrible-idea/> (2010, September).
- [29] E. Hammer. Web authorization protocol. <https://github.com/hueniverse/oz> (2013, May).
- [30] Dick Hardt. The OAuth 2.0 authorization framework. 2012.
- [31] Harvard. About harvard’s authentication system. <http://iam.harvard.edu/get-started/authentication> (2015, March).
- [32] Bredndan Hayes. Web SSO vs. enterprise SSO. <http://www.pathmaker-group.com/web-ssso-vs-enterprise-ssso-what-do-i-need/> (2012, Februari) seen 2015-02-20.
- [33] Colm O Heigeartaigh. Web authorization protocol. <http://coheigea.blogspot.nl/2012/10/xml-signature-wrapping-attacks-on-web.html> (2012, October) seen 2010-09-30.
- [34] Jeff Hodges, Collin Jackson, and Adam Barth. Http strict transport security (hsts). Technical report, 2012.
- [35] Jeff Hodges, Chris McLaren, Prateek Mishra, Bob Morgan, Tim Moses, and Evan Prodromou. SAML security considerations. *OASIS*.
- [36] Egor Homakov. The most common OAuth2 vulnerability. <http://homakov.blogspot.nl/2012/07/saferweb-most-common-oauth2.html> (2012, July).
- [37] Egor Homakov. OAuth2.a or let’s just fix it. <http://homakov.blogspot.nl/2012/08/saferweb-oauth2a-or-lets-just-fix-it.html> (2012, August) seen 2015-02-20.

- [38] Timothy A Howes, Mark C Smith, and Gordon S Good. *Understanding and deploying LDAP directory services*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [39] Jani Hursti. Single Sign-On. Helsinki University of Technology (1997).
- [40] Jasig. CAS protocol. <http://jasig.github.io/cas/4.0.x/protocol/CAS-Protocol.html> (2014, December).
- [41] WANG Jing. Covert redirect vulnerability. http://tetraph.com/covert_redirect/ (2014, May).
- [42] Michael Jones, Paul Tarjan, Yaron Goland, Nat Sakimura, John Bradley, John Panzer, and Dirk Balfanz. Json web token (JWT). 2012.
- [43] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, 2006.
- [44] D. Kearns. The death (and life) of a protocol. <https://blogs.kuppingercole.com/kearns/2012/07/31/the-death-and-life-of-a-protocol/> (2012, July).
- [45] Kash Kiani. Four attacks on OAuth - how to secure your Auth. SANS (2011, March).
- [46] Pawel Krawczyk. Secure SAML validation to prevent XML signature wrapping attacks. Cornell University (2014, January).
- [47] Manuel Lemos. Is your oauth 2.0 application secure? <http://www.phpclasses.org/blog/package/7700/post/4-Is-Your-OAuth-20-Application-Secure.html> (2014,May) seen (2015-06-12).
- [48] Kelly D Lewis. Web single sign-on authentication using SAML. *arXiv preprint arXiv:0909.2368*, 2009.
- [49] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 threat model and security considerations. 2013.
- [50] Eve Maler and Drummond Reed. The venn of identity: Options and issues in federated identity management. *IEEE Security & Privacy*, (2):16–23, 2008.
- [51] Oscar Manso, Morten Christiansen, and Gert Mikkelsen. Web based identity management systems. 2014.
- [52] Charles R McClure and John Carlo Bertot. *Evaluating networked information services: Techniques, policy, and issues*. Information Today, Inc., 2001.
- [53] Michael McIntosh and Paula Austel. XML signature element wrapping attacks and countermeasures. In *Proceedings of the 2005 workshop on Secure web services*, pages 20–27. ACM, 2005.
- [54] Sun Microsystems. SAML v2.0 basics. <https://www.oasis-open.org/committees/download.php/20520/SAMLV2.0-basics-Oct2006.pdf> (2006, October).
- [55] Steven P Miller, B Clifford Neuman, Jeffrey I Schiller, and Jermoe H Saltzer. Kerberos authentication and authorization system. In *In Project Athena Technical Plan*. Citeseer, 1987.
- [56] nelsonic. learn json web tokens. <https://github.com/docdis/learn-json-web-tokens> (2015, April) seen (2015-05-04).

- [57] J. Oberheide. Cosign SSO vulnerability. <https://jon.oberheide.org/blog/2007/04/12/cosign-ssvulnerability/> (2007, April).
- [58] University of Michigan. Cosign authentication process. http://webservices.itcs.umich.edu/mediawiki/cosign/index.php/Main_Page (2006, December).
- [59] OWASP. Cross-site request forgery. <https://www.owasp.org/index.php/CSRF> (2014, December).
- [60] OWASP. Cross-site request forgery (csrf) prevention cheat sheet. https://www.owasp.org/index.php/CSRF_Prevention_Cheat_Sheet (2015, June) seen (2015-06-12).
- [61] OWASP. Open redirect. https://www.owasp.org/index.php/Open_redirect (2012, March).
- [62] Joon S Park and Ravi Sandhu. Secure cookies on the web. *IEEE internet computing*, 4(4):36–44, 2000.
- [63] Penn. CoSign troubleshooting. <https://secure.www.upenn.edu/computing/resources/category/web/article/cosign-troubleshooting> (2013, April) seen 2015-02-20.
- [64] o. pfaff. Openid connect - an emperor or just new cloths? <http://www.slideshare.net/oliverpfaff/openid-connect-an-emperor-or-just-new-cloths> (2012, April).
- [65] Nicholas Piasecki. When in doubt: OpenID is a bad idea for your web site. <http://www.twobotechnologies.com/blog/2012/08/cloud-security-standards.html> (2009, July) seen 2015-02-20.
- [66] Erik Poll, editor. *Forced/forceful browsing*.
- [67] Erik Poll, editor. *MitM attacks on HTTPS sessions*.
- [68] Erik Poll, editor. *Session Management*.
- [69] Andy Powell and David Recordon. OpenID: Decentralised single sign-on for the web. *Ariadne*, (51):9, 2007.
- [70] Joost Reede. On a-select and federated identity management systems. 2007.
- [71] David Ross and Tobias Gondrom. Http header field x-frame-options. Technical report, 2013.
- [72] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2:1–13, 2010.
- [73] Mike S. 10 reasons why OpenID Connect will be ubiquitous. <http://www.gluu.org/blog/10-reasons-openid-connect-will-be-ubiquitous/> (2014, February).
- [74] Mike S. SSO protocol adoption predictions. <http://www.gluu.org/blog/gluu-web-authentication-ssvprotocol-adoption-predictions/> (2013, December).
- [75] Draft N Sakimura, J Bradley, M Jones, B de Medeiros, and E Jay. OpenID Connect standard 1.0-draft 20. 2011.
- [76] Anil Saldhana. SAML vs OAuth. *DZone*, 2013.

- [77] Mathew J. Schwartz. OAuth, OpenID flaw: 7 facts. *darkReading*, 2014.
- [78] Brent Shaffer. Oauth 2 scope. <https://bshaffer.github.io/oauth2-server-php-docs/overview/scope/> (2015, April) seen (2015-05-11).
- [79] Hank Simon. SAML: the secret to centralized identity management. 2004.
- [80] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking SAML: Be whoever you want to be. In *USENIX Security Symposium*, pages 397–412, 2012.
- [81] Travis Spencer. Cloud security standards. <http://www.twobotechnologies.com/blog/2012/08/cloud-security-standards.html> (2012, August) seen 2015-02-20.
- [82] Travis Spencer. Deep dive into OAuth and OpenID Connect. <http://nordicapis.com/api-security-oauth-openid-connect-depth/> (2014, December).
- [83] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390. ACM, 2012.
- [84] Matthias Trisl. Openam. 2014.
- [85] Staffordshire University. shibboleth. <http://projects.staffs.ac.uk/suniwe/project/shibboleth.html> (2008, January).
- [86] W3.org. Token based authentication. http://www.w3.org/2001/sw/Europe/events/foaf-galway/papers/fp/token_based_authentication (2001).
- [87] Brian Wellington. Domain name system security (dnssec) signing authority. 2000.
- [88] MATIAS WOLOSKI. 10 things you should know about tokens. <https://auth0.com/blog/2014/01/27/ten-things-you-should-know-about-tokens-and-cookies/#token-cross-domains> (2014, January) seen 2015-04-8.
- [89] Josh Wyse. Why JSON is better than XML. <http://cloud-elements.com/json-better-xml/> (2014, August) seen 2015-02-20.
- [90] Kurt Zeilenga. Lightweight directory access protocol (LDAP): Technical specification road map. 2006.
- [91] Bas Zoetekouw and Martijn Oostdijk. OpenID Connect for surfconext. *SURF*, 2012.