Karlsruhe Institute of Technology

Certificate-based OpenSSH for Federated Identities

Master's Thesis
submitted by

Lukas Brocke

to the KIT Department of Informatics,
Steinbuch Centre for Computing (SCC),
Department Data Analytics, Access and Applications (D3A)

Reviewer:	Prof. Dr. Achim Streit
Second Reviewer:	Prof. Dr. Bernhard Neumair
Advisors:	Dr. Diana Gudu, Dr. Marcus Hardt

29. March 2023 – 29. September 2023

I declare that I have developed and written the enclosed thesis completely by myself and have not used sources or means without declaration in the text.

Karlsruhe, 29.09.2023

.....

(Lukas Brocke)

Abstract

Despite being the most widely used Secure Shell Protocol (SSH) implementation, OpenSSH only supports a very limited number of authentication mechanism including passwords, public keys, and Kerberos. The extension of OpenSSH with support for authentication using federated identities addresses the security risks associated with password-based authentication, prevents cumbersome management of public keys, and streamlines user management by enabling Single Sign-On (SSO) capabilities across diverse systems and platforms. We present oinit, a collection of programs extending OpenSSH to support any OpenID Connect identity provider for authentication. Our certificate-based solution integrates seamlessly with standard OpenSSH and does not require any changes in users' existing workflows or used programs.

Zusammenfassung

Trotz der weiten Verbreitung als Secure Shell Protocol (SSH) Implementierung unterstützt OpenSSH nur eine sehr begrenzte Anzahl von Authentifizierungsmechanismen, darunter Passwörter, öffentliche Schlüssel und Kerberos. Die Erweiterung von OpenSSH mit Authentifizierung mittels föderierter Identitäten adressiert die Sicherheitsrisiken von passwortbasierter Authentifizierung, verhindert die umständliche Verwaltung von öffentlichen Schlüsseln und optimiert die Benutzerverwaltung, indem Single Sign-On (SSO) Funktionalitäten über verschiedene Systeme und Plattformen hinweg ermöglicht werden. Wir präsentieren oinit, eine Sammlung von Programmen zur Erweiterung von OpenSSH um Authentifizierung mit OpenID Connect Identity Providern. Unsere zertifikatsbasierte Lösung integriert sich nahtlos in das Standard-OpenSSH und erfordert keine Änderungen in den bestehenden Arbeitsabläufen oder verwendeten Programmen der Benutzer.

Contents

1 Introduction **1**

2 Foundations **3**
2.1 Secure Shell Protocol (SSH) and OpenSSH . . . 3
2.2 Hypertext Transfer Protocol (HTTP) . . . 4
2.3 Representational State Transfer (REST) . . . 4
2.4 JSON Web Token (JWT) . . . 4
2.5 Federated identities . . . 6
2.6 OpenID Connect . . . 6
2.7 oidc-agent . . . 8
2.8 motley_cue and mccli . . . 8

3 Related Work **9**

4 Problem analysis **11**
4.1 Goals and limitations . . . 11
4.2 Considerations . . . 12
4.3 Authentication methods . . . 13
4.3.1 Password . . . 13
4.3.2 Keyboard-interactive . . . 14
4.3.3 Generic Security Service API (GSSAPI) . . . 15
4.3.4 Public key . . . 16
4.3.5 Host-based . . . 17
4.3.6 Certificate . . . 17
4.4 Obtaining OpenSSH certificates . . . 23
4.5 Dynamic usernames . . . 25

5 Implementation **27**
5.1 Design decisions . . . 27
5.2 Architecture . . . 29
5.3 Two-step process . . . 31
5.3.1 Preparation . . . 32
5.3.2 Connection . . . 33
5.4 Certificate authority . . . 34
5.4.1 API endpoints . . . 35
5.4.2 Configuration . . . 38
5.5 Command-line client . . . 38
5.5.1 Adding and deleting hosts . . . 39
5.5.2 Connecting to hosts . . . 39

5.6 User switching . . . 40
5.6.1 *setuid* bit and dropping privileges . . . 40
5.6.2 Unix tools . . . 41
5.6.3 Secure Shell Protocol (SSH) command . . . 42
5.7 User workflow . . . 43

6 Evaluation **45**
6.1 Testing . . . 45
6.2 Latency . . . 46
6.3 Security . . . 46
6.3.1 Service user interactive access . . . 46
6.3.2 TTY injection . . . 47
6.4 Verification of goals . . . 49
6.5 Limitations . . . 51
6.6 Installation . . . 52

7 Conclusion **53**
7.1 Summary . . . 53
7.2 Future work . . . 53

Bibliography **55**

Appendix **59**

Introduction

The Secure Shell Protocol (SSH) is still the most used network protocol for login and execution of commands on remote servers, with OpenSSH being the most prominent and widespread implementation [1, 2]. Several authentication and authorization protocols like OAuth [3], OpenID [4], Kerberos [5, 6] and SAML [7] exist and are used for Internet services today. However, OpenSSH still only offers a limited number of authentication methods including passwords, public keys, and Kerberos [8]. Passwords are known to suffer from security problems like low entropy and re-use. Public key-based authentication is cumbersome at large scale, is missing feature like key expiration and introduces new challenges to users, e.g., private key loss and compromise. While Kerberos can be used with OpenSSH, its configuration is complex and is difficult to set up between multiple realms or organizations. Other solutions like LDAP [9] suffer from similar limitations.

At the same time, authenticating to services with an existing account has become normal in the world wide web. Users are used to being able to log in to external services and websites using their existing Google, Apple, or university account [10, 11, 12]. This is especially true in the context of federated services deployed at research institutions, where a single existing home account at any institution grants a user access to services of partnering institutions. For example, students and researchers at universities in Baden-Württemberg benefit from access to services like data storage (bwSync&Share[1]) and computation resources (bwCloud[2]) hosted across different institutions and cities. Similarly, federated services are a core concept in distributed and grid computing like the Worldwide LHC Computing Grid (WLCG)[3] project, where collaboration between numerous institutions and data centers worldwide is necessary.

Our goal is to extend OpenSSH with support for authentication using federated identities, based on the established OpenID Connect protocol. This brings advantages of federated identities including Single Sign-On (SSO), scalability and interoperability to OpenSSH. It thus allows users to conveniently use federated identities for authentication, while the burden of user management and authentication is offloaded to their home organizations. We limit ourselves to OpenSSH and rely on existing programs for communication with OpenID Connect identity providers (`oidc-agent`) and user provisioning (`motley_cue`). Our aim is to seamlessly integrate into users' existing SSH workflows, while at the same time not requiring any changes to the OpenSSH source code.

[1] `https://www.scc.kit.edu/en/services/bwSyncAndShare.php`
[2] `https://www.bw-cloud.org`
[3] `https://wlcg.web.cern.ch`

This thesis is structured as follows. Chapter 2 (Foundations) introduces protocols and programs we build upon, while chapter 3 (Related Work) discusses existing approaches for this problem. In chapter 4 (Problem analysis), we define our goals and limitations. Further, we detail the analysis of OpenSSH and possibilities for an integration of OpenID Connect. Chapter 5 (Implementation) describes the architecture and realization of our solution. We evaluate our implementation regarding security and our defined goals in chapter 6 (Evaluation). The thesis completes with and mentions possible further work in chapter 7 (Conclusion).

Foundations

2.1 Secure Shell Protocol (SSH) and OpenSSH

The Secure Shell Protocol is a network protocol intended for login and the execution of commands on remote computers. As the name implies, SSH provides cryptography to allow secure connections on insecure or untrusted networks. SSH is designed as a client-server architecture [13] and is based on the Transmission Control Protocol (TCP). The original SSH protocol version 1, referred to as SSH-1, was developed by Tatu Ylönen at the Helsinki University of Technology in 1995. The cryptographically improved but incompatible version 2 of the protocol (SSH-2) was developed and later standardized by the Internet Engineering Task Force (IETF) in 2006. It is preferred today by most implementations, and SSH-1 is only supported by some clients for backwards-compatibility.

OpenSSH [8] is a suite of programs and today's most popular implementation of the Secure Shell Protocols, however support for SSH-1 was dropped in version 7.6[1] (03.10.2017) [1, 2]. Started as a fork of Ylönen's SSH-1 implementation, it was originally developed for the OpenBSD operating system, however a version called OpenSSH Portable is available and installed by default on most Linux distributions, Microsoft Windows, Apple macOS as well as various other operating systems. Next to the OpenSSH daemon `sshd` (server) and the remote login client `ssh`, OpenSSH includes programs such as `scp` (secure file copy), `sftp` (secure file transfer) and `ssh-agent` (authentication agent).

The OpenSSH authentication agent (implementing the agent protocol [14]) is a service program able to store private keys for authentication in memory. This allows a user to (temporarily) add keys to the agent and then use them multiple times, without the need to re-enter additional information such as a passphrase for decryption. OpenSSH also supports forwarding of this agent via the login client `ssh`, allowing stored keys to be used on a remote computer without the need to transmit or store private information there. Port forwarding can be used to forward TCP connections via `ssh`, allowing other network protocols not implementing encryption themselves to benefit from SSH's cryptography. While a common use case of SSH is to login to a remote computer and opening a shell for interactive access, SSH can also directly execute remote commands [15]. Multiple popular command-line programs such as `git` and `rsync` can make use of this, effectively using SSH as a transport protocol.

OpenSSH supports multiple core and extension authentication methods. These meth-

[1] `https://www.openssh.com/releasenotes.html#7.6`

ods include password-based authentication [16], a generic challenge-response method (`keyboard-interactive`) [17], GSSAPI authentication/key-exchange [18] for Kerberos, host-based as well as public-key authentication including OpenSSH certificates [16]. Section 4.3 describes these authentications methods in detail.

2.2 Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) is a communication protocol used to exchange hypermedia information between clients and servers [19], originally developed by Tim Berners-Lee at CERN [20]. Like SSH, most versions operate over TCP, while only the most recent iteration HTTP/3 is based on QUIC and therefore User Datagram Protocol (UDP) instead. It makes use of Transport Layer Security (TLS) for encrypting connections.

HTTP is the fundamental building block of the World Wide Web, used by clients to request websites from servers, usually in the form of HyperText Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript and media files. To request and send data, clients make use of HTTP methods such as GET and POST. The server then responds with a status code and the requested data or a confirmation. Today, HTTP is not just used for websites but also as building block for Application Programming Interfaces (APIs) and other communication protocols such as REST, Simple Object Access Protocol (SOAP) and WebDAV.

2.3 Representational State Transfer (REST)

REST is an architectural style for designing web-based APIs. It defines constraints and conventions for building web services, emphasizing the use of standard HTTP methods such as GET, POST, PUT, and DELETE to perform operations on resources represented as URLs. Communication in REST is stateless, meaning each request from a client to a server must contain all the information needed to understand and process it. This architectural style has gained widespread adoption due to its compatibility with the World Wide Web's infrastructure and its ability to facilitate loosely coupled, scalable, and interoperable systems. As REST is an architectural style rather than a protocol, no official standard but only classification models based on fulfillment of REST principles exist.

2.4 JSON Web Token (JWT)

JSON Web Token (JWT) is a proposed data format standard based on the JavaScript Object Notation (JSON) format, intended to be used in web contexts [21]. JWTs can be signed by its issuer, allowing them to be verified by another party.

As shown in figure 2.1, a JWT consists of three parts: a header, payload, and signature. The header defines the type of token and the algorithm used for generating the signature, for example `RS256` for Rivest-Shamir-Adleman (RSA) using the SHA-256 hash function.

The payload contains multiple JSON fields called claims. While some claim names are registered in the RFC (such as *iss* for the issuer and *exp* the expiration time), the payload may also contain any custom claims defined by the JWT issuer. The signature part is generated by the defined algorithm and based on the header and payload.

The three parts can be individually base64url-encoded [22] and concatenated using a period, resulting in a compact string that can be used in Uniform Resource Locators (URLs), HTTP headers and other contexts.

```
$ echo $TOKEN
eyJhb[...]trIn0.eyJle[...]R1In0.Ly20G[...]6mOJw # cut in length here

# header
$ jq -R 'split(".") | .[0] | @base64d | fromjson' <<< $TOKEN
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "o6XzGhu2PQJHB0XAoRFkdw4gwq3f3B8Li7LAe4yqhKk"
}

# payload
$ jq -R 'split(".") | .[1] | @base64d | fromjson' <<< $TOKEN
{
  "exp": 1693580042
  "iat": 1693578600,
  "auth_time": 1692781200,
  "jti": "832a5af1-a6b2-41ab-b95e-774bd14387b8",
  "iss": "https://oidc.scc.kit.edu/auth/realms/kit",
  "sub": "ea1b8e21-3654-4178-bd04-98c6adf58951",
  "typ": "Bearer",
  "azp": "7b3b85df-1965-41b9-b4e2-476f0eb0d5df",
  "session_state": "43aa4866-945c-4a60-b61f-ca1810e2e690",
  "acr": "1",
  "scope": "openid offline_access profile base",
  "sid": "43aa4866-943c-4a60-b61f-ca1710f2e690",
  "eduperson_scoped_affiliation": [
    "member@kit.edu",
    "student@kit.edu"
  ],
  "preferred_username": "uelri",
  "eduperson_principal_name": "uelri@student.kit.edu"
}
```

Figure 2.1: Example of encoded and decoded JWT.

2.5 Federated identities

Federated identities or federated identity management refers to the concept of enabling user authentication and authorization across multiple services/applications using a single set of credentials. Instead of registering and authenticating at a service provider directly, a user is registered at and managed by an identity provider. To use a service, the user logs in at an identity provider, who sends the user's identity information to the service provider for an authorization decision. This concept therefore relies on mutual trust relationships between identity and service providers.

The use of federated identities is convenient for users, as a single set of credentials can be used for multiple services. This is often combined with SSO. For service providers, the use of federated identities can reduce administrative burdens, as no user and credential management is necessary. Due to the use of interoperable protocols, a service provider can also allow authentication using multiple different identity providers without additional cost or scalability problems.

Federated identity management is heavily used in research communities due to the increasing need to share research data and services across collaborators from different institutions across the world [23]. To standardize federated identity management systems for research based on existing technologies, attempts such as the AARC Blueprint Architecture have been developed [24]. It is based on widely used protocols such as SAML, OAuth 2.0, and OpenID Connect, defines user attributes used for authentication/authorization and defines a protocol translation layer that integrates a proxy, discovery service and means to translate identity tokens between different technologies.

2.6 OpenID Connect

OpenID Connect is the third generation of OpenID, an authentication protocol intended for web and mobile applications [4]. It is based on the OAuth 2.0 framework, defines a REST API and uses JWTs.

OpenID Connect allows users registered at an identity provider (or OpenID provider) to authenticate themselves (e.g., via username and password), resulting in an identity token as well as an access token that can be given to a relying party. Optionally, a refresh token can be used to re-request expired identity/access tokens. An identity token, encoded as a JWT, contains signed information such as a name or email address in the form of claims[2]. A relying party, such as a website, can use this information to reliably authenticate users without the need to store login information (usernames, passwords, ...) itself. In contrast, access tokens (usually also encoded as JWT, however not required by the OpenID Connect protocol) do not contain personal information but rather allow a relying party to access some resource on behalf of the user. Access tokens are often used as Bearer tokens in the context of REST APIs.

OpenID Connect supports multiple possible authentication flows, which define how a

[2] `https://openid.net/specs/openid-connect-core-1_0.html#StandardClaims`

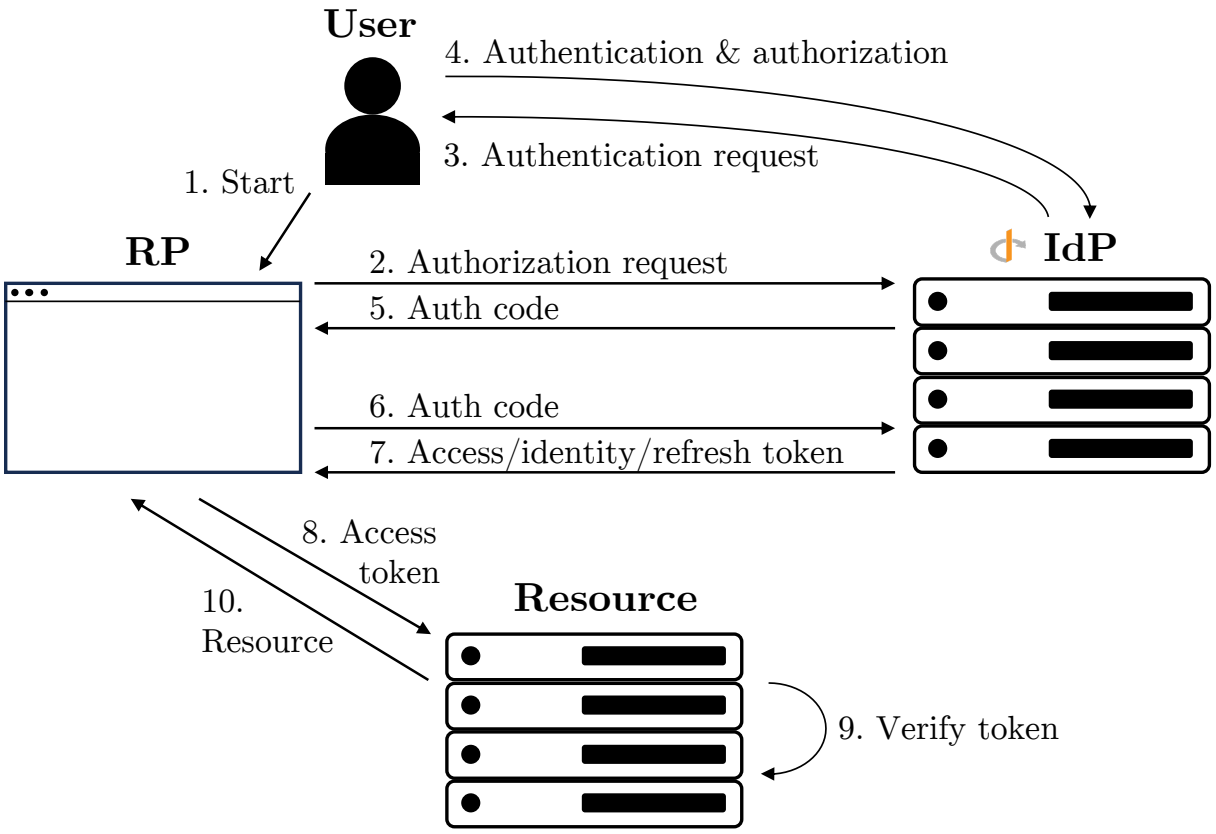

Figure 2.2: OpenID Connect authorization code flow to obtain an access token, which is used to access a protected resource.

user authenticates him-/herself using an OpenID Connect client. Examples include the password flow (send password to identity provider), the authorization code flow (login at identity provider website, redirect back to client application) and device flow (use second device with browser). Figure 2.2 shows the basic functionality of the OpenID Connect authorization code flow. The user starts the authorization code flow using a relying party (RP), such as a web browser (1). This RP requests an authorization code from the identity provider (2), which has to authenticate the user first (3). Usually, this means that a set of credentials (username & password) must be entered. After successful authentication of the user and authorization of the RP (4), an auth code is generated and sent back to the RP (5). This auth code can then be used to obtain access, identity, and refresh tokens from a different IdP endpoint (6, 7). To access some protected resource, the access token is included in a request (8). After verification of the token signature, validity date and other claims (9), access is granted and the resource sent back to the RP (10).

Many popular service providers such as Google ("Sign In with Google")[3] and Apple ("Sign In with Apple")[4], as well as universities such as the Karlsruhe Institute of Technology[5] implement OpenID Connect as identity provider.

[3] https://developers.google.com/identity/openid-connect/openid-connect

[4] https://developer.apple.com/documentation/sign_in_with_apple/sign_in_with_apple_rest_api

[5] https://www.scc.kit.edu/en/services/openid-connect.php

2.7 oidc-agent

`oidc-agent` is a collection of programs that help requesting and managing OpenID Connect access tokens from the command-line [25]. The project follows the design of `ssh-agent` in that a central agent service called `oidc-agent` communicates with OpenID Connect identity providers and stores received access tokens in memory.

The included `oidc-gen` program generates account configurations with information about the provider, authentication flow, scopes and more. The `oidc-add` program can load account configurations into the agent, the client program `oidc-token` can then be used to request access tokens for these account configurations. Securely stored refresh tokens are automatically used to request new access tokens if expired, without the need to re-authenticate at the identity provider.

The `oidc-agent` offers an Inter-Process Communication (IPC) API via Unix sockets that can be used to load account configurations and request access tokens [26]. Libraries for C, Python and Go are available that make use of this API.

2.8 motley_cue and mccli

`motley_cue` is a program to (de-)provision Linux user accounts given OpenID Connect access tokens after previous authorization [27]. It can be configured to accept tokens from different identity providers and supports multiple user backends including local Unix accounts, LDAP and bwIDM[6]. Authorization decisions are based on levels of assurance and various access tokens claims, such as virtual organization or group memberships. Multiple strategies considering different token claims (name, given name, email address, memberships, ...) can be used to map OpenID Connect identities to account usernames and groups. `motley_cue` offers a REST API to deploy, suspend and manage accounts given an OpenID Connect access token as Bearer token.

An integration with SSH is possible by installing a Pluggable Authentication Module (PAM)[7] and configuring the OpenSSH daemon to offer challenge-response authentication (`keyboard-interactive`) [28]. This will prompt users of the `ssh` command to enter an access token instead of a password. A modification of OpenSSH source code is not necessary.

`mccli` is a small wrapper program around `ssh` that aims to make SSH access to servers using `motley_cue` easy [29]. It can request an OpenID Connect access token via `oidc-agent`, call the `motley_cue` REST API to provision a local user account and then invoke the `ssh` program with the correct username and arguments. Additionally, `mccli` also offers an identical wrapper for the `scp` program. Like `motley_cue`, it does not require a modification of OpenSSH source code.

[6] `https://www.bwidm.de/` (german)
[7] `https://github.com/EOSC-synergy/pam-ssh-oidc-packaging`

Related Work

Several different approaches for an integration of federated identity authentication into OpenSSH already exist.

In the COmanage open-source project, developed by the Internet2 Middleware Initiative, users can authenticate themselves using a chosen OpenID Connect identity provider at a web-based registry [30]. After approval of a newly registered account, a SSH public key can then by uploaded by the user. It is stored in a LDAP server, which SSH servers such as the OpenSSH daemon can be configured to use as user database. To authenticate in a SSH connection, the SSH client uses a corresponding private key and fixed username determined based on his federated identity. A similar approach is taken in the Token Translation Service (WaTTS) developed at the Karlsruhe Institute of Technology (KIT) for the INDIGO Data Cloud project [31, 32]. Using a WaTTS SSH plugin, users can generate or upload a SSH key pair after OpenID Connect authentication [33]. The public key is then deployed to SSH servers, at which the user can authenticate using his/her private key. These approaches require multiple interaction steps by the user, who must authenticate in a web browser before uploading a manually generated SSH key. The SSH connection is then based on a pre-determined username, and uploaded public keys do not expire. A variety of other similar approaches exist, which often rely on a LDAP server as user backend.

To integrate OpenID Connect authentication into the SSH connection itself, several approaches based on Linux Pluggable Authentication Modules (PAMs) exist. E. Bonelo developed a custom PAM module based on the CYCLONE platform [34, 35]. When connecting to a SSH server using `keyboard-interactive` authentication, the PAM module starts a HTTP server and prompts the user to visit a URL. After authentication with an OpenID Connect identity provider in the browser, the PAM module obtains a JWT access token for validation and requests further user info from the identity provider. If the obtained federated identity e-mail address is listed in a user's personal configuration file inside the home directory, authorization is granted. This approach requires an existing Unix account that has been configured with allowed e-mail addresses.

A similar PAM module was developed by UK Research and Innovation (UKRI) based on previous work by M. Univerzita [36, 37]. It allows users to authenticate using the OAuth 2.0 device flow after visiting a generated URL in a browser. SSH clients must specify a username to use, which is compared with a configurable attribute from the OAuth user info for authorization.

`motley_cue` is a mapper service for OpenID Connect identities, developed by D. Gudu

at KIT [27]. It can be used alongside PRACE-LAB's PAM module to integrate into SSH, where the user is then prompted for an access token instead of a password [28]. The PAM module authorizes users based on OpenID Connect user info (such as preferred username) or by calling `motley_cue`, which considers virtual organizations/group memberships listed in the access token. A command-line wrapper program called `mccli` can be used on the client side to obtain OpenID Connect access tokens (via `oidc-agent`), deploy a user on-demand, and pass tokens to `ssh` automatically.

Attempts at modifying the OpenSSH source code to include other authentication methods have been made as well. This includes GSI-OpenSSH, a patched version of OpenSSH that supports authentication using Globus GSI (Grid Security Infrastructure) [38]. The project has been discontinued in 2018 and since mostly replaced with a PAM-based solution called Globus Auth SSH, later renamed XSEDE OAuth SSH [39, 40]. Next to the PAM module, it relies on client wrapper programs `oauth-ssh-token` and `oauth-ssh` to obtain and use OAuth tokens from Globus Auth. Identities are mapped to remote Unix usernames using a unique suffix per identity provider or a text-based map file. The SciToken SSH PAM module was later developed by Y. A. Gao et al. to overcome some limitations of XSEDE OAuth SSH [41].

Some commercial solutions focused on the use in corporate environments have also been developed in recent years, with some parts made available open-source. These include Smallstep SSH and Teleport, which are based on SSH certificates for authentication [42, 43]. Smallstep offers a client command-line program to obtain access tokens from OpenID Connect identity providers (called provisioners) [44]. A self-hostable CA issues SSH certificates that are then used for authentication. Remote user accounts must be created manually, whereas the usernames must match the e-mail address local part used to authenticate at the access token provisioner. Similarly, Teleport by Gravitational Inc. offers a command line client to authenticate at different OpenID Connect providers. Based on received access tokens, certificates for multiple services (Kubernetes, PostgreSQL, ...) including SSH can be obtained and used for authentication. Alternatively, Teleport offers a PAM module that can create user accounts on-demand, if a username was specified by the SSH client [45].

Problem analysis

In this chapter, we address the challenges to overcome when integrating OpenID Connect authentication into OpenSSH. First, we define several goals that a viable implementation should fulfill.

4.1 Goals and limitations

G1: No change in users' workflows
Users must not be required to change their existing workflow but be able to use the program and other SSH-based programs as common. This means that wrapper programs must not be required and SSH-based programs like `git` and `rsync` must work without any configuration. Additionally, no SSH built-in functionality such as the execution of remote commands, agent forwarding or jump servers should be broken. Integration into the `ssh` program should be tight, so that no manual user intervention such as copying or inserting tokens is required.

G2: Multiple OpenID Connect identity providers
A SSH server must be able support multiple OpenID Connect identity providers for authentication, which the SSH client can select from.

G3: Retain the benefits of federated identities
Advantages of federated identities and OpenID Connect, including SSO capabilities, interoperability, token expiry and revocation must be retained. At the same time, a solution must not suffer from the same drawbacks as existing authentication methods. This includes re-use of password or risk of private key loss/compromise.

G4: Dynamic usernames based on federated identity
User accounts must be deployed on-demand, that is on first SSH connection. The username used for connecting must be allowed to be chosen by the SSH server and not the user. This implies that users must be able to connect via SSH without knowing their, possibly yet to be deployed, username.

G5: No source code changes
It must not be required to change the source code of the SSH implementation, as this poses additional challenges such as distribution of the modified code. Additionally, it requires gaining trust of users and system operators in running that modified software, which may be used to give users access to security-critical systems.

G6: Secure and tamper-proof
The system must be secure, meaning that the SSH connection must not be tampered with. It must not be possible to abuse the system to gain SSH access without permission, or to impersonate another user account without adequate rights.

We limit our integration efforts to work with OpenSSH, as it is the most widely used implementation for client and server [1, 2]. Supporting other SSH implementations such as Dropbear SSH [46] may require completely different approaches and is therefore of scope for this thesis. Our program must be able to run on Unix-like operating systems, including popular Linux distributions and Apple macOS. Support for other operating systems, especially Microsoft Windows, may be considered but often proves to be difficult due to fundamental differences in architecture and available features.

As the intention of our integration is to extend and work alongside existing solutions without necessarily replacing them, we may base our solution on existing work to prevent re-implementations without benefits. This holds true for `motley_cue`, which supports user authorization and account provisioning for multiple backends and provides a REST API for interaction. For retrieval of access token from OpenID Connect providers, we may use `oidc-agent` and its libraries.

4.2 Considerations

To integrate federated identity authentication using OpenID Connect into OpenSSH, several aspects must be considered:

1. **User identity mapping**: It is necessary to map federated identities to Unix account names based on claims contained in the JWT. Claims differ between identity providers and unique values across identity providers are not guaranteed. Additionally, valid Unix account names are usually limited to 32 ASCII characters, which is a smaller space than all possible federated identity IDs.

2. **Technology harmonization**: OpenID Connect is based on the exchange of tokens, whereas OpenSSH authentication uses passwords or asymmetric cryptography. Therefore, it is necessary to either extend OpenSSH, transport tokens via existing authentication mechanisms, or to exchange a token with credentials usable by OpenSSH.

3. **Authentication and authorization**: While authentication is done by a chosen identity provider, authorization decisions based on virtual organization/group memberships or other JWT claims must be made before a SSH connection.

4. **Account provisioning**: After authentication and authorization, user accounts need to be provisioned automatically and on-demand. Group memberships of the federated identity must be respected as well. The newly generated username must then be propagated back to OpenSSH.

5. **Credentials management**: Any used OpenID Connect and OpenSSH credentials must be securely stored and accessible for use. Also, it is necessary to consider expiration and possible revocation of any credentials.

6. **User experience**: All benefits of OpenID Connect, such as the ability to use different identity providers and SSO need to be integrated into OpenSSH.

7. **Security**: The security of OpenSSH authentication must not be broken. Additionally, some form of logging for audits should be possible.

Some of these aspects can be addressed using existing tools, such as `oidc-agent` for OpenID Connect authentication/credential management and `motley_cue` for user identity mapping, some authorization decisions and account provisioning. Therefore, we focus on the remaining un-covered aspects as well as the seamless integration of all tools. In the following sections, we present our analysis of OpenSSH to address the transport/exchange of access tokens (section 4.3), the seamless integration into OpenSSH (section 4.4) and the dynamic user provisioning (section 4.5), while combining everything into a seamless user experience.

4.3 Authentication methods

As we do not want to change OpenSSH source code and do not wish to change users' SSH workflows, we must analyze OpenSSH's built-in authentication methods to evaluate their possible use for federated identity authentication. Apart from the `none` authentication method, the most recent version of OpenSSH supports authentication via `password`, `keyboard-interactive`, GSSAPI (`gssapi-with-mic`), `publickey`, `hostbased` and certificate [16, 17, 18].

4.3.1 Password

In password-based authentication, the user is prompted to enter a password, which is then transferred to the OpenSSH daemon in clear text (the packet containing the password is encrypted at transport level). The daemon uses the transmitted password to compare against a password database such as the `/etc/passwd` file or another PAM backend. Figure 4.1 shows how password authentication looks from the perspective of a user. On first connection to an unknown host, the user is asked to confirm the connection by verifying the presented fingerprint (out-of-band). Further connections will not ask for confirmation, as the host and its fingerprint are added to a `known_hosts` file. This security principle is called Trust On First Use (TOFU), which OpenSSH relies on for most authentication methods. If the entered password is accepted by the OpenSSH daemon, the user will get access to an interactive login shell.

Unfortunately, OpenSSH only allows to enter the password interactively or via the `-p` argument of `ssh`. It is not possible to enter a password via the standard input (`stdin`), environment variables or the configuration file `~/.ssh/config`. We therefore could not

```
$ ssh user@example.com
The authenticity of host 'example.com (93.184.216.34)' can't be established.
ECDSA key fingerprint is SHA256:OhzHBV6tWS.../.../...
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'example.com' (ECDSA) to the list of known hosts.
user@example.com's password: *************
```

Figure 4.1: Example of a SSH connection using password-based authentication.

find a way to transport an access token as password, without requiring the user to copy and paste the token. A small wrapper program calling `ssh` with the `-p` argument is possible (such as `sshpass`[1]), that would mean however that users must use the wrapper program instead of `ssh` directly.

4.3.2 Keyboard-interactive

The `keyboard-interactive` authentication method is a generic, multi-round challenge-response procedure. It allows the OpenSSH daemon to request arbitrary pieces of text-based information from the client, while specifying their prompt (e.g., "token:" instead of "password:") and secrecy (secret input will be masked with asterisks). `keyboard-interactive` authentication is intended to be used with a PAM backend on the server side. Uses of this method include requesting security tokens or one-time passwords from users. Like password authentication, `keyboard-interactive` relies on the TOFU principle.

Due to the configurable nature and the ability to use PAM, `keyboard-interactive` authentication may be used to transport OpenID Connect access tokens. In fact, the `mccli` wrapper uses this method to transport a previously obtained access token to the OpenSSH daemon, who validates it with a custom PAM backend [28]. `mccli` calls the `motley_cue` REST API before passing the access token to the `ssh` command by simulating user input. Figure 4.2 shows the demonstration server[2] prompting for an access token when connecting with the `ssh` command directly, as well as the `mccli` command necessary to automate the input of an access token.

```
$ ssh ssh-oidc-demo.data.kit.edu
(user@ssh-oidc-demo.data.kit.edu) Access Token:

$ mccli ssh ssh-oidc-demo.data.kit.edu --oidc kit
```

Figure 4.2: Example of a `ssh` and `mccli` command used to connect to a demonstration server with `keyboard-interactive` authentication enabled.

While transporting an access token via SSH to a custom PAM is possible, we identified

[1] `https://sourceforge.net/projects/sshpass/`
[2] `https://ssh-oidc-demo.data.kit.edu/`

two problems. Without requiring the user to actively copy and paste an access token into the `ssh` prompt, a wrapper program like `mccli` or `sshpass` is always necessary. Additionally, input to `keyboard-interactive` has a maximum length of 1023 characters, which access tokens can exceed. This is mentioned by the `mccli` developers[3], however no workaround exists.

4.3.3 Generic Security Service API (GSSAPI)

The Generic Security Service Application Program Interface (GSSAPI) is a standardized API that provides authentication and security services for client-server network communication protocols [47]. It enables developers of security applications to offer a GSSAPI library, which developers can then use to integrate into their applications without the need to implement complex security mechanisms themselves. The GSSAPI is comparable to Simple Authentication and Security Layer (SASL) used in SMTP, IMAP and LDAP. It is most prominently used in the Kerberos authentication protocol [6, 48]. OpenSSH implements the GSSAPI mechanisms to support Kerberos authentication, as defined in RFC4452 [18]. A proposed standard for integrating OpenID into GSSAPI exists, but is not supported by OpenSSH [49].

To evaluate the possible use of a custom GSSAPI-based library for transportation of an OpenID Connect access token, it is important to understand how to integrate such a new authentication mechanism into OpenSSH. The support of Kerberos via GSSAPI in the `ssh` client is implemented by dynamically linking to the GSSAPI library. When compiling from source, the required C header files `gssapi_krb5.h` and `gssapi.h` must be present[4]. The compiled `ssh` binary then relies on the Kerberos GSSAPI library to be installed on the users operating system (e.g., `libgssapi_krb5.so.2` in the case of GNU Debian GNU/Linux[5]).

Integrating another GSSAPI-based authentication mechanism into `ssh` therefore requires linking against a different library by adapting the `Makefile` [50]. While other SSH clients such as PuTTY allow the user to specify multiple GSSAPI libraries at runtime[6], OpenSSH does not. A patch for enabling dynamic runtime loading of GSSAPI libraries in `ssh` has been proposed by Aaron Sowry in 2013, however was not accepted by the OpenSSH Portable maintainer[7]. In his reply, Damien Miller reasoned this by being concerned by "a number of potential problems", including binary incompatibility, and changing library paths between different computers. We found no evidence of further work relating to his comment about future plans to moving OpenSSH's supported authentication methods into helper programs.

Another approach to circumvent OpenSSH's limitation of only supporting one GSSAPI implementation at the same time was implemented by the Massachusetts Institute of Technology (MIT) and the University of Illinoi's National Center for Supercomputing

[3] `https://ssh-oidc-demo.data.kit.edu/faq.html`

[4] `https://packages.debian.org/bookworm/amd64/libkrb5-dev/filelist`

[5] `https://packages.debian.org/bookworm/amd64/libgssapi-krb5-2/filelist`

[6] `https://the.earth.li/~sgtatham/putty/0.79/htmldoc/Chapter4.html#config-ssh-auth-gssapi`

[7] `https://bugzilla.mindrot.org/show_bug.cgi?id=2121`

Applications (NCSA). NCSA distributes a patch for OpenSSH allowing the simultaneous use of Kerberos and GSI authentication via GSSAPI [38], built on top of MIT's *mechglue* library[8]. However, this requires applying the provided patch to OpenSSH, changing the source code. Also, the patch was last updated in 2015 for OpenSSH 7.0p1. NCSA proposed adding their GSI authentication implementation to OpenSSH in 2015, however the patch was also denied in 2020 with Damien Miller stating that there are no plans for integrating any other GSSAPI-based authentication methods[9].

We found source code modifications to be necessary for the OpenSSH daemon as well, which we want to avoid (see **G5**). The `auth2-gss.c` source file implements generic GSSAPI calls for use with the `gssapi-with-mic` authentication mechanism. However, `gss-serv.c` lists supported GSSAPI mechanisms, which currently only contains Kerberos. Adding another mechanism therefore requires changes to OpenSSH's source code. The developers explain this by the need to translate GSSAPI identities into local Unix accounts, which is specific to the used mechanism[10]. This is confirmed by the presence of `gss-serv-krb5.c`, which implements this translation.

4.3.4 Public key

Authentication using asymmetric cryptography is often preferred over passwords, as it requires the possession of some key compared to only the knowledge of a password. For this authentication method, users are first required to generate a private/public key pair using the `ssh-keygen` program. Two files in an OpenSSH-specific format are created in the user's `~/.ssh/` directory by default, for example `~/.ssh/id_ed25519` (private key file) and `~/.ssh/id_ed25519.pub` (public key file). An example key pair is shown in figure 4.3. Several key types such as RSA and Edwards-curve Digital Signature Algorithm (EdDSA) are supported. The generated public key file content then must be transferred to the OpenSSH server and stored in a file accessible to `sshd`, by default in the user's home directory at `~/.ssh/authorized_keys`. The private key needs to remain on the client computer accessible only to the user itself, enforced by strict file permissions. It can also optionally be protected with a passphrase. To authenticate against the OpenSSH daemon in a SSH connection, the client must proof ownership of the private key. This is done by providing a cryptographic signature to the daemon, which can verify it using the public key present in `~/.ssh/authorized_keys`. Like all previously presented authentication methods, public key authentication also relies on the TOFU principle.

Unfortunately, a private/public key pair generated by `ssh-keygen` does not contain any customizable data, which information related to a federated identity needed for authentication (e.g., OpenID Connect access token, username) could be embedded in. A custom comment can be passed to `ssh-keygen` on creation; however, it is just stored in plain-text alongside the actual key in the generated file. The manual states that it can be used for key identification but is never transferred to or used by the OpenSSH daemon [51].

[8] `https://web.mit.edu/kerberos/krb5-1.12/doc/plugindev/gssapi.html`
[9] `https://bugzilla.mindrot.org/show_bug.cgi?id=2495#c2`
[10] `https://marc.info/?l=openssh-unix-dev&m=153574004725602&w=2`

```
$ cat id_ed25519
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUA...
-----END OPENSSH PRIVATE KEY-----

$ cat id_ed25519.pub
ssh-ed25519 AAAAC3Nz...a083z This is a comment
```

Figure 4.3: Example of an `ed25519` (EdDSA algorithm with SHA-512 and Curve25519) OpenSSH private (`id_ed25519`) and public key (`id_ed25519.pub`).

4.3.5 Host-based

Compared to the public key authentication, host-based authentication uses a private/public key pair specific to an entire computer instead of a single user. A user authenticating to an OpenSSH daemon therefore reads a computer-wide private key (usually stored in `/etc/ssh/`) to generate a signature, which the daemon verifies using a public key. While this authentication method is useful in some scenarios, such as in a closed computing cluster, it is difficult to securely configure and differentiation between individual users is no longer possible. We therefore find this method unsuited for our use case.

4.3.6 Certificate

While OpenSSH certificates were introduced with version 5.3 in 2010, they are still relatively unknown to most users. Compared to X.509 certificates, they are stored in a custom, simpler format and only consist of a public key, some identity information, a list of principals as well additional permissions. Interestingly, certificates do not specify their own authentication mechanism name but instead also use `publickey`. The `ssh-keygen` program can generate and sign OpenSSH certificates using a private/public key pair called certification authority (CA) [51].

Figure 4.4 shows the contents of an example OpenSSH certificate. The *type* of a certificate is either n user or host certificate. Additionally, it specifies which algorithm the certificate is based on. *Public key* and *Signing key* specify the OpenSSH public key that the certificate is based on and signed with (CA) respectively. The manual describes the *Key ID* field as an identifier logged by `sshd` when used for authentication. It is also used in a key revocation list when wanting to revoke certificates. Similarly, the 64-bit *Serial* number serves as identification for revoking certificates. It is intended to be used to distinguish certificates signed with the same CA key and identical *Key ID* but is set to 0 by default. Certificates are either valid forever (by default) or can specify a time range (validity interval) in the *Valid* field. A certificate is not accepted by `sshd` before its start time or after its end time. The *Principals* field lists one or multiple Unix usernames for which the certificate is valid. For host certificates, this field contains valid hostnames instead. Additionally, OpenSSH certificates contain optional *Critical Options* and *Extensions*. The manual lists *force-command* as the only possible *Critical*

Options, which forces the execution of a command by `sshd` instead of executing a user-specified command or spawning an interactive shell. *Extensions* allow and disallow certain OpenSSH features including PTY allocation as well as agent, port and X11[11] forwarding.

```
$ ssh-keygen -L -f user-key-cert.pub
user-key-cert.pub:
        Type: ssh-ed25519-cert-v01@openssh.com user certificate
        Public key: ED25519-CERT SHA256:rkSKv...
        Signing CA: ED25519 SHA256:xw9aV... (using ssh-ed25519)
        Key ID: "user@example.com"
        Serial: 0
        Valid: from 2023-09-01T14:30:00 to 2023-09-02T14:30:00
        Principals:
                user
        Critical Options:
                force-command whoami
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

Figure 4.4: Example of an OpenSSH user certificate.

Compared to public keys, OpenSSH certificates include some benefits and solve some security problems:

- Certificates are issued for specific users or hostnames, whereas a private/public key pair can be re-used by different users and hosts.
- Certificates can include an optional validity interval, which means they can automatically expire. Public keys do not support this and must be manually revoked if desired.
- Certificates can include restrictions, such as disallowing port or agent forwarding. For public keys, this is only possible by modifying the `~/.ssh/authorized_keys` file and making sure the user itself cannot modify this file.
- Certificates can contain a custom forced command. Like restrictions, public keys only support this by modifying the `~/.ssh/authorized_keys` file.
- Certificates do not rely on the TOFU principle. Instead, the `ssh` client also verifies the signature of a certificate presented by the OpenSSH daemon.

[11] Forwarding of graphical applications via the X Window System

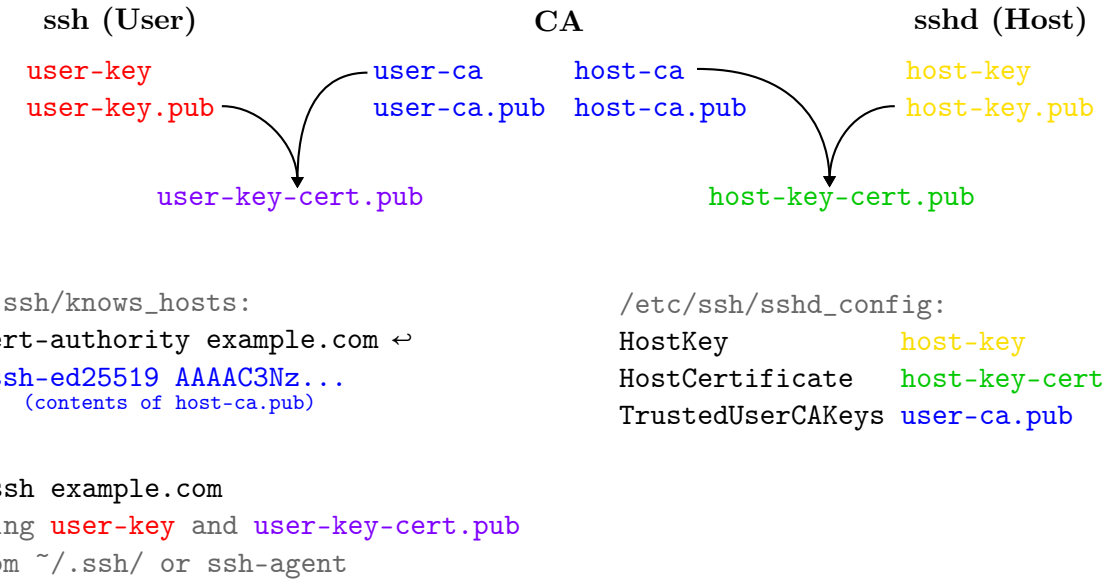

Figure 4.5: Example steps and configuration of an OpenSSH certificate authentication setup (using separate user and host CA key pairs).

- Certificates fields such as *Key ID*, *Serial*, *Principals* and even the *Critical Options* and *Extension* can contain arbitrary values. As the certificate can be signed, these fields cannot later be modified without breaking the signature.

It must be noted that using OpenSSH certificates for authentication requires some configuration of both the OpenSSH daemon as well as clients. Figure 4.5 illustrates the steps and configuration necessary for certificate authentication. This example uses the recommended setup of two separate key pairs for the user CA and host CA, however it is possible to use a single key pair. Also, this setup implies a CA instance separate from the OpenSSH server. This way, it is possible to issue host certificates for multiple independent OpenSSH servers based on the same host CA key pair. For scenarios with a single OpenSSH server, the same server can also take on the tasks of the CA.

The illustrated setup is based on four OpenSSH private/public key pairs: a user key pair, user CA and host CA key pairs as well as a host key pair. The `ssh-keygen` commands to generate all key pairs and certificates are listed in the appendix. To authenticate itself against clients, the `sshd` requires a host certificate. This certificate is issued by the CA by signing the public host key with the host CA private key. `sshd` needs to be configured to present this certificate to clients using the `HostKey` and `HostCertificate` keyword in its configuration file `/etc/ssh/sshd_config`. Additionally, it is necessary to specify that user certificates signed by the user CA should be accepted (with the `TrustedUserCAKeys` keyword). Like the host certificate, a user certificate is issued by signing the user's public key with the user CA private key. To let `ssh` known which host certificates to trust, the contents of the host CA public key must be added to `~/.ssh/known_hosts` with a special `@cert-authority` directive. A connection to the OpenSSH daemon then uses the issued user certificate as well as the user's private key. If either the user or host certificate were issued with a limited validity, they must be re-issued manually after their expiry.

When a client authenticates to an OpenSSH daemon, both parties exchange certificates

which are validated by the other side. A `ssh` client verifies the received host certificate using the listed principals and by validating its signature against the CA public key listed in the local `~/.ssh/known_hosts` file. The `sshd` daemon verifies the user certificate by checking the listed principals and signature using the user CA public key configured with `TrustedUserCAKeys`. If both sides accept the certificate, authentication succeeds. `sshd` also checks allowed connection features listed in the extensions and executes the *force-command* critical option, if specified. Compared to all previously presented authentication mechanisms, certificate authentication does not rely on TOFU by asking the user to verify a fingerprint presented by the OpenSSH daemon. Rather, the user must trust a host CA to only issue host certificates to trusted servers. Similarly, the OpenSSH server administrators must trust a user CA to only issue certificates to trusted users.

OpenSSH certificates are interesting for our use case, as some fields allow custom content to be specified when issued. Therefore, embedding an OpenID Connect access token inside a certificate might be possible. We ruled out the *Serial* field, as it can only contain 64-bit numbers, in which access token do not fit. Similarly, we ruled out the *Principals* field as it is evaluated by `sshd` to contain the correct username or hostname. Embedding an access token as additional principal creates security problems, as forging usernames is very easy. Therefore, we found the fields *Key ID* and the *force-command* critical option to be possible candidates, which we evaluate in the following sections.

KeyID field

The *Key ID* field can contain any string with arbitrary length. Embedding an OpenID Connect access token is therefore possible. The challenge is finding a way to extract this token in or by `sshd`.

The OpenSSH daemon configuration supports an `AuthorizedPrincipalsCommand` keyword, which specifies a command to execute when a connection using OpenSSH certificates is received by `sshd` [52]. This command is supposed to return a list of allowed principals, which are then compared with the principals listed in the received user certificate. Because `sshd` can pass certain tokens to this command via arguments, including the *Key ID* field of the certificate (with the `%i` placeholder), we found that it might be possible to abuse this command by extracting the access token from the *Key ID* field to run some user account generation. However, we found no way to propagate a created username back to `sshd`. Using `AuthorizedPrincipalsCommand`, it is only possible to allow a connection by returning at least one username that is present in the certificate's *Principals* field. Returning a freshly created new username that is not listed in the certificate will always deny the connection attempt.

We also identified a security problem with this approach. `sshd` invokes the command listed in `AuthorizedPrincipalsCommand` by creating a new subprocess (via *fork & exec*). All arguments given to this command, such as an OpenID Connect access token extracted from the *Key ID* field, are therefore publicly visible for all users on the system[12]. While access tokens are usually valid only for a few minutes, any user with access to the system is still able to steal tokens by observing `AuthorizedPrincipalsCommand`s being run by

[12] In process viewers such as `top`/`htop` or directly from the `/proc/` directory.

`sshd`. Attempts at not passing the *Key ID* field via argument but another technique also failed. `sshd` did not accept modified commands such as

```
AuthorizedPrincipalsCommand TOKEN=%i /usr/bin/extract-token
```
(environment variable)

or

```
AuthorizedPrincipalsCommand /usr/bin/extract-token
                            <( TOKEN=%i printenv TOKEN )
```
(temporary file descriptor)

as the provided command is strictly parsed to only allow a certain syntax (an absolute path to a file, followed by whitespace-separated arguments without special characters) for security reasons. Other techniques such as the use of pipes:

```
AuthorizedPrincipalsCommand TOKEN=%i printenv TOKEN |
                            /usr/bin/extract-token
```

result in a shell being invoked with the entire command passed as argument to `-c`, resulting in a subprocess which again leaks the token to every user:

```
bash -c "TOKEN=<access token> printenv TOKEN | /usr/bin/extract-token"
```

Additionally, the `ssh-keygen` manual states that the contents of *Key ID* are logged by `sshd`, which is not desirable for access tokens. Embedding into and extracting an access token from another certificate field than *Key ID*, such as custom *Extensions* results in the same security problem. We conclude that with this approach, only public and non-sensitive contents should be embedded in an OpenSSH certificate, which rules out personal access tokens.

force-command option

Like the *Key ID* field, the *force-command* critical option can be specified when issuing OpenSSH certificates. The `ssh-keygen` manual states that *force-command* can include a custom command to be run by `sshd` as the connecting Unix user after successful authentication. This command overrides the invocation of a login shell or the execution of a command that might have been specified by the user with

```
ssh example.com "some-command <args>".
```

It is therefore possible to embed a custom command and pass an OpenID Connect access token as program argument. While an `AuthorizedPrincipalsCommand` can allow or deny a connection, the difference is that the *force-command* is only executed after the user was already authenticated and a successful SSH connection has been established. This allows

```
USER  Command
root  ├ sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root  │ └ sshd: user [priv]
user  │    └ sshd: user@pts/0
user  │       └ bash -c 'TOKEN=... /usr/bin/extract-token'
```

Figure 4.6: Extract of process tree showing *force-command* being executed in a shell subprocess. Note that in this exact case, `bash` replaces itself with `/usr/bin/extract-token` shortly after startup.

running a custom program by `sshd` to handle the passed OpenID Connect access token, without tampering with the SSH connection itself. Exiting the custom program also automatically closes the SSH connection.

Unfortunately, we identified an identical security problem as to the previous approach. `sshd` creates a shell subprocess via *fork & exec* to execute the given *force-command*. Any arguments to this command, like an OpenID Connect access token, are therefore visible to any user. Again, adapting the command the use an environment variable instead of a program argument (`force-command="TOKEN=... /usr/bin/extract-token"`) or other tricks are not possible, as everything is passed to the shell via the `-c` argument and therefore publicly visible (shown in figure 4.6). Some shells, including `bash`, replace themselves via `exec` directly instead of forking first when no pipes, stdin/stdout/stderr redirection or logical operators are present in the command passed, to not waste system resources. While this happens at shell startup and takes only a short time, it is still possible to observe the original arguments including the token before replacement. A custom shell suffers from the same problem and does not solve this security problem.

To conclude, we find that OpenSSH's built-in authentication mechanisms all come with certain limitations when wanting to transport OpenID Connect access tokens. The password and `keyboard-interactive` mechanisms require manual interaction by the user, which we want to avoid (see **G1**). Additionally, `keyboard-interactive` is limited to a character length problematic for access tokens. A custom GSSAPI-based authentication mechanism requires changes to the source code of at least `sshd`, which we also want to prevent (see **G5**). Host-based authentication is not suited for our use-case due to the difficulty of differentiating between multiple users on the same client system. Public keys do not offer the ability to transport custom content such as an access token, therefore they can only be used as temporary grant in exchange for a token. To retain the benefits and features of federated identities (see **G3**), this however requires additional work such as revocation of keys. OpenSSH certificates with their built-in features such as expiration and custom *force-command*s seem like a possible option for the idea of this thesis, however it is only possible to embed non-sensitive content due to possible security problems when extracting it. In the sections, we evaluate how certificates could still be used for OpenID Connect authentication, even with the limitation of not being able to embed an access token directly.

```
Host *
        SendEnv LC_* LANG

Host example.com
        Port 1234
        User user
        IdentityFile ~/.ssh/id_ed25519
```

Figure 4.7: Extract of an example `ssh` user configuration file (`~/.ssh/config`).

4.4 Obtaining OpenSSH certificates

As our research shows that OpenSSH certificates might be usable four our thesis idea, it is necessary to find a way to prepare certificate-based authentication while at the same time not requiring any changes to users' workflows (see **G1**). This especially means that `ssh` commands executed by users should work as expected, and no wrapper program around `ssh` must be required. First, it is necessary to invoke a custom external program handling the generation/issuing of a user certificate when a `ssh` is run, however without the user noticing. Second, the certificate must then be stored or prepared to be automatically used by the SSH connection.

The `ssh` manual describes that after interpretation of command line options given to the command, both a user's local (`~/.ssh/config`) and system-wide configuration file (`/etc/ssh/ssh_config`) are read [53]. A variety of configuration keywords are supported, separated into sections by the `Host` keyword. Using this syntax, configuration options can be applied to specific hosts only. Figure 4.7 shows an example configuration, where a non-standard port, a fixed user and private key file should be used for connections the host `example.com`. Additionally, some locally set environment variables are sent in connections to each host by making use of the `Host` keyword's pattern matching support.

The configuration file supports the `ProxyCommand` and `ProxyJump` keywords, which are intended to be used for configuring a custom proxy used by `ssh` when connecting to a host. As the command configured in this keyword is run before the actual SSH connection, it is possible to execute a custom program that generates/request an OpenSSH certificate before opening the actual proxy and allowing the SSH connection to work. A drawback of this approach for our use case is the need to configure the `ProxyCommand` keyword separately for every host the client wants to connect to, resulting in multiple `Host` sections. Also, running a local proxy (like `nc`[13]) presents a small overhead, while the proxy is not actually necessary and only abused to run a custom program before the SSH connection.

In addition to the `Host` keyword, the `ssh` configuration also support as `Match` keyword alongside some conditions. Apart from some built-in conditions like `canonical` and `final`, the `exec` condition allows the specification of an external command to run. This command must return an exit code of 0 to indicate a match, while any non-zero exit code means no match. Like the `ProxyCommand`, this command is executed before the actual

[13] `netcat` is a utility program for network-related tasks, including proxy support.

```
1 Match exec "/bin/get-certificate %h %p"
2         User user
```

Figure 4.8: Example use of the `Match` keyword.

SSH connection and therefore allows to run custom code like generation of an OpenSSH certificate. Figure 4.8 shows how `Match exec` can be used to realize this. The specified command/program can also accept tokens (such as `%h` for hostname and `%p` for port), which are replaced with actual values by `ssh` when invoked. If the command returns 0, the `Match` block is considered true and all configured keywords like `User` are applied to the connection. Compared to the `ProxyCommand` keyword, this approach does not require any local proxy to run. In addition, only a single `Match` block is necessary, as the custom command can decide based on the hostname/port whether a host supports OpenID Connect authentication in the first place. If not, a non-zero exit code can be returned, which skips the entire `Match` block. As configuration files are read top to bottom and keywords defined in matching `Host`/`Match` sections override any later matching sections, this `Match exec` should be placed on top of the `ssh` configuration file.

We found only one pitfall of this approach, which is that in a `ssh` configuration, the use of a `CanonicalizeHostname` or `Match final` keyword results in the entire configuration to be re-parsed. Therefore, it must be considered that the custom command/program may be executed multiple times before a single `ssh` connection.

With the ability to execute an external program to generate or request an OpenSSH certificate, it is necessary to store this certificate and instruct `ssh` to use it. While each certificate could be written into their own file, it is difficult to then tell `ssh` which file to use. It is not possible to pass additional command-line arguments like `-i`[14] to `ssh` from the configuration file. Writing the certificate file location into the `ssh` configuration file also is not possible, as it has already been parsed at that point. Sharing a single, fixed file path instead results in problems if multiple OpenSSH certificates are present at the same time (for example if the user connects to two separate OpenSSH servers).

Instead, it is possible to load a certificate and its corresponding private key into the `ssh-agent`. The SSH connection will then automatically select an appropriate certificate from the agent. Additionally, the `ssh-agent` can be instructed to delete certain certificates and private keys after some lifetime. Therefore, expired certificates can be automatically removed.

In conclusion, we find that the `ssh` configuration file can be modified to include a `Match exec` keyword. This allows the execution of an external program which can generate or issue a certificate before the actual SSH connection. Loading this certificate into the `ssh-agent` enables `ssh` to use it directly after for authentication.

[14]This option instructs `ssh` which identity (private key, public key, certificate) to use for the connection.

4.5 Dynamic usernames

In **G4**, we outlined that a viable implementation should support dynamic usernames. That is, a user should be able to connect to an OpenSSH server without knowledge of his/her username, but then be presented with an interactive login shell (by default) running as a personal user account. Different OpenID Connect access tokens belonging to the same federated identity should result in the same personal user account.

The SSH authentication protocol defines that a username must be included with authentication requests [16]. Further, the RFC requires that "if the 'user name' does not exist, the authentication request MUST NOT be accepted.". Therefore, not specifying a username and letting `sshd` decide on which account to use is not possible. While according to the specification the username field can be left empty (empty string), it is up to `sshd` on how to handle this[15]. When enabled in the configuration file, the OpenSSH daemon can make use of PAM for authentication. However, OpenSSH does not implement user switching via PAM[16][17]. In addition, PAM authentication is only supported for the password, `keyboard-interactive` and Kerberos (GSSAPI) authentication methods but not for public keys and certificates[18]. Trying multiple authentication methods with different usernames (first method to deploy user, second method to authenticate) also is not possible, as `sshd` does not allow this in accordance with the authentication protocol specification[19].

In addition to the protocol preventing non-specification or changing of usernames, we found no way to implement this behavior without source code changes, as both the `ProxyCommand` and `Match exec` keywords are unable to do so. The username used by `ssh` is already determined based on the local username, command line options or some `User` keyword in the configuration file, before either of the commands are invoked. As it is not possible to change the username *before* the authentication attempt (on the client side) or *during* the authentication attempts (via PAM), we conclude that switching usernames can only happen *after* successful authentication.

In our previous research (section 4.3.6), we showed that the *force-command* critical option embedded in OpenSSH certificates is executed after authentication. This command can therefore be used to switch to a personal user account, based on either a custom program or existing Unix tools such as `su`. This approach implies two consequences:

1. Due to security problems outlined before, the *force-command* should only contain non-sensitive information. Instead of embedding an access token in the certificate, we therefore apply the idea of embedding the final username in the certificate. This requires obtaining a username before the actual SSH connection, which must then be switched to after a successful authentication.

2. As `sshd` requires an existing username for the SSH connection, another fixed service

[15] `https://groups.google.com/g/comp.security.ssh/c/EzvfhfpMVPw`
[16] `https://bugzilla.mindrot.org/show_bug.cgi?id=1215#c22`
[17] `https://listman.redhat.com/archives/pam-list/2011-June/msg00010.html`
[18] `https://bugzilla.redhat.com/show_bug.cgi?id=1492313`
[19] `https://security.stackexchange.com/a/121312`

```
1 Match exec "/bin/get-certificate %h %p"
2         User service
```

Figure 4.9: Example configuration of service user combined with the `Match exec` keyword.

> user must be used. This user is only used for the SSH connection itself before switching to a personal user account specified in the certificate *force-command*. It must not be overwritten by the user using command-line options or a configuration section.

For the service user to be used, it must be configured in the `ssh` configuration file. Figure 4.9 shows how a user named `service` can be configured, combined with the previously presented `Match exec` approach for invoking an external program (section 4.4). Using this configuration, only a single `Match` section is required to work for all OpenSSH hosts supporting OpenID Connect-based authentication.

It must be noted that a similar approach can also be implemented using public keys instead of OpenSSH certificates, as OpenSSH's `~/.ssh/authorized_keys` file (or other files configured with `AuthorizedKeysFile`) also allows the specification of a command to be executed for every public key [54]. However, this approach requires constant modification and growth of the file, as new (temporary) public keys are added to a service user's authorized keys. In comparison, the use of OpenSSH certificates does not require modification of any files, including the `sshd` configuration (apart from configuration necessary for certificate authentication itself). Certificates also include other benefits such as optional expiration and not relying on TOFU.

Implementation

In this chapter, we describe the implementation of a suite of programs to extend OpenSSH with support for OpenID Connect authentication. We base our architecture and design decisions on research presented in the previous chapter.

5.1 Design decisions

To integrate OpenID Connect authentication into OpenSSH, we decided to use the following approach.

On the client, the `ssh` configuration file is modified to invoke an external program using `Match exec` whenever `ssh` is executed. The external program obtains an OpenID Connect access token by calling `oidc-agent`. The access token is transferred to a certificate authority, which calls `motley_cue` to deploy and return a username if the user is authorized. An OpenSSH certificate is issued, in which the username is embedded via the *force-command* critical option. This certificate is transferred back to the client, where it is loaded into the local `ssh-agent`. The `ssh` client authenticates itself at the OpenSSH daemon using certificate-based authentication and a fixed service user. After successful authentication, the *force-command* is executed by `sshd` to switch from the service user to the personal user account deploy by `motley_cue`.

We chose to use `Match exec` to invoke an external program during a SSH connection, as this allows the user to keep his/her `ssh` workflow. No wrapper script is required and all built-in `ssh` features work as expected. At the same time, no manual user intervention such as inserting access tokens is required, which we outlined in **G1**. Compared to the `ProxyCommand`-based approach, no locally running proxy is required and a single `Match` section in the `ssh` configuration file is sufficient.

For obtaining OpenID Connect access tokens, we decided to make use of `oidc-agent`. It supports arbitrary identity providers (**G2**) and includes SSO-like capabilities by being able to load configurations into a running background process (**G3**). Additionally, it can be easily integrated into other programs due to its IPC API.

`motley_cue` is used to deploy user accounts, as it can be configured to support more than one OpenID Connect identity provider at the same time (**G2**). Multiple user backends and strategies for user creation are supported, therefore no re-implementation is necessary. Basing user creation on `motley_cue` also means that the use of `mccli` is also still possible simultaneously.

To prevent any changes to OpenSSH source code (**G5**), we decided to make use of a built-in authentication mechanism. Even though some configuration of `ssh` and `sshd` are required, we chose to use OpenSSH certificates because of their benefits compared to public key authentication. These include improved security (no TOFU), optional expiration (**G3**) and the ability to embed custom content. Exchanging access tokens with temporary private/public key pairs instead would also be possible, however key expiration and revocation must then be implemented manually using e.g., `sshd`'s `AuthorizedKeysCommand` and `RevokedKeys` configuration keywords. By using an existing authentication mechanism, we also make sure that the SSH connection is not tampered with and still secure (**G6**).

To store any received OpenSSH certificates along with private keys on the client, we chose to make use of the `ssh-agent`. This allows `ssh` to automatically select and use the correct certificate. No modification of the configuration file is required and expired certificates can automatically be removed from the `ssh-agent`. Also, not writing any key or certificate to disk reduces the risk of compromise.

Due to restrictions of the OpenSSH authentication protocol, we make use of the OpenSSH certificate *force-command* critical option to switch to a personal user account after successful authentication (**G4**). By embedding this command inside certificates, no further configuration of `sshd` is necessary. This also means that the SSH protocol is not tampered with in any way (**G6**).

By choosing this approach, we address all aspects mentioned in section 4.2:

1. **User identity mapping**: `motley_cue` is used to map federated identities to local Unix accounts based on multiple JWT claims. It is integrated using the provided REST API.

2. **Technology harmonization**: OpenID Connect access tokens are exchanged with custom OpenSSH certificates which are then used for authentication.

3. **Authentication and authorization**: `oidc-agent` handles authentication at the identity provider once, then stores a refresh token to request access tokens. An account configuration is loaded into memory and only needs to be unlocked once after reboot. We rely on `motley_cue` for authorization decisions, which is configurable to use different attributes of a federated identity. Only if access to the OpenSSH server is granted, a certificates is issued for the client allowing a SSH connection.

4. **Account provisioning**: A certificate authority is responsible for provisioning user accounts on-demand by calling `motley_cue`. Generated usernames are embedded in an OpenSSH certificates and later used by a custom *force-command* to switch to the personal user account.

5. **Credentials management**: All OpenID Connect credentials are managed by `oidc-agent`, whereas OpenSSH credentials (certificate and private key) are loaded into `ssh-agent` for use by `ssh`.

6. **User experience**: Our approach integrates seamlessly into the `ssh` command and allows the selection of an identity provider. `oidc-agent` provides SSO functionality

by keeping loaded account configurations in memory and requesting new access tokens using a stored refresh token.

7. **Security**: By relying on a built-in OpenSSH authentication mechanism, we retain OpenSSH's security guarantees. Issued certificates and authorization decisions are logged by the CA and `sshd` respectively, allowing audits.

However, this approach also includes two implications:

1. As including OpenID Connect access tokens inside a certificate results in the risk of exposing them when extracted, the final username must be embedded instead. Due to the nature of some username generation strategies, `motley_cue` does not support reserving usernames. Therefore, user accounts cannot be deployed by `sshd` via the *force-command*. Instead, the CA must deploy the user in exchange for an access token and embeds the resulting username in the issued OpenSSH certificate. User creation is not usually a task of a certificate authority; however, this is necessary to not leak access tokens to other users. This necessary design decision does not impact the authentication process and is not noticeable by the user.

2. Due to the OpenSSH authentication protocol, a fixed service user is required for the SSH connection, before `sshd` switches to the previously deployed personal user account. This is configured once in the `ssh` configuration file, but not noticed by the user otherwise. When executing `ssh`, the user must not specify a custom username to use. This however can be detected and displayed as an error to the user.

5.2 Architecture

Figure 5.1 shows the architecture we decided on for our solution. Next to the existing SSH client and server, we introduce a certificate authority. This CA takes over tasks of the OpenSSH certificate authority required due to the use of OpenSSH certificates (see 4.3.6). Additionally, it is responsible for communication with one or multiple `motley_cue` instances running on OpenSSH servers.

Based on this architecture, we identified the need to implement three separate programs, described below.

- A certificate authority issuing OpenSSH certificates (called `oinit-ca`).
- A command-line program to use alongside `ssh` (called `oinit`).
- A program responsible for user switching on the server, extending `sshd`'s functionality (called `oinit-switch`).

This suite of programs is referred to as oinit. While functionally completely different, this name was chosen in reference to the `kinit` program which is used to obtain a ticket for Kerberos authentication in SSH.

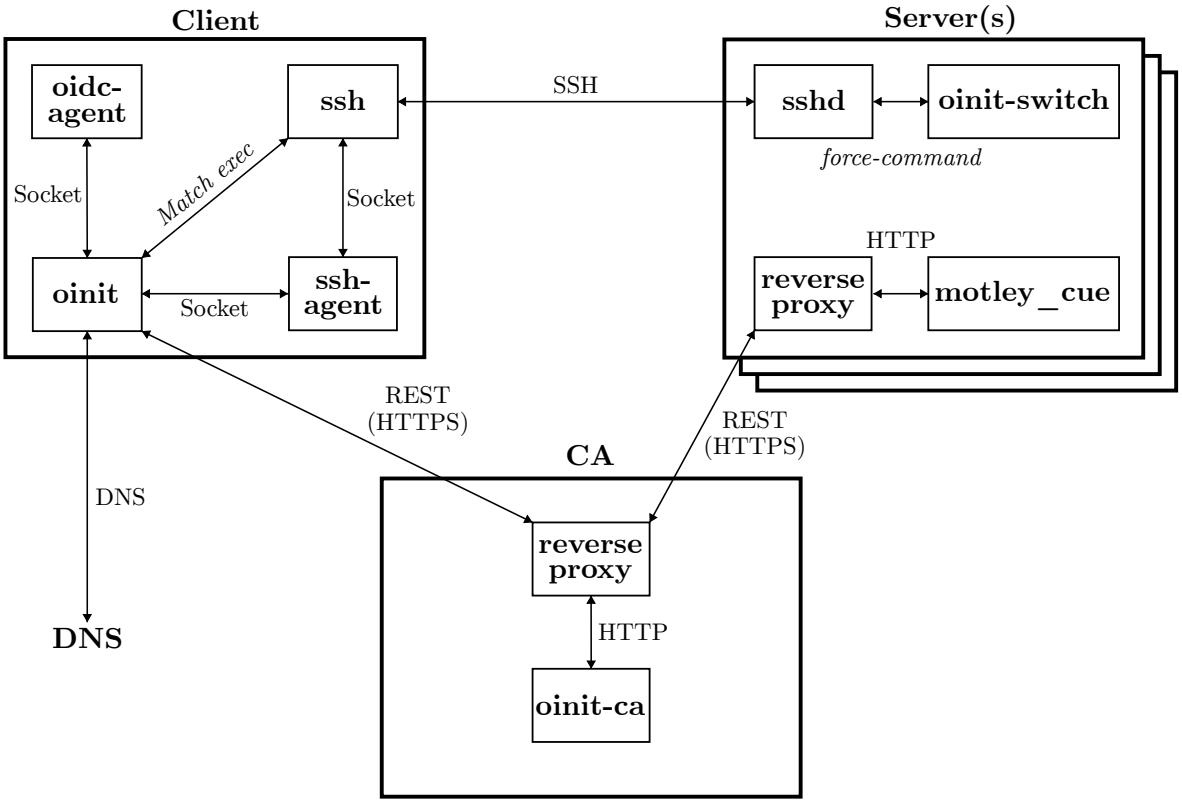

Figure 5.1: Architecture of oinit.

On the client, the `oinit` command-line interface allows the user to enable OpenID Connect authentication for specific OpenSSH servers. The Domain Name System (DNS) is used to determine the CA responsible for an OpenSSH server, for which authentication should be enabled and prepared. This process is described in detail in the following section (5.3). `oinit` is invoked via the `Match exec` configuration keyword whenever the user issues a `ssh` command. Communication with `oidc-agent` and `ssh-agent` is realized with Unix sockets. For communication with the CA, REST via HTTPS is used.

The `oinit-ca` offers a REST API via HTTP and runs behind a reverse proxy which supports HTTPS. The CA issues user and host certificates for multiple clients and OpenSSH servers. For this, it can be configured with URLs of multiple `motley_cue` instances. `oinit-ca` uses `motley_cue`'s REST API to request supported providers and to deploy user accounts. The CA is designed to be able to operate separately from any OpenSSH server. Due to its REST API, it is loosely coupled and independent from any SSH connection. For simpler scenarios with a single OpenSSH server, `oinit-ca` however can also be operated on the same physical or virtual server as `sshd` and `motley_cue`.

On all servers supporting OpenID Connect authentication, `motley_cue` is deployed behind a reverse proxy for HTTPS support. The OpenSSH daemon is configured to accept user certificates issued by the CA, however no further custom configuration is required. After successful authentication of a client, `sshd` calls `oinit-switch` as it is embedded in user certificates with the *force-command* critical option.

We decided to implement all programs in the Go programming language, mainly be-

cause of three reasons:

- `crypto/ssh` **package**: The built-in `crypto/ssh` package implements a SSH client and server and therefore supports, among other things, generation and parsing of keys, as well as creation and signing of certificates. In addition, the package implements the agent protocol [14], which allows direct interaction with the authentication agent without the need to rely on installed programs such as `ssh-add`. A drawback of this implementation is the limitation to Unix sockets, therefore some operating systems such as Microsoft Windows are not supported.

- `oidc-agent` **library**: An official library for interacting with `oidc-agent` called liboidcagent-go[1] is available. The library communicates with the agent via Unix sockets and supports requesting access tokens directly from Go code. Like the `crypto/ssh` package however, it currently does not support the Microsoft Windows operating system.

- **Typed, compiled, cross-platform language**: Go is a typed language which compiles to statically linked binaries, which means that users do not need to have any dependencies such as an interpreter installed, as is the case with e.g., Python programs. The language can compile to programs for all major operating systems and CPU architectures, for this it includes features for easy cross-platform development such as build constraints and build configuration via environment variables[2].

Like OpenSSH Portable, all programs are part of a single code repository which allows the same code to be shared between multiple programs. The repository follows the *Standard Go Project Layout*[3], a non-official but widely recognized project structure. The implementation of `oinit-ca`, `oinit` and `oinit-switch` is described in sections 5.4, 5.5 and 5.6.

5.3 Two-step process

The `oinit` command-line program must allow the user to enable certificate-based authentication for specific OpenSSH servers that support it. Additionally, some configuration of `ssh` is required to enable OpenSSH certificate authentication (as outlined in section 4.3.6). Therefore, the first SSH connection to an OpenSSH server is preceded by a one-time preparation step described in section 5.3.1. After that, any SSH connection to this server can use authentication based on certificates and follows the same steps describe in section 5.3.2. Figure 5.2 shows in which step certificate generation and `ssh` configuration take place.

[1] `https://github.com/indigo-dc/liboidcagent-go/tree/master`
[2] `https://pkg.go.dev/cmd/go`
[3] `https://github.com/golang-standards/project-layout`

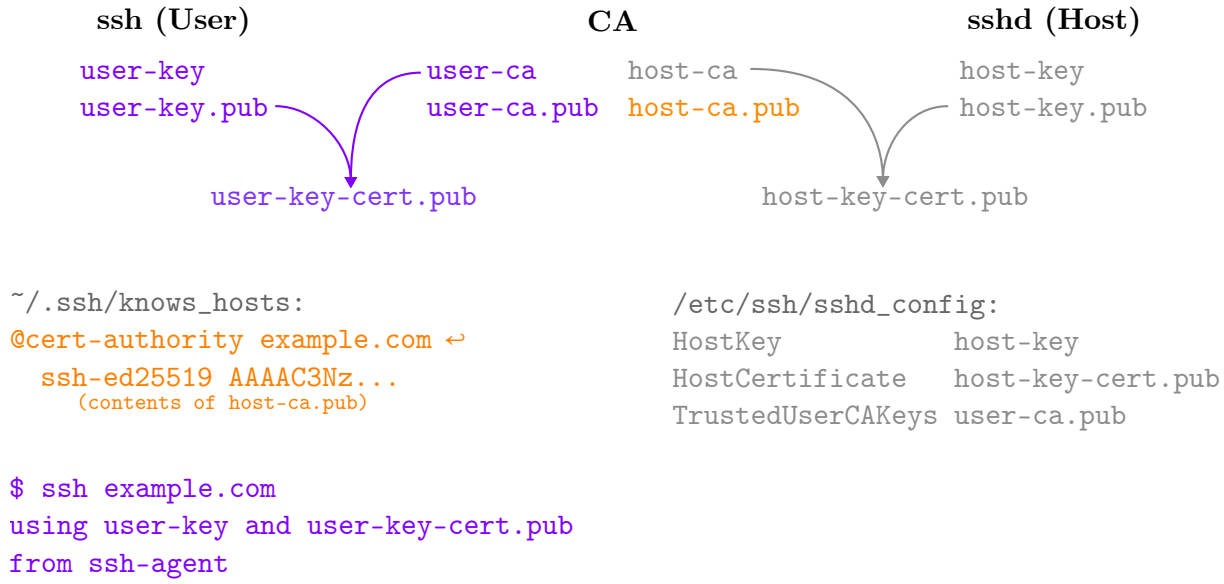

Figure 5.2: Steps undertaken in preparation and later connection steps. Uncolored steps are configured manually before when `oinit-ca` is installed and the OpenSSH daemon is configured.

5.3.1 Preparation

Figure 5.3 shows the steps undertaken in preparation. When enabling OpenID Connect authentication for an OpenSSH server (host), the `oinit-ca` responsible for that server must be found first. The user might specify the HTTP URL manually, if not DNS is used to determine it. For this, the OpenSSH host administrator must publish a TXT record containing the CA URL beforehand. If the OpenSSH host address is `login.example.com`, `oinit` will search for a TXT record publish for the domain `_oinit-ca.login.example.com` or `_oinit-ca.example.com`. Additionally, users can enable OpenID Connect authentication for multiple servers using a wildcard pattern like `*.example.com`, which matches all subdomains of `example.com`.

As the `~/.ssh/known_hosts` file must be modified to support certificate authentication (see section 4.3.6), the OpenSSH host CA public key must be determined. For this, `oinit` calls `oinit-ca` via its REST API and appends the received public key to the known hosts.

`oinit` also writes the CA URL and OpenSSH host address to its own `~/.ssh/oinit_hosts` file. This is necessary to distinguish between hosts with and without OpenID Connect authentication enabled. Additionally, this way the CA URL must not be determined again every time `oinit` is invoked.

Lastly, the preparation step also modifies the `ssh` configuration file at `~/.ssh/config`. A `Match exec` section is added to invoke `oinit` automatically every time a `ssh` command is run. This is only done once and only if the configuration does not already contain `oinit`'s `Match exec` section.

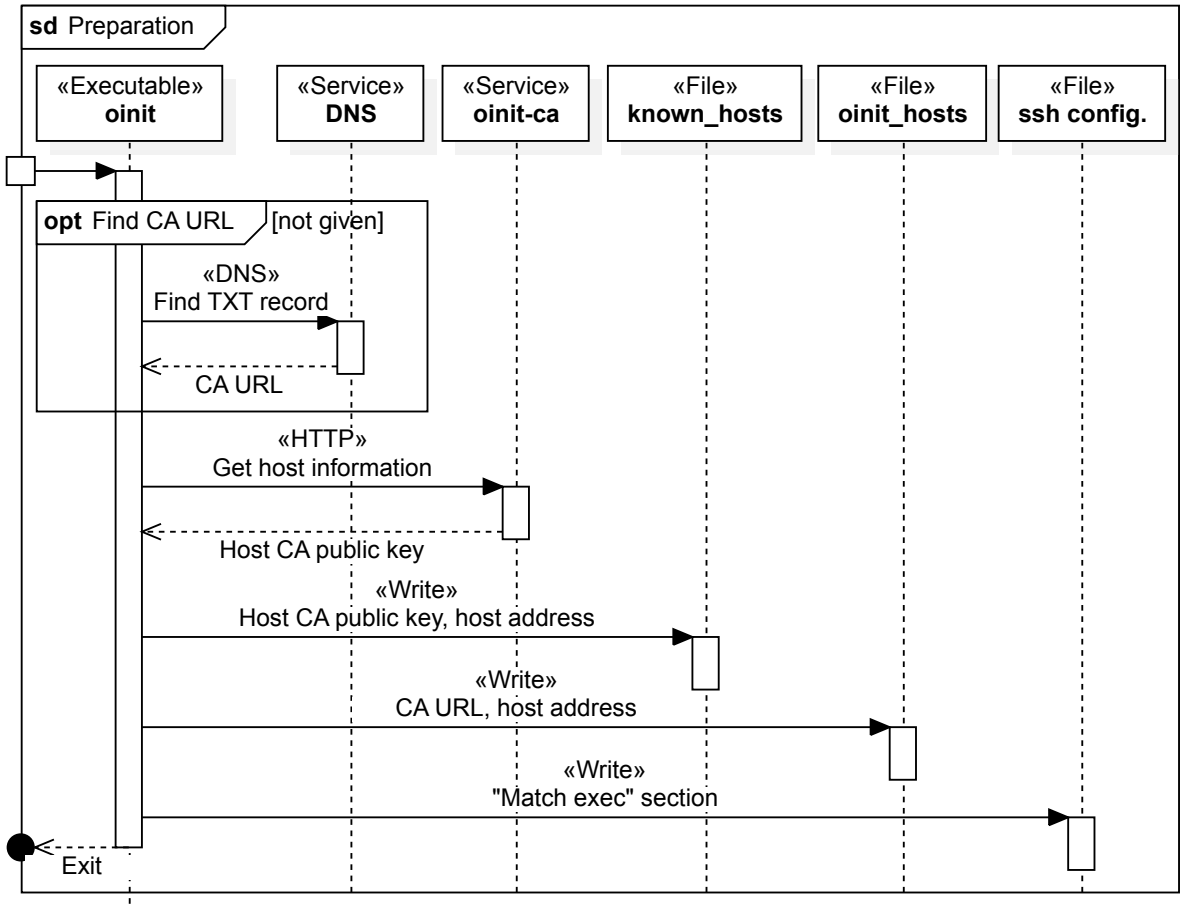

Figure 5.3: Sequence diagram of steps undertaken in preparation process. Note that the reverse proxy in front of `oinit-ca` is not displayed.

5.3.2 Connection

As shown in figure 5.4, `oinit` is automatically invoked via the `Match exec` section added in the preparation process whenever a `ssh` command is run. `oinit` first reads its `~/.ssh/oinit_hosts` to determine whether the targeted host has OpenID Connect authentication enabled. If not, `oinit` exists with a non-zero status code to indicate that `ssh` shall continue with evaluation of the remaining configuration file. Otherwise, a process with the end goal of obtaining an OpenSSH certificate is started.

First, `oinit` makes sure that no certificate is already present in the agent. If it is, then the user has already connected to this host before and received a certificate that is still valid. `oinit` can therefore exit and does not need to request a new certificate.

Next, an OpenID Connect access token is required. The user might have given a token via environment variables to `oinit`, in which case this step is skipped. `oinit` first requests a list of supported identity providers and required scopes from `oinit-ca`. As the CA knows `motley_cue`'s address and port, it forwards this request. The received list of providers and scopes is then forwarded back to `oinit`. The user is then prompted to select a provider. Already configured and loaded accounts in `oidc-agent` are considered and

presented to the user in this step. Additionally, an optional pre-selection of a provider via environment variables is supported. After a selection is made by the user, `oidc-agent` is called to obtain an access token.

`oinit` then requests an OpenSSH certificate from `oinit-ca` using the obtained access token and the public key of a freshly generated SSH key pair. `oinit-ca` uses then access token to request deployment of a user account from `motley_cue`. If a user account for the federated identity belonging to the access token already exists, no new account is generated. With the returned username from `motley_cue`, `oinit-ca` then generates a new OpenSSH certificate based on the given public key. It is signed with the user CA private key and sent back to `oinit`, which loads both the certificate and the previously generated private key into `ssh-agent`. `oinit` exists with a status code of zero, which indicates to `ssh` that the `Match exec` section matches, and defined keyword must be applied to the SSH connection. This include the defined service user "oinit", which is used for the connection.

`ssh` automatically selects the freshly received certificate and private key from `ssh-agent` to authenticate at `sshd`. After successful authentication, the *force-command* is executed and invokes `oinit-switch`. With the username generated by `motley_cue` before, `oinit-switch` prepares the user environment and switches to the personal user account. Any SSH command which might have been specified by the user is run, otherwise an interactive shell is presented. When the executed command or the shell exits, the SSH connection is closed as usual.

5.4 Certificate authority

`oinit-ca` is responsible for three tasks:

1. Offering public information about multiple OpenSSH servers (host CA public key, supported providers & scopes)

2. Deploying users in exchange for OpenID Connect access tokens by calling `motley_cue`

3. Issuing OpenSSH certificates containing `oinit-switch` and the deployed username as *force-command*

`oinit-ca` is implemented as a standalone executable that offers a REST API built using the Gin framework[4]. For communication with `motley_cue`, a simple API client is implemented. Otherwise, only built-in Go packages such as `crypto/ssh` are used.

As `oinit-ca` only implements unencrypted HTTP, however transferred data contains sensitive information such as OpenID access tokens, a reverse proxy such as nginx, traefik, Caddy or the Apache web server that terminates HTTPS should be used.

[4] `https://github.com/gin-gonic/gin`

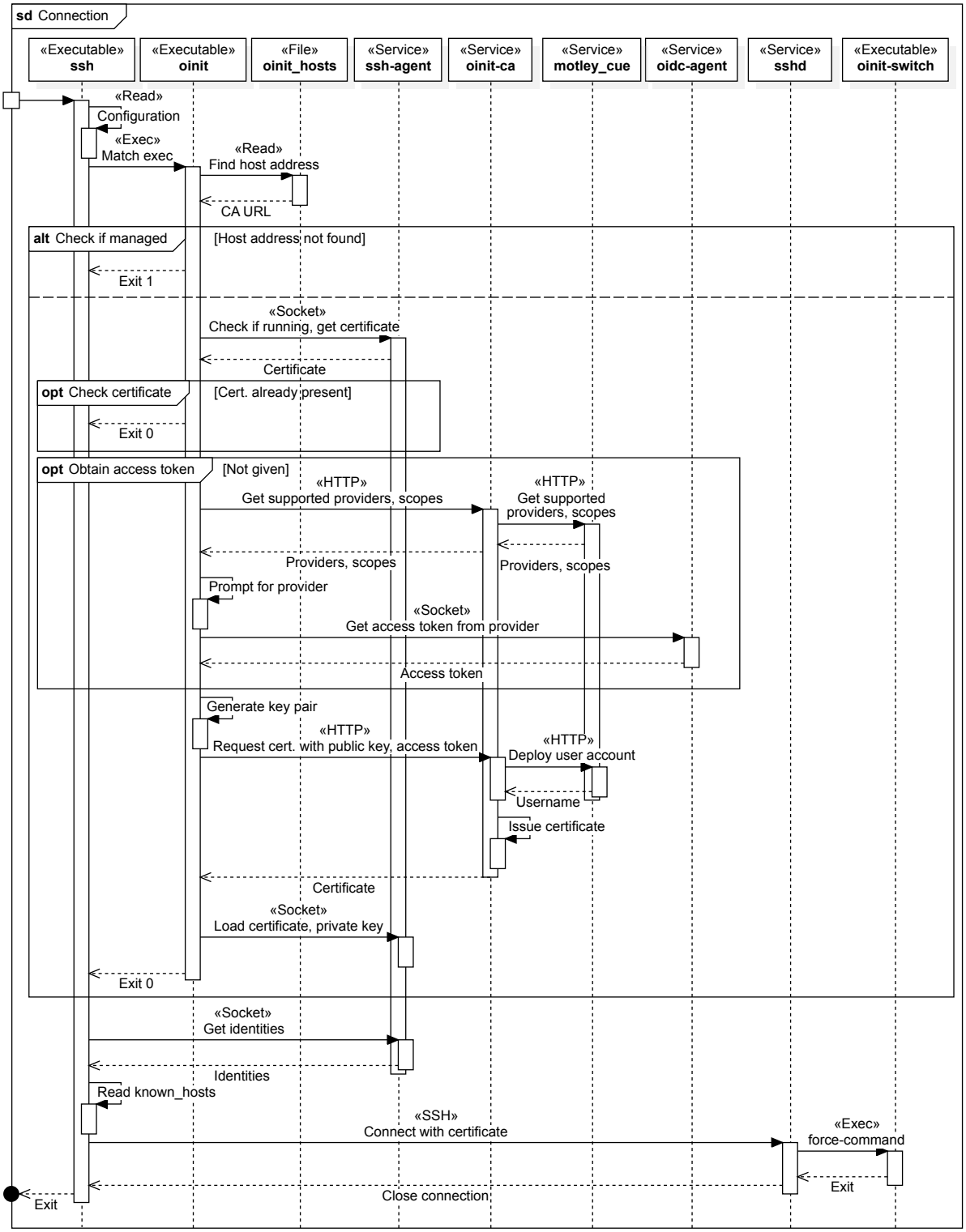

Figure 5.4: Sequence diagram of steps undertaken in connection process. Note that the reverse proxies in front of `oinit-ca` and `motley_cue` are not displayed.

5.4.1 API endpoints

Three REST endpoints covering all CA tasks are implemented in `oinit-ca`. All request and response bodies are formatted in JSON. JSON is commonly used for REST APIs,

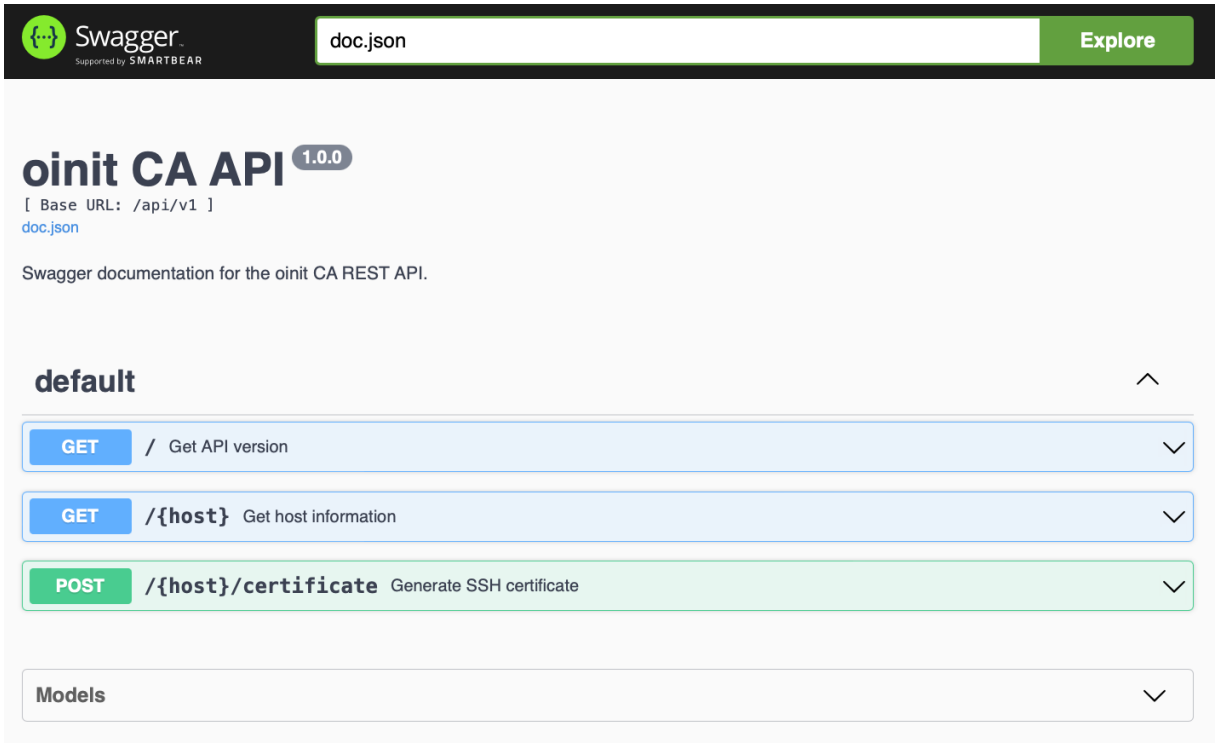

Figure 5.5: `oinit-ca` Swagger documentation.

follows a simple specification and has widespread support in major programming languages. Figure 5.5 shows the Swagger UI based on the OpenAPI specification[5] generated and included in `oinit-ca`. A small API client for `oinit-ca` is also implemented for use in the `oinit` command line client.

Get API version

The `GET /version` endpoint returns the API version supported by the `oinit-ca` instance (currently 1.0.0). This is useful for future revisions of the API, as clients such as `oinit` can dynamically change their behavior based on available endpoints.

Get host information

For a given OpenSSH host address (such as `login.example.com`), the `GET /{host}` endpoint returns both the host CA public key and supported providers (along with required scopes). The host CA public key is requested by `oinit` in the preparation process (see section 5.3.1) and added to `~/.ssh/known_hosts`. The list of supported providers is used in the connection process (see section 5.3.2) and displayed to the user for selection. For this information, `oinit-ca` calls the `motley_cue` instance running on the OpenSSH host. A small API client supporting required `motley_cue` endpoints is implemented for this. To improve performance, the response by `motley_cue` is cached for a configurable time (10 minutes by default).

[5] `https://www.openapis.org`

Generate SSH certificate

The `POST /{host}/certificate` endpoint is used by `oinit` in the connection process (section 5.3.2). Given an OpenID Connect access token and OpenSSH public key, `oinit-ca` calls `motley_cue` to deploy a personal user account. If the user is authorized to use the service (configured in `motley_cue` based on e.g., levels of assurance, virtual organization/group memberships), a new OpenSSH certificate is generated based on the given public key. This certificate is then signed with the according user CA private key and returned as serialized string.

An example OpenSSH user certificates issued by `oinit-ca` is shown in figure 5.6. The *Public key* field contains the public key given in the API request, while *Signing CA* is a signature created using the user CA private key. The *Key ID* field is set to `oinit@<host address>` with the host address being the address of the OpenSSH server. This field is used by `oinit` to determine which host a certificate was issued for (and therefore whether requesting a new certificate is required or not), as OpenSSH certificates do not offer this information by default. A *Serial* number is not used and therefore set to the default value of 0. The certificate validity interval is set according to the `oinit-ca` configuration (see section 5.4.2). The list of principals includes both the "oinit" service user used by SSH connections, as well as the username generated by `motley_cue`. While not necessary, this allows the user to connect using his personal user account instead of the service user, if known. Special care is taken in `oinit-switch` to account for this. The *force-command* is set to the `oinit-switch` program along with the generated username as program argument. Lastly, the *Extensions* are reduced to only two permissions compared to the default list for security reasons. The default list includes permission for port and X11 forwarding, which we do not expect to be required in most scenarios. `oinit-ca` can be easily extended to allow a configuration of these extensions.

```
$ ssh-keygen -L -f example-user-key-cert.pub
example-user-key-cert.pub:
        Type: ssh-ed25519-cert-v01@openssh.com user certificate
        Public key: ED25519-CERT SHA256:rkSKv...
        Signing CA: ED25519 SHA256:xw9aV... (using ssh-ed25519)
        Key ID: "oinit@login.example.com"
        Serial: 0
        Valid: from 2023-09-01T14:30:00 to 2023-09-02T14:30:00
        Principals:
                oinit
                user
        Critical Options:
                force-command oinit-switch user
        Extensions:
                permit-agent-forwarding
                permit-pty
```

Figure 5.6: Example of an OpenSSH user certificate issued by `oinit-ca`.

5.4.2 Configuration

`oinit-ca` supports a configuration file that specifies which OpenSSH hosts are managed by the CA. In addition, the configuration file contains the required `motley_cue` REST API URLs for all hosts. Figure 5.7 shows an example configuration file in the INI file format. The configuration file is separated into *host groups* via INI sections, such as `[example.com]`. The host group name itself is not used and rather helpful for administrators. A host group contains an arbitrary number of OpenSSH hosts along with their `motley_cue` API URLs, which the `oinit-ca` instance supports and issues certificates for. The figure shows the definition of two hosts, one of which makes use of wildcard matching to match any OpenSSH host with a domain ending in `.pool.example.com`. All hosts in a host group share the same configuration options which include the host and user CA public key pair used for OpenSSH certificates, a duration for which `motley_cue` responses are cached, and the validity of issued user certificates. Validity intervals start with the current time/date and can be configured to end after either a fixed duration or to inherit the end time from the OpenID Connect access token.

An arbitrary number of host groups with unique names can be configured. Additionally, all mentioned configuration options can be defined on top of the file (outside any host group) to act as default values for all host groups.

```
1  [example.com]
2  # <host address>    = <motley_cue API URL>
3  login.example.com   = https://login.example.com:8443
4  *.pool.example.com = https://login.pool.example.com:8443
5
6  host-ca-privkey = /etc/oinit-ca/host-ca
7  host-ca-pubkey  = /etc/oinit-ca/host-ca.pub
8  user-ca-privkey = /etc/oinit-ca/user-ca
9  user-ca-pubkey  = /etc/oinit-ca/user-ca.pub
10 cache-duration = 600    # in seconds
11 cert-validity  = token # inherit from access token
```

Figure 5.7: Example `oinit-ca` configuration file.

5.5 Command-line client

The `oinit` command-line program must allow the user to enable OpenID Connect authentication for specific OpenSSH servers. In addition, it is also invoked by `ssh` to prepare said authentication. Therefore, `oinit` implements multiple sub-commands including `add`, `list`, `delete` and `match`.

5.5.1 Adding and deleting hosts

The `oinit list` sub-command lists all OpenSSH servers for which OpenID Connect authentication is currently enabled. This includes servers added by the user (to `~/.ssh/oinit_hosts`), as well as a system-wide configuration (`/etc/ssh/oinit_hosts`). While the system-wide configuration can only be manually modified by a system administrator, it is useful for multi-user systems or computer pools.

To add new hosts, users can run the `oinit add <host address> [CA URL]` sub-command. If no CA URL is given, it is automatically determined using DNS. The `oinit add` sub-commands implements the preparation process described in section 5.3.1. It calls `oinit-ca` to receive a host CA public key, which is written to `~/.ssh/known_hosts` using a special `@cert-authority` syntax (see figure 5.2). The enabled host address alongside the CA URL is written to the custom `~/.ssh/oinit_hosts` file shown in figure 5.8. Note that this is not the `motley_cue` API URL, which is unknown to `oinit`. Additionally, the `~/.ssh/config` is modified once to include a `Match exec` section invoking the `oinit match` sub-command (see figure 5.9).

OpenID Connect authentication can also be disabled again using the `oinit delete` sub-command, which removes the host address from `~/.ssh/oinit_hosts` as well as existing certificates from the `ssh-agent`.

```
$ cat ~/.ssh/oinit_hosts
login.example.com:22 https://login.example.com
```

Figure 5.8: Example `~/.ssh/oinit_hosts` file.

```
$ cat ~/.ssh/config
# This 'Match' block was added by oinit.
#
# Please make sure it stays positioned on top of your ssh config
# file, assuring it will be applied before other 'Host' or 'Match'
# blocks that may interfere with oinit.
Match exec "oinit match %h %p"
        User oinit

... (remaining configuration)
```

Figure 5.9: `ssh` configuration file with `oinit`'s `Match exec` section.

5.5.2 Connecting to hosts

The `oinit match <host> <port>` sub-command is invoked by `ssh` with host and port replaced by actual values of the target OpenSSH server. This sub-command implements the connection process described in section 5.3.2.

The `crypto/ssh` package is used for communication with `ssh-agent`, as well as generation of an OpenSSH key pair and de-serialization of received certificates. For communication with `oidc-agent`, the liboidcagent-go library is used. A simple API client is used to interact with `oinit-ca`. Otherwise, only built-in Go packages are used.

A notable detail is that everything written to the standard output (`stdout`) is not displayed to the user, but rather hidden by the `sshd` process. To circumvent this, especially because `oinit match` must prompt the user for an identity provider to use, the program writes to the attached Teletypewriter (TTY) directly.

5.6 User switching

The `oinit-switch` program is invoked by OpenSSH, due to it being embedded in the issued certificates as *force-command* option. It is responsible for transparently switching to the correct Unix user (referred to as target user here), setting up the user environment and invoking a shell. Additionally, `oinit-switch` must be able to execute commands directly instead of starting an interactive login session in case a command was given to OpenSSH.

When `oinit-switch` is invoked, the username of the target user is given as program argument. As all connections to OpenSSH use the oinit user, the program is executed as that user. The oinit user is set up without any special permissions or access to the sudo[6] command, the challenge therefore is to find a way to allow switching to target users without further input such as a password.

5.6.1 *setuid* bit and dropping privileges

As switching to or executing programs as an arbitrary other user without prompting for a user password requires root permissions, our first approach was to make use of the *setuid* kernel feature. By setting the *setuid* bit on executable file (with the command `chmod u+s`), the file is executed as the user owning the file, instead of as the user issuing the command to execute it. This results in the spawned Linux process having its *effective uid (euid)* set to the executable owner's id, while the *process uid (uid)* remains as the one of the issuing user. The *setuid* bit is used in many programs in a default Linux-based operating system. For example, the `passwd` program allows any user to change his/her own password after a prompt for the current password. For it to be allowed to write the new hashed password to the `/etc/passwd` file (for a local Unix account), which is owned and only writable by the root user, the `passwd` program has the *setuid* bit set and is therefore executed with permissions of the root user. By using root as the owner of the `oinit-switch` program and setting the *setuid* bit, the program is therefore executed with root permissions. When run by OpenSSH, this allows the program to then make use of the similarly named `setuid` system call to drop its root privileges and instead be executed with the *uid* and *euid* of the target user. Any further interaction of the

[6]Short for *superuser do* or *substitute user do*; a common program used on Unix operating systems to execute commands as the super user (root user).

program with the operating system or kernel is therefore executed as the target user, re-gaining root permissions is not possible. To prevent any already logged in user from exploiting the `oinit-switch` program to switch to an arbitrary user, the program makes sure that the user executing the program is oinit by checking the *uid*. After switching to the target user, `oinit-switch` then prepares the environment by changing to the target users' home directory and setting appropriate values for environment variables such as HOME, SHELL, USER and PATH. Variable values such as the user's home directory and preferred shell are read from the `/etc/passwd` file. `oinit-switch` then continues to spawn an interactive login shell or execute a command directly, if given to OpenSSH.

While this approach does work for local Unix accounts, any other source of users configured on the system (such as LDAP, which `motley_cue` supports) will not. Additionally, this sketched approach does not yet consider the `/etc/shadow` file, which contains aging information about local Unix accounts. This is problematic, as `motley_cue` writes a past date into the *account expiration date* field when suspending local Unix users. By not respecting this file, `oinit-switch` still allows suspended users access to the system. Also, so far little attention has been paid to the configuration of the user environment. While the minimal required environment variables HOME, SHELL, USER and PATH are set to sensible values, no system configuration files such as `/etc/login.defs` are considered.

5.6.2 Unix tools

Because the extension of `oinit-switch` to reliably support PAM/Name Service Switch (NSS) and various system configuration files (including but not limited to `/etc/shadow` and `/etc/login.defs`) is out of scope for this thesis, error-prone and a re-implementation in Go does not yield a conceivable benefit, we instead opted to rely on existing and well-established Unix tools. In our research of well-established and commonly available Unix tools suited for use in `oinit-switch`, we considered `runuser`, `su` and `sudo`. While the manual page of `runuser` states that it is intended to be used by the root user, using the *setuid* bit on the `oinit-switch` executable file and then setting the *process uid* to 0 at runtime allows us to use the program anyway. However, the manual page also explains that most versions of `runuser` (like on Debian GNU/Linux) do use PAM for session management, however no authentication information are read from PAM, which disqualifies it for use in `oinit-switch`.

In contrast, the `su` program (from whose source code `runuser` is compiled with reduced features) is intended to be used by any user and does support PAM authentication information. `su` also considers system login configurations (such as the `/etc/login.defs` and `/etc/shadow` files) for local Unix accounts. Access to suspended or limited accounts is therefore not granted. Additionally, the program can be configured in a dedicated PAM configuration file (usually `/etc/pam.d/su`). This enables a configuration based on the `pam_succeed_if.so` module that allows a certain user to switch to arbitrary users without being prompted for the target user password (see figure 5.10). This also eliminates the need for the *setuid* bit on the `oinit-switch` executable. `su` is part of the `util-linux` package, which is installed by default on all major Linux distributions. We therefore decided to use `su` in `oinit-switch`.

```
# Allow 'su <target>' without password if executing user is named oinit
auth [success=ignore default=1] pam_succeed_if.so use_uid user = oinit

# However, target user must not have uid 0 (root user)
auth sufficient                 pam_succeed_if.so uid ne 0
```

Figure 5.10: Extract from configuration file `/etc/pam.d/su` which allows user oinit to switch to an arbitrary user (except root) without being prompted for a password.

We also considered the commonly used program `sudo`. Like `su`, it does support switching to and the execution of commands as another user (with the `-u` argument). As it also considers any PAM authentication configuration and can be configured to allow password-less execution for certain users, `sudo` is a viable alternative. We however decided against it, as its configuration file (`/etc/sudoers`) does not allow excepting root as target user for password-less execution, which poses a security risk if not handled carefully. In addition, it is not always installed on Linux-based operating systems by default and therefore offers not benefit compared to `su` in our use case.

5.6.3 SSH command

In addition to switching users, `oinit-switch` must also consider that a user might have passed a command to execute, instead of requesting an interactive shell. Normally, the OpenSSH daemon `sshd` detects the passed command and executes it automatically. Due to the use of the *force-command* option in the oinit authentication flow however, any command specified by the user is overwritten and ignored. Because of this, `oinit-switch` must manually check for a possibly given command to execute. According to the `sshd` manual page, the original command is however always preserved in the SSH_ORIGINAL_COMMAND environment variable [54]. `oinit-switch` is therefore able to check if the variable is set and then execute it.

After some preparation and error checking, `oinit-switch` executes the `su` command with the correct arguments. Instead of making use of Go's `Output()` or `Run()` helper functions from the `os/exec` package, `oinit-switch` directly executes the `execve` system call, provided via the `Exec()` function from package `syscall`. This approach was chosen because Go's `os/exec` package uses follows a *fork*[7] & *execve* approach. The use of *fork* results in an additional sub-process as a copy of itself, which is then immediately replaced by the `su` command. As the `oinit-switch` process only waits for its sub-process (the `su` command) to exit and then exits itself, it is unnecessary and wastes system resources such as memory. The direct use of the `execve` system call without a *fork*/*clone* prevents this additional process.

Figures 5.11 and 5.12 show the difference between these approaches. Note that in the latter figure, the `oinit-switch` process replaced itself with `su` and is no longer visible. The seemingly duplicate `sshd: oinit@pts/0` process executed as user oinit is explained

[7] *fork* is the standard C library *libc* function name, the actual system call is named *clone*.

```
USER  Command
root  ├ sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root  │ └ sshd: oinit [priv]
oinit │   └ sshd: oinit@pts/0
oinit │     └ oinit-switch user
root  │       └ su - user
user  │         └ -bash
```

Figure 5.11: Extract of process tree for an interactive `ssh` login using the `os/exec` package (*fork & execve*).

```
USER  Command
root  ├ sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root  │ └ sshd: oinit [priv]
oinit │   └ sshd: oinit@pts/0
root  │     └ su - user
user  │       └ -bash
```

Figure 5.12: Extract of process tree for an interactive `ssh` login using *execve* directly.

by a privilege separation technique employed by sshd[8]. One can also observe that the `su` process follows the *fork & execve* approach instead of replacing itself via *execve* directly. This is explained by `su` needing to do PAM-related clean up tasks after the exit of its child process[9].

5.7 User workflow

After installation and configuration of all oinit programs, as well as `motley_cue` and `sshd`, `ssh` is ready to use OpenID Connect authentication for specific OpenSSH servers. First, the desired OpenSSH server must be added with the `oinit add` sub-command (figure 5.13). `oinit list` shows that the added host is now managed by `oinit`. Opening a SSH connection (without specifying a username) invokes `oinit` automatically. The user is asked to select from a list of supported OpenID Connect identity providers. Configured and loaded accounts in `oidc-agent` are also shown to the user.

After selection, the user may be asked for a passphrase by `oidc-agent`. If an identity provider is selected, for which no `oidc-agent` configuration exists, it is dynamically created[10]. This may include opening one or multiple windows (not shown), after which focus is returned to the terminal.

The user is informed that a temporary OpenSSH certificates was received, after which

[8] `https://serverfault.com/a/585782`

[9] `https://github.com/util-linux/util-linux/issues/325#issuecomment-227398119`

[10] Since `oidc-agent` version 5.

```
$ oinit add example.com
i Determined CA from DNS: https://ca.example.com
+ example.com:22 was added.

$ oinit list
i The following hosts are managed by oinit:
    example.com:22

$ ssh example.com
[1] https://aai-dev.egi.eu/auth/realms/egi
[2] https://aai.egi.eu/auth/realms/egi
[3] https://accounts.google.com
[4] https://iam.deep-hybrid-datacloud.eu
[5] https://login-dev.helmholtz.de/oauth2 (Accounts: helmholtz)
[6] https://oidc.scc.kit.edu/auth/realms/kit (Accounts: kit)
[7] https://wlcg.cloud.cnaf.infn.it
? Please select a provider to use [1-7]: 6
+ Received a certificate which is valid until 2023-09-01 14:30:00 +0200 CEST
nogroup001@example.com:~$ whoami
nogroup001

$ scp test.txt example.com:~
test.txt                  100%   50KB 549.6KB/s   00:00

$ ssh example.com "whoami && ls -l *.txt"
nogroup001
-rw-r--r-- 1 nogroup001 nogroup 5 Aug  9 15:54 test.txt
```

Figure 5.13: Output of `oinit add`, `oinit list` and a subsequence SSH connection. `nogroup001` is the username generated by `motley_cue`.

he/she is presented with an interactive login shell, running as a personal user account. Programs relying on SSH, such as `rsync` and `git` or `scp` (shown in figure) work as expected. Commands directly specified by the user are also executed as the personal user account.

Evaluation

In this chapter, we evaluate our implementation regarding correctness, performance, security, and the achievement of the self-defined goals.

6.1 Testing

To verify the correctness of individual functions throughout all oinit programs, we implemented unit test cases. We make use of Go's built-in testing capabilities (the `testing` package and `go test` command) as well as the widely used toolkit testify[1]. All unit tests are run as part of a Continuous Integration (CI) pipeline in the source code repository.

As the correct interaction of all oinit programs among themselves as well as `ssh`, `sshd`, `ssh-agent`, `oidc-agent`, and `motley_cue` is critical, we also run an integration test on three different operating systems. This test simulates an entire workflow by installing and configuring all programs, enabling OpenID Connect authentication for a local OpenSSH daemon and finally executing a command via SSH, verifying that authentication and command execution succeeded.

To evaluate the `oinit-ca` REST API, we ran a collection of fuzzing tools[2] randomly testing all routes with different HTTP request headers and bodies. While no problems directly related to oinit could be found, we followed recommendations of changing some HTTP response status codes to better suited ones. The fuzzing tool was able to force some HTTP 500 response codes[3] for excessively long request headers, however these errors are known limitations of Go's built-in HTTP implementation, therefore not related to oinit directly and do not represent a problem for real-world scenarios.

To get notified about bugs, performance issues and possible security vulnerabilities, we also regularly run the static analysis programs `staticcheck`[4], `go vet` and GitHub's CodeQL Go analyzer as part of the CI pipeline.

[1] `github.com/stretchr/testify`
[2] `https://endava.github.io/cats/`
[3] This code indicates an internal server error
[4] `https://staticcheck.dev`

6.2 Latency

To offer a good user experience, authentication using the oinit programs should be fast with as little latency as possible. As oinit does not implement any computationally intensive algorithms but rather interconnects multiple services across multiple servers, the network latency plays a major role in the perceived performance. Based on the sequence diagrams shown in 5.3, we aimed to reduce the number of network requests.

First, whenever `oinit add` is run to enable OpenID Connect authentication for an OpenSSH host, the automatically determined `oinit-ca`/reverse proxy URL is permanently stored in `~/.ssh/oinit_hosts`. This prevents DNS requests every time a connection to the enabled server is made. If the CA URL does change at some point, users are required to delete and add the host via `oinit del`/`oinit add` again.

Second, we introduced a temporary cache for the request of supported providers from `oinit-ca` to `motley_cue`. This prevents a chained HTTP request, for which the SSH client must wait. We do not expect the supported providers to change regularly for a `motley_cue` instance, therefore the first response is cached for a configurable time. By caching the `motley_cue` response at the CA and not the `oinit` client program, all clients communicating with the CA benefit from decreased response times.

We found that all remaining network requests are necessary and cannot be cached.

6.3 Security

As defined in **G6**, authentication using oinit must be secure and tamper-proof. By using HTTPS for all communication between REST APIs, we make sure that no sensitive data (e.g., access tokens) can be read by a third party. For the SSH connection itself, oinit relies on standard OpenSSH certificate authentication without modifications, we therefore rely on OpenSSH's security guarantees.

Still, we identified two possible security problems related to the use of certificate authentication and user switching. As an OpenSSH administrator delegates authorization decisions to a certificate authority, he/she must trust the CA to correctly issue user certificates with the embedded *force-command*. Section 6.3.1 describes the problem and mitigation of a missing *force-command*. On the OpenSSH server, we rely on `oinit-switch` to securely switch to a personal user account. A security problem we identified due to the use of `su` is described in section 6.3.2.

6.3.1 Service user interactive access

While the oinit service user has no administrative rights, it is allowed via PAM to switch to other user accounts without requiring their password. Interactive shell access to the oinit user therefore poses an opportunity for abuse, which must be prevented.

By default, the user is created without a password and without any configured authorized public keys for SSH access. Nobody except system administrators is therefore

able to directly access the oinit user. However, as this service user is used for all SSH connections before switching to the correct target user (see section 5.6), oinit relies on the execution of the *force-command* present in certificates to not grant interactive access. For example, a *force-command* set to `force-command: 'oinit-switch example'` instructs the OpenSSH daemon to execute `/bin/bash -c 'oinit-switch example'` as the oinit user[5]. If the daemon would not execute the *force-command*, for example due to a bug, interactive shell access as the oinit user would be granted. Similarly, the oinit certificate authority could mistakenly issue a certificate without the *force-command* being set, which would also result in interactive access. While this should never happen, we still want to prevent this attack vector.

To mitigate these unlikely but possible scenarios, we implement a custom shell to replace `/bin/bash` for the oinit user. This program, called `oinit-shell`, must prevent interactive access however still allow the execution of the `oinit-switch` *force-command* for user switching to work. To achieve this, `oinit-shell` carefully checks its arguments to only allow the execution of `/usr/bin/oinit-shell -c 'oinit-switch ...'` but no other command. Passing no arguments also results in an error to prevent interactive access. This approach is inspired by `git-shell`[6], a shell restricting SSH access to only a few commands required for the `git` program to work. After validating its arguments, the passed `oinit-switch` command is executed via the `execve` system call, which replaces the currently running process and initializes a new stack and heap. Without prior forking, this hides the custom shell in the process tree.

To enable `oinit-shell`, it must be configured as the default shell for the oinit user at creation time or later using the `chsh` command.

6.3.2 TTY injection

In Unix based operating systems, the Teletypewriter subsystem is a core concept of interaction with processes. It stems from the early days of computers when physical typewriters were used to interact with computers. In modern kernels such as Linux, the TTY is a subsystem responsible for handling user input including line editing commands such as backspace, clear and reprint. The concept of pseudo terminals (*pty*) with a master and slaves (*pts*) was introduced to allow terminal emulation in userland. A single so-called TTY device[7] can be attached to multiple Linux processes at the same time. While one process receives its input from the device (`stdin`) and outputs to it (`stdout`), the other processes run in the background.

The `ioctl` system call allows interaction with the TTY subsystem and pseudo terminals. Problematic is its `TIOCSTI` command, which can insert arbitrary input to a process's attached TTY. Figure 6.1 shows that by default, the `su` command used in `oinit-switch` for user switching (see section 5.6) is vulnerable to an attack enabled by the `TIOCSTI`

[5]This assumes GNU Bash (*Bourne Again SHell*: `https://www.gnu.org/software/bash/`) as the default shell, however this behavior is equivalent for all shells.

[6]`https://git-scm.com/docs/git-shell`

[7]A TTY device is the TTY driver plus a so-called *line discipline configuration*, residing as a file in the `/dev` directory (e.g., `/dev/pts/0`).

command. A malicious user could stop the `-bash` process using `SIGSTOP`[8] and then make use of the `ioctl` system call to send input to the parent `su` process as if entered by hand, which is run as the root user[9].

```
USER  TTY      Command
root  (no tty) ├ sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root  (no tty) │ └ sshd: oinit [priv]
oinit (no tty) │    └ sshd: oinit@pts/0
root  pts/0    │       └ su - user
user  pts/0    │          └ -bash
```

Figure 6.1: Extract of process tree showing `su`'s default behavior. Note the identical `pts/0` pseudo terminal for both `su` and the `bash` subprocess.

While the `TIOCSTI` command can be optionally disabled since Linux 6.2[10], we expect most Linux based operating systems to still be vulnerable to this attack. In its manual page, the authors of `su` state that the `--pty`/`-P` argument can be used to protect against terminal injection using the `ioctl` system call [55]. The attack is mitigated by creating a separate pseudo terminal slave for the subprocess of `su`, which can be seen in figure 6.2. This is a simple mitigation which we decided to use in `oinit-switch` for interactive sessions.

```
USER  TTY      Command
root  (no tty) ├ sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root  (no tty) │ └ sshd: oinit [priv]
oinit (no tty) │    └ sshd: oinit@pts/0
root  pts/0    │       └ su - user --pty
user  pts/1    │          └ -bash
```

Figure 6.2: Extract of process tree showing `su` with `--pty` argument. Note the different `pts/1` pseudo terminal for the `bash` subprocess.

However, we found that using the `--pty` argument with `su` breaks some non-interactive SSH sessions. This includes programs relying on SSH, such as `scp`, `git` and `rsync`. We identified the cause being that `ssh` by default does not request a TTY device from the OpenSSH daemon when executing command directly. In our case, that means that the `oinit-switch` or its replacing `su` process is started without an attached TTY. The use of `su --pty` for `ioctl TIOCSTI` mitigation however results in a pseudo terminal device being created for the subprocess, nonetheless. This device sets some default input/output related terminal configuration including column/row counts, line break and end of line (EOL) characters. Due to this, the text-based data exchanged by programs such as

[8] A signal sent from the kernel to a process to stop it.

[9] `https://www.errno.fr/TTYPushback.html`

[10] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=83efeeeb3d04`

`scp`, `git` and `rsync` with a respective process on the OpenSSH server via SSH contains unexpected line breaks, resulting in errors.

While we experimented with manual TTY configuration, we found no easy way to prevent this behavior. As we also did not intend to change the source code of programs relying on SSH to be compatible again, we decided to not use `su`'s `--pty` argument for non-interactive commands. However, as `ssh` allows forcing TTY creation from the OpenSSH daemon even for non-interactive access, users aware of this decision could gain access to an interactive session without any mitigation. For example, a user running `ssh -tt example.com /bin/bash` would gain access to an interactive shell process without the `su --pty` mitigation and could therefore run an `ioctl TIOCSTI` attack.

To fix this issue, an option is to whitelist certain commands and allow them to be run without the mitigation. We decided against this, as this requires extensive knowledge of all programs that users might want to run via SSH and updating the `oinit-switch` source code for every new command. Instead, we decided to modify `oinit-switch` to not allow any TTY to be present when executing commands. This does break the legitimate use case of requesting a TTY for directly running interactive commands, such as `ssh -tt example.com htop`[11]. However, we expect the number of users aware of the `ssh -tt` argument to be small in the first place. Additionally, normal interactive access is still always possible by first connecting via `ssh example.org` and then executing `htop` via the interactive shell on the server.

6.4 Verification of goals

In section 4.1, we defined six goals that, according to our own opinion, should be fulfilled by an OpenID Connect authentication mechanism for OpenSSH. These goals are aimed towards a secure implementation and a good user experience. We verify if and how our chosen approach and implementation fulfills these goals.

G1: No change in users' workflows
oinit integrates seamlessly into any `ssh` command, which can be used as always. No wrapper script or program is required to authenticate at the OpenSSH server using OpenID Connect. Users are only required to enable the oinit workflow for one or multiple servers (using wildcards) once. Therefore, users are not required to change their existing workflows, with the one notable exception being that the remote username must not be specified, as this will overwrite the service username used by oinit and prevent successful authentication.

All built-in OpenSSH functionalities such as jump hosts, agent forwarding, and the execution of commands are supported with OpenID Connect authentication. External programs relying on SSH, such as `scp`, `git` and `rsync` can also be used without problems. Due to the mitigation of a security problem, the only built-in functionality not working correctly is the request of a TTY when executing commands, as described in section 6.3.2.

No manual intervention in the form of inserting or copying access token is required

[11] `htop` is an interactive process viewer.

from the user. The user is only prompted to select from a list of supported OpenID Connect identity providers, which can be prevented by a pre-selection using environment variables, if desired.

We conclude that no change in users' workflows is necessary in most cases.

G2: Multiple OpenID Connect identity providers
When a connection to an OpenID Connect authentication-enabled OpenSSH server is made, the user is prompted to select from a list of supported identity providers. This list stems from the configuration of `motley_cue`, which can support multiple OpenID Connect identity providers. We therefore conclude that oinit fulfills this goal.

It must be noted that in its current implementation, oinit always uses an OpenSSH certificate for a given server if present. If a user wants to connect to the same server using a different identity provider, the existing certificate can be manually removed from `ssh-agent`. Unfortunately, `ssh` does not support explicitly selecting a `ssh-agent` identity to use when connecting to a server, therefore using two different identity providers for a single OpenSSH server *simultaneously* without removing an identity is not possible.

G3: Retain the benefits of federated identities
oinit makes use of `oidc-agent`, which enabled SSO-like capabilities by loading so-called configurations into memory. That means that no password or other form of unlocking is necessary after the first time. As oinit supports any OpenID identity provider that `oidc-agent` and `motley_cue` are configured for and only forwards access tokens, interoperability is also granted. Additionally, credential expiry is retained as certificates issued by the oinit CA expire after a configurable time (or at the same time as the OpenID Connect access token). Revocation of credentials is also supported, as oinit does not allow access after user accounts are suspended or limited by `motley_cue`.

As oinit is based on OpenSSH certificates, drawbacks of other authentication methods including passwords and private keys are prevented. The issued certificates are only valid for a limited time and therefore not re-usable limitlessly. As all certificates are signed by a trusted OpenSSH certification authority, forging them is also very difficult. Additionally, the risk of any credential compromise is reduced by never storing any key or certificate on disk. The required OpenSSH private/public key pair is generated and held in memory only, as is any received OpenSSH certificate that is only loaded into `ssh-agent` and never written to file.

G4: Dynamic usernames based on federated identity
Personal user accounts are deployed when a SSH connection is first made to an OpenSSH server. While the user accounts are deployed by an HTTP call preceding the actual SSH connection, this is a design decision reasoned by security reasons (see section 5) and not noticed by the user. Usernames are generated by `motley_cue` based on the claims contained in the access token (which is a JSON Web Token) and therefore chosen by the server. We conclude that user accounts are indeed deployed on-demand, which allows users to connect without knowledge of their personal account username.

G5: No source code changes
oinit extends OpenSSH's functionality by automatically modifying configuration files and making use of built-in features, especially the certificate *force-command* critical option.

No changes to the OpenSSH source code are therefore required. The built-in features oinit relies on (especially certificates and EdDSA keys) have been introduced years ago, therefore no recent version is required.

G6: Secure and tamper-proof
oinit relies on the security of OpenSSH certificates and does not tamper with the SSH connection itself. As discussed in section 6.3, oinit deploys mitigation techniques to prevent unwanted service user interactive access and TTY injection attacks, that could be used to impersonate other user accounts.

We conclude that our implementation mostly fulfills the defined goals, therefore oinit is a viable solution to extend OpenSSH with OpenID Connect authentication. However, the taken design decisions imply some limitations, which we discuss in the following section.

6.5 Limitations

oinit is based on some OpenSSH-specific features, including OpenSSH certificates and `ssh`'s `Match exec` configuration keyword. Therefore, oinit can only be used with the OpenSSH implementation of SSH. While other implementations do support some of OpenSSH's features (for example, PuTTY supports OpenSSH certificates since 2022), they cannot be used as a drop-in replacement for `ssh` or `sshd` in our architecture. We already outlined this limitation alongside the defined goals in section 4.1.

Regarding operating systems, oinit is limited to Unix systems including Linux distributions and Apple macOS. For the `oinit` client program, this is because the used `crypto/ssh` Go package only supports Unix sockets for communication with `ssh-agent`. Similarly, the liboidcagent-go library only supports Unix sockets as well. While both `ssh-agent` and `oidc-agent` are available for non-Unix operating systems such as Windows, missing Go packages/libraries supporting Windows' named pipes make an integration difficult. For the OpenSSH server, oinit is limited to Linux distributions only, as `motley_cue` currently is limited to Linux.

The use of OpenSSH certificates for oinit also introduces additional complexity and limitations. By default, an OpenSSH daemon is not configured to accept certificates and some configuration is necessary (see section 4.3.6). Also, because `sshd` only supports one certification authority simultaneously, the use of oinit prevents the use of OpenSSH certificates for authentication of unrelated users. The same user CA could be used for oinit and authentication of unrelated users, however that requires the possession of the user CA private key which implies the oinit CA being hosted on the same server as OpenSSH.

In general, the use of OpenSSH certificates introduces multiple trust relationships. OpenSSH server administrators give away the user authentication decision to the CA and must trust it to only issue user certificates to trusted users. Similarly, SSH users must trust the CA to not issue host certificates to untrusted servers. In comparison, password and public key authentication rely on Trust On First Use (TOFU). For oinit, this means that if a separate, trusted CA is not desired or possible, the CA must be hosted on the same server as the OpenSSH server.

6.6 Installation

All oinit source code is open source, hosted on GitHub and licensed under the MIT license[12]. The repository also includes documentation for the installation and configuration of all programs[13]. All four oinit binaries `oinit`, `oinit-ca`, `oinit-switch` and `oinit-shell` can be compiled from the same repository using standard Go tools. Alternatively, Linux packages with pre-built binaries are available for Debian- and RPM-based distributions[14]. A Docker container image containing `oinit-ca` is available from the GitHub Container Registry[15] as well.

When installed, the Linux packages install configuration files, create a system user, generate OpenSSH key pairs and set up systemd. However, a few manual configuration steps like the generation of a host CA public key for every desired OpenSSH server are still required. These steps are mentioned in the documentation.

[12] `https://github.com/lbrocke/oinit`
[13] `https://github.com/lbrocke/oinit/wiki`
[14] `https://repo.data.kit.edu`
[15] `https://github.com/lbrocke/oinit/pkgs/container/oinit-ca`

Conclusion

In this chapter, we summarize the presented thesis and give suggestions for possible future work to extend oinit.

7.1 Summary

The goal of this thesis was to extend standard OpenSSH with support for authentication using federated identities. For this, we opted to rely on the established OpenID Connect protocol, which includes benefits such as SSO, scalability and interoperability. Our goals included to prevent any source code changes, while at the same time to provide a seamless integration into standard OpenSSH. We did not want users to be required to change any of their existing workflows. Additionally, we wanted to allow the usage of any OpenID Connect identity provider, for which user accounts are deployed dynamically and on-demand.

In this thesis, we presented our analysis of OpenSSH, including its existing authentication mechanisms and configuration options. Based on this research, we presented a concept and architecture on how to integrate OpenID Connect authentication into OpenSSH without any source code changes. We described the implementation of four individual programs, which interact with OpenSSH as well as `oidc-agent` and `motley_cue` to achieve this thesis' goal. Also, we outlined multiple security problems and our approach of their mitigation. As a result, we present a suite of programs called oinit, that fulfills the goals we outlined for ourselves.

All oinit programs are publicly available under the MIT license, we additionally provide Linux packages and documentation for easy installation. For OpenSSH administrator and users, oinit provides a simple way to extend SSH with OpenID Connect authentication. It can be used alongside other solutions, such as `mccli`, depending on use case and environmental restrictions.

7.2 Future work

oinit is fully usable on Unix operating systems such as Linux distributions and Apple macOS. However, due to limitations of libraries and packages oinit is based on, other operating systems are not supported. This includes Microsoft Windows, for which OpenSSH

and `oidc-agent` are available. To make oinit available for more users, the implementation of named pipes for support of `ssh-agent` and `oidc-agent` are desirable.

In its current implementation, the oinit certificate authority does allow the configuration of issued OpenSSH certificates. This includes the OpenSSH user CA to sign certificates with, as well as the certificate validity. CA administrators may want to restrict or extend issued certificates even more, for example by making use of the certificate extensions feature to allow/restrict the use of agent/port/X11 forwarding or TTY allocation. Therefore, the oinit CA could be extended to allow for further configuration.

Our research and implementation of oinit is limited to OpenSSH, as it is the most widely used implementation of SSH. Due to the use of some OpenSSH-specific feature, other implementations such as Dropbear SSH and PuTTY cannot be used as a drop-in replacement. In future work, attempts at adapting oinit to support other SSH implementations could be made.

Bibliography

[1] 7th Zero. *Results: SSH Statistics Gathering Project*. https://7thzero.com/blog/ssh-statistics-gathering-project-2017-results. [Online; accessed 04-September-2023]. 2017.

[2] Oliver Gasser, Ralph Holz, and Georg Carle. "A deeper understanding of SSH: Results from Internet-wide scans". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. 2014, pp. 1–9. DOI: 10.1109/NOMS.2014.6838249.

[3] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. http://www.rfc-editor.org/rfc/rfc6749.txt. RFC Editor, Oct. 2012. URL: http://www.rfc-editor.org/rfc/rfc6749.txt.

[4] OpenID Foundation. *How OpenID Connect Works - OpenID Foundation*. https://openid.net/developers/how-connect-works/. [Online; accessed 30-July-2023]. 2023.

[5] C. Neuman et al. *The Kerberos Network Authentication Service (V5)*. RFC 4120. http://www.rfc-editor.org/rfc/rfc4120.txt. RFC Editor, July 2005. URL: http://www.rfc-editor.org/rfc/rfc4120.txt.

[6] MIT. *Kerberos: The Network Authentication Protocol*. https://web.mit.edu/kerberos/. [Online; accessed 04-September-2023]. 2023.

[7] Cantor S. et al. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0*. OASIS Standard saml-core-2.0-os. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf. OASIS, Mar. 2005. URL: http://docs.oasis-open.org/%20security/saml/v2.0/saml-core-2.0-os.pdf.

[8] The OpenSSH contributors. *OpenSSH*. https://www.openssh.com. [Online; accessed 18-July-2023]. 2023.

[9] J. Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511. http://www.rfc-editor.org/rfc/rfc4511.txt. RFC Editor, June 2006. URL: http://www.rfc-editor.org/rfc/rfc4511.txt.

[10] Ruti Gafni and Dudu Nissim. "To Social Login or not Login? Exploring Factors Affecting the Decision". In: *Issues in Informing Science and Information Technology* 11 (Jan. 2014), pp. 57–72. DOI: 10.28945/1980.

[11] LoginRadius. *Consumer Digital Identity Trend Report 2022*. https://www.loginradius.com/resource/consumer-digital-identity-trend-report-2022. [Online; accessed 04-September-2023]. 2022.

[12] janrain and Blue Research. *The Value of Social Login - Solving the Engagement Gap*. https://paperform.co/ebook/Industry-Research-Value-of-Social-Login-2013.pdf. 2014.

[13] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. http://www.rfc-editor.org/rfc/rfc4251.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4251.txt.

[14] Damien Miller. *SSH Agent Protocol*. Internet-Draft draft-miller-ssh-agent-04. IETF Secretariat, Dec. 2019.

[15] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. http://www.rfc-editor.org/rfc/rfc4254.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4254.txt.

[16] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. http://www.rfc-editor.org/rfc/rfc4252.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4252.txt.

[17] F. Cusack and M. Forssen. *Generic Message Exchange Authentication for the Secure Shell Protocol (SSH)*. RFC 4256. RFC Editor, Jan. 2006.

[18] J. Hutzelman et al. *Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol*. RFC 4462. RFC Editor, May 2006.

[19] HTTP Documentation. *HTTP Working Group*. https://httpwg.org/specs/. [Online; accessed 03-September-2023]. 2023.

[20] Tim Berners-Lee. *The HTTP Protocol As Implemented In W3*. https://www.w3.org/Protocols/HTTP/AsImplemented.html. [Online; accessed 03-September-2023]. 1991.

[21] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. http://www.rfc-editor.org/rfc/rfc7519.txt. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7519.txt.

[22] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. http://www.rfc-editor.org/rfc/rfc4648.txt. RFC Editor, Oct. 2006. URL: http://www.rfc-editor.org/rfc/rfc4648.txt.

[23] Christopher John Atherton et al. *Federated Identity Management for Research Collaborations*. Version 2.0. June 2018. DOI: 10.5281/zenodo.1307551. URL: https://doi.org/10.5281/zenodo.1307551.

[24] AARC Community members and AppInt members. *AARC Blueprint Architecture 2019 (AARC-G045)*. Nov. 2019. DOI: 10.5281/zenodo.3672785. URL: https://doi.org/10.5281/zenodo.3672785.

[25] Gabriel Zachmann and contributors. *Introduction - oidc-agent*. https://indigo-dc.gitbook.io/oidc-agent/. [Online; accessed 30-July-2023]. 2023.

[26] Gabriel Zachmann and contributors. *API - oidc-agent*. https://indigo-dc.gitbook.io/oidc-agent/api. [Online; accessed 30-July-2023]. 2023.

[27] Diana Gudu and contributors. *motley cue*. https://motley-cue.readthedocs.io/en/latest/. [Online; accessed 30-July-2023]. 2023.

[28] PRACE-LAB. *PRACE-LAB / pam*. https://git.man.poznan.pl/stash/projects/PRACELAB/repos/pam/. [Online; accessed 30-July-2023]. 2023.

[29] Diana Gudu and contributors. *mccli*. https://mccli.readthedocs.io/en/latest/. [Online; accessed 30-July-2023]. 2023.

[30] Internet2 Middleware Initiative. *COmanage*. https://spaces.at.internet2.edu/display/COmanage/Home. [Online; accessed 16-September-2023]. 2023.

[31] INDIGO. *Introduction - Token Translation Service*. https://indigo-dc.gitbook.io/token-translation-service/. [Online; accessed 16-September-2023]. 2018.

[32] INDIGO. *indigo-dc/tts: WaTTS - the INDIGO Token Translation Service*. https://github.com/indigo-dc/tts. [Online; accessed 16-September-2023]. 2018.

[33] Bas Wegh and Lukas Burgey. *indigo-dc/tts_plugin_ssh: a simple ssh plugin for WaTTS*. https://github.com/indigo-dc/tts_plugin_ssh/tree/master. [Online; accessed 16-September-2023]. 2018.

[34] Erik Berdonces Bonelo. "OpenID Connect Client Registration API for Federated Cloud Platforms". English. Master's thesis. Aalto University. School of Science, 2017, p. 62. URL: http://urn.fi/URN:NBN:fi:aalto-201706135534.

[35] Yuri Demchenko et al. "CYCLONE: A Platform for Data Intensive Scientific Applications in Heterogeneous Multi-cloud/Multi-provider Environment". In: *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. 2016, pp. 154–159. DOI: 10.1109/IC2EW.2016.46.

[36] Jarosław Surkont et al. *PAM module for OAuth 2.0 Device flow*. [Computer Software] https://doi.org/10.11578/dc.20220727.6. Aug. 2020. DOI: 10.11578/dc.20220727.6. URL: https://doi.org/10.11578/dc.20220727.6.

[37] UK Research and Innovation (UKRI). *slaclab/pam_oauth2_device: PAM module OAuth2 Device flow*. https://github.com/slaclab/pam_oauth2_device. [Online; accessed 16-September-2023]. 2022.

[38] National Science Foundation et al. *GSI-Enabled OpenSSH*. http://grid.ncsa.illinois.edu/ssh/. [Online; accessed 28-August-2023]. 2019.

[39] Jason Alt et al. "OAuth SSH with Globus Auth". In: *Practice and Experience in Advanced Research Computing*. PEARC '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 34–40. ISBN: 9781450366892. DOI: 10.1145/3311790.3396658. URL: https://doi.org/10.1145/3311790.3396658.

[40] Extreme Science and Engineering Discovery Environment (XSEDE). *XSEDE/oauth-ssh: SSH with Globus Auth*. https://github.com/XSEDE/oauth-ssh. [Online; accessed 16-September-2023]. 2022.

[41] You Alex Gao, Jim Basney, and Alex Withers. "SciTokens SSH: Token-Based Authentication for Remote Login to Scientific Computing Environments". In: *Practice and Experience in Advanced Research Computing*. PEARC '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 465–468. ISBN: 9781450366892. DOI: 10.1145/3311790.3399613. URL: https://doi.org/10.1145/3311790.3399613.

[42] Inc. Smallstep Labs. *Smallstep SSH — Single Sign-On SSH With Zero Key Management*. https://smallstep.com/sso-ssh/. [Online; accessed 16-September-2023]. 2023.

[43] Gravitational Inc. *Using Teleport with OpenSSH in agentless mode | Teleport Docs*. https://goteleport.com/docs/server-access/guides/openssh/. [Online; accessed 16-September-2023]. 2023.

[44] Inc. Smallstep Labs. *step-cli | Automate Certificates & Common Cryptography Primitives*. https://smallstep.com/cli/index.html. [Online; accessed 16-September-2023]. 2023.

[45] Gravitational Inc. *Configure SSH with Pluggable Authentication Modules | Teleport Docs*. https://goteleport.com/docs/server-access/guides/ssh-pam/. [Online; accessed 16-September-2023]. 2023.

[46] Matt Johnston. *Dropbear SSH*. https://matt.ucc.asn.au/dropbear/dropbear.html. [Online; accessed 05-September-2023]. 2023.

[47] J. Linn. *Generic Security Service Application Program Interface Version 2, Update 1*. RFC 2743. RFC Editor, Jan. 2000.

[48] L. Zhu, K. Jaganathan, and S. Hartman. *The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2*. RFC 4121. http://www.rfc-editor.org/rfc/rfc4121.txt. RFC Editor, July 2005. URL: http://www.rfc-editor.org/rfc/rfc4121.txt.

[49] E. Lear et al. *A Simple Authentication and Security Layer (SASL) and Generic Security Service Application Program Interface (GSS-API) Mechanism for OpenID*. RFC 6616. RFC Editor, May 2012.

[50] OpenSSH contributors. *Portable OpenSSH*. https://github.com/openssh/openssh-portable. [Online; accessed 30-July-2023]. 2023.

[51] Tatu Ylonen et al. *ssh-keygen(1) — OpenSSH authentication key utility*. July 2023. URL: https://man.openbsd.org/ssh-keygen.1.

[52] The OpenSSH contributors. *sshd_config(5) - OpenBSD manual pages*. https://man.openbsd.org/sshd_config. [Online; accessed 29-August-2023]. 2023.

[53] The OpenSSH contributors. *ssh_config(5) - OpenBSD manual pages*. https://man.openbsd.org/ssh_config. [Online; accessed 29-August-2023]. 2023.

[54] The OpenSSH contributors. *sshd(8) - OpenBSD manual pages*. https://man.openbsd.org/sshd. [Online; accessed 29-August-2023]. 2023.

[55] util-linux contributors. *su(1) - Linux manual page*. https://man7.org/linux/man-pages/man1/su.1.html. [Online; accessed 29-August-2023]. 2023.

Appendix

Commands used in example OpenSSH certificate setup

```
# Generate user key pair
$ ssh-keygen -t ed25519 -f user-key -C user-key

# Generate host key pair
$ ssh-keygen -t ed25519 -f host-key -C host-key

# Generate CA key pairs
$ ssh-keygen -t ed25519 -f host-ca -C host-ca
$ ssh-keygen -t ed25519 -f user-ca -C user-ca

# Issue host certificate 'host-key-cert.pub'
$ ssh-keygen -h -I example.com -n example.com -V +1w -s host-ca host-key.pub

# Issue user certificate 'user-key-cert.pub'
$ ssh-keygen -I user@example.com -n user -V +1d -s user-ca user-key.pub
```

List of Acronyms

API Application Programming Interface

CSS Cascading Style Sheets

DNS Domain Name System

EdDSA Edwards-curve Digital Signature Algorithm

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

IPC Inter-Process Communication

JSON JavaScript Object Notation

JWT JSON Web Token

LDAP Lightweight Directory Access Protocol

NSS Name Service Switch

PAM Pluggable Authentication Module

REST Representational State Transfer

RSA Rivest-Shamir-Adleman

SASL Simple Authentication and Security Layer

SOAP Simple Object Access Protocol

SSH Secure Shell Protocol

SSO Single Sign-On

TCP Transmission Control Protocol

TLS Transport Layer Security

TOFU Trust On First Use

TTY Teletypewriter

UDP User Datagram Protocol

URL Uniform Resource Locator

List of Figures

2.1 Example of encoded and decoded JWT. 5
2.2 OpenID Connect authorization code flow. 7

4.1 Example of a SSH connection using password-based authentication. 14
4.2 Example of `ssh` and `mccli` with `keyboard-interactive` authentication. . 14
4.3 Example of an `ed25519` OpenSSH private and public key. 17
4.4 Example of an OpenSSH user certificate. 18
4.5 Example steps and configuration of OpenSSH certificate authentication. . . 19

5.1 Architecture of oinit. 30
5.2 Steps undertaken in preparation and later connection steps. 32
5.3 Sequence diagram of steps undertaken in preparation process. 33
5.4 Sequence diagram of steps undertaken in connection process. 35
5.5 `oinit-ca` Swagger documentation. 36
5.6 Example of an OpenSSH user certificate issued by `oinit-ca`. 37
5.7 Example `oinit-ca` configuration file. 38
5.8 Example `~/.ssh/oinit_hosts` file. 39
5.9 `ssh` configuration file with `oinit`'s `Match exec` section. 39
5.10 Extract from configuration file `/etc/pam.d/su`. 42
5.11 Extract of process tree for an interactive `ssh` login using the `os/exec` package. 43
5.12 Extract of process tree for an interactive `ssh` login using *execve* directly. . 43
5.13 Output of `oinit add`, `oinit list` and a subsequence SSH connection. . . 44

6.1 Extract of process tree showing `su`'s default behavior. 48
6.2 Extract of process tree showing `su` with `--pty` argument. 48