POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Implementation of a Single Sign-On System with Open Source Software

Relatore
prof. Francesco Laviano

Candidato
Ludovico Pavesi

Anno accademico 2018-2019

Summary

This thesis will focus on the implementation of a single sign-on (SSO) system in a real-world scenario. Single sign-on allows users to log in to a number of heterogeneous software applications and computers with the same set of credentials (e.g. username and password).

These systems have fundamental importance in organizations where a large number of "internal users" (e.g. employees, students, etc.) needs to access an heterogeneous system of software components, but are often overlooked in smaller organizations.

SSO systems have many benefits:

For users Reduced password fatigue, both due to having less credentials to remember, and to the system ability to recognize their session and execute the sign-on procedure without prompting for password, when switching from one application to the other.

For tech support Having less passwords usually means less calls to reset forgotten passwords.

For security Users submit their sensitive credentials, for example their plaintext password over an encrypted connection, only to the SSO server and never to other applications. If implemented correctly, this greatly reduces the risks of a vulnerability in other applications: the only data that an attacker could gain is limited to what is known to that application, which does not include any credentials that could be used to access other applications (e.g. hashed passwords). Moreover, it is easier to create, delete and temporarily lock user accounts.

For policy Relegating authentication to a single, centralized system also means moving all the authentication policies to a single, centralized system, making their enforcement easier. For example enforcing a minimum password length or a two factor authentication process (2FA) becomes very simple, having to configure only the SSO software rather than every single application.

For legal compliance It may be legally required to erase data related to a user under some circumstances: having a single location where data is located and should be erased makes this process simpler and less error prone.

For application developers Many large organizations have custom-built internal applications for their employees or other users (e.g. students in a university). Having a SSO system means that there's no need to replicate user account management and sign-on functionality in every application, since it is sufficient to connect it to the central SSO server, usually through a ready-made library.

For federation Very important in some fields, is the possibility to connect such SSO servers together in a federation, to allow users to sign-in to a different organization with an account from their home organization.

Despite these advantages and the fact that such systems have been deployed in many large organizations since the early 2000s, a series of problems hampering their adoption still exists:

Complexity These systems are usually large and complex, with many components needed to meet the requirements of different use cases, and often provide support both for current protocols (e.g. SAML2 and OpenID Connect) and legacy or non-standard protocols (e.g. SAML1, OpenID, CAS, Kerberos). Moreover, many of these systems provide multiple back-ends for data storage (e.g. relational databases, LDAP), and some of them also provide a form of access control: that is, they can evaluate policies to make authentication and authorization decisions.

Lack of support in off-the-shelf applications With the gradual demise of most desktop based applications, replaced by cloud or mobile applications, it should become gradually easier to integrate SSO systems that naturally require a network to function, since the SSO server is not the same as the application server: however, many applications, especially ones targeted at small organizations, still have lacking or mediocre support for SSO systems, often supporting only one protocol and not even in a complete manner.

Lack of public documentation While most SSO software comes with a complete manual that describes how to configure every aspect of the software, there are few tutorials written by third parties on how to configure such systems. This is in part due to the lack of demand and necessity for that many SSO systems as there is for web servers with their plethora of tutorials that describe every possible aspect of their configuration, for example. This is also due to the fact that such systems are often implemented in large enterprises by their employees or consultants and

they have no little to no incentive to provide large amounts of documentation for free to the general public.

This may not seem a problem, but theoretical knowledge and a manual are not enough. Both have their undeniable merits and are still important when faced with the task of implementing such a system, but there is also a large fraction of the work that is practical, and "getting there" by pure trial and error is not always feasible: it requires time to understand how to design the system, how to install and configure the software components, how to make it work for one's use case, how to ensure that the system is secure and reliable, and so on.

The result is that developers often don't even bother to support these protocols and systems, while business owners and IT administrators in smaller organizations fail to recognize the advantage of these systems given all the drawbacks and effort necessary for their implementation.

While this document cannot serve as a tutorial for the software that has been described, since it cannot be updated according to new releases of the software, it may still help somebody faced with similar tasks to gain a better understanding of what are the possible choices and what could be the motivation behind some of them. Obviously these will not be "the correct choice" for every scenario or the only possible reference, but will hopefully provide another example on how things could be done, what worked and what didn't.

Moreover, almost all of the code and configuration that has been written along this thesis has been released as open source, under a free software license, and made reusable where possible: while it may help someone else to gain insight on the practical problems of configuration and deployment of some software components, it is also a small step in the direction of providing the "tools and automation" to "quick start" some of these software components and build upon them.

The work is organized as follows: chapter 1 provides an introduction to the real world scenario in which the system has been implemented and motivations for this decision. Then, the three main components of the system are presented: chapter 2 discusses the implementation of the user account database with a LDAP server, chapter 3 discusses the implementation of the SSO server itself and finally chapter 4 provides some details on the user account management software that has been created.

Contents

1 Motivation 9
1.1 The “as is” situation 9
1.2 Process redesign 11
1.3 Deciding an implementation strategy 13
1.3.1 LDAP 13
LDAP authentication in applications 14
1.3.2 Single sign-on 16
SSO authentication 17
1.3.3 Additional security considerations 19
1.3.4 Software support 19
1.3.5 User provisioning 21
1.4 Implementation choices 23
1.4.1 A note on open source 23
1.4.2 Methodology 24

2 LDAP server 26
2.1 Comparison of available software 26
2.1.1 Community of developers 26
2.1.2 Community of users 27
2.1.3 Documentation 28
2.1.4 Performance and reliability 29
2.1.5 Compatibility 30
2.1.6 Comparison results 31
2.2 Designing the directory 31
2.2.1 Structure 32
2.2.2 Schema 32
2.2.3 Custom classes 37
2.3 Configuration management 39
2.3.1 Ansible 40
2.3.2 Vagrant 41
2.3.3 Choosing a role 41

2.3.4 Making a role or two . . . 44
Setting LDAP attributes . . . 44
TLS and certificate management . . . 45
TLS enforcement . . . 48
Automated tests . . . 49
2.3.5 Replication . . . 50
2.3.6 Examples . . . 51
2.4 ACI attributes . . . 52

3 SSO server 55
3.1 Comparison of available software . . . 55
3.1.1 Community of developers . . . 56
3.1.2 Community of users . . . 56
3.1.3 Documentation . . . 58
3.1.4 Performance and reliability . . . 58
3.1.5 Compatibility . . . 58
3.1.6 Comparison results . . . 59
3.2 First attempt: Keycloak . . . 59
3.2.1 Installation . . . 59
3.2.2 Configuring the LDAP backend . . . 60
3.2.3 Configuring the relational database . . . 62
3.3 Second attempt: WSO2 IS . . . 63
3.3.1 Installation . . . 63
3.3.2 Configuring the LDAP backend . . . 64
3.3.3 Configuring the keystore . . . 64
3.3.4 Configuring clients . . . 66
3.4 Final considerations . . . 66
3.4.1 Single sign-out . . . 66
3.4.2 Session management . . . 67
3.4.3 Adaptive authentication . . . 69

4 User management application 70
4.1 Programming language and frameworks . . . 71
4.2 General structure and libraries . . . 72
4.3 Refresh token issues . . . 75
4.4 Service accounts and permissions . . . 77
4.5 User input validation . . . 79
4.6 Registration and invite code . . . 81
4.7 Password change . . . 83

5 Conclusions 85

Bibliography 87

List of Abbreviations

ABAC	Attribute-based access control
ACI	Access Control Item
ACME	Automated Certificate Management Environment
API	Application programming interface
CA	Certificate authority
CSS	Cascading Style Sheets
CSV	Comma-separated values
DN	Distinguished name
DNS	Domain Name System
GDPR	General Data Protection Regulation
HR	Human Resources
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IdM	Identity management
IETF	Internet Engineering Task Force
IP	Internet Protocol
(WSO2) IS	Identity Server
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union - Telecommunication Standardization Bureau
JDBC	Java Database Connectivity
JDK	Java Development Kit
JS	JavaScript
JSON	JavaScript Object Notation
JVM	Java virtual machine
JWT	JSON Web Token
LAN	Local area network
LDAP	Lightweight Directory Access Protocol
LDIF	LDAP Data Interchange Format
NSS	Network Security Services
OID	Object Identifier

OTP	One-time password
OTS	Off-the-shelf
PDF	Portable Document Format
PHP	PHP: Hypertext Preprocessor
RBAC	Role-based access control
REST	Representational State Transfer
RFC	Request for Comments
SaaS	Software as a service
SAML	Security Assertion Markup Language
SCHAC	SCHema for ACademia
SCIM	System for Cross-domain Identity Management
SIR	Scheda Identificazione Rischi occupazionali
SQL	Structured Query Language
SSF	Security Strength Factor
SSH	Secure Shell
SSO	Single sign-on
TLS	Transport Layer Security
URI	Uniform Resource Identifier
UUID	Universally unique identifier
VPN	Virtual private network
XML	Extensible Markup Language
XSS	Cross-site Scripting
YAML	YAML Ain't Markup Language

Chapter 1

Motivation

1.1 The "as is" situation

The background for this project was a "student team", WEEE Open[1], at Politecnico di Torino. Student teams are groups of students that, under the direction of a professor and with funding from the university, work on a project that is usually relevant from an engineering, architectural or social point of view. WEEE Open team, in particular, salvages old computers, repairs them and donates them to no-profit organizations, schools, and other institutions.

Despite being just a group of students in a university, such a team could be considered an organization somewhat similar to a small business, with a simple structure and a primary activity: repairing and donating computers and other hardware. There are also some support activities related to "human resources" management and IT.

In particular, the support activities involve maintaining an instance of a cloud file sharing software (NextCloud), a custom-made inventory management software (Tarallo[2]), a Telegram bot that provides useful information to team members, and some other smaller software components that mainly interact with these. Moreover, a record of all the team members has to be kept and updated each time someone leaves or joins the team.

These are stovepipe systems, each one with its authentication and authorization methods: each new member gets a separate account for each software, and when a member leaves all the accounts have to be deleted.

[1] `http://weeeopen.polito.it/`

[2] `http://github.com/weee-open/tarallo`

Even promoting or demoting someone means making multiple manual modifications: granting or revoking access to restricted folders on the file sharing software, adding or removing administrator permissions on the inventory software and on the bot, and so on.

This process has also prevented the team from adopting more software tools, e.g. a wiki, which would be yet another stovepipe system to manually manage.

Moreover, when someone joins the team, a form (SIR) has to be filled and signed both by the team member and the leader professor, and a multiple choice test on "safety in laboratories" has to be scheduled and completed by the new student.

The resulting onboarding process is cumbersome, complicated and with lots of data duplication, and is shown in figure 1.1.

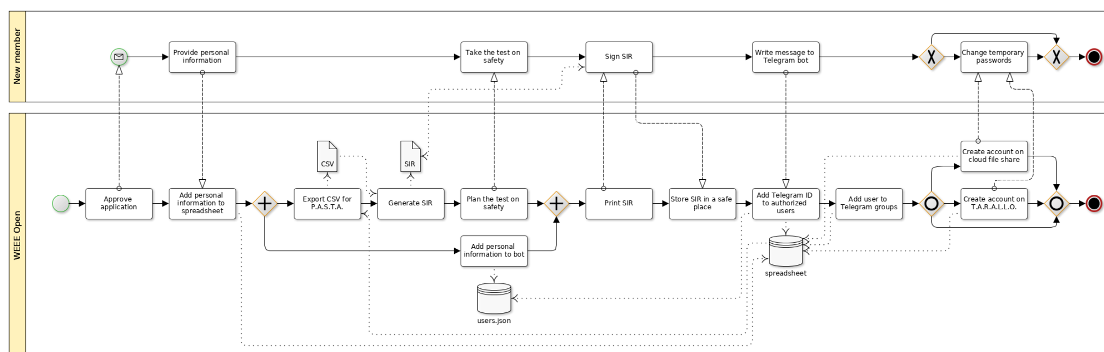

Figure 1.1. The "as is" onboarding process

When the application for a new team member is approved, they are contacted and asked for some of their personal information, which is needed in order to keep a record of all team members and to fill the SIR form. This data is added to a spreadsheet that over the years has grown to 25 columns. Then, part of the spreadsheet is exported in CSV format and imported into Pasta,[3] a software made by the team that fills the form automatically and provides a PDF file that is subsequently printed.

The test on safety is then planned with the new team members and technical staff of the university that has to supervise the test. In the meantime, another part of the user personal information is added to the Telegram bot[4], in a JSON file: this is so that the bot can refer to users by their real name, rather than by their username.

[3]https://github.com/weee-open/pasta

[4]https://github.com/WEEE-Open/weeelab-telegram-bot/

This file is also used by weeelab,[5] yet another custom application that keeps track of time that members spend in the laboratory and to do so it also requires a user list.

Once the test on safety has been taken, the SIR is signed and stored for later processing. The new member then has to write a message to the Telegram bot, which will reply that the user is not yet authorized: however, doing so, users reveal their Telegram ID to the bot, which stores them in a temporary file. The ID is then manually copied from that file to the JSON file. The bot is able to recognize users only from their Telegram ID, which Telegram servers provide every time they deliver a message to the bot. The user is then added to Telegram groups by another team member, yet another manual process.

Finally, accounts are created on different software: sometimes this part is deferred until it is actually needed and some users never get an account on some software that they will never use, but it is still a manual and time-consuming process. Accounts are created with a temporary password which is given to the user along with the instruction to change it, and it has been observed that sometimes people don't do that and just keep the temporary password forever.

Some improvements have already been performed in the past. For example, having a software that fills the SIR form automatically saved a dramatic amount of time. Another improvement has been merging the weeelab and the bot users list in the same JSON file, which removed one file to manually update. However, the process is still very time consuming, or rather a lot more time consuming than it could be. The sheer quantity of manual steps that have to be performed at different times also makes the process error prone.

Moreover this process has to be repeated around 50 times per year, and a similar reverse process also has to be repeated around 50 times per year when someone leaves the team. Turnover rate is in fact very high, but this is a problem that affects every team.

Finally, in the last three years these processes have been performed primarily by a student that is hopefully graduating soon, so a long term and more automated solution is desperately needed.

1.2 Process redesign

While the gradual improvements performed in the past somewhat simplified and automated the process, a radical redesign is needed to achieve shorter times, less

[5] `https://github.com/WEEE-Open/weeelab/`

manual steps, less errors and increased security (e.g. no more temporary passwords left forever). By removing as many manual steps as possible and centralizing the user database, these results should be achievable.

The human factor should also be considered: having more automated steps will reduce the workload on the support activities and so improve the morale and increase time that can be devoted to the primary activity of the team. Having a single source for all accounts will also reduce password fatigue from remembering different passwords for different services.

While the requirement to have all new members take the test on safety and sign the SIR form is mandated by the university and cannot be changed in any way, all the user database that exist in some form or the other could be replaced with a central user database, then an application that handles registration in a more automated manner can be built and Pasta can be integrated with it. This redesigned process is shown in figure 1.2.

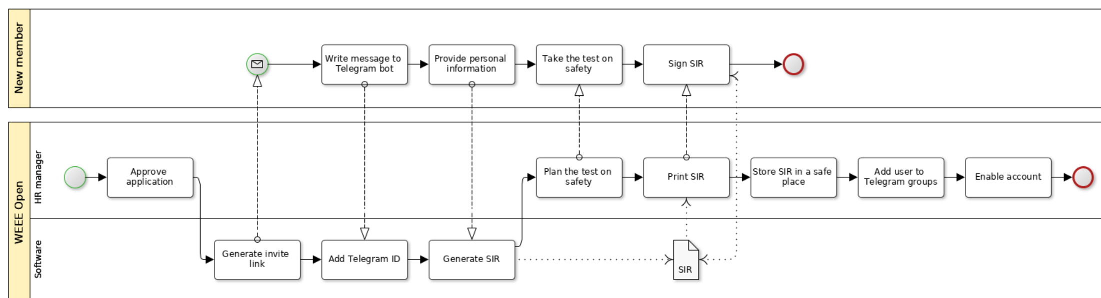

Figure 1.2. The “to be” onboarding process

“HR managers” are a subset of team members that handle these tasks, which will remain the same as before, while the tasks labeled as “Software” will be performed by a custom-made application.

All the separate user databases have been removed: only one database is present, although not shown in the figure. No action needs to be performed on different software applications: they will read user account information from the database and will perform user authentication according to the database.

As part of this process, the bot will also need to write information to the database on its own, in the same way that other applications may read from it. This is because the Telegram ID is not readily visible from the Telegram application, but bots can read it from every message they receive from Telegram servers: the idea is to have the bot store that information in the user database. As an alternative, the bot may print the ID and the user has to paste it into a form.

1.3 Deciding an implementation strategy

The process in figure 1.2 does not specify any particular technology or software to realize it, since different choices are possible and will be discussed in this section.

1.3.1 LDAP

To centralize all the accounts and the authentication, it is possible that a LDAP (Lightweight Directory Access Protocol) server may be sufficient.

LDAP is an application layer protocol to access and manage a directory service[1]. A directory is akin to a phone book, as it usually contains a list of people and their contact information: phone number, email address, office location, etc. Sometimes it may contain information on networked devices, like computers and printers.

LDAP was originally described in 1995 and was conceived as a simpler method to access X.500 directories, implementing a subset of the X.500 protocol from 1988[2]. A few years and some iterations later, a RFC was published for version 3 of the protocol[1], which is the one still used nowadays.

Rather than being a gateway to an X.500 server as it was originally designed, today LDAP is mainly used as a "standalone" software, with the LDAP server itself providing the backing database.

But why would someone use a directory access protocol to perform centralized authentication? LDAP servers allow users to authenticate before performing operations on the data stored by the server. This operation is named "bind" in LDAP terminology.

In its simplest form, a bind operation allows users to supply a DN – distinguished name, a unique identifier for an entry in the directory – and a plaintext password, possibly over an encrypted connection. The server compares the received password to the password hash it has stored for that DN and responds to the user with a confirmation that the authentication was successful, or an indication of which error was encountered in case of failure. The user can then perform operations on the server, according to the authorization policies defined for that user.

This carries similar security vulnerabilities to HTTP basic authentication, or HTML form based authentication with a back-end database that stores usernames and passwords:

- An attacker may gain access to the database and obtain DNs and hashed passwords
- An attacker may gain access to the backend server, which in LDAP is the same software as the database server, and intercept all DNs and plaintext passwords

- If the security of the protocol that transports LDAP (e.g. TLS) is compromised, an attacker may decrypt traffic and obtain DNs and plaintext passwords

It should be noted that LDAP as a protocol allows other authentication methods, for example some challenge-response mechanisms and TLS client certificates, but most OTS applications that allow centralized authentication through LDAP rely on user passwords only.

The advantage of using LDAP for authentication is that directories have been commonly deployed in corporate environments for decades, with their main use as directories, so with the growth in number of different software components that require user authentication it was easy to connect them to the directory and turn it into an authentication server.

LDAP authentication in applications

The method generally used[3] to perform authentication is:

1. Users supply their credentials (e.g. username and password) to an application (e.g. a website, through an HTML form)
2. The application performs a bind to the LDAP server with its own service account, which has its own password or any other form of credentials supported by the server (e.g. TLS client certificates)
3. The application performs a search for the username, to find its bind DN.
4. The application performs another bind, with the user DN and supplied password.
5. According to the result of that bind, the user is considered authenticated or not.

Optionally, the LDAP application may also query any number of LDAP attributes for the user and perform authorization based on these. This can range from a simple list of groups that the user is part of, to realize role-based access control (RBAC), to more complex attribute-based access control (ABAC) which is a superset of RBAC. However, only the attributes useful for authorization are stored in the central LDAP server: the authorization policies and ultimately the authorization decision are still left to the application. While this step could also be implemented with a central authorization server, but this is outside the scope of this discussion: the only supported method for most applications is RBAC based on LDAP group membership, with all authorization decisions performed by the application.

The third step, the search of the user DN, is necessary because DNs should be treated as opaque identifiers by applications. In smaller LDAP deployments they

are often human readable, simple and "guessable" to a point. They may resemble `uid=janice.doe,ou=people,dc=example,dc=com`, where `uid` is an attribute of the user entry with a value of `janice.doe`. However, there's no guarantee that the DN isn't `cn=Janice Doe,ou=people,dc=example,dc=com`, or `cn=Janice Doe,ou=accounting,ou=people,l=Berlin,l=Germany,dc=example,dc=com`, or `nsUniqueId=824ea447-9eff-4d32-a701-009188960af4`, or that it may not change its structure in the future: making any assumption about DNs is bad practice and user DN, if needed, should be searched every time from one or more of its attributes. Which attributes are used for the search (e.g. `uid` for username) can be configured statically in the application, usually.

Security considerations were outlined in section 1.3.1, however, one more point should be addressed: if a single application uses a single relational database or LDAP database for its credentials storage, when any of these components or the communication between any of these components is compromised, the credentials may be stolen and reused against the same application, or attempted to use against unrelated applications if the same password has been reused.

However, if multiple applications share the same relational database or the same LDAP database, when credentials are compromised, attackers may obtain a set of credentials that are surely valid and will be accepted by multiple applications.

This, on the one hand, reduces the security of the system compared to a series of stovepipe systems where users have different credentials. On the other hand, it may somewhat increase security compared to credentials reuse on the stovepipe systems, since in case of breach it is much easier and faster to lock all compromised accounts centrally, or reset passwords and force users to change their password on the next login.

The fact that applications need a system account on the LDAP server implies that the credentials to that account have to be stored in plain text or encrypted in a reversible manner on the application server. This is usually not a significant security risk: all modern LDAP servers allows to restrict permissions for an account[4], so the system account can be allowed only to search for users' DN using only the attributes it needs, and to read only the attributes it needs for authorization, if that's not done through the user account. There's no need to read the hashed password, since the plaintext password is sent to the server – through a secure channel, hopefully – and the task to validate it against the hash is performed by the LDAP server.

However, another disadvantage is that applications have to maintain their own sessions and cannot check if users credentials are still valid without prompting for credentials again. This may be a problem in case a password has been compromised and used by an attacker to log in: users may change their password, but additional

mechanisms, specific to each application, are needed to audit current sessions and allow users or system administrators to terminate other sessions. This increases the burden on application developers, users and administrators, or decreases overall security of the system if they are not implemented.

From an usability standpoint, while LDAP authentication makes easier to remember credentials and update them since they are shared across applications, it still provides a sub-optimal experience when switching from one application to the other, since users still need to sign in and sign out of applications independently. This still induces password fatigue, by making users type their passwords again and again, and makes it difficult to log out of all the applications simultaneously.

An improvement in security and usability is, arguably, a real SSO system.

1.3.2 Single sign-on

A real SSO system mainly consists of an authentication server and a backing store, be it LDAP or a relational database, which could also be embedded in the server.

The main diffence from a "fake" SSO like a LDAP server is that users perform authentication by submitting credentials only to the SSO server, which in turn tells information about the user (usually named "claims" or "attributes", depending on the protocol) the application. This is true mostly in a "web SSO" scenario, where a web browser is available and the SSO server provides an HTML form or similar methods to let user input their credentials: this is the case for all cloud applications, for mobile applications (by launching the browser application) and possibly for desktop applications, so it is probably the most common SSO scenario and the only one that has been considered in this document.

It should be noted that this is only valid for interactive authentication – that is, when users are sitting in front of a screen and can provide their credentials – but doesn't really work for machine to machine communication, e.g. for a service that needs to call APIs that require authentication. Both the most common and modern SSO protocols, SAML2 and OpenID Connect, provide some methods to perform also this kind of authentication, usually by storing credentials in plain text in the client. The client then submits these credentials to the SSO server, or to the called application that in turn submits them to the SSO server. This clearly reduces the security of the system, since credentials need to be stored in plain text, so they should be used only when interactive authentication is not possible.

The advantages of SSO systems, compared both to stovepipe systems and "fake" SSO, have been outlined in the introduction to this document.

SSO authentication

There are many protocols, both standard an proprietary, but the general idea for interactive authentication in web SSO is as follows:

1. User accesses an application, e.g. by navigating to the application URI with a browser
2. The application has no active session for the user and requires authentication: it redirects the user to the SSO server URI
3. The SSO server provides some method for users to authenticate, e.g. an HTML form for username and password
4. The user authenticates
5. The SSO server redirects the user back to the application, and provides some signed claims or attributes about the user to the application
6. The application verifies the signed claims or attributes and performs all the necessary steps to authorize the user and start a session

The method used to provide claims to the application depends on the protocol, and it may use a front channel or a back channel.

As an example, SAML2 usually uses a front channel method, in which it adds a signed and encoded XML response containing the required attributes to the application where it redirects the user upon successful authentication: the HTTP protocol and the user agent – for example a web browser – are used as transport between the SSO server and the application server.

OpenID Connect supports multiple methods, or "flows", for authentication: the most common one for applications with a back-end running on a server, the "Authorization Code Flow", involves the SSO server sending the application an authorization code via query parameters while redirecting after successful authentication; however, the back-end application uses the authorization code to request the ID token from the OpenID Connect server through a back channel, i.e. by making an HTTP request directly to the server, without involving the user agent.[6]

The claims or attributes can be any information regarding the user, e.g. username, email address, age, groups, roles, etc. Since they are signed by the SSO server, the application needs to check the signature to ensure that the claims have not been tampered with. Both in SAML2 and OpenID Connect, this is done by providing the SSO server public keys – manually or with automated discovery – to all applications, which will trust these keys.

The claims may contain additional metadata, for example an OpenID Connect token (the standardized data structure that holds the claims and their signature)

also contains these mandatory parameters:

1. The issuing authority of the token, i.e. an identifier of the SSO server, in case an application can accept authentication from multiple SSO servers
2. The "audience" of the token, i.e. and identifier of the intended client application(s), to prevent attacks that involve stealing the token and using it against other applications
3. An identifier for the subject of the token, that is: an identifier for the user
4. An expiration time, to mitigate against attacks that involve reusing a stolen token
5. The time at which the token was issued

There are also more parameters available, which are optional, or mandatory only in some circumstances.

Claims may be optionally encrypted for confidentiality, but this is rarely done and both the SAML2 specification and OpenID Connect specification[5][6] make it optional, since it would offer the same protection as an encrypted connection between user agent, SSO server and application server, e.g. via TLS, which is a standard security measure and is generally implemented. The OpenID Connect specification mandates that all operations have to be performed over TLS.

Furthermore, the application may periodically contact the SSO server to check that the claims are still valid and the session is still active.

It should be noted that, while maintaining a session for the application is responsibility of the application itself, the SSO server may also maintain a session, e.g. with browser cookies. This is relevant from an user experience point of view: when users navigate from one SSO-enabled application to the other, the aforementioned process is repeated, but the the SSO server may decide that, if the session is still valid, it may skip the prompt for user credentials, issue the necessary tokens or responses, and immediately redirect the user to the application. Users only see some redirects that normally take a few seconds at most, then they are instantly signed in to the other application, without any need to enter their credentials or perform any manual authentication step.

Of course, it is usually possible to configure this behavior to balance security and convenience: SSO servers may be instructed to require credentials only if the previous authentication has been performed more than a specified amount of time before, or if the application that requires authentication is particularly sensitive, or to perform only one step in a two-factor authentication process.

One thing should be noted from a security standpoint, however: while applications

never see the user credentials and have no access to the plaintext passwords, nor to any hashed or encrypted form of these, the application still needs to be trusted: nothing prevents the application, for example, from obtaining an ID token and using it forever without checking the expiration time, or from accepting any claim without validating the signature.

1.3.3 Additional security considerations

Table 1.1 was written to compare potential security vulnerabilities of different systems, which may be not as clear to compare as usability or user experience on different systems.

If attackers compromise X, what can they obtain?	Stovepipe systems	LDAP	SSO
Application files or database, read only	Hash	Password (system)	Nothing
Application files or database, full control	Password	Password	Nothing
Authentication database, read only	N/A	Hash	Hash
Authentication database, full control	N/A	Password	Password

Table 1.1. Security properties comparison of stovepipe systems, LDAP and SSO

The table compares only risks related to user credentials, not other data. Using LDAP authentication almost provides the sum of vulnerabilities from stovepipe systems and SSO.

N/A means that there is no separate authentication database in stovepipe systems, since it is part of the application database.

1.3.4 Software support

Aside from all security and user experience considerations, one more aspect should be considered: software support for both LDAP and SSO protocols.

For this, an evaluation has been performed, by taking into account all the applications that the team currently uses or wants to use and how well they support each protocol. This evaluation also includes two programming languages – PHP and Python – that have been used for most of the team's custom applications: libraries that provide support for those protocols has been considered, since team members will have to use such libraries to add support for SSO to custom applications.

A score ranging from 0 to 3 has been assigned to each application and protocol:

- 3: Support included in the application or in standard libraries, with adequate documentation
- 2: Support included but without documentation (which will make installation and configuration more complicated), or there is a plugin or external library that is well maintained and documented but not included by default
- 1: Plugin or library without documentation or apparently unmaintained
- 0: No support at all, or at least nothing has been found in the documentation and on the Internet

Then, for each protocol, a sum has been computed. The result is shown in table 1.2 and refer to publicly available information available at the time of writing, March 2019. Other plugins or undocumented features may exist, but only simple web searches and the official manual of the software have been consulted.

Software	LDAP	SAML2	OpenID Connect
NextCloud	3	2	0[6]
Gitea	3	0	1
DokuWiki	3	1	2
Wekan	2	0	3
PHP	3	2	2
Python	2	2	2
Sum	16	7	10

Table 1.2. Comparison of authentication methods support across applications

It is interesting to note that every application has good support for LDAP authentication, usually better than their support for SSO protocols.

More applications which are not used yet and not planned for the short-medium term were also examined but not included in this comparison, since in the long term their support for such protocols may change or the decision to use such applications may change. However, similar results have been found even for these.

From this comparison, is clear that if the SSO solution is chosen, the server has to support both the currently used and standard protocols: SAML2 and OpenID

[6]OpenID Connect is supported but only in the scenario where NextCloud acts as the SSO server, not as a client (relying party), from what I could understand from the documentation.

Connect, since some applications only support one of them.

1.3.5 User provisioning

The additional requirement of user provisioning should be considered.

Some applications may need a full list of the users and groups available in the system, for example to allow users to exchange messages between each other, to share folders, and so on.

In a traditional web application setup, where user accounts are stored in a relational database, the application can access the same database both to authenticate a user and to obtain a list of users and their attributes.

With LDAP, the same is possible, if required: an application can just perform more searches to obtain a complete list of users.

In a SSO scenario – both with SAML2 and OpenID Connect – this is not possible, since the application has no access to the user database. It can only obtain claims provided by the SSO server and related to a single user. To solve this problem, the simplest approach is for the application to treat any sign-on from a previously unknown user a new registration.

Many open source web applications, like most of the ones considered in table 1.2, can be run "standalone" so they have provisions for user registration and authentication from a local source, e.g. a relational database. With this approach, they can easily support SSO protocols like SAML2 and OpenID Connect by just validating the claims, converting them to a data format suitable for the application and calling the appropriate internal methods: to log in or to register a new user.

This approach has been implemented, for example, in Gitea: OpenID Connect is supported, but a local user account is created during first sign-on.

This may be acceptable in applications where users don't need to interact with each other directly, for example in most of the team's custom applications, or when the application is provided "as a service" to the public and OpenID Connect just provides a way to perform "social login"[7] to simplify registration and authentication for users, where there is no external centralized database of users.

However, in an organization some sort of user database is available and only users that are known to the organization should be authorized to access the applications. Moreover, this is also an inconvenience when users need to interact with each other through the application. For example, in a file sharing application, some user may

[7] e.g. signing in with an account from a social network. These mechanism are usually implemented through OpenID Connect or proprietary variants of the protocol.

want to share a file with another user: if the other user has never logged in to the application, they will not have an account and the application will not allow others to share files with them.

Another important point that should be considered is that, while in a "social login" scenario, users should manage their accounts themselves and could delete them if they don't want to use the service anymore, in an organization accounts are normally deleted only when a person is no longer part of the organization. If OpenID Connect is used only to create a local account, old accounts should be deleted manually from each application, increasing the workload on system administrators or possibly violating the law if data is retained indefinitely.

The solution is to perform automated user provisioning separately from authentication. In a sense, even the creation of a local account on one of the applications based on claims provided by the SSO server can be considered a form of automated user provisioning. However, while the "account creation" part is automated, updating or deleting the account is generally manual, in such systems.

A more comprehensive approach that is fully automated is still possible in a SSO scenario. One way to fulfill it is to leverage the SCIM protocol. SCIM is an HTTP-based protocol[8] that provides a RESTful API to exchange information related to users, groups or other resources in JSON format[9]. It was standardized in 2015 by the IETF with two RFCs.

The problem is that, while it is supported by most SSO software, including the open source Gluu, WSO2 IS and CAS, support in open source applications is completely lacking: as an example, none of the applications mentioned in table 1.2 has any support for SCIM, despite some of them still requiring a local user database.

An alternative, which is slightly more supported across applications, is to use LDAP for user provisioning. This involves creating a system account for the application to search the directory, and obtain a user list from there, either "just in time" when needed by the application, or periodically to update a local user database.

The latter is what NextCloud does: it is possible to configure LDAP for authentication and user provisioning which periodically copies user information to a local database and deletes accounts that have been deleted from LDAP in a fully automated manner. The most interesting part, however, is that the SAML2 plugin for NextCloud can be configured on top of that to be the only authentication source: users are only allowed to sign-on through the configured SSO provider using the SAML2 protocol and the SSO server just needs to provide a unique identifier for the user (username, UUID, etc.) that can be compared with some LDAP attribute. In this way, LDAP is used exclusively for user provisioning (to create, update and delete local NextCloud accounts) and SAML2 exclusively for authentication.

1.4 Implementation choices

Since SSO, compared to LDAP authentication, provides better user experience, better security and other advantages, a decision was made to implement a SSO system, despite its complexity compared to a LDAP server for authentication.

However, LDAP will still be used as the storage for user accounts.

It should also be noted that all the applications have good support for LDAP as an authentication source, so having a separate LDAP server may be a viable strategy if any serious difficulty is encountered while configuring one application to work with the SSO server: at least user provisioning would still be automated and user passwords would be the same, despite losing all advantages of a real SSO system. This will allow to keep the onboarding process as much automated as possible and prevent the proliferation of user account databases that happened in the past among the team's applications.

Moreover, for the user provisioning part, LDAP is required since it's the only supported protocol in all the considered applications that support user provisioning.

Finally, this increases the flexibility of the system in case the chosen SSO server is replaced with another one in the future: there would be no need to migrate the users database, just to replace the SSO software and point the new one to the LDAP server. On the other hand complexity also increases, since two software components have to be installed, configured, managed and integrated.

1.4.1 A note on open source

All components of the system should be free and open source, both because team has basically no money for this project, and to reap the usual benefits that come with open source projects: avoiding vendor lock-in and sudden raises in licensing costs or fees, protection against the product being discontinued by the company (if the source code is available under a permissive license, the community of users may decide to continue maintaining the software even if it gets discontinued), the existence of a public community of users of that product.

It should also be noted that most of the software considered in chapters 2 and 3 is developed by software vendors rather than by volunteers. Most organizations that use this kind open source software also pay for support or training for their employees or similar professional services: this is one of the main business methods that allow a company to make profit from the open source software it develops.[10]

While source code may be available for free, and that was a deciding factor in this case, companies rarely reap this reduced cost benefit directly: if they wanted to use an open source software without paying for support or training they are usually

allowed to do so, but their employees have to learn to use the product, configure it, deploy it, maintain it, and this soon becomes more expensive than paying someone else who is already experienced with that software to do it.

In this case, instead, the ability to learn how such software works – or the general concepts and protocols, which may be even more useful than experience with a specific software – with an hands-on approach was considered an advantage rather than a disadvantage.

Even if paying for commercial support from a vendor would have been outside of what the team could afford, many SaaS solutions exist and would have been easier to deploy: these were also excluded since costs were still often very high, targeted at medium to large businesses rather than a small no-profit organization with nonexistent budget for this project.

1.4.2 Methodology

Ultimately, this is a problem of selecting an appropriate OTS (off-the-shelf) software product for integration: that is, an existing software product that is generally available to the public. While te more common acronym is COTS (commerial off-the-shelf), only free and open source products have been considered, so the OTS acronym has been preferred throughout the document.

Selecting and integrating such components is rarely a linear process, since many trade-offs exist between available features and other properties, for example documentation, ease of use, software architecture, community, etc.[11]

Compare this with the many existing procedures and methodologies to formalize the process of building custom software (e.g. waterfall, agile, scrum, etc.): these usually assume that features can be planned and then built according to the plan.

Even if the plan changes along the way as the software project progresses, it is still true that features should be created as needed. When reusing existing software components, some features may be missing and require changing the requirements rather than implementing the feature, or may exist but work differently from what was planned, again requiring a change to the requirements or to other components.

This is also the case for open source components: availability of the source code and the possibility to modify it is more of a guarantee, rather than a necessary aspect in the process. In fact, especially in small projects that use open source software components, it has been shown that developers rarely need to or want to modify the source code.[12]

A better approach is to consider simultaneously requirements, available components, their features and architecture.[11] Often this involves going back and forth to evaluate or analyze different aspects or different components, as one sees fit.[12]

While the process of selecting a LDAP server (chapter 2) has been presented in a mostly linear manner to make the results more clear, in fact it hasn't been a linear process: every available LDAP server that has been considered was discovered, tested and evaluated in a different order, sometimes considering more than one of them at a time, searching for comparisons, reading the documentation, going back and forth between every step and different software to better compare them.

This is also true for the selection of a SSO server (chapter 3), and even more visible: the chosen SSO was replaced midway through with another one, since more relevant aspects were evaluated while installing and configuring it that swayed the decision.

The considerations are also in part applicable the task of building a custom application (chapter 4), where some features could be built as required, while other were influenced by the features and architecture of available libraries and the SSO server.

Chapter 2

LDAP server

2.1 Comparison of available software

Four main contenders in the category of open source LDAP servers were located, these are:

- OpenLDAP
- ApacheDS
- OpenDJ
- 389DS

The requirements for this projects are simple and satisfied by all these servers: possibility to define a custom schema, support for TLS, support for group membership through the `MemberOf` attribute or similar, ability to restrict user permissions.

However, there are more aspects to consider other than features. A comparison with different categories was made and to each software 0 or 1 point(s) were assigned for each category. Finally, a sum has been computed for each software. Table 2.1 summarizes this comparison, while the following sections explain in greater detail how the score has been assigned.

2.1.1 Community of developers

All projects are actively maintained. OpenLDAP is maintained by the OpenLDAP Foundation, ApacheDS by the Apache Software Foundation, 389DS by Red Hat, Inc. For OpenDJ, however, the situation gets more complicated.

The last company to maintain OpenDJ close-sourced it in November 2016 and rolled back the publicly available version. The project was subsequently forked

Category	OpenLDAP	ApacheDS	OpenDJ	389DS
Community of developers	1	1	0	1
Community of users	1	1	0	1
Documentation	1	0	1	1
Performance and reliability	0	1	1	1
Compatibility	0	1	0	1
Final score	3	4	2	5

Table 2.1. Comparison of open source LDAP servers

by the community[13]. Actually, two main forks emerged: WrenDS, which at the time of writing seems rather lacking in activity and contributions[1], and OpenDJ maintained by the Open Identity Platform Community[2]. The latter seems to be still actively maintained[3] and it is the only version that has been considered in the comparison.

It should be made clear that this category compares how many resources are allocated to each project development: having a large community or a small number of employees working on the product is better than a small number of volunteers working in their spare time, due do the larger amount of work that can be done by a large number of people or by employees. While it can be argued that OpenDJ still has a large community of users, it has fewer developers than in the past and is fragmented between different forks and the now closed source version.

2.1.2 Community of users

All projects have a mailing list for users where they can ask questions and get an answer from fellow users or developers. However, the mailing list for OpenDJ seems to be shared among all products forked by the Open Identity Platform Community and, more importantly, the archives seem to be inaccessible[4]. A chat is available

[1] In June 2019, the last code contribution was listed in August 2018, with only two commits related to documentation done after that: `https://github.com/WrenSecurity/wrends/`

[2] `https://www.openidentityplatform.org/`

[3] `https://github.com/OpenIdentityPlatform/OpenDJ/`

[4] The link provided on their website, pointing to `https://groups.google.com/d/forum/open-identity-platform`, ends up in a page with an error message telling the user that they are not authorized. Last checked in June 2019.

on Gitter[5] but activity seems a bit lacking, since it contains exactly 8 messages in total since its creation in March 2018.

Another interesting aspect that could be considered is who (which people, organizations, etc...) use which product on which scale.

If a product has a large user base or is integrated in other projects it may be an indication of its reliability, ease of use or or other features and, more importantly, the more users there are, the higher the chance is that any bug that is encountered has already been reported and a workaround or a fix is available. Moreover, if any error that cannot be solved trivially by reading the error message is encountered, there's also an higher chance that someone has already encountered the error and a solution is available somewhere. The benefit is that searching for an answer on the web, on the issue tracker or in mailing list archives is much faster than posting a question or a bug report and waiting for an answer.

For example, ApacheDS is also integrated into WSO2 IS to provide a default LDAP server, and OpenDJ is integrated into Gluu but they seem to maintain yet another fork.[6]

The INFN (Istituto Nazionale di Fisica Nucleare) has a large deployment of 389DS, which in 2015 was comprised of 18 instances connected in different ways (master-master and master-slave replication, database chaining) and 33521 entries.[14]
389DS is also used as a component in FreeIPA and its commercial counterpart, Red Hat Identity Manager.

Just from the sheer number of third-party tutorials that are available, it is safe to assume that OpenLDAP has a large user base.

2.1.3 Documentation

Another important point that should be considered is quality of documentation[12]: even if commercial support and training is available for some of these products, the budget for this project is nonexistent. Publicly available documentation is then very important, since will be one of the few available references that will be used to configure, deploy and maintain the software.

Both OpenLDAP and OpenDJ have a manual maintained by their volunteers.

ApacheDS has some guides, however some parts are missing, redirecting to an error page or to a page containing just a "TODO" text. For example, all subsections of

[5] `https://gitter.im/OpenIdentityPlatform/OpenDJ`, last checked in June 2019

[6] `https://github.com/GluuFederation/gluu-opendj/`

section 4.1.2 of the Advanced User Guide[7] give an error page or an empty page, while sections 4.2.5.1, 4.2.5.3 and 4.2.5.4 end up in a "TODO" page. This has been checked both in March 2019 and in June 2019 and no change has been observed, meaning that this is just not a temporary error. With the help of the Internet Archive Wayback Machine[8] it is also possible to notice that the situation was the same in 2013, for example. Although not all these sections are relevant in our use case, the state of the manual does not inspire confidence.

389DS has a collection of tutorials on its website.[9] Moreover, it is the "upstream" project for Red Hat Directory Server, a commercial LDAP server, which has a few manuals and guides[15][16] that also apply to 389DS, since the difference between products is minimal. This is also pointed out on 389DS website.

Third-party documentation, e.g. tutorials available on the Internet, has not been considered for this comparison, since these may be outdated and often they are too simplistic and geared towards getting the software to run as quickly as possible, without providing many of the details that an official manual or guide can provide.

2.1.4 Performance and reliability

Performance is not very important in this scenario, since there will be around 50 users at most and around 100 entries at most in the directory: all these servers should be able to handle it. However, reliability is a more important concern and some insight into that aspect may be gained from performance considerations. In fact it is very hard do find evidence on which software is more reliable, other than a few anecdotal and opinion-based posts on the web – actually, it is impossible to the point that nothing else was found – but there is some hope to find a benchmark that directly compares performance, since that kind of data is easier to measure and more objective.

The only benchmark that could be found compares OpenDJ, OpenLDAP, ApacheDS and a commercial version of OpenLDAP[17]. The author claims that "All applications are optimized, configured, tweaked and tuned for maximum performances". In the first test, performed by attempting a large number of concurrent logins with "100 000 users in the LDAP server", OpenLDAP crashed at 40 concurrent logins.

While which of these servers is faster is not relevant in our comparison, the results are still interesting: not because we expect to reach high numbers of concurrent logins, but because it hints that OpenLDAP is not very reliable. Granted, these

[7]`https://directory.apache.org/apacheds/advanced-user-guide.html`

[8]`https://web.archive.org/`

[9]`http://port389.org`

were extremely stressful test conditions, beyond what we will ever reach in production, but this evidence, combined with the existence of commercial versions of OpenLDAP and the existence of at least one fork of OpenLDAP geared toward increased performance in high load situations and code quality improvements[10] may raise questions about the overall quality and reliability of OpenLDAP.

As for 389DS, for which no benchmarks or reliability metrics could be located, the existence of the deployment at INFN[14] could indicate that it will be reliable enough for this project, too.

2.1.5 Compatibility

The LDAP server software will be deployed on a CentOS virtual machine on a hosting provider, since that's one of the cheapest options available and we already have experience managing CentOS servers. Moreover, the deployment and configuration management should be automated as much as possible through Ansible (see section 2.3.1 for more details) since, again, the team already has experience with that configuration management tool. In fact, "developer familiarity"[12] is usually a relevant factor when choosing OTS software components to integrate into open source projects, especially in smaller projects where time and human resources are limited.

Both OpenLDAP and 389DS are available in standard CentOS repositories or in the EPEL (Extra Packages for Enterprise Linux) repository maintained by the Fedora project. There are many Ansible playbooks written by third parties that allow to deploy and configure them.

However, with the release of Red Hat Enterprise Linux 7.4 in August 2017, the OpenLDAP package has been deprecated. Since CentOS is based on the open source parts of Red Hat Enterprise Linux, it is possible that future versions of CentOS may not include OpenLDAP packages or it may work in a less than optimal way: for this reason no points were assigned to OpenLDAP in this category of the comparison.

There are no packages available in the standard CentOS repository nor in the EPEL for ApacheDS and OpenDJ. However they are Java applications, that are mostly self contained and mainly depend on the availability and installed version of the JVM and JDK on the system, and for this reason they should still be easy enough to install and maintain.

While no up-to-date Ansible playbook to deploy ApacheDS could be located, writing one according to the installation procedure in the manual proved to be quite

[10] `https://github.com/leo-yuriev/ReOpenLDAP`

simple. The resulting playbook has been published under an open source license.[11]

Doing the same for OpenDJ proved more difficult, however success was still achieved and a playbook has been published.[12]

Up to around 2015 there was an official playbook to install and configure OpenDJ, however it was retired in favor of a deployment strategy based around containers (e.g. Docker containers). After the fork, the Open Identity Platform Community seems to want to continue on this route, by providing an official Docker image[13]. Moreover, Gluu also manages OpenDJ with Docker containers. Given our limited experience with containers, compared to the configuration management one, and the fact that support for installations directly on the OS has been basically phased out in favor of containers, no points will be assigned to OpenDJ in this category.

2.1.6 Comparison results

389DS is the clear winner in this comparison, then ApacheDS comes in as a close second, with OpenLDAP and OpenDJ behind. It should be noted that the comparison is based on our particular use case and requirements. For example, if a very large deployment was needed and commercial support was considered instead of documentation, the results may have been significantly different.

Moreover these results should be taken with a grain of salt, since assigning arbitrary scores to an arbitrary number of categories can be done in many other ways that may provide any number of different final scores. However, testing the installation of all these software components and reading anecdotal opinions provided a summary that agrees with the final score: 389DS is the best compromise between ease of use and support for our other tools and software components and it has good documentation, ApacheDS was very easy to deploy and configure but the manual really seems a bit lacking compared to the other choices, OpenDJ is fragmented between too many fork with the original product having gone closed source, and OpenLDAP, while being very easy to install and configure, is a better choice on Linux distributions other than CentOS.

2.2 Designing the directory

Now that the software has been chosen, it is possible to design the directory: that is, the schema and structure.

[11] `https://github.com/lvps/apacheds-test`

[12] `https://github.com/lvps/opendj-test/`

[13] `https://hub.docker.com/r/openidentityplatform/opendj`

2.2.1 Structure

The first step is to identify which data should be stored in the directory: users, groups and service accounts.

A very simple structure has been chosen, similar to the default example structure provided by 389DS: `dc=weeeopen,dc=it` will be the root suffix since the team owns the weeeopen.it domain – `dc` stands for "domain component" – and every service, including the directory itself, will be located in a subdomain.

Then, only three container entries are created under the root suffix:

- `ou=Groups,dc=weeeopen,dc=it`
- `ou=People,dc=weeeopen,dc=it`
- `ou=Services,dc=weeeopen,dc=it`

These will contain entries that represent groups, user account and service accounts, respectively.

The former two container entries come by default with 389DS if the user chooses to create example entries during setup, the third one is similar to these for uniformity. The "ou" attribute, which is the only significant attribute located in these container entries other than the ACI attributes (see section 2.4), comes from the `organizationalUnit` class. These attribute and object class are often used for container entries even if they are not exactly organizational units like "finance" or "human resources" are. Choosing any other object class or any other attribute for the DN would have been the same, from a technical point of view.

Service accounts have been separated by users so as to prevent anyone from signing in to the SSO server with these accounts, and to prevent them from showing up in users lists. This mostly reduces the clutter in users list, but should also improve security, since these accounts are one less method that an attacker may use to gain access to the system.

2.2.2 Schema

The next step is to decide which kind of information to store in the directory and which attributes should be used to encode it: that is, to design the schema.

One of the possible approaches for this task is starting from which applications will need to access the directory, which pieces of information they need and finally map this information to standard or custom LDAP attributes and object classes.[16]

For each one of the applications that will need to access the LDAP server, both directly to read or modify attributes and through the SSO server to obtain claims, a list was made with the pieces of information that they need for each user.

These considerations apply mainly to users, since service accounts only need a bind DN and a password, and groups only need a name and a members list: they are trivial to design and implement.

- weeelab
  - Username
  - Nickname(s)
  - Name
  - Surname
  - Student ID number (matricola)
- Telegram bot
  - Telegram ID
  - Username
  - Name
  - Surname
  - Group (for authorization when accessing administrator functions)
- Tarallo
  - Username
  - Group (for authorization when accessing administrator functions)
- Pasta
  - Name
  - Surname
  - Student ID number (matricola)
  - Degree course
  - Date of birth
  - Place of birth
  - Mobile phone number
  - Student or PhD student (the template is different)
  - Date of the test on safety
- NextCloud

  - Username
  - Name
  - Surname
  - Group (for authorization when accessing some shared files and directories)
- DokuWiki
  - Username
  - Group (for authorization when accessing specific pages)
- Gitea
  - Username
  - Group (for authorization when accessing administrator functions)
- Other information, not mandatory but potentially useful
  - State: account activated or not, locked or not locked, something similar, for increased flexibility when creating accounts (so they can be created but activated at a later time) or to "soft-delete" accounts if it may be needed to recover them later
  - List of team Telegram groups that the user has joined: this could be represented with LDAP groups, and the bot could be potentially enhanced to manage this information

Every user will also have an hashed password, although no application has access to user passwords and no application should normally authenticate users by performing a bind operation, that is only done through the SSO server.

No data should be readable anonymously, users should be able to view their own personal information and edit some relevant parts of that, applications generally should have read-only access and only to the data they need.

The SSO server should have access to username and groups only, since that's the only information that needs to be provided through claims to applications that don't support user provisioning. The custom applications will be enhanced to add support for user provisioning.

An application to allow users to edit their personal information is needed, and in fact it has been created as part of this work. The application is described in more details in chapter 4.

It is easy to notice from the list of information above that many of such pieces of information are "duplicated" across applications: one of the goals of the entire

project is to remove such duplicates, by moving them to the LDAP server and using it as the only source of information.

Some users, which are referred to as “HR managers”, should have permissions to manage other users: that is, to control who is in which groups, to edit some personal information if the user so requires and similar tasks. For example, name and surname should be editable only by an administrator: if users are allowed, they may temporarily replace their name and surname with someone else and impersonate other users in some applications, even though they cannot gain the permissions and group membership of those users just by changing their name.

The next step is to decide which permissions to grant to each user, group and application. This has been done in table 2.2, for the main pieces of information. The other, potentially useful ones, will not be considered from now on since they are not needed in the initial deployment and most of them can be added easily later on, or may be reconsidered when the software and infrastructure that needs them is actually built and deployed.

Piece of information	Write permission	Read permission
Username	Registration	Self, Managers, Applications
Name	Registration, Managers	Self, Managers, Applications
Surname	Registration, Managers	Self, Managers, Applications
Student ID number	Registration, Self, Managers	Self, Managers, pasta
Degree course	Registration, Self, Managers	Self, Managers, pasta
Nickname(s)	Self, Managers	Self, Managers, weeelab
Place of birth	Registration, Self	Self, pasta
Date of birth	Registration, Self	Self, pasta
Mobile phone number	Registration, Self	Self, pasta
Date of the test on safety	Managers	Self, Managers, pasta
Telegram ID	Self, Managers	Self, Managers, bot
Groups	Managers	Self, Managers, Applications

Table 2.2. Read/write permissions for each piece of information stored in the LDAP server

Where “Applications” appears in table 2.2, it means that more than an application needs a specific permission: refer to the previous list for the applications that need access to which pieces of information.

“Registration” means that this information is written during the registration process, by the application that is detailed in chapter 4. “Self” means that users can

view or edit that information, through that application. This was ultimately decided to be possible only through the service account for the application, but letting user accounts have these permissions was also a possible choice, for more details see section 4.4.

Managers, which means “HR Managers”, don’t need access to some personal information of the users, but it may still be added in the end: managers print and generally handle the paperwork with these information, so they could just read it from there anyway.

These pieces of information need to be mapped to LDAP attrbutes and their respective object classes. This has been done in table 2.3. Enforcement of the permissions is described in section 2.4 instead.

Piece of information	Class	Attribute
Username	inetOrgPerson	uid
Name	inetOrgPerson	givenName
Surname	inetOrgPerson	sn
Student ID number	schacLinkageIdentifiers	schacPersonalUniqueCode
Degree course	weeeOpenPerson	degreeCourse
Nickname(s)	weeeOpenPerson	weeelabNickname
Place of birth	schacPersonalCharacteristics	schacDateOfBirth
Date of birth	schacPersonalCharacteristics	schacPlaceOfBirth
Mobile phone number	inetOrgPerson	mobile
Date of the test on safety	weeeOpenPerson	safetyTestDate
Telegram ID	telegramAccount	telegramID
Groups	telegramAccount	telegramNickname

Table 2.3. Pieces of information mapped to LDAP attributes

`inetOrgPerson` is one of the LDAP standard classes[18]. It is commonly available by default with LDAP servers and 389DS makes no exception.

`schacPersonalCharacteristics` and `schacLinkageIdentifiers` are already existing classes which are usually not available by default in LDAP servers. They are instead part of SCHAC (SCHema for ACademia). See section 2.2.3 for a discussion of why they were chosen.

`weeeOpenPerson` and `telegramAccount` add the few attributes that could not be represented with the default and SCHAC classes.

2.2.3 Custom classes

Rather than defining a custom class for all the attributes that aren't available in `inetOrgPerson`, a decision was made to use other common classes – although not standardized in an RFC or not commonly included with LDAP servers – to represent as much data as possible.

The motivation is potentially better compatibility with other software that may be encountered in future that recognizes and supports these classes. Moreover, reusing attributes that have already been defined and possibly standardized means that somebody else has already decided the best way to represent that data. Finally, this may simplify maintenance in the long term since other system administrators may be already familiar with that schema, while nobody could be already familiar with a custom schema that is used only inside our organization.

There are two commonly used, public schemas for IdM systems that also have a focus on higher education[19]: SCHAC[14] and eduPerson[15].

A simple evaluation has been performed: which one of them has most of the attributes that are need for this project? The results are available in table 2.4. Only attributes that are possibly standard across organizations have been compared, to have any chance to find them in an already existing schema.

Piece of information	eduPerson attribute	SCHAC attribute
Student ID number	eduPersonUniqueId	schacPersonalUniqueCode
Degree course		
Place of birth		schacDateOfBirth
Date of birth		schacPlaceOfBirth

Table 2.4. Available attributes suitable for our use case in eduPerson and SCHAC

As it is clear from table 2.4, SCHAC is more suitable for this use case.

It is worth mentioning that SCHAC is released already in LDIF[20] format[16]. LDIF is the standard format in which LDAP schema is defined and distributed, however the format used for SCHAC seems to be specific to OpenLDAP and could not be imported as-is into 389DS.

A conversion was necessary. The only SCHAC LDIF file compatible with 389DS

[14] `https://wiki.refeds.org/display/STAN/SCHAC`

[15] `https://www.internet2.edu/products-services/trust-identity/eduperson-eduorg/`

[16] `https://wiki.refeds.org/display/STAN/SCHAC+Releases`

that could be located on the Internet was hosted by INFN[17] but every attribute and class is missing its OID (Object Identifier), while they are available in the SCHAC LDIF file. This is supported by 389DS, but having globally unique OIDs, the ones found in the original LDIF file, is potentially better for compatibility between clients and servers.

For this reason, a new conversion of the schema has been performed. The result is a SCHAC LDIF file compatible with 389DS, containing all the OIDs from the original schema. This LDIF file has been published in a public git repository.[18]

For the other missing attributes, two different schema files have been created, with the following classes:

- `telegramAccount` and `telegramGroup`
- `weeeOpenPerson`

Attributes related to Telegram are located in a separate LDIF file and object class to favor reuse by other individuals and organizations without adding the team's specific attributes from the `weeeOpenPerson` class to their schema. To obtain a valid OID for these classes and attributes, two free methods are available: one is to register with IANA and obtain a manually assigned OID under their arc[19], the other is to generate an UUID and use it as an OID under the `2.25` arc provided by ISO and ITU-T for this purpose.[21] The latter requires no registration and no human interaction from third parties.

The main disadvantage of the second method, compared to the first, is that generated OIDs are very long since UUIDs are 128 bit in length[22] and some old software may impose a limit on OID length, despite no such limit existing in any specification. The latter method was still chosen, since it is faster to implement given the lack of human interaction. If incompatible software is encountered, it is still possible to switch to IANA-assigned OID with minimal modifications to the schema.

An UUID had to be generated: `4bdd6dd3-b842-4f9c-9f3f-0178cb980e52`

Then it was converted to the OID format[20], obtaining an OID arc for the WEEE Open organization: `2.25.100841824846419382782883384063386193490`.

[17] `https://wiki.infn.it/cn/ccr/aai/doc/objectclasses`

[18] `https://github.com/weee-open/schema`

[19] `https://pen.iana.org/pen/app`

[20] Tools are available to perform this task, e.g. `https://misc.daniel-marschall.de/tools/uuid_mac_decoder/index.php`

Two branches were created according to common practice[16]: `.1` for attributes and `.2` for object classes.

`weeeOpenPerson` is a structural class which specifies `inetOrgPerson` as its superclass, and it provides the `safetyTestDate`, `degreeCourse` and `weeelabNickname` attributes.

`telegramAccount` and `telegramGroup` are auxiliary classes, since other entities may have a Telegram account or represent a Telegram group. `telegramAccount` contains the `telegramNickname` and `telegramID` attributes.

All attributes are optional rather than mandatory: this is also a best practice which makes the schema more flexible.[16] All attributes except `weeelabNickname` are single-valued, since more than one value doesn't make sense. Multiple nicknames in weeelab have always been possible, so the tradition has been carried over.

The `schacUserPresenceID` attribute has been considered as an alternative to `telegramNickname`, however `schacUserPresenceID` has its equality syntax defined as `caseExactMatch`: that is, case-sensitive. Telegram nicknames, however, are case-insensitive, so `telegramNickname` has been defined to use `caseIgnoreMatch` equality.

All these schema files have been published in the same public repository.[21]

A few more attributes have been defined and are visible in the repository but will not be described here: these are not used yet and may be subject to change in the future.

Finally, some tests have determined that UUID based OIDs are supported without any apparent restriction or problem on 389DS, Apache Directory Studio and the PHP LDAP library. No other LDAP client or server implementations have been tested.

2.3 Configuration management

One of the objectives of this project was to automate configuration management as much as possible: for this task, Ansible was chosen, given the experience of some team members with such tool. Moreover, to make it easier to test configuration changes in a development environment, Vagrant has also been used.

[21] `https://github.com/weee-open/schema`

2.3.1 Ansible

Ansible[22] is an open source tool for software configuration management, app deployment and other related tasks.

A few key concepts in Ansible are modules, roles, playbooks and tasks.

A playbook is a text file in YAML[23] format that defines a sequence of tasks that has to be performed. Task definition is declarative, but dependency management is imperative: tasks are executed in the order they appear in the file.

Each task specifies a module name and some parameters. Ansible connects via SSH to the remote server or workstation to configure, copies the entire module there and executes it with the specified parameters. Finally, it reads the module output to potentially perform some other actions, for example interrupt the playbook execution if the task failed.

Most of the modules available by default are idempotent[23]: this means that a module can be executed one or more times, with the same parameters, and the final state of the system will be the same. For example, a module that copies a configuration file for a software to a remote server, will add the configuration file if it doesn't exist, or will perform no action if the file already exists and has the same content, or will update the file if content differs: the final state is always that the file is present on the target server and has the correct content.

To avoid lengthy repetitions of the same tasks across different machines it is possible to create roles, which are a collection of tasks and other data (files, templates, variables, etc.). These can be applied to a machine in a plyabook. For example, a playbook may apply the "web server" and "database server" roles to a machine.

Ansible – and configuration management tools in general – is useful in reducing the number of manual tasks that have to be performed and providing some form of documentation on the infrastructure. Documentation that is machine-readable and always reflects the actual state of the infrastructure if no manual modifications are performed. In general these tools are also useful to manage a large number of machines with similar or identical configuration, which however is not relevant in the considered scenario.

[22] `https://www.ansible.com/`

[23] `https://yaml.org/spec/1.2/spec.html`

2.3.2 Vagrant

Vagrant[24] is an open source tool to create, configure and manage virtual machines or containers in development environments. It allows, for example, to define a set of virtual machines and their operating system and IP address, then it will automatically create them according to requirements. The operating system disk images used to initialize virtual machines are provided by volunteers or organizations and are available in a public repository.[25] Other private repositories exist, but the public one was enough for this project.

Once a virtual machine has been started, Vagrant supports multiple configuration management tools to configure the operating system and install the required applications. Ansible was chosen for this task. This combination allows to quickly test modifications to Ansible roles and playbooks in an environment that closely resembles the production servers.

Moreover, since Vagrant automates most tasks related to a virtual machine, it is possible to destroy and recreate a virtual machine quickly to restart the configuration process from a clean state.

These advantages that come from combining Vagrant and Ansible were immensely useful to quickly test changes and revert them, potentially more than offsetting the time that was spent to configure the environment and playbooks.

2.3.3 Choosing a role

To leverage previous work done by other people as much as possible, a decision was made to search an existing role to deploy and configure 389DS before attempting to make a new one.

Indeed, some roles to install and configure 389DS already exist. A comparison similar to the one in section 2.1 has been done in table 2.5, considering some key features or important aspects of the role.

Specific features were used instead of generic categories because, as the comparison has shown, there is no single role that provides all the relevant features. As a baseline, all considered roles should be able to install 389DS and configure some necessary parameters of the software, for example the default suffix of the directory, so these haven't been included in the comparison.

Roles in the table are named according to the GitHub username or organization name of the repository owner, plus the role used by INFN which is not hosted at

[24] `https://www.vagrantup.com/`

[25] `https://app.vagrantup.com/boxes/search`

GitHub. This was done because the role themselves had similar or identical names.

Feature	net2grid[26]	neoncyrex[27]	colbyprior[28]	LennertMertens[29]	INFN[30]
Available on Ansible Galaxy	1	1	0	1	0
Last commit less than 1 year ago[31]	0	0	1	1	1
Configures replication	0	1	1	0	0
Configures TLS	0	0	1	0	1
Configures MemberOf plugin	0	0	1	0	0
Configures custom schema	0	0	0	0	0
Final score	1	2	4	2	2

Table 2.5. Comparison of Ansible roles to deploy and configure 389DS

A brief description of what each feature means and why it is important follows.

Available on Ansible Galaxy This means that the role is available in a public, official repository, Ansible Galaxy[32]. An advantage is that Ansible provides tools to easily manage and include in playbooks roles from the Galaxy repository, but most importantly roles located there are more probably intended for general use by their authors, rather than being specific for their use case.

Last commit less than 1 year ago It is possible that the roles which haven't received recent commits are unmaintained and don't work anymore, or that may stop working in the future despite still working right now. After deployment, as many components of the system as possible should be managed with Ansible: using a role that is still maintained means that, if something breaks due to incompatible changes in a future 389DS or Ansible version, the maintainer could fix it without effort on our part. Roles that are currently maintained may go unmaintained in the future, but roles that are already unmaintained will probably stay that way.

Configures replication This is not a strict requirement, but may still be very useful: 389DS has support for replication and having a second server, replicated in

[26]`https://github.com/net2grid/ansible-role-389-ds`

[27]`https://github.com/neoncyrex/389-ldap-server`

[28]`https://github.com/colbyprior/389-ldap-server`

[29]`https://github.com/LennertMertens/ansible-role-389ds`

[30]`https://baltig.infn.it/infn-aai/389-infn-aai`

[31]At the time of writing, March 2019

[32]`http://galaxy.ansible.com/`

real time, may prove useful for failover or local caching.

If one of the servers becomes unavailable for any reason, the other one can take over and handle requests. This still requires some configuration, be it in applications, DNS or by placing a reverse proxy in front of both servers, but getting replication to work is a first step and possibly the most complex one.

For local caching, a replica could be set up on a computer inside the laboratory: other software that is running there, for example weeelab, can query that server with a very low latency since they are in the same LAN.

In general replication may have other advantages, for example related to load balancing, but given our scenario these are the only relevant advantages.

Configure TLS Since the directory will be accessible for user provisioning from software potentially located on other servers, some form of transport security is required. 389DS and all the applications and libraries considered in section 1.3.4 support LDAP over TLS, so this is the probably simplest choice.

Moreover, 389DS can be configured to reject binds if TLS encryption has not been negotiated with the client: this should increase security by mitigating the risk of misconfiguration since 389DS is aware if encryption is in place or not. With any other method, e.g. an encrypted VPN, 389DS would only receive unencrypted connections, without knowing if encryption is done elsewhere or not.

Configures MemberOf plugin The MemberOf plugin is included in 389DS but disabled by default. It makes group membership management easier: if an entry is added to a group or removed from a group, the plugin automatically adds or deletes an attribute in the entry – the MemberOf attribute – that has the group DN as a value. Most software that does user provisioning through LDAP and supports groups expects this kind of attribute, so letting 389DS manage it should make things simpler.

The plugin is actually very simple to enable, but having it already done is still simpler than adding such a feature.

Configures custom schema There are custom schema files to deploy, both made by the team like the `weeeOpenPerson` object class and made by others but not included in 389DS like SCHAC. None of the surveyed roles support importing arbitrary custom schema files.

Other considerations Both `colbyprior` and INFN roles contain some additional configuration specific to their use case, which is not surprising since both

roles are not available on the Ansible Galaxy so they are probably not meant for general use.

2.3.4 Making a role or two

A decision was made: to create a new role with all the required features. It started as a fork of `colbyprior` role, which in turn is a fork of `neoncyrex`, but ultimately it was rewritten almost entirely and a separate role for replication was made from scratch.

In order for it to be useful for other people and organizations, it was made as generic as possible and made available on Ansible Galaxy[33,34]. The source code is also available in public git repositories[35,36].

The roles still retains other features that were present in `colbyprior` role but were not useful in our case. If these could have been useful in other use cases, they were left. Instead, features specific for their use case, for example a backup script that uploaded files to a specific cloud provider, were removed. This should help in making the role more generic and reusable.

Some of the most interesting features and details of these roles, both from an operational and technical point of view, are described in the following sections.

Setting LDAP attributes

Ansible provides a `ldap_attr` module that allows to create, edit, replace and delete LDAP entries. The supported parameters are listed in the manual[37] and one of these is `state`. At the time of writing, March 2019, it supports three values: `present`, `absent` and `exact`. These will ensure – i.e. bring the entry to a final state where this condition is true – that a value for an attribute is present among others, is absent or that the attribute has all and only the specified values.

For values `present` and `absent` the module performs a LDAP compare operation internally, which was not supported by 389DS on the `cn=config` tree at the time of writing, March 2019. Support was later added in April 2019[38], but as of June 2019 a new version including that feature is not available on CentOS yet.

[33] `https://galaxy.ansible.com/lvps/389ds-server`

[34] `https://galaxy.ansible.com/lvps/389ds_replication`

[35] `https://github.com/lvps/389ds-server`

[36] `https://github.com/lvps/389ds-replication`

[37] `https://docs.ansible.com/ansible/latest/modules/ldap_attr_module.html`

[38] `https://pagure.io/389-ds-base/issue/49390`

Since most configuration operations happen on the `cn=config` tree, most Ansible roles – including `colbyprior` and `neoncyrex` roles – used another strategy: they copied a LDIF file to the server, invoked the `ldapmodify` command line utility and checked the return code. This approach does indeed work, but is hard to maintain since configuration is split between the playbook and a LDIF file and incurs in more overhead since two different modules have to be executed for each operation: the "template" module to copy the file and the "shell" module to invoke `ldapmodify`.

However, when `state` has the `exact` value, the module does not perform a LDAP compare operation; instead, it performs a search operation to read all values for that attribute and then performs add, delete or replace operations as needed. This can be confirmed by looking at the module source code[39]. Possibly for this reason it is also the only state used in the INFN role.

The main drawback is that, if an attribute value needs to be added and the previous ones preserved, this is not possible with the `ldap_attr` module alone, given this limitation. However this proved not to be a problem, since no case was encountered in any of the two roles where attributes needed to be added or deleted leaving other values as they were.

Both roles only use `ldap_attr` with `state` set to `exact`: this, compared to the LDIF file approach, reduced the overall number of tasks for the same features provided by the role. There may still be a component of "personal taste" on which of these two solutions is more elegant or better in any sense, but in my opinion this solution, despite still not being perfect, made the role more readable and maintainable.

TLS and certificate management

The process of enabling TLS requires involves two main steps: adding the server certificate and private key to the NSS (Network Security Services) database and adding some attributes and entries under `cn=config` to instruct the server to use that certificate.[15]

The second part is simple to implement with the methods described in section 2.3.4, while the first part is more complicated. The NSS database is part of the NSS libraries[40], which are open source and included in various projects as an OTS component. There are some command line tools which are part of the libraries to manage NSS databases, however no Ansible module exists for such tasks.

[39]For example, in Ansible 2.8 (which is the latest stable release as of June 2019), the module source code is available at `https://github.com/ansible/ansible/blob/stable-2.8/lib/ansible/modules/net_tools/ldap/ldap_attr.py`

[40]`https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS`

It is possible to invoke the command line tools from Ansible, which is what the `colbyprior` and INFN roles have done. However, during tests, two potential problems were noticed:

1. While the whole sequence of tasks is idempotent – if the certificate is already present in the database, the command line tools recognize it and don't add it again – Ansible may not recognize that
2. If the certificate changes, it is added again to the NSS database even if it should replace the previous one

With to regards to point 1, the process requires import into the NSS database a file in PKCS#12 format[15][24] that contains the private key and certificate. This is a temporary file created on the server, from a key and certificate provided in another format, which is deleted after being added. The final state of the process is always the same: the key and certificate are in the database, the temporary file is gone. However, on the next run, Ansible will find that the PKCS#12 file does not exist and create it again. This adds up to a counter of modules that performed changes, which is diplayed at the end of the playbook execution. Ideally, if the final state did not change, the count should be 0.

While it is clearly a not optimal solution, it is possible to add a `changed_when` parameter to any task, to instruct Ansible on when it should count the task among the ones that performed a configuration change. This can be set to `false` to ignore all changes. The advantage is that Ansible will not report spurious changes, while there are still other tasks that change the overall configuration when enabling or disabling TLS, so Ansible won't show a play with 0 changes either if the final state of the system has changed. This was not done neither in `colbyprior` nor in INFN module.

An attempt has been made at extracting the certificate serial number and issuer from the NSS database, since these should be enough to uniquely identify a certificate and its private key, and compare them with the supplied certificate and private key: this proved to be very complicated and fragile, due to the difficulty of reliably getting that information from command line tools that act on an NSS database that may be empty, contain only one certificate, or contain multiple certificates, and the difficulty of extracting such information from the supplied certificate with other command line tools.

A better alternative could be the creation of an Ansible module that handles NSS databases and exposes an idempotent interface, then the approach defined above would become more feasible. The ideal solution would probably be to allow the module to accept other formats of key and certificates and perform the conversion step to PKCS#12 format internally, to make the entire task idempotent rather than breaking it up into multiple tasks that are idempotent only when considered

in sequence.

Concerning point 2, it has been observed during tests that NSS command line tools, when the user runs a command to add a certificate that has the same nickname as another certificate already in the database, it will not be replaced. The certificate nickname is a short human-readable string that identifies the certificate and private key in the NSS database and it has to be specified in 389DS so that it can use the correct certificate and key to serve TLS connections. A fixed name was chosen in the Ansible role.

If a certificate with an existing nickname is added, NSS will compare some other attributes of the certificate, possibly the issuer and serial number or the signature, or the entire certificate: if they match, nothing is done and the operation is reported as successful. If they don't, the certificate is added alongside the old one and no error is reported.

This is not an acceptable behavior for this use case: if the two certificates have the same nickname, 389DS – or the NSS library contained within – will chose one of them according to some unspecified criteria, or at least not specified in 389DS documentation. For example, if the old certificate is expired or revoked and the new one is valid, 389DS may choose the old certificate and serve connections with an invalid certificate.

Moreover, since no error is reported, it is impossible to asses the result of the operation without checking the content of the NSS database afterward.

One possible solution is to remove all certificates with that nickname and then add the new one, but the task that imports the certificate will always report the configuration as changed, unless silenced by setting the `changed_when` parameter to false. If that is done, however, the playbook will not report any changes when a certificate has been changed, which is not a desirable behavior since it may confuse system administrators and hides an actual configuration change in the system.

The chosen solution was to add the certificate, then count the certificates with the same nickname in the NSS database: if more than one is found, they are all removed (certificates to remove can be identified only by nickname: if they all have the same name, NSS libraries will remove them one at a time) and the specified certificate and key are added again.

This has been tested with self-signed certificates that differed in signature, private key, serial number and date of issuance and expiration but had the same subject, and the process was confirmed to work correctly: changes are reported only when certificates are removed and added again, and 389DS will start using the new certificate once it is restarted, which is done automatically in the role.

Replacing and old certificate with a new one in an automated manner is important

in our use case: these roles will be used not only for deployment but also for maintenance, and a significant maintenance goal is to manage certificates through the ACME protocol[25]. This is possible for example with Ansible, which has specific modules that support such protocol, but there are also other tools available. When a certification authority emits a new certificate for the server, for example because the previous one is about to expire, tools that support the ACME protocol will retrieve it, but it has to be correctly stored into the NSS database replacing the old one for the operation to succeed. Managing this process with an Ansible role is desirable and should now be possible. This will be tested more thoroughly in the near future when a production deployment is attempted, with certificates from a public certification authority that supports the ACME protocol.

TLS enforcement

One of the purposes of enabling TLS was to enforce it on all connections to 389DS. There are two settings in 389DS that may provide this result: one to require that all connections have a minimum SSF (Security Strength Factor), a value that usually corresponds to the number of bits in the symmetric key used for TLS encryption, and one to require that all bind operations happen over a secure connections.

Each task that changes some settings in the `cn=config` tree, for example these two settings, will open a new connection from the `ldap_attr` module to 389DS. The connection parameters are set initially and are configurable with role variables. Once these two options are activated, however, no more unecrypted connections are possible: if these were used up to that point, the next task that needs to configure settings will fail.

It is not always possible to use a secure connection from the start, however: if 389DS has just been installed by the role, TLS is disabled by default since there is no certificate present in the NSS database.

Checking if 389DS has been just installed is also not a reliable operation: if the playbook execution fails after the installation, once it is executed again it will detect 389DS as already installed, despite the possibility that TLS configuration was not completed.

Moreover, since many of the options are reversible – i.e. they can be enabled or disabled and the tasks will perform the necessary operations to reach that state – it is possible that TLS has been disabled despite 389DS being already installed.

It is also not enough to check if the role has been configured to enable or disable TLS: these operations may need to be performed over TLS or not over TLS up to the point where it is finally enabled or disabled.

The chosen solution was to perform a series of tasks that, after 389DS has been

installed, check the state of these options: instead of connecting to the server to check – which may fail since it is not known if TLS will be required or not – these values are searched inside the `cn=config` database, which is just a plain text file. If TLS is enforced, TLS is enabled in the connection. If TLS is not enforced, it may also be not configured, so operations are performed without TLS: this is a very limited security risk, since the module is run on the server, so an attacker would need to capture packets on the virtual loopback interface to obtain the data – e.g. administrator bind DN and plaintext password – exchanged in the unencrypted connection.

Once TLS configuration is applied, the role will switch to operations over TLS if it needs to enable TLS enforcement, or will continue over an unencrypted connection. Since two settings may be needed to enforce TLS, two connections are needed: after the first one, unecrypted connections may be already disallowed, but when TLS has been enabled it is always possible to connect over TLS, even if not required by the TLS enforcement settings.

The role also performs a check at the beginning and fails with an error if TLS is set to be disabled and TLS enforcement is set to be enabled, or the user would lock themselves out of the server.

Automated tests

Automated tests have been employed in the `389ds-server` role. Molecule[41] was the software chosen for this task.

Molecule is a software for automated testing of Ansible roles. It works by creating a virtual machine or a container according to a definition provided by the user, provisioning it with Ansible to test the role, and possibly running some tests on the provisioned system. The provisioning part is done twice and the second run should not result in any change reported by any task, to ensure that the entire role is idempotent.

Some parts of the role still perform changes during every run (see section "TLS and certificate management"), but this fact is concealed to Ansible and the user since these are temporary changes and the role, considered in its entirety, is idempotent.

The final tests are mostly limited to checking that 389DS has been started and is listening on the correct ports in the container, since if anything failed during the provisioning an error should have been reported immediately and the process terminated: Ansible always stops the playbook or role execution as soon as an error is encountered.

[41] `https://github.com/ansible/molecule`

A container was chosen because that's the only supported option by Travis CI[42], a continuous integration tool that runs the entire test suite after every commit to the public repository.

The `389ds-replication` role has no tests because Molecule doesn't provide a way to create two containers and have them interact with each other in a manner compatible with 389DS, from what could be understood from the documentation. Having two separate instances of 389DS is a requirement to correctly test replication. It may be possible to test it with two instance on the same machine or in the same container, but this has not been attempted: configuring replication is done through some sequential steps that all require the previous one to work, compared to the multitude of different features that may or may not affect each other handled by the `389ds-server` role: it's much easier to manually test if replication is still working or not after a change by verifying that the role can be still applied without errors, compared to testing multiple features in different configurations.

The `389ds-server` role contains three different test scenarios to test various features, including TLS configuration with a self-signed certificate. Each scenario builds a container with Ansible-container and starts it with Docker, performs provisioning twice and then destroy the machine: these steps are all handled and automated by Molecule.

The tests don't cover every possible setting and execution path to avoid slowing down the test suite too much: all the successful tests took an average of 8 minutes and 7 seconds to run on Travis CI.[43] More tests can be added in future if the need arises, for example to detect regressions after bug fixes.

2.3.5 Replication

In 389DS, replication is structured as follows: a server can be a consumer or a supplier or both, in relation to any other server. When a client performs any operation that modifies data on a server, the server stores the modifications in its database and then sends out updates to all its consumer servers, which update their internal database. If consumer servers are suppliers to other server, they repeat the process of sending updates to their consumers. Replication operations are always initiated by a supplier that "pushes" its updates to a consumer, never a consumers that requests updates.[15]

There is another mode of operation – the hub – and other modes of replication – e.g. if a server contains multiple databases, it is possible to replicate only a part of

[42]`https://travis-ci.com`

[43]Raw data: 6 test suite runs, taking 7 m 55 s, 8 m 5 s, 7 m 56 s, 8 m 15 s, 8 m 5 s, 8 m 26 s

them – but these are not supported by the `389ds-replication` role or any other role that had been located.

A server that is both a supplier and consumer is referred to as a "master"[15], while a server that is only a consumer is usually referred to as a "slave" and is generally configured to reject user operations that will modify the database, since it cannot send the changes anywhere else. However, in a scenario with only two servers where one is a supplier and the other is a consumer – which could be the team's use case if a replica is hosted inside our laboratory, for increased performance and partial failover – the supplier is also referred to as a "master".

For this reason, it is my opinion that the "master" and "slave" terminology is confusing and the "supplier" and "consumer" terminology is more clear. While `neoncyrex` role uses the "master" and "slave" terminology, `colbyprior` role uses "read-write master" and "read only replica" which is more clear and has been suggested by one of 389DS developers.[44] In the `389ds-replication` role, instead, only "supplier" and "consumer" terms have been used. The term "master" has been mentioned in some comments since 389DS documentation also mentions the term in similar situations.

The replication role has been split from the main role since they are mostly independent: any other role can be used to deploy 389DS, or even a manual installation is possible, and then replication can be configured by the `389ds-replication` role, without evaluating all the configuration options provided by the `389ds-server`, which may be conflicting with the ones provided by a different role or by a manual installation. This should make the role more reusable in different situations.

Another reason is that a server could be, for example, a supplier for multiple other servers, especially in a master-master replication scenario. This requires setting up multiple replication agreements on the supplier, one for each of its consumers. Both in `colbyprior` and `neoncyrex` roles one replication agreement is set up with another server after install. To set up more replication agreements, the role has to be applied again: a flag is provided to skip all installation related tasks, since 389DS is already installed. Splitting the role in two enables the possibility to apply the installation role once and the replication role multiple times as needed, making the intent of the playbook more clear.

2.3.6 Examples

Given the difficulty of finding any third-party tutorials especially on replication, or any pre-configured development environment, another "examples" repository has

[44] `https://github.com/colbyprior/389-ldap-server/pull/1`

been created[45] with Vagrant files and Ansible playbooks to set up different 389DS configurations: single, master-master replication with two masters and "supplier and consumer" replication with two servers. The repository contains enough documentation to get the Vagrant-managed virtual machines up and running.

Most of the process is automated: a single command will bring up the virtual machines and perform all installation and configuration. Only a manual step is needed ad the end to begin replication, because when replication is first started a supplier server has to send its entire database to all of its consumers and overwrite theirs. This requires careful consideration in a production environment if a server has just been added, in order to use the production database to overwrite the empty database of the new server and not vice versa. For this reason the role does not perform this operation automatically, but the documentation for both the role and the "examples" repository explain what has to be done in order to start replication.

2.4 ACI attributes

Authorization to perform LDAP operations on 389DS can be granted with ACI attributes (Access Control Item)[15][26]. ACI attributes can be added to any entry and will be applied to any entry in that subtree. They may filter operations according to the operation itself (search, read, write, compare, etc...), the bind DN of the user performing the operation, the attributes involved in the operation and some other information: IP address of the client, time of the day, SSF of the connection, etc.

By default, 389DS does not allow any operation unless explicitly allowed by an ACI attribute.[15] There are two types of ACI: "allow" and "deny". Given the default behavior of denying everything that is not explicitly allowed, it is generally advised to use only "allow" ACI attributes.[15]

To simplify ACI management, a Python script was created, that allows to write ACI in a more structured manner and, more importantly, formats them as a list of YAML strings that can be pasted into an Ansible playbook. The script, along with the ACI attributes, is available in the same repository with the schema files.[46]

The policy is organized as follows: it gives some read permissions to system accounts for WSO2 IS, Nextcloud and Crauto (see sections 3.3.2, 1.3.5 and 4.4 for more details) so that they can read the user and group attributes they need.

Additionally, Crauto is given write permissions broad enough to also create and

[45] `https://github.com/lvps/389ds-examples`

[46] `https://github.com/weee-open/schema`

remove users, since it handles registrations and user management in general. None of these accounts has sufficient permissions to read the hashed user password. Nobody has permission to edit the service accounts in any way, including to change their password. Most permissions are given to service accounts directly since each application will use one of these accounts to perform its operations. Users always have the permission to bind on the LDAP server: this cannot be restricted and the SSO server performs a LDAP bind operation with the user bind DN and password to check that the password is correct.

It should be noted that there is always at least one special administrator account which has every permission and ignores ACI attributes: this is used to create service accounts and set the ACI attributes since no other account has sufficient permissions to do that. It is also used to apply configuration changes through Ansible. Its password is not stored in plain text on any server, since it is only needed on the workstation that runs Ansible.

Finally, a simple password policy has been defined and stored in an Ansible playbook along the task that updates ACI attributes.[47] The password policy rejects passwords that are too short and temporarily locks accounts with too many failed bind attempts, to mitigate against brute force attacks.

It is advisable to have a test suite based on the policies that the ACI should implement.[4] For this reason, a test suite for the ACI and password policies has been defined and is available in the same repository as the schema and ACI definitions.

The test suite is a Python script based on pytest[48] that connects to a 389DS instance – running on a virtual machine, but this is not mandatory – and tries to perform a series of LDAP operations with different bind DN and on different entries. There is a total of 84 tests, counting the parametrized versions of the same test as a different test.

Parametrized tests mean that a single test is repeated, in this case, with different bind DN but the same result is always expected. For example, a test checks that password change to a user account is denied: that operation is attempted from the same user account, another user account, another user account but in a privileged group, WSO2 IS user and Nextcloud. These accounts should not be allowed to change password since this is handled only by Crauto, more details are provided in sections 4.4 and 4.7.

Privileged users had specific permissions in an early version of the policy but don't

[47] `https://github.com/weee-open/sso`

[48] `https://docs.pytest.org/en/latest/`

anymore, however they have specific additional permissions in Crauto: the tests were not removed when the policy was changed since they will allow to easily catch any mistake in the future if some permissions on the LDAP server are ever added again to that group.

ACI syntax is already checked by 389DS as soon as an ACI attributes is added or updated, so there's no need to test that aspect separately.

Chapter 3

SSO server

The objective for this part of the project is to get a SSO server up and running, allowing users to perform log in both via SAML2 and OpenID Connect on two different applications. Users should be able to open the other application and get logged in without performing any manual step (e.g. providing credentials) once they log in to the first one. More applications will be added gradually after the system has been deployed to production.

To do this, a SSO server had to be selected, installed and configured.

3.1 Comparison of available software

Since support for both OpenID Connect and SAML2 is required (see section 1.3.4), there are four open source contenders in this category:

- OpenAM
- WSO2 Identity Server (WSO2 IS)
- Keycloak
- Central Authentication Service (CAS)

Feature-wise, all projects have what is required for this project: support for OpenID Connect and SAML2, and the ability to use a LDAP server as a database for claims and to authenticate users with a bind operation. Other open source SSO servers with a large user base exists, for example Shibboleth, but they only support SAML2 or support OpenID Connect through plugins and extension that are not suitable for production, yet.

Gluu was also excluded because, despite having all the required features, it is

an integrated product which embeds both a LDAP database that can be used in production and the SSO components: the ability to choose a LDAP server independently from the SSO server has been considered more important than better component integration, since it allows to limit vendor lock-in and make it easier to replace any of the two components, if needed.

A comparison similar to the one in section 2.1 has been performed, the results are available in table 3.1.

Category	OpenAM	WSO2 IS	Keycloak	CAS
Community of developers	0	1	1	1
Community of users	0	1	1	1
Documentation	1	1	1	1
Performance and reliability	n/a	n/a	n/a	n/a
Compatibility	0	1	1	1
Final score	1	4	4	4

Table 3.1. Comparison of open source LDAP servers

For OpenAM, only the Open Identity Platform fork has been considered.[1]

3.1.1 Community of developers

OpenAM was part of the product suite that included OpenDJ and has suffered the same fate, more details are available in section 2.1.1. Its forked version is still under active development by the community[2], but the same considerations from section 2.1.1 still apply and have influenced the points assigned to it for this category.

The other projects are backed by companies (WSO2 IS by WSO2, Keycloak by Red Hat, Inc.) or foundations (CAS by the Apereo Foundation): the bulk of the development work is carried out by the companies themselves or members of the foundation.

3.1.2 Community of users

CAS and Keycloak have specific mailing lists for users.

[1] `https://www.openidentityplatform.org/`

[2] `https://github.com/OpenIdentityPlatform/OpenAM`

WSO2 maintains a mailing list for their users, but the website also suggests using Stack Overflow instead, noting that they "monitor and participate in the discussion there".[3]

A few third-party tutorials exist for every product, but none has so many that it is possible to infer that the community for that software is larger compared to the others. The same problems already described for OpenDJ also exist for OpenAM (see section 2.1.2). The Gitter chat seems to be a bit more active[4] compared to the one for OpenDJ, but the community is too fragmented to grant points in this category.

Products that integrate these components have also been searched, but it was difficult to find any relevant one.

Companies and institutions using them, however, may be easier to find: since these software often have some public-facing pages, e.g. the sign-in page, it is possible to search on any search engine, e.g. Google, some text from such pages. The results have to be examined carefully to verify that the found page is actually of that software, since it may just be any other page with the same text.

This method has been applied in particular to WSO2 IS, searching some text from the sign-in form in May 2019, and it turned out 6 universities and 5 other organizations in the first two pages of results on Google. The other results on these pages were duplicates, test instances or unrelated pages. The Google search result page claims that there are 333 results in total, for the specified text. Different results are obtained when searching different text from the same page, or text from other pages (error pages, management console login page, etc...) but there are always a few universities and other types of organizations. These results should be interpreted with a grain of salt, since some of these may just be test instances not marked as such and there may be other organizations that use it but their sign-in pages are not index or not available from on the public Internet.

For Keycloak this was more difficult, since the longest text on the sign-in page is "Username or email", which is also very generic. Searching text from the Administration Console welcome page, which appears before signing in, also yielded to nothing but a single organization. Despite these not very encouraging results, Keycloak seems to have the largest number of third-party tutorials available.

For OpenAM the text on the sign-in page is also very generic and no other pages are exposed before sign-in, so it was impossible to find anything on the web.

For CAS, it is possible to assume that at least some of the organizations that are

[3] https://wso2.com/mail/

[4] `https://gitter.im/OpenIdentityPlatform/OpenAM`

part of the Apereo Foundation use that product; there are other products hosted by the foundation.

3.1.3 Documentation

All four products provide comprehensive manuals in their repositories or websites.

3.1.4 Performance and reliability

Performance has been considered only because it gives an indication about reliability, for details see 2.1.4.

However, it was even more difficult to find any information on reliability of these SSO servers than on LDAP servers. The only document that could be found was a benchmark from the same author that provided the benchmark of LDAP servers.[27] In the article, the only relevant SSO servers that are compared are WSO2 IS and OpenAM, and the author concludes that OpenAM has better performance. Both products didn't crash during the benchmark, and that is the only relevant information for this use case. Nothing could be found for Keycloak and CAS.

Given the lack of available data, it was decided not to assign any points for this category.

3.1.5 Compatibility

None of these products is available in the official CentOS repository nor in the EPEL. All of the projects are written in Java and depend on the availability of the JVM and JDK on the server.

WSO2 provides a repository suitable for CentOS that contains WSO2 IS in packaged form, and some Ansible scripts[5] to install and configure WSO2 IS. The scripts, however, are not a role: they should be downloaded, customized if needed and then executed. This increases the complexity of integrating them into other Ansible scripts and updating them when updates are committed to the repository.

The other products can all be installed like most Java applications, e.g. by extracting a compressed archive and running the launcher script, or by deploying them to Tomcat or JBoss application servers. Which methods are supported depends on the application.

[5] `https://github.com/wso2/ansible-is`

There are 6 Ansible roles available in Ansible Galaxy[6] to install Keycloak. Furthermore there are two Ansible modules included by default that allow to manage and configure Keycloak via its APIs: `keycloak_client` and `keycloak_group`.

3.1.6 Comparison results

Apart from excluding OpenAM, or at least the version from Open Identity Platform Community, for this use case the other SSO servers are comparable. For this reason, an alternative approach has been chosen: begin to configure one of these servers and go on until possible or until the development environment works correctly. If it doesn't work, switch to another one of these servers.

The development environment comprises of a Nextcloud instance, the custom user management application that is described in chapter 4, a 389DS instance with master-master replication (to test failover, if supported), and the SSO server. Access to both Nextcloud and the user management application should be possible through the SSO server, with the SAML2 and OpenID Connect protocols.

3.2 First attempt: Keycloak

The first software to be tested has been Keycloak, more specifically version 6.0.0, the most recent one available at the time of writing.

3.2.1 Installation

The first task was to make an Ansible playbook to install Keycloak inside a Vagrant virtual machine.

One of the available roles had to be chosen, as a starting point. None of them stands out for particular features since all of them cover what is needed in our use case, so one that had both an high score – Ansible Galaxy assigns a score to roles according to votes cast by users and automated checks for code quality[7] – and a documentation that looked comprehensive was chosen: `nkinder.keycloak`.[8]

A playbook was written to set some variables as described in the documentation and to apply the role and it was verified that it works as expected.

[6] Last checked in June 2019, see `https://galaxy.ansible.com/search?deprecated=false&keywords=keycloak`

[7] The scoring system is explained in greater detail in the documentation: `https://galaxy.ansible.com/docs/contributing/content_scoring.html`

[8] `https://galaxy.ansible.com/nkinder/keycloak`

The playbook gets applied by Vagrant, which manages the development environment of four virtual machines: two for 389DS, one for Keycloak and one for Nextcloud. The custom application described in chapter 4 runs on the host machine instead, to simplify development.

As usual, the playbook has been published as open source.[9] It contains a few more tasks, for example to add the self-signed 389DS certificates to Keycloak keystore so it can recognize them. This would not be needed in production with real certificates from a public and trusted CA, but adding them made it easier to test Keycloak compatibility with 389DS.

3.2.2 Configuring the LDAP backend

Configuring Keycloak to use the LDAP backend was the second task: it has been performed from the Administration Console, a web interface from which administrators can configure most settings for Keycloak, with the intention of coming up with a working configuration, exporting it and then using Ansible to apply that configuration to the production server, albeit with some tweaks, e.g. changing the LDAP server URI to the production one. It should have also been possible to apply the same configuration to the development environment, starting from a fresh installation of Keycloak, to ensure that everything worked correctly before moving on to production.

Here, the first problem was encountered: apparently, there is no way to configure Keycloak to use StartTLS when connecting to the LDAP server, despite that being the only standardized way to install the TLS layer in a LDAP connection.[1]

Keycloak instead supports LDAPS, which means that the LDAP connection is established over a TLS, rather than starting the LDAP connection first and then enabling TLS. This is supported by almost every LDAP library, client and server, including 389DS. For this reason it was not a huge problem, but using standard protocols would have been better, at least from a theoretical point of view: LDAPS has been in use since before the StartTLS extension was standardized and is not going away soon, but it is still not defined in any standard and StartTLS is possibly preferred as an alternative.

It should be noted that support has been added to Keycloak in May 2019[10], however version 6.0.0 was released in April 2019, so it didn't contain that feature yet.

Apart from this minor inconvenience, configuring Keycloak to use the LDAP server in read-only mode seems to work well.

---

[9] `https://github.com/WEEE-Open/sso/blob/master/keycloak.yml`

[10] `https://github.com/keycloak/keycloak/commit/54909d3ef4d7e36f7b37ad3647632576bb796374`

There are two modes in which Keycloak can gather data from the LDAP server. In one of these, it needs to periodically synchronize its internal users database with the LDAP user list. Keycloak contains an embedded relational database, H2, that holds the configuration for the server and the user database.[28][29] This means that, when a user is added or updated, it may take a while before Keycloak performs a synchronization and sees the updated attributes. Passwords are never synchronized and they are only validated by performing a LDAP bind operation on the LDAP server: this means that, if a password is changed, the user can use the new password to log in immediately, without waiting for synchronization, and the old password is rejected. Moreover, 389DS prevents bind operations with locked accounts, so that change is also instantly enforced by 389DS instead of waiting for synchronization.

In the other mode of operation Keycloak does not synchronize its internal database but queries the LDAP server every time.[29] However when this mode was configured it was observed that, while Keycloak does correctly allow users to sign-in, they are never shown in the Admin Console, which seems to list users only from the internal database. Enabling user synchronization ("Import Users") would solve this problem, although this is not needed due to the custom user management software that was built (see chapter 4).

Importing users could potentially speed up sign-in and other operations if the LDAP server is located on a different machine since the internal database would act as a cache. However, the LDAP and SSO server in production will probably run on the same machine, so network latency is negligible and keeping a copy of user attributes provides little to no advantage.

Having a password policy set on 389DS and most attributes provisioned through LDAP simplifies setting up the SSO server, since it does not need to perform additional validation or to handle these attributes: they are just excluded from what the SSO server is allowed to read, so it only has access to the attributes that need to be translated into SAML2 attributes or OpenID Connect claims. These restrictions are implemented through ACI attributes (see section 2.4).

Another problem is that Keycloak, as of version 6.0.0, does not support setting multiple LDAP server addresses for failover: if one server fails, there's no automatic way to instruct Keycloak to retry the operation on a replica server. This can be worked around with a reverse proxy in front of both servers, which could also run on the same machine as Keycloak and perform load balancing and provide other useful features, but this would increase the complexity of the deployment by having more software components to configure and manage.

3.2.3 Configuring the relational database

It is recommended to replace the embedded relational database with an external database in production.[28] For this task, MariaDB has been chosen: it is a free and open source relational database which is available for most Linux distributions in the official repository, including CentOS. Since the version included with CentOS is usually older than the latest stable release, there are repositories maintained by the MariaDB Foundation that provide the latest stable version for CentOS and other Linux distributions.

MariaDB was also chosen due to previous experience of the team members in configuring and managing it. Installing MariaDB and creating a database was done with an existing role, which has been added to the same playbook that installs Keycloak. In fact, MariaDB and Keycloak are installed on the same machine in development, and will be installed on the same machine in production: this is a method to keep costs down by reducing the number of machines, and to make access to the database more secure against misconfiguration or other vulnerabilities that don't require direct access to the server to be exploited, since it will be accessible only from the server itself and not from the network.

To connect Keycloak and MariaDB, a JDBC connector is required.[28] This is a software component, named MariaDB Connector/J, and it is available from the MariaDB Foundation.[11]

The only method provided by the MariaDB Foundation to install it is to manually download the file, since it does not seem to be available in the CentOS MariaDB repository at the time of writing, April 2019.

For this reason, an Ansible role has been made to automatically download a specific version of the connector and create a configuration file to load it into Keycloak, or more precisely into the WildFly application server which is used internally by Keycloak. The role has been published as open source.[12]

The playbook[13] has been modified to include that role and additional configuration required by Keycloak.

Despite this being a very simple task, the role helped a lot in quickly testing different versions of the connector, since most of the stable releases available at the time of writing, April 2019, seems to have incompatibilities or problems with Keycloak. In fact, some of them caused Keycloak to crash on startup or when performing some operations in the Admin Panel. All the errors available in Keycloak logs were

[11] `https://downloads.mariadb.org/connector-java/`

[12] `https://github.com/lvps/ansible-wildfly-mariadb-connector-j`

[13] `https://github.com/WEEE-Open/sso/blob/master/keycloak.yml`

related to the database, most of them pointing out that the the database connection had been abruptly closed, but the real cause for this event is still unknown. Nothing relevant could be found in MariaDB logs.

Searching these errors on the Internet didn't provide any definitive answer: bug reports for various other software seems to hint that some MariaDB Connector/J versions have this kind of problems with some other versions of open source components used internally in Keycloak and other projects, but no clear indication of which versions are exactly incompatible has been found. It is also not clear if this is a related problem or just another problem with similar symptoms.

Almost all versions of MariaDB Connector/J ranging from 2.2.3 to 2.4.1 have been tested, sometimes even multiple times with slight variations in configuration, without finding any conclusive answer to the issue, other than version 2.2.3 seems to work best. However, this is still far from optimal: 2.2.3 is an older version, it cannot be kept forever in production as MariaDB and Keycloak get updated, and at this time is impossible to tell if any future version of the connector will work better or not before testing it, adding more maintenance work in the future. It is true that any update to any software may introduce a bug that affects the production server, but given the fact that most newer versions of MariaDB Connector/J cause problems to Keycloak and its components, it is reasonable to assume that the next version will likely still be affected by similar problems.

An alternative would be to try another relational database, for example PostgreSQL, but the team members have no experience configuring and managing it, which would increase the work that needs to be done on their part. Another possibility is to keep using the embedded H2 database.

A third alternative would be to try out another SSO software. This would also allow to evaluate the actual complexity of installation and features of both products by comparing them.

3.3 Second attempt: WSO2 IS

3.3.1 Installation

A playbook had to be created to install and configure WSO2 IS. A decision was made to make a new playbook rather than using the official Ansible scripts, since this would be easier to integrate with the rest of the Ansible roles, and to start from a clean installation without any customization to follow the manual more closely and gain more insight into the configuration process. The playbook has

been published in the usual repository.[14]

The Ansible playbook enables the repository provided by WSO2, installs the package, creates a system user and group – as it is done in the official Ansible scripts – and a systemd service. This is enough to get WSO2 IS running.

A few configuration files are also deployed by the Ansible playbook: they disable the embedded LDAP server, configure WSO2 IS to use the external LDAP server and some other parameters, for example setting the hostname to the real value rather than "localhost".

3.3.2 Configuring the LDAP backend

Configuring the LDAP backend is done through a configuration file managed by Ansible.

WSO2 IS supports both StartTLS for LDAP and LDAPS, but neither has been configured in the development environment since configuring WSO2 IS to trust the self-signed certificates was deemed too difficult. In the development environment it will use an unencrypted connection, while in production it will be configured to use StartTLS, since the LDAP server certificate will come from a public and trusted CA.

WSO2 IS also integrates a relational database, in fact the same one as Keycloak: H2. Again, this is used to hold some settings that that are not read from the configuration files and optionally to store user account information. When a LDAP server is configured as the primary user store, which is what has been done, users are only stored to and queried from the LDAP server.

However, a decision was made to leave the H2 database as it was, since the documentation does not recommend specifically to replace it with another database for production use. This however will not provide an exact comparison with Keycloak, since it will not be tested if WSO2 IS exhibits the same behavior as Keycloak when configured to use MariaDB Connector/J.

3.3.3 Configuring the keystore

Another important part of the playbook is the creation of a separate Java keystore with a certificate and a private key. WSO2 IS, by default, uses a single keystore – containing one private key and its related self-signed certificate – both for TLS authentication when acting as an HTTP server (e.g. for the Management Console, sign-in page, etc...) and when signing SAML2 responses and other documents.

[14] `https://github.com/weee-open/sso`

Since SAML2 certificates are usually copied to clients through a secure channel and only the configured certificate is trusted,[5] a self-signed certificate with a distant expiry date is a simple solution that requires little maintenance. It has to be self-signed because public CAs rarely offer certificates that last 1 year or more, especially for free; moreover, it is useless for it to be emitted from a public CA, since relying parties aren't even required to validate or take into account the content of the certificate, except for the public key for signature validation or any other information useful to identify the correct certificate among the ones available to the relying party.

The TLS certificate, instead, if WSO2 IS is exposed directly to the internet rather than sitting behind a reverse proxy, needs to come from a public CA in order to increase security, since browser cannot confirm the signature of self-signed certificates.

A certificate from a public CA is signed by possibly some intermediate CA certificates and those are ultimately signed by the root certificate of a public CA that the browser trusts: in this way it is possible for the browser to validate the server certificate and confirm that it is not forged and has not been tampered with. For this reason, a certificate from a public CA is needed, and these usually have a relatively short validity time, then they have to replaced with a new certificate, possibly issued by the same CA. This process will be automated in production as described in section 2.3.4, but using the same certificate to sign SAML2 responses would increase complexity since all SAML2 clients need to be updated as soon as the WSO2 IS certificate is replaced. Using only a self-signed certificate would require a reverse proxy in front of WSO2 IS to server HTTP request with a certificate from a trusted CA, managed by the reverse proxy: this would also increase complexity, since there would be one more software – the reverse proxy – to configure and maintain.

A simpler solution is to configure WSO2 IS to use two keystores with two different certificates and private keys: one for TLS, the other to sign SAML2 responses and other documents. This requires just some initial setup to create the keystores and configure WSO2 IS to use a specific keystore for each task, but once initial setup is done only the TLS keystore needs to be updated when a new certificate is obtained from the CA, without changing any setting on WSO2 IS or other applications. In the development environment both certificates are self-signed, but in production the TLS certificate will come from a public CA.

Finally, it should also be noted that the default keystore is available in the public repository along with WSO2 IS source code. For this reason it must be replaced for production use.[30]

In the development environment, new certificate and key pairs have been generated

and committed to the repository, but for production these will not be used: instead, new ones will be generated and not committed to any public repository, since the private key has to stay private.

3.3.4 Configuring clients

One client had to be configured to use SAML2 and one to use OpenID Connect, to experiment with both protocols and their configuration. For SAML2, Nextcloud was chosen and configured for both single sign-on and user provisioning as described in section 1.3.5. For OpenID Connect, Crauto was chosen as the test project, see chapter 4 for more details.

This was a matter of configuring a few settings for each client (service provider) in the Management Console. Configuring Nextcloud required a little back and forth to obtain a working configuration, but fortunately both Nextcloud and WSO2 IS logs were detailed enough to understand what had to be corrected in the configuration on either side.

While this section should be the central part of this work, surprisingly there is very little to say: it worked as intended. When attempting to sign-in to Nextcloud or Crauto, the user is redirected to WSO2 IS and presented with either a sign-in form or immediately redirected to the service provider if the session is still valid.

This is now a complete SSO system, for which deployment is mostly automated. Nextcloud and Crauto installation is still manual but out of scope for this project, configuring service providers is still manual albeit not very complicated and could possibly be automated, the rest is completely automated even though just for a development environment.

3.4 Final considerations

3.4.1 Single sign-out

While the original scope was to stop at single sign-in, having single sign-out to work is very desirable in this use case, so at least an attempt was made. First, normal sign-out had to be enabled.

Various sign-out processes are defined for both SAML2 and OpenID Connect. The one that are relevant in this scenario are the ones where a service provider application initiates the process.

This is done by terminating the application session and redirecting the user to a SSO sign-out page, so the SSO software can terminate its session with the user.

Optionally, the user is redirected again to an application page that informs them that log out has succeeded. This has been tested and works on both applications.

However, some problems have been encountered when trying to set up *single* sign-out. When a users log out from one application, they should be logged out from other applications: every application should support single sign-out for this to work and usually the SSO server cannot guarantee that the process has been completed correctly, e.g. because one of the applications may be temporarily offline so it cannot process the sign-out request.

As part of single sign-out, tests performed with Crauto have shown that OpenID Connect refresh tokens are not revoked once sign-out has been performed. The server can apparently continue to issue new refresh tokens and ID tokens until the refresh token expires. However, the user is correctly logged out: attempting to sign-out again fails with WSO2 IS claiming that there's no active session, which is to be expected, and if another sign-in is attempted to any application WSO2 IS presents the user with a sign-in form, which is the correct behavior.

The same test has been performed with Keycloak, and it has been shown that it invalidates the refresh token upon sign-out: if a refresh is attempted with a token from a session that has been terminated via sign-out, an error is returned. This feature could make the single sign-out process more robust, if configured in a specific way. The gist is to issue ID tokens with a short validity period and refresh tokens with a longer validity period. At the cost of increased load on the SSO server due to the refresh requests to process more often, when a user logs out there's an higher chance that the session is terminated on every application: if one application could not perform single sign-out, e.g. because it is temporarily offline during sign-out or because it simply does not support single sign-out, when it tries to refresh the ID token it receives an error and should terminate the application session.

Moreover, while WSO2 IS supports various single sign-out mechanisms both for OpenID Connect and SAML2, cross-protocol sign-out is not supported in version 5.8.0, which is the latest stable release at the time of writing. This means that if a user signs out from a SAML2 application, OpenID Connect sign-out is not performed, and vice versa. There is an open issue[15] on the official WSO2 IS repository. At the time of writing, June 2019, the issue has been added to the "5.9.0-GA" milestone, implying that this feature will be implemented in the 5.9.0 release.

3.4.2 Session management

Users may have multiple sessions active on the SSO server and with applications, for example if they signed in with their smartphone and with a computer.

[15] `https://github.com/wso2/product-is/issues/3090`

Moreover, they may terminate these sessions without signing out, for example because they configured their browser to delete cookies when it is closed and they close the browser. In that case the session is still active, but all cookies stored in the browser that linked it to that user session are lost. If applications use other mechanisms to store session data, it may be more or less difficult to terminate the session on the client side without notifying the server.

Users should be able to view a list of their sessions that the SSO server still considers active: this allows users to terminate sessions for which they no longer have client-side data, or in general any session that they no longer want to be active but for which they have no access, e.g. if they signed in from a browser on a workstation in the laboratory and then left without performing sign-out, they may use their smartphone to access a session management panel and perform this operation. This is also a security measure: if credentials get compromised for some users, they may e.g. change their password, and then check that there are no unrecognized active sessions, which may have been started by an attacker, and terminate them if there are any. If a list of previously active sessions is maintained, it would also be possible to check if an attacker has gained access to the account through compromised credentials or not.

Both WSO2 IS and Keycloak provide user management panels, where users can change their password, edit their profile and view active sessions.

In Keycloak, this panel allows users to view their currently active sessions and terminate them. It is only possible to terminate all sessions instead of only one of them, but this shouldn't be a problem. There is no functionality to view past sessions, which is a limitation. There are APIs available to implement these same functions into other custom software, for example Crauto.

In WSO2 IS, this feature is also available in the panel, but it gathers its data from the Analytics Server[30], which is a component of WSO2 IS that has not been installed in the development setup. The information should be available in the internal relational database, but there are no APIs to query such information, except for the ones provided by the Analytics Server. This is a problem, since installing and configuring the Analytics Server would greatly increase complexity of the deployment.

While this features is somewhat important, it is not immediately required for the initial deployment since the system does still work without it: the decision between installing the Analytics Server or replacing WSO2 IS with another SSO server has been postponed.

3.4.3 Adaptive authentication

Another interesting aspect can be considered: WSO2 IS supports adaptive authentication. That is, the SSO server may decide which authentication factors to require from users during sign-in based on a set of policies. For example, if users sign-in at an unusual time of the day, the SSO server may require a second factor of authentication.

Support for adaptive authentication is very limited in Keycloak 6.0.0 since custom rules be implemented only by writing a custom plugin[16] or by enabling a preview feature that is not ready for production, yet.

WSO2 IS also supports step-up authentication, which is a form of adaptive authentication: that is, users may sign-in with their username and password only and then, when they access for example the administration panel of a SSO-enabled application, the application may instruct the SSO server to ask for a second factor of authentication. This functionality could be useful in some of the team's custom software case and will be considered for future implementation.

[16]e.g. `https://qiita.com/naokiiiii/items/c5bc7ed2204507f4ad6a`

Chapter 4

User management application

Another component is still needed: a user management application.

The SSO server doesn't have access to all the attributes stored in the LDAP server that other applications may need, so its integrated user management tools aren't sufficient for this task.

It would still be possible to assign these permissions to the SSO server, but since the chosen SSO server may be changed in the future, the effort of building a simple user management panel tailored to the use case requirements may pay off, allowing to choose any other SSO server without considering the quality and features of its user management panel which is not going to be used. Moreover, this is going to make that potential migration more transparent to users and HR managers, which will see a different login page but still the same management panel.

Additionally, the registration process has some specific requirements that go beyond a simple registration form, and these are:

- Accounts should be created as locked, with the `nsAccountLock` LDAP attribute set to true, to prevent logins until the account is activated by an HR manager
- Account creation should be possible only through an invite link which is personal, only one account can be created through the same invite link
- Some form fields should be pre-populated with data that is already available from the team recruitment process, for new member's convenience

Given these requirements, which will rarely be satisfied in their entirety by a user management component in a SSO software or any standalone user management software, a decision was made to create a new software to handle registrations.

The user management part was also deemed useful enough to be done, since it would require limited effort if built in the same application that handles registrations. However, there are also more requirements for the user management part:

- Users should be able to see all their personal information and possibly download it for GDPR compliance
- Users should be allowed to edit only some of their personal information
- Users in a special group (HR managers) should be given permission to lock and unlock accounts and edit some personal information
- HR managers should be able to reset other users' passwords
- Users should be able to change their own password
- HR managers should be able to download the filled SIR form for a user. This could be done through another application that leverages the Pasta software, but integrating it in the user management software will make the onboarding process even simpler

The software, named "Crauto" after a poll among team managers, has been published in a public git repository.[1] A description of the architecture and features the software follows.

4.1 Programming language and frameworks

Making such a software was also a chance to try to integrate a custom software component with the SSO server and LDAP server. Most of the team's custom software is written in Python or PHP: using one of these languages could help in testing and better understanding what will need to be done in the future to also integrate these applications.

The existing Python software will need to be integrated with LDAP only (Telegram bot, weeelab) or leverage the SSO protocols as a way to authenticate for API access to one of the PHP applications (Tarallo). Instead, one of the PHP applications (Tarallo) will need to authenticate users with the SSO server in a manner similar to what Crauto should do, but it has no need to access the LDAP server.

For this reason a decision was made to use PHP: connecting Tarallo to SSO had a higher priority than the other software that has been mentioned, since it would simplify user management compared to the the "as is" situation and testing out PHP

[1] `https://github.com/WEEE-Open/crauto/`

libraries for SAML2 or OpenID Connect would be a relevant step in the direction of integrating Tarallo with the SSO process.

Additionally, since Nextcloud had already been connected with SAML2, a decision was made to use OpenID Connect, to test the configuration of that protocol both from SSO and from an application side.

Moreover, since Tarallo is built without a framework but a set of different libraries, another decision was made to use no framework for this project, to exclude any OpenID Connect library that depends on a specific framework.

For this reason, a deliberate attempt as been made at keeping the project as simple as possible.

4.2 General structure and libraries

The application comprises of a few public-facing pages, some classes that provide core functionality, a some templates and a few other files (configuration files, static assets like CSS files, etc.).

Each page contains some business logic and a reference to a template that renders the HTML. Templates are reused across pages, this is why splitting them from the page itself has been considered useful.

Pages are placed in a directory accessible by the web server: when it receives a request for that file, it invokes the PHP interpreter that executes the file and returns its output, which is usually an HTML page. The web server then returns that output to the client as normal.

This was an area were the simplest approach has been deliberately chosen. Most large PHP projects have a router class, collection of classes, function or script, to which every request is sent: this enables the software to handle requests for any resource even if it does not directly map to a file on the web server. While many libraries exist to create routers and most frameworks offer that functionality, this was deemed an unnecessary complexity for this project, since there are no requirements to serve pages that don't clearly map to a file.

The main drawback is that, without additional configuration on the web server, no web pages with a different name than a file existing in the web server directory can be served. This may result in some unsightly URI paths which are visible in the browser address bar, e.g. `/people.php?uid=example.user`. However, the elegance of these paths was not a concern for an application that will be rarely ever accessed by regular users.

To give a coherent structure to the project, the `pds/skeleton` specification 1.0.0[2] has been followed. It specifies, among other things, which directory may be used for files that can be served by the web server, and which ones should be used for other types of files. By restricting the web server to serve a single directory in the project clients cannot access configuration files, templates and classes directly. Accessing these files usually has no effect in any case, but this completely eliminates the risk of a misconfiguration that may accidentally print out the contents of a configuration file, for example.

Despite no frameworks being used, some libraries are still used: Plates[3], a PHP library for templating, and Bootstrap[4], an HTML, CSS and JS library for the user interface. To manage these libraries and their dependencies, the Composer dependecy manager has been chosen.[5]

For LDAP, the LDAP library that is part of the PHP core distribution was used. It offers some PHP functions that are wrappers around the C functions provided by the OpenLDAP client library.

For OpenID Connect, a library had to be chosen since it would simplify the development compared to implementing the client part from scratch. In particular, the client part involves validating the cryptographic signature of different tokens and their content[6], which is a task best left to an already existing, open source and well-tested library, to avoid introducing security vulnerabilities.

The OpenID Connect web site lists some of the compatible libraries[6]. For PHP, there are four open source libraries listed at the time of writing, June 2019:

- `phpOIDC`[7]
- `OpenID-Connect-PHP`[8]
- `oauth2-server-php`[9]
- `Drupal OpenID Connect Plugin`[10]

---

[2] https://github.com/php-pds/skeleton/tree/1.0.0

[3] http://platesphp.com/

[4] https://getbootstrap.com/

[5] https://getcomposer.org/

[6] https://openid.net/developers/libraries/

[7] https://bitbucket.org/PEOFIAMP/phpoidc/src/default/

[8] https://github.com/jumbojett/OpenID-Connect-PHP

[9] https://bshaffer.github.io/oauth2-server-php-docs/

[10] https://www.drupal.org/project/openid_connect

The Drupal connector was ruled out since it is a plugin for the Drupal CMS. `oauth2-server-php` only implements the identity provider (SSO server) part of the specification, not the relying party (a client application) so it is not suitable for Crauto.

`phpOIDC` is the only certified library among these, but the readme file points out that “the focus was on the interoperability, less focus was given to the security issues”, negating one of the potential advantages of using an existing library. Moreover, it seems to require a database even for the relying party implementation, which would make Crauto more complicated to deploy and manage.

`OpenID-Connect-PHP` instead is much simpler and requires no database, so that was the chosen library.

Implementing the authentication code flow was done as follows: the “Crauto” service provider was created in WSO2 IS, the client key and secret were randomly generated by WSO2 IS, and a few lines of code were written to call `OpenID-Connect-PHP` library functions, provide them the key, secret, server URI and a few other parameters. The library then proceeds to perform the authentication code flow, with some specific steps to make it work in PHP:

1. Generate “nonce” and a “state” parameters[6] and store them in a PHP session
2. Prepare a request with these and any other needed parameters
3. Redirect the user to the OpenID Connect authorization endpoint on WSO2 IS
4. User performs authentication and consents to send claims to the Crauto application
5. WSO2 IS redirects to the same page that the user came from, because the library has been configured to supply that URI to WSO2 IS
6. The user is sent back to the page, with a query parameter that contains the authorization token appended to the URI
7. The same code path is executed, until the point where the library checks if an authorization token is present among the query parameters: before it was not present so it went on to generate nonce and state and redirect the user, now they are present, so instead it makes a request to the token endpoint to exchange the access token for an ID token. The request is done through a “back channel”: that is, the request is prepared and sent from PHP without involving the user agent
8. Check that the stored “state” parameter matches the one present in query parameters along with the authorization token: this prevents some attacks where a malicious user injects a valid authentication token from another request in

the redirect, or unsolicited responses with forged tokens

9. Decode and parse the JWT ID token (values are stored in a PHP array) and validate its signature and contents according to the specification. The nonce parameter, which provides protection against replay attacks[6], is also checked, which is a good thing despite its use being optional in the OpenID Connect specification
10. The validated token is stored in the object where the `authenticate` function has been called

From the point of view of the programmer, if the code before the `authenticate` function doesn't have any side effects, it can be thought as executing it once and no ID token exists before `authenticate` is called while one is available later, although due to redirects the code before `authenticate` function is actually executed twice.

This approach proved to be very simple to integrate in an application. After the authentication step is completed, Crauto reads the parsed ID token and stores some of its attributes in a PHP session for later use: the user ID, full name and groups – which are all claims coming from the LDAP server through the SSO server – the expiration time, the refresh token and a copy of the ID token itself since it is required for sign-out.

The sign-out part does work in a similar manner, so it is not interesting to describe here. It should also be noted that, while sign-out works, single sign-out does not work yet: see section 3.4.1 for more details.

The refresh token part is more interesting and will be described in section 4.3.

4.3 Refresh token issues

One of the methods to handle the ID token is to consider a user authenticated until that token is valid or until a sign-out request has been received. If the token expires, the application should request a new ID token to the SSO server: this way, the SSO server has a chance to ask the user for its password or other authentication factors if required by policy, or to reject the request if the user has signed out but no single sign-out request was sent to the application, for example because the application does not support single sign-out.

There are multiple methods to perform this operation: one of them is to perform a redirect to the SSO server. If all goes well, the SSO server immediately redirects the user back to the page where they were without asking for credentials, since their session is still valid. Another is to call such an endpoint with a special parameter that instructs the SSO server to skip the prompt for credentials if possible according to its policy. This could be done with an HTTP request from the client every once

in a while, and if it fails – i.e. the SSO server wants the user to provide credentials – the normal authentication flow is performed. This is often done in single-page JS applications which have no back-end. Finally, refresh tokens can be used: if the back-end application has such a token, it can use it to request a new ID token. If the session is not valid anymore, e.g. because the user has performed single sign-out, the server should respond with an error. If the session is valid, the SSO server responds with a new ID token and also a new refresh token.

WSO2 IS allows to configure a time after which refresh tokens expire, even if the SSO session is still valid, to reduce the risk of a stolen refresh token being reused multiple times by a malicious user, even though in normal conditions in Crauto, and in any application that uses refresh tokens, these tokens are stored on the server but never visible to users.

The `OpenID-Connect-PHP` library provides support for refresh tokens: there is a `refreshToken` method that can be used to request an authorization token from a refresh token. The refresh response may contain a new ID token and not just an authorization token[6] and WSO2 IS implements it in this manner. However, `OpenID-Connect-PHP` returns the complete response that also contains the ID token, but does not perform any validation on the ID token, to the best of my understanding. There are also no public methods to invoke validation functions on a given ID token. From the documentation and examples available on the web it is not clear how this process should be handled with that specific library, either.

A workaround has been found, although quite hideous in my personal opinion: the library `authenticate` method can be configured to perform Implicit Flow Authentication, in which it receives the ID token directly inside a query parameter and validates it. After inspection of the library, it has been determined that it reads the “state” and “nonce” parameters from the current session and query parameters from the `$_REQUEST` superglobal array.

`$_REQUEST` is set at the beginning of the execution of the PHP script with the parsed request parameters, but it can be modified later, and this just what is done: these parameters are added to the superglobal array and the others to the session, then the `authenticate` method is invoked to perform an implicit flow, then these temporary parameters are deleted. This surprisingly worked, even multiple refreshes have been attempted in sequence and all of them correctly obtained a new ID token and refresh token. The validation performed on the ID token is the same as in the Authentication Code Flow, although there is no nonce. This should not be a security concern, since the nonce is used to prevent unsolicited responses which may come in the form of requests performed by an user agent: the ID token, here, comes as a direct response from a back channel request to the SSO server, which is impossible to hijack in this manner.

However, the entire refresh process still feels brittle and, in a sense, wrong. It has been useful to test how refresh tokens work, but it may be removed altogether and refresh tokens disabled for this application before moving to production.

4.4 Service accounts and permissions

Some choices had to be made: does Crauto authenticate against the LDAP server with its own service account or with the user account? For which operations: reading the users list, users editing their own profile, HR managers editing another user, changing password, registering new users?

The main drawback of a service account is that its bind DN and password have to be stored in plain text on the server. They can be stored in a manner not accessible by clients, but if an attacker gains access to the server they may obtain these credentials and use them to authenticate against the LDAP server, while if user accounts are used the attacker would have to capture that information as it is used, possibly gaining access only to a limited user account and not to a service account with broad permissions.

It should be noted, however, that the permission to edit any user account would either be given to HR managers – they don't have the target user password which is not stored in plaintext anywhere, so they cannot use that account to perform the operation – or to the service account. An attacker that gains access to an HR manager account would have permissions similar to the ones of the service account.

Moreover, Crauto still needs the plaintext password – either for users or for the service account – to perform a bind operation against the LDAP server. An alternative which leverages the SSO server for increased security against this specific vulnerability would be to never access the LDAP server directly, but send updates to the SSO server for example through the SCIM protocol. However, some form of authentication is still needed so that the SSO server may recognize that the SCIM request is coming from a trusted source: this requires a plaintext secret of some sort (password, token, TLS client certificate and key, etc.) on the Crauto server, which yields similar security vulnerabilities as a service account on the LDAP server.

Another possible solution is to perform requests to a SSO server API with some form of authentication that is done on a per-user basis and leave to the server the responsibility to check if the operation is authorized or not. This could be implemented with OpenID Connect or the OAuth2 "three-legged" process with Authorization Code Flow, which is the basis for OpenID Connect Authorization Code Flow. For example, when the user tries to edit another user's profile, they may be redirected to the SSO server for authentication, perform that step or just see some redirects happen if the SSO server decides to skip it (see section 1.3.2), get sent back to Crauto that then performs the rest of the Authorization Code Flow,

to obtain the ID token (OpenID Connect) or the access token (OAuth2). Then it could use the received token as a means to authenticate against the SSO server APIs, for example with the SCIM protocol, and perform the required operations. The token is still stored in plain text on the Crauto server for some amount of time[11], but it should have a very short validity time, so as to mitigate the risk of an attacker obtaining it. Refresh tokens should not be available, in such a scenario. The problem with this approach is that authorization is delegated to the server and, at least in the SSO software considered in this project, this seems to complicated, requiring modifications to the SSO server with custom plugins and extensions.

For example WSO2 offers SCIM and non-standard API and accepts three types of authentication[12]: HTTP basic, TLS client certificate and OAuth2 access token. Moreover, the SCIM 2.0 protocol is supported and it offers a specific endpoint for users to modify themselves. Finally, WSO2 IS offers more granular permissions, so it is possible that certain endpoints can be restricted to only certain users. However, the server does not perform any validation on the attribute values: the API would still need to be restricted so that only the Crauto server can access it to reduce the risk of a malicious user sending a request from elsewhere with invalid attributes or attributes that cannot be normally edited by users. It is also possible to extend WSO2 IS to perform validation of the parameters, but this is a very complicated task and making any modification, even as a plugin, to an OTS product always incurs in the risk, however small, that it may break upon a new release of the software, requiring constant maintenance.

For these reasons, direct access to the LDAP server was preferred. However, the choice between a service account and a user account still has to be performed.

To obtain a list of users and their attributes, and even for attributes related to the current user, a service account is probably the best choice: performing a bind operation with the users' account would mean that Crauto has to require the user password on each page since LDAP connections are closed automatically by PHP after every page script terminates or that the server has to keep a copy of the user's plaintext password to perform a bind operation every time it needs to. Even if the plaintext password is discarded upon sign-out, this is still a plaintext password stored on a server for a significant amount of time, especially if a user forgets to sign-out.

Editing user information and requiring a password is bad user experience: users

[11]The user ID token used to sign in to Crauto is also stored on the server as part of the session, but it is useless to access other applications or to perform more operations with the SSO server, since its "audience" is restricted to Crauto

[12]`https://docs.wso2.com/display/IS580/Authenticating+and+Authorizing+REST+APIs`

have already logged in through OpenID Connect, why should they type their password again? It is possible to have Crauto ask them so authenticate again with the SSO server if the ID token is too old, but asking for user password every time is annoying for users and increases password fatigue. However, a service account for this task would need to have very broad permissions, basically to edit every attribute that an HR manager is able to edit.

Another point to consider is that, if user accounts have permissions to perform “modify” operations or read attributes, they may connect to the LDAP server with their own client and perform such operations without any input validation. This can be mitigated by allowing bind operations only from some machines by checking the client IP addresses, employing a VPN, etc... Finally, the SSO server will always have the permission to perform binds with the LDAP server using user accounts, so if it gets compromised then the plaintext user passwords may be obtained by an attacker that can then bind with the LDAP server and perform any operation without validation. This can be prevented only if user accounts have limited permissions and a service account for Crauto is used.

Ultimately, the method of constantly asking for user password to perform this kind of operations is uncommon in regular web applications and whether to give more permissions to Crauto or SSO server is a matter of where more risks should be concentrated, since they cannot be reduced: in a small and simple application (less code usually results in less bugs) that however has been developed by a single person, or a large application that has accumulated improvements for years and has been scrutinized by hundred of developers but is very large and complex?

There is no easy answer to this question, but another aspect may be considered: flexibility. If the SSO software is replaced in the future, the chosen solution should be as much as decoupled as possible from the server to help that; even though OpenID Connect, OAuth2 and SCIM are standard protocols, a decision may be made to migrate to a SSO server that does not support SCIM, for example.

Ultimately, a decision was made to use a service account for all operations, with a caveat for password change described in section 4.7: it is difficult to say if and which one of the two methods carries more risks, however this was the easiest one to implement – especially from the point of view of writing ACI attributes (see 2.4), which should reduce the risk of misconfiguration – and the one that provided better user experience in every scenario.

4.5 User input validation

One interesting point to note is that, while for relational databases that support SQL queries are usually prepared as strings which are then passed to a library

that handles communications with the database, the PHP LDAP library for some operations requires an array with the data.

This is true for all "modify" operations on attributes, i.e. add, remove and replace, for the "compare" operation, and to add and remove entries. The advantage of this approach is that it completely prevents without any effort on the programmer's part all LDAP injection attacks[31], which are similar to the more common SQL injection attack. In a SQL injection attack a malicious user may provide an especially crafted input to the server and, if the input is not escaped correctly, it may change the semantics of the query where it is placed, since the query is considered just as a string for which the application server does not know the semantics.

Instead, if the operation has to be encoded as an array, there's no way for a malicious user to provide a crafted input that changes the array structure or affects other array keys and values. The library that handles such an array, then, can escape any parts that need to be escaped, or encode them in a manner that does not influence the semantics of the operation before sending it to the LDAP server. For this reason, no escape function exists in PHP to sanitize user input for these specific operations on attributes and entries, since it is not needed.

However, it should be noted that the LDAP search operation requires a "filter", which still has to be built as a string and sent to the LDAP library as such. For this reason search operations are potentially vulnerable to LDAP injections, but care has been taken to employ the `ldap_escape`[13] standard function, which escapes any search parameter in a way that allows them to be inserted in a search filter.

Finally, if DNs are built by string concatenation with components taken from user input, they are vulnerable to LDAP injection, too: however, building DNs is bad practice (see section 1.3.1) so it hasn't been done, except for creating a new entry. For the latter case, `ldap_escape` has been used since it supports escaping parts of a DN.

Care has also been taken to escape all user-provided data displayed in every web page, including data stored in the LDAP server or coming from OpenID Connect claims. If not sanitized, a malicious user may craft a string that includes in the page a remote JS file, allowing them to perform an XSS (Cross-site Scripting) attack[32] or other types of attacks that involve the injection of arbitrary HTML.

The data has only been inserted in HTML parts of the page, never in JS parts, so the escaping has been done through a Plates library function which in turn calls the `htmlspecialchars` PHP standard function. This is sufficient to prevent any HTML injection.

[13] `https://www.php.net/manual/en/function.ldap-escape.php`

Extensive validation has been done on user input on the server side, including white list validation for input that allows it, e.g. country of birth (which can be selected from a list) and degree course (which can also be selected from a list). After registration this kind of information can be modified freely, although it is still escaped where required, since that data needs to be accurate and come from a white list only between user registration and when the account is unlocked, since in that period the SIR is generated.

Moreover, most of that data cannot be modified by users but only by HR managers. The choice to give a free text input to managers has been done for flexibility: for example, if a new degree course is created and somebody from that course joins the team, managers can manually type the name of the course without editing the list of available courses, which may not be possible in the short term since it requires direct access to the files on the server and HR managers generally don't have that kind of access.

Validation on the client side has been considered only as a convenience, to warn users that their input is incorrect before sending it to the server, never as a security measure.

4.6 Registration and invite code

Registrations should be allowed only when the URI contains an invite code and some parts of the form should be pre-populated with data that is already available – name, surname, student ID – to provide a better user experience. This data comes from the "recruitment form", yet another custom application that can be extended to have access to the LDAP server.

Even more useful is that the Telegram ID and nickname fields are pre-populated: however, for this to work, the new member has to contact the Telegram bot which can in turn obtain the Telegram ID for the user and store it somewhere.

A method to pre-populate the fields has to be decided. Various choices are possible.

One of them would be to add the values in a query parameter of the registration page URI, possibly encoded. The recruitment form and the Telegram bot may use different query parameters. For this to work, a link has to be generated and then provided to the Telegram bot by the user: the bot adds its query parameter and returns the updated URI.

Another possibility is to add a LDAP attribute that holds both the invite code and the other data and the two applications append their data. The attribute should be multi-valued (i.e. more than one attribute of that type can be added to the entry) and stored in a single place, e.g. in the Crauto system account entry.

Another possibility is to create LDAP user entries, each with an invite code and only the available data, in a specific container entry.

Finally, regular user entries could be created with available data and the missing data added during registration.

All these choices have advantages and disadvantages: the query parameter solution does not need to give write permissions on the LDAP server to the bot and the recruitment form and could even be made completely stateless if the invite code is some form of signed token that can be verified by Crauto, instead of being stored somewhere in its exact form.

The single attribute solution would require more permissions but provide shorter URI and would not produce a different URI to add data: this could potentially less confusing for users, e.g. if they already have the page open they could just refresh it.

The LDAP user entries in a specific container would require less broad permissions, since the bot and the recruitment form can have write permissions only on the attributes that they need to write and in a container entry that is only used during "recruitment", instead of on the entire attribute with all the data. This could be relevant if one of the applications is compromised, however even having full access to these attributes does not pose a security risk: they are only used to pre-populate a form.

Creating regular user entries is the simplest solution, but the most difficult to implement in a secure, since these applications need write access to entries in the regular users subtree and other applications may not be correctly configured to ignore these accounts.

Ultimately, a decision was made to use LDAP user entries in a separate subtree from regular users. While this has some disadvantages compared to the query parameters method it provides a slightly better user experience.

Another container entry, `ou=Invites,dc=weeeopen,dc=it`, was added along the ones described in section 2.2.1. Another custom schema object class has been created: `inviteCodeContainer` with the `inviteCode` attribute: this is an auxiliary class which is added to "invited user" entries that have regular user classes, so any attribute could be potentially added. Since these are in a separate subtree from regular users, however, most applications are not allowed to search or view them and they are not considered in users lists.

The DN component of the invited user entries is `cn` because it is a mandatory attribute for the `weeeOpenPerson` class since it is mandatory in its `inetOrgPerson` superclass.
The resulting DNs are similar to `cn=John Doe,ou=Invites,dc=weeeopen,dc=it`.

It would have been possible to use `inviteCode` instead, however since it is bad practice to build DNs (see section 1.3.1) a search would still be necessary to obtain the DN from the invite code: the CN attribute was chosen since it may be required to edit these entries by hand during testing and knowing immediately which person they refer to is more useful. Moreover, this prevents accidental creation of duplicate invites for the same person. Invites are short-lived, around 24 hours at most, so the odds of two persons with the same name and surname that are joining the team at the same time is very low and, since these entries can be edited manually by connecting to the LDAP server in such a case, this risk was deemed insignificant.

Finally, attributes are read from the invite entry and used to pre-populate the registration form. If there is no Telegram ID among the attributes, a warning message is displayed telling the user to contact the bot before filling the rest of the form. Only a subset of attributes is considered, all other attributes are discarded.

After a regular user account has been created as part of the registration process, the invite entry is deleted. The operation is not atomic, both because 389DS does not seem to support LDAP transactions and because the risk connected with someone exploiting this is very low: accounts still need to be activated by an administrator and account creation is confirmed out-of-band with the new member, so any account that shouldn't exist can be manually deleted.

4.7 Password change

Users are allowed to change their password. Moreover, HR manager are allowed to change users password: this is required initially since there is no other method for password recovery planned in the short term: when such methods are implemented in a future version, this functionality may be removed, although leaving users the possibility to change their own password, obviously.

The change is performed with the system account to eliminate the risk of a malicious user obtaining the password and gaining direct access to the LDAP server and updating the password, effectively bypassing any second factor of authentication if set up in the SSO server.

When users are changing their own password, even if they are HR managers, the previous password is asked and used to perform a bind operation: if the bind succeeds, the password is updated. This was done to minimize the the time between the last password check and the password change, eliminating the risk of an attacker finding a signed-in account and changing the password.

It was possible to evaluate the time passed since last authentication and ask the SSO server to authenticate the user again, however that would not minimize the time as much as submitting the old and new password in the same HTML form,

and would require to store the new password somewhere in plaintext while the authentication redirects are happening. That is: if the user submits the form and the authentication is deemed too old, the server has to store the new password, redirect user to authentication and then when the user returns, recover the new password and perform the update. The plaintext password cannot be kept in memory only as the PHP script terminates when the user is redirected to the SSO server and another script is started when the user returns and accesses a page: it can be stored in a PHP session, which ultimately ends in an unencrypted file on the server disk. It could be deleted as soon as the operation is completed, which may take a few seconds, but it cannot be securely wiped.

The redirect to SSO has to be performed when the new password is submitted, not when loading that page, or a malicious user that finds a computer with an account logged in, even if a long time has passed and that would trigger Crauto to redirect to the SSO server for authentication, may send a request directly to that page, bypassing any authentication. This can be done for example with the developer tools that are available in any modern browser.

For these reasons, the approach of asking for the old password was chosen: the old password is only ever stored in memory, it is used only to bind with the LDAP server to check that is valid and discarded as soon as the script ends. The new password is only sent to the server inside a LDAP replace operation over a secure TLS connection.

The plaintext of the new password would need to pass through Crauto in any case since it has to perform the update, even with the previously mentioned SSO authentication mechanism. Even letting the SSO server perform the password update would still allow Crauto to view the plaintext of the new password, unless the SSO server APIs are called from a client-side script running in the user browser. However, for security considerations of letting users access that kind of API, see section 4.4.

Chapter 5

Conclusions

The objective of setting up a SSO system that supports both SAML2 and OpenID Connect has been reached.

The onboarding process, that lead to the decision to implement the SSO system, has been greatly improved: with a centralized accounts database and user registration form, a large number of manual steps have been eliminated. Moreover, with a true SSO solution rather than simple centralized authentication with LDAP, the usability and overall security of the system should be improved.

Some work is still needed to integrate Pasta into Crauto and to integrate the Telegram bot into the process, however this is outside the scope of this document and SSO in general.

The system also needs to be deployed into production and more applications need to be connected. Deployment is planned for the short term, since minimal modifications are required to convert the Ansible playbooks from the development environment to production environment. Applications will be gradually connected once this is done, phasing out all the local user databases.

In the medium term, one important feature that will be configured and enabled will be two-factor authentication with OTP. In the long term, adaptive authentication and step-up authentication will be considered to again enhance the usability and security of the SSO system.

While designing the directory and developing the user management was a straightforward process, selecting a LDAP server and a SSO server was a far less linear process, which is to be expected when selecting and integrating OTS software components. In particular, both the SSO software that have been tested have advantages and disadvantages that were discovered while testing them and it is not even clear which one will be the better one for the described use case.

Another important result has been the creation of Ansible roles to manage 389DS: while they are only a part of the full SSO system, it is planned to maintain them indefinitely by providing bug fixes and possibly new features, for the benefit of the open source community in general.

The decision to split the LDAP and SSO server and decouple them as much as possible was ultimately useful, allowing to change the SSO server without significant modifications to the rest of the system, both during this part of development and possibly in the future if another SSO server is chosen.

Automation tools, namely Vagrant and Ansible, were also very important in reducing the overall effort to test different aspects of the system and different configurations and in quickly bringing the system back to a clean and known working state when needed. Their usefulness should not be underestimated when integrating OTS software products, due to these advantages.

Bibliography

[1] J. Sermersheim, "Lightweight Directory Access Protocol (LDAP): The Protocol." RFC 4511, June 2006.

[2] T. A. Howes, "The Lightweight Directory Access Protocol: X.500 Lite," Technical Report 95–8, University of Michigan Center for Information Technology Integration (CITI), Ann Arbor, Michigan, July 1995.

[3] C. Hall, "LDAP Authentication & Authorization Dissected and Digested." `https://thecarlhall.wordpress.com/2011/01/04/ldap-authentication-authorization-dissected-and-digested/`.

[4] A. Findlay, "Best Practices in LDAP Security," in *LdapCON*, 2011.

[5] "Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite," Standard sstc-saml-metadata-errata-2.0-wd-05, OASIS, 2015.

[6] N. Sakimura, J. Bradley, M. B. Jones, B. de Mendeiros, and C. Mortimore tech. rep.

[7] C. Ng and K. Englert, "The difference between iam's user provisioning and data access management."

[8] P. Hunt, K. Grizzle, M. Ansari, E. Wahlstroem, and C. Mortimore, "System for Cross-domain Identity Management: Protocol." RFC 7644, Sept. 2015.

[9] P. Hunt, K. Grizzle, E. Wahlstroem, and C. Mortimore, "System for Cross-domain Identity Management: Core Schema." RFC 7643, Sept. 2015.

[10] D. Mainardi and G. Macario, "L'impresa innovativa e open source." Seminar with slides available at `https://www.dropbox.com/s/q2uccngn0tjwykr/LIeOS.pdf`, Oct. 2016.

[11] L. Brownsword, T. Oberndorf, and C. A. Sledge, "Developing new processes for cots-based systems," *IEEE Software*, vol. 17, pp. 48–55, July 2000.

[12] T. R. Madanmohan and Rahul De', "Open source reuse in commercial firms," *IEEE Software*, vol. 21, pp. 62–69, Nov. 2004.

[13] "Let's fork the ForgeRock suite." `http://www.timeforafork.com/`.

[14] D. Maselli, "INFN-AAI Stato dell'infrastruttura centrale." Workshop INFN CCR, 2015. `http://www.lnf.infn.it/~dmaselli/share/INFN/AAI/AAI%20-%20stato%20infrastruttura%202015.pdf`.

[15] M. Muehlfeld, P. Bokoč, T. Čapek, P. Kovář, and E. Deon Ballard, *Red Hat Directory Server 10 Administration Guide*. Red Hat, 2019.

[16] M. Muehlfeld, P. Bokoč, T. Čapek, and E. Deon Ballard, *Red Hat Directory Server 10 Deployment Guide*. Red Hat, 2019.

[17] The HFT Guy, "LDAP Benchmark: OpenDJ vs OpenLDAP vs Symas OpenLDAP vs ApacheDS." `https://thehftguy.com/2015/10/05/ldap-benchmark-opendj-vs-openldap-vs-symas-openldap-vs-apacheds/`.

[18] M. C. Smith, "Definition of the inetOrgPerson LDAP Object Class." RFC 2798, Apr. 2000.

[19] F. Tröger, "A Reference Schema for LDAP-based Identity Management Systems," in *LDAPcon*, 2007.

[20] G. Good, "The LDAP Data Interchange Format (LDIF) - Technical Specification." RFC 2849, June 2000.

[21] "Information technology – procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers," Recommendation ITU-T X.667, ITU's Telecommunication Standardization Sector (ITU-T), Geneva, CH, Oct. 2012.

[22] P. J. Leach, R. Salz, and M. H. Mealling, "A Universally Unique IDentifier (UUID) URN Namespace." RFC 4122, July 2005.

[23] J. Geerling, "Ansible for devops," 2014.

[24] K. Moriarty, M. Nystrom, S. Parkinson, A. Rusch, and M. Scott, "PKCS #12: Personal Information Exchange Syntax v1.1." RFC 7292, July 2014.

[25] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic Certificate Management Environment (ACME)." RFC 8555, Mar. 2019.

[26] A. Findlay, "Writing Access Control Policies for LDAP," in *UKUUG Spring Conference*, 2009.

[27] The HFT Guy, "IAM Benchmark: OpenAM vs WSO2 vs SimpleSAMLphp vs Shibboleth." `https://thehftguy.com/2015/10/14/iam-benchmarks-openam-vs-wso2-vs-simplesamlphp-vs-shibboleth-2/`.

[28] "Keycloak Server Installation and Configuration Guide." `https://www.keycloak.org/docs/6.0/server_installation/index.html`.

[29] "Keycloak Server Administration Guide." `https://www.keycloak.org/docs/6.0/server_admin/index.html`.

[30] "WSO2 Identity Server Documentation." `https://docs.wso2.com/display/IS580/WSO2+Identity+Server+Documentation`.

[31] "LDAP Injection." `https://www.owasp.org/index.php/LDAP_injection`.

[32] "Cross-site Scripting (XSS)." `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`.