Fuzzing OpenID Connect implementations

Harm Roukema
s1029070

August 19, 2026

First supervisor/assessor:
Dr. ir. Erik Poll

Supervisor:
Amineh Akhavan Saraf

Supervisor:
François Kooman

Second assessor:
Dr. ir. Jan Tretmans

Radboud University

Summary

Single-Sign On protocols, like OpenID Connect (OIDC), allow users to login to multiple applications by only logging in once. How can we test that OIDC implementations are secure?

We test the security of the OIDC implementations with fuzzing, a process of feeding random input into a program in the hope of uncovering failures [42] which may result in vulnerabilities. OpenID Connect is an extension of OAuth 2.0. We discuss both OAuth 2.0 and OpenID Connect, and which types of vulnerabilities occur in their implementations. To demonstrate these vulnerabilities, we develop an OAuth 2.0 CTF (Capture The Flag, an intentionally vulnerable OAuth 2.0 implementation) with five challenges[1]. Then we link these types of vulnerabilities to fuzzing techniques, to see which techniques are most effective. We improve upon learning-based fuzzing, a fuzzing variant that involves learning state machines as defined by Aichernig et al. [1], by fuzzing state-by-state and fuzzing multiple OpenID Connect implementations[2], namely the SimpleSAMLphp OIDC module, the SimpleSAMLphp AuthOAuth2 module, the Shibboleth OIDC module, and the Apache mod_auth_openidc module. We find that the SimpleSAMLphp AuthOAuth2 module does not adhere to OAuth 2.0 best practice [24] and might be vulnerable to a state leak attack as defined by Fett et al. [9] in some configurations.

We also discuss browser-swapping attack as defined by Jonas Primbs [34], attacks that target the OAuth 2.0 and OIDC protocols itself. We argue that browser-swapping attacks are a legitimate threat due to their similarity to the ConsentFix attack on Microsoft Azure [16]. We create a demo of browser-swapping attacks that works on the four fuzzed OIDC implementations. We implement a known protection against browser-swapping attacks in the SimpleSAMLphp OIDC module, namely the `form_post` response mode and a configuration option to enforce this[3].

[1]The CTF is open-source on GitHub: `https://github.com/Harm-r/oauth-ctf`

[2]The fuzzer is open-source on GitHub: `https://github.com/Harm-r/oidc-learning-based-fuzzing`

[3]The pull request can be found on GitHub: `https://github.com/simplesamlphp/simplesamlphp-module-oidc/pull/337`

Contents

Glossary

access token A credential used to access protected resources. It represents an authorization issued to the client/RP, and includes specific scopes and durations of access, enforced by the resource and authorization servers/OP. It may for example be a random string. Page 17, 21

ad hoc mutation A mutation specifically designed for a specific target or vulnerability. Ad hoc mutations are generally more effective at uncovering a specific vulnerability, but are less likely to find novel vulnerabilities and are not generally applicable to other targets or protocols. Page 49, 61

attacker model The assumed capabilities of the attacker. The attacker model of our fuzzer is given in Section 7.1. Page 55

authorization code Specific to the authorization code flow, exchanged for an access token. With an authorization code, the client can be authenticated, and the access token does not need to be transmitted through the user-agent. An authorization code must be short lived and single-use. Page 18, 20

authorization server The server that authorizes the user in OAuth 2.0 and issues an access token, e.g. Google when signing in with Google on X. Similar to the OP in OIDC. For more information, see Section 2.2.2. Page 16, 76

black-box With little information of the underlying system, e.g. without access to the source code. Page 48

browser-swapping attack An attack on the OAuth 2.0 (and therefore, also OpenID Connect) protocol in which a victim's session is compromised via a stolen or leaked authorization code. The attacker generally gains access to this authorization code via social engineering (e.g. phishing), but might also through other ways, by e.g. viewing system logs. It is very similar to the ConsentFix attack on Microsoft Azure [16]. For more information, see Chapter 9. Page 1, 12, 74

Capture The Flag (CTF) An intentionally vulnerable application meant to practice security testing on. Players of a CTF can gain points by submitting "flags" (e.g. `CTF{this_is_a_flag}`), random strings that are obtained by breaching the security of the application, e.g. stored on an admin panel. Page 12, 13, 44, 85

claim A piece of information that is asserted about an entity (e.g. end-user), as described in Section 2.4.3. Generally a key-value pair in a JWT payload. Page 23, 31, 42

client Application that requests a protected resource on behalf of the resource owner (e.g. user) in OAuth 2.0, e.g. X when signing in with Google on X. Not to be confused with the user. Similar to the RP in OIDC. For more information, see Section 2.2.2. Page 16, 79

client identifier (`client_id`) Identifier (e.g. random string) for a client or RP issued at client registration by the authorization server or OP, not secret. Page 18, 20

Cross-Site Request Forgery (CSRF) An attack in which a malicious web application tricks the victim's browser into making an authenticated request to an honest web application [2]. Page 20, 38, 44

Dynamic Application Security Testing (DAST) A type of testing that tests an application at runtime [23]. Fuzzing is a type of DAST. Page 47

end-user The user that tries to login/authenticate in OIDC. Page 10, 55, 56

flow A process of authenticating a user in OIDC. Either authorization code flow, implicit flow or hybrid flow. Similar to "Grant" in OAuth 2.0. See Section 2.4.2 for more information. Page 15, 17, 56, 72

fuzzing Fuzzing is the process of feeding random input into a program in the hope of uncovering failures [42], which may result in vulnerabilities. There are multiple types of fuzzing. For more information, see Section 6.1. Page 1, 11, 65

grammar Describes the input to a stateful system, both the message format and sequence of messages. Sometimes the term can also be used to mean only the description of the message format. In this thesis, these cases are clarified each time. Page 48

grant An (authorization) grant is a credential representing the resource owner's (e.g. end-user's) authorization and is used by the client to obtain an access token. Examples are the authorization code grant from the authorization code flow and the implicit grant from the implicit flow. Page 17

ID token OIDC's primary extension to OAuth 2.0 that enable authentication of the end-user. An ID token is a JWT that contains claims, asserted pieced of information, about an end-user. Page 24, 25, 42

JSON Web Token (JWT) Allows parties to exchange JSON data ("claims") in a compact, URL-safe way [18]. JWTs can be authenticated with a signature using JWS (JSON Web Signature) [17] and encrypted with JWE (JSON Web Encryption) [19]. The ID token in OpenID Connect from Section 2.4.1 is a JWT. For an example on how to create a JWT, see Section 2.3. Page 23, 24, 42

learning-based fuzzing A fuzzing method by Aichernig et al. that combines automata learning and fuzz testing to test black-box systems [1]. For more information, see Chapter 3. We give our reasons for using learning-based fuzzing in Section 6.3 and describe our exact setup in Chapter 7. Page 34, 53

nonce Optional string that associates a client session with an ID token, mitigates replay attacks. Passed unmodified to the ID token. Page 27, 43

OAuth 2.0 An authorization framework that enables applications to request limited access to resources on another application, without exposing the user's credentials to the requesting application [32] [13]. OpenID Connect is an extension of OAuth 2.0. For more information, see Section 2.2. Page 1, 5, 10, 12–14, 74

OIDC implementation Implemented OIDC in code, as opposed to the protocol specification. With OIDC implementations, we refer to both the implementations at the RP and the OP. The implementations that we fuzzed are the SimpleSAMLphp OIDC module (OP), the SimpleSAMLphp AuthOAuth2 module (RP), the Shibboleth OIDC module (OP), and the Apache mod_auth_openidc module (RP). Page 1, 10–12, 38, 59, 70, 84–86

OpenID Connect (OIDC) An SSO protocol, "a simple identity layer on top of the OAuth 2.0 protocol"[37], adding authentication of the end-user. For more information, see Section 2.4. Page 1, 5, 10–13, 24, 41, 74

OpenID Provider (OP) Authenticates the end-user in OIDC, e.g. Google when signing in with Google on X. Similar to the authorization server in OAuth 2.0. Page 10, 25, 55, 56, 74, 79

redirection URI (`redirect_uri`) optional parameter containing the redirection endpoint in the authorization request, registered during client registration. Page 20, 38

Relying Party (RP) Provides the application that the end-user wants to use in OIDC, e.g. X when signing in with Google on X. Similar to the client in OAuth 2.0. Page 10, 25, 40, 55, 56, 74

request object (`request`) A single, self-contained JWT parameter to be optionally signed and/or encrypted containing other OIDC authorization request parameters, as described in Section 2.4.4. Page 32

resource owner Entity capable of granting access to a protected resource in OAuth 2.0, e.g. end-user. For more information, see Section 2.2.2. Page 16

resource server The server hosting the protected resources, responding to protected resource requests using access tokens. Can be the same server as the authorization server. For more information, see Section 2.2.2. Page 16

response mode (`response_mode`) optional parameter specifying how parameters are to be returned by the OP, e.g. `query` for returning parameters in the query string. More information in Section 2.4.2.1. Page 27

response type (`response_type`) Required parameter at authorization endpoint to specify desired grant type, either either "code" for the authorization code grant or "token" for the implicit grant. Page 19, 20

scope Used at the authorization and token endpoints to specify the scope of the access request. Has to include "openid" for OIDC. Page 19, 20, 40

Server-Side Request Forgery (SSRF) An attack in which an attacker can create HTTP requests from a vulnerable web application server to the internet or its local network [25]. Page 41, 42

Single-Sign On (SSO) An authentication method that allows users to login to multiple applications by only logging in once. Page 1, 10

state Recommended parameter used by the client to maintain state between the request and callback, reused in the response. Should be used to prevent cross-site request forgery. Page 20, 40

stateful system A system that takes a sequence of messages as input, where inputs might change some internal state of the system [5]. Page 48

Static Application Security Testing (SAST) A testing technique in which source code is scanned for vulnerabilities, without actually executing the program. Page 40, 47

system under test (SUT) In this thesis, a combination of an OP implementation and RP implementation that is fuzzed. Page 3, 48, 56

user-agent A program that retrieves and interacts with web content by e.g. following redirects. A browser, like Firefox, is an example of a User-Agent. Page 16

Chapter 1

Introduction

Have you ever wondered what happens when you click the "Login with Google" button on a website? How can a website allow you to login with credentials from another provider like Google, Apple or Microsoft? Under the hood of these buttons, Single-Sign On (SSO) protocols like OpenID Connect (OIDC) allow you to use one set of credentials to access multiple applications, often with just a single click. Approximately 4.5% of the top 1 million domains support SSO [15]. Radboud University uses SSO themselves, for example when allowing students to login to BrightSpace with SURFconext[1].

Meanwhile, attackers abuse SSO protocols to access protected resources, for example by stealing OAuth tokens in the Salesloft Drift application in 2025 [22], or in the ConsentFix attack on Microsoft Azure [16]. In addition, vulnerabilities in SSO implementations can have serious impact. For example, they can result in an authentication bypass in Authlib[2] or a signature bypass in SimpleSAMLphp[3]. How can we test that the implementations of SSO protocols are secure?

In our research, we focus on OpenID Connect implementations, as OpenID Connect is one of the newest and most widely deployed single sign-on protocols [10]. It is an extension of OAuth 2.0. The OIDC protocol has been proven to be secure by Fett et. al [10], which is why we focus on OIDC implementations instead. For now, we will distinguish between three parties in the OIDC flow: the *end-user*, the user that tries to login, the *Relying Party (RP)*, that provides the application that the end-user wants to use, and the *OpenID Provider (OP)*, which authenticates the end-user. For example, if a user wants to sign in to X (formerly Twitter) using Google, X is the RP (Relying Party) and Google the OP (OpenID provider). An overview is shown in Figure 1.1. Note that other SSO protocols use different terminology for these parties: the RP is also called the client in OAuth 2.0 or service provider

[1] `https://www.surf.nl/en/services/identity-access-management/surfconext`

[2] `https://github.com/authlib/authlib/security/advisories/GHSA-7wc2-qxgw-g8gg`

[3] `https://simplesamlphp.org/security/202512-01`

in SAML, while the OP is also called the authorization server in OAuth 2.0 and the identity provider in SAML.

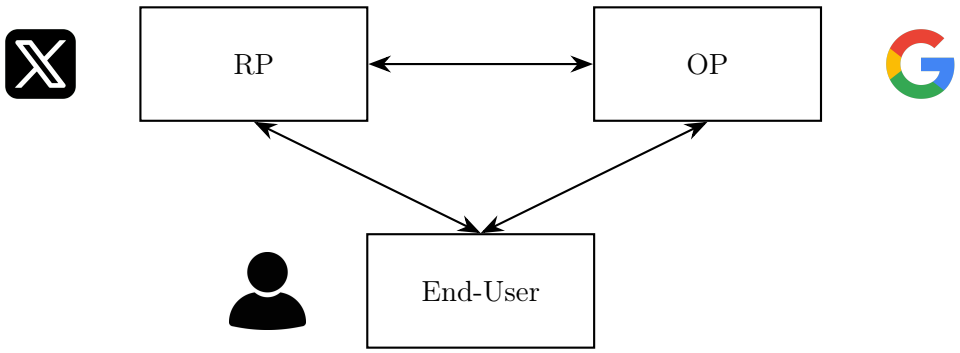


Figure 1.1: Interaction between the end-user, RP, and OP

Why should we use OpenID Connect? An SSO protocol like OIDC is a potential solution to password management [30]. Without SSO protocols like OIDC, users need to keep track of many different credentials for many relying parties. OIDC allows users to easily login at multiple relying parties with a single set of credentials.

Because of its role in authentication, security of OIDC is essential. Vulnerabilities in OIDC implementations could allow for attacks like an account takeover or an authentication bypass. Meanwhile, although proven secure by Fett et al. [10], we find the OIDC protocol to be, in our opinion, quite complex, as it is a large protocol with lots of features, as described in Section 2.4. This makes it easy for developers to make mistakes in OIDC implementations. These factors combined with the large adaption of OIDC make it an interesting topic for security research.

We decide to test the security of OIDC implementations with fuzzing, a process of feeding random input into a program in the hope of uncovering failures [42]. OIDC implementations can already be tested with the OpenID Conformance Suite [29]. However, these tests are designed to test conformance to the OIDC standard, not to find novel attacks or differences in implementations, for which fuzzing is more suitable. There has been little research on using fuzzing to test OIDC implementations specifically, although dynamic testing of OAuth 2.0 implementations has been studied by Yang et al. [41].

This brings us to our research question: *"How do we fuzz OIDC implementations for vulnerabilities?"*. We tackle this research question by addressing two sub-questions:

- Which types of vulnerabilities are there?
- Which fuzzing techniques could we use to find which types of vulnerabilities?

By "OIDC implementations" we mean both the implementations at the RP and the OP. For more information on our attacker model, see Section 7.1.

This thesis was written during an internship at the Trust & Identity Incubator at GÉANT. The T&I Incubator "aims to develop, foster and mature new ideas in the trust and identity space in research and education"[4] and mainly consists of employees of European NRENs (National Research and Education Networks). An NREN is an internet service provider for research and education in a country. NRENs often offer additional services for e.g. identity and security. SURF[5] is the Dutch NREN. GÉANT is a European association that creates digital infrastructure for research, education and innovation[6]. It consists mostly of European NRENs.

In this thesis, we start by providing background information on OAuth 2.0, OpenID Connect and JWT, a type of token that is used in OIDC, in Chapter 2. We cover related work in Chapter 3, and continue by discussing types of vulnerabilities in OIDC implementations in Chapter 4. We created a custom vulnerable OAuth 2.0 implementation (Capture The Flag) for testing, which we discuss in Chapter 5. We look into possible fuzzing techniques and which types of vulnerabilities they might find in Chapter 6. Our fuzzing setup is explained in Chapter 7, of which we show the results in Chapter 8. During our research, we also learned about browser-swapping attacks, which we discuss in Chapter 9. Finally, we list possible future work in Chapter 10 and draw our conclusions in Chapter 11.

[4]For more information on the T&I Incubator, see `https://trustidentity.geant.org/ti-incubator/`

[5]`https://www.surf.nl/`

[6]For more information on GÉANT, see `https://geant.org/`

Chapter 2

Background on OIDC

This chapter provides background information on OpenID Connect (OIDC). We first give our recommendation on how to learn OIDC from scratch in Section 2.1. OpenID Connect is built on top of OAuth 2.0: it includes the entire OAuth 2.0 protocol, but also adds more functionality. To understand OpenID Connect, it is therefore useful to first look into OAuth 2.0, which we do in Section 2.2. Because a specific type of token in OpenID Connect, the ID token, is a JWT, we also explain what a JWT is and how it works in Section 2.3. Then we look into OpenID Connect in Section 2.4.

Writing this chapter helped us a lot in learning about OAuth 2.0 and OIDC at the start of our research. It is mainly a *summary* of *only the relevant parts* of the specifications for this thesis, and is not a complete overview of the protocols.

Note that OAuth 2.0 is not an SSO protocol, because it does not provide authentication. It allows for sharing resources with third-party applications, but the identity of the end-user is never proven. OpenID Connect *does* authenticate the end-user and is therefore an SSO protocol.

2.1 How to learn OIDC?

There are a lot of different complementary OAuth and OIDC specifications, and it is difficult to know which are relevant and how you should approach learning them. In this section, we give some recommendations on how to approach learning OAuth and OIDC.

Our main recommendation is to combine reading with hands-on practice: look at the HTTP traffic of an existing OIDC implementation, implement (part of) the specification yourself, play our OAuth Capture The Flag (CTF) from Chapter 5 or create your own CTF challenges. This is a lot more fun than just reading and helps you to understand the somewhat abstract documentation in practice. Solving CTF challenges has also been shown to be an effective way to learn new technologies [26].

To learn OIDC, you need to understand OAuth 2.0. The OAuth 2.0 specification [13] can be a bit technical, so we recommend to read Section 2.2 first. Reading should be supplemented with hands-on practice, for example by going over the OAuth 2.0 Playground[1] while looking at the HTTP requests, or by implementing the specification yourself. If you are interested in the security aspects of OAuth 2.0, make sure to read Section 4.2.1, RFC 9700 "Best Current Practice for OAuth 2.0 Security" [24], and to play our OAuth CTF from Chapter 5. There are other numerous OAuth CTF challenges online you can practice with, for example the labs by PortSwigger [32]. You can even create your own CTF challenges like we did.

If you do not care about OIDC, and only want to implement OAuth, you should take a look at the OAuth 2.1 specification draft [14], which includes multiple security improvements.

After you understand OAuth 2.0 and want to learn OIDC, you should read Section 2.4 or take a look at the OpenID Connect specification [37] for more details. Make sure to also include hands-on practice with e.g. the OpenID Connect Playground[2]. To learn about the security of OIDC, you can read Section 4.2.2 or practice a lab by PortSwigger [33].

If you implement OIDC, make sure to test your implementation with the OpenID Conformance Suite [29].

2.2 OAuth 2.0

OAuth 2.0 is an authorization framework used with HTTP that enables applications to request limited access to resources on another application, without exposing the user's credentials to the requesting application [13] [32]. When we write OAuth, we refer to OAuth 2.0. In this section, we only discuss the relevant parts of the specification for this thesis. For more information, please consult the specification [13].

There is also a draft of OAuth 2.1 [14], with some differences like the removal of the implicit grant type. We only discuss OAuth 2.0, because OpenID Connect is based on it.

A formal security analysis of the OAuth 2.0 protocol was conducted by Fett at al. [9]. They discovered four attacks that broke the security of OAuth, and recommended fixes to these attacks under which they proved OAuth to be secure. These fixes were added to OAuth in RFC 9700 "Best Current Practice for OAuth 2.0 Security" [24], of which Fett is a co-author. Although proven secure under these fixes, OAuth 2.0 is still vulnerable to browser-swapping attacks as described in Chapter 9.

A user might want to grant a third-party application access to their protected resources. Traditionally, this requires the user to share its credentials

[1] `https://www.oauth.com/playground/`

[2] `https://www.openidconnect.net/`

with the third-party, e.g. with X (the RP) in Figure 1.1. This creates several problems: [13]

- Third-party applications are required to store the credentials for future use, typically in plain text.
- This requires servers to support password authentication, which can be less secure.
- Third-party applications gain overly broad access to the user's protected resources.
- The user cannot revoke access to an individual third party without revoking access to all third parties.
- Compromise of the third-party application results in compromise of the user's password and the protected resources.

OAuth addresses these issues by supplying the third-party application with an access token with the consent of the user. This access token denotes a specific scope, lifetime, and other access attributes, and is revocable.

2.2.1 Example

We start our explanation of OAuth 2.0 with an example of a flow, namely the authorization code flow in Figure 2.1.

In this example, a user wants to login on X using their Google credentials ("Sign-in with Google"). We do not expect you to understand all steps and terminology in Figure 2.1 right away, we will explain each part step-by-step and refer back to it:

- You can see multiple parties interact. These roles are explained in Section 2.2.2.
- This figure shows the authorization code flow, which uses the authorization code grant. An overview of the available grants is given in Section 2.2.3.
- You can see an authorization code in step D and an access token in step E. These are explained in Section 2.2.4.
- You can see three endpoints: `/authorize` and `/token` at the authorization server and `/callback` at the client. The different endpoints are explained in Section 2.2.6.
- We explain the other parts of the Figure, the individual steps, in Section 2.2.7.1. This also includes examples of the actual HTTP requests and responses.

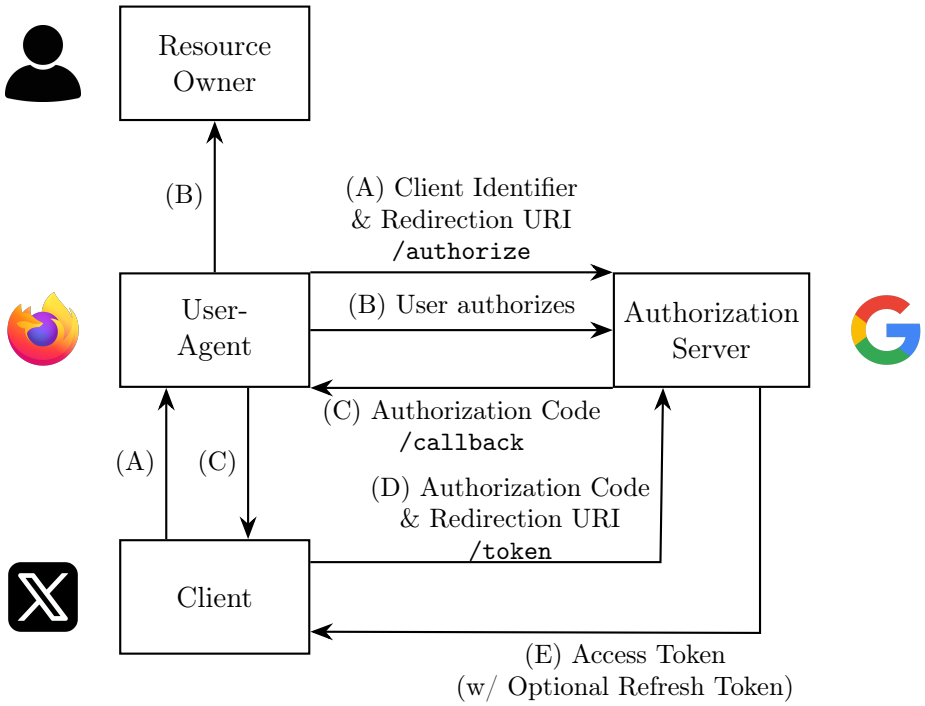


Figure 2.1: Authorization Code Flow

2.2.2 Roles

OAuth 2.0 defines four roles [13]:

- **Resource owner**: entity capable of granting access to a protected resource, also known as **end-user** in case this entity is a person.
- **Resource server**: the server hosting the protected resources, responding to protected resource requests using access tokens.
- **Client**: an application making protected resource requests on behalf of the resource owner and with its authorization. In OIDC, the client is called the RP.
- **Authorization server**: the server issuing access tokens to the client after successfully authorizing the resource owner. In OIDC, this server is called the OP.

The resource and authorization servers can be the same server.

Note that Figure 2.1 also includes the user-agent. The **user-agent** is a program that retrieves and interacts with web content, in this case by following redirects. A browser, like Firefox, is an example of a User-Agent.

When a user wants to grant a third-party application access to their protected resources, e.g. a Google user singing in with Google on X:

- The **resource owner** is the **user** signing in with Google on X.
- The **resource server** is the Google server, providing account information to X.
- The **client** is X, gaining access to the account information of the user at Google.
- The **authorization server** is Google as well, authorizing the user.

2.2.3 Grant types and Flows

Figure 2.1 shows the authorization code flow using the authorization code grant. An **(authorization) grant** is a credential representing the resource owner's authorization, used by the client to obtain an access token, and is expressed using one of four grant types:

- **Authorization code**: the authorization server is used as an intermediary. Allows for authentication of the client as well as direct transmission of the access token to the client.
- **Implicit**: simplified, optimized for clients in a browser. Client is issued an access token directly, no authorization code is used. Client is not authenticated (only potentially via redirection URI). The access token will be exposed to the user-agent. Less round trips, less security.
- **Resource Owner Password Credentials**: the resource owner's username and password are directly used to obtain an access token.
- **Client Credentials**: client credentials can be used as an authorization grant when the protected resources are under the control of the client.

We discuss the authorization code grant and implicit grant in more detail in Section 2.2.7.

A **flow** is the process of obtaining authorization in OAuth 2.0 (or authentication in OIDC). It is often combined with a grant. For example, we say "Authorization code flow" when we refer to the entire authorization process using the authorization code grant.

2.2.4 Tokens

In step E of Figure 2.1, we see that an access token with optional refresh token is sent to the client. **Access tokens** are credentials used to access protected resources. They represent an authorization issued to the client, and include specific scopes and durations of access, enforced by the resource and authorization servers. It may be an identifier (e.g. random string) used to

retrieve authorization information or include the information in a verifiable manner (with a signature).

Refresh tokens are credentials used to obtain access tokens in case the current access token becomes invalid or expires, or to obtain additional access tokens, issued by the authorization server. Using refresh tokens is optional. They are only sent to the authorization server, never to the resource server.

In the authorization code flow in Figure 2.1, an **authorization code** is exchanged by the client (X) for an access token. This is specific to the authorization code flow and does not apply to other flows like the implicit flow. With an authorization code, the client can be authenticated, and the access token does not need to be transmitted through the user-agent. An authorization code must be short lived and single-use.

2.2.5 Client registration

Before initiating the protocol, the client registers with the authorization server. The client shall specify the client type, its client redirection URIs and any other information required by the authorization server. There are two client types: **confidential clients**, when the client can maintain confidentiality of their credentials (e.g. a web application), and **public clients**, when the client is incapable of maintaining confidentiality of their credentials (e.g. native application). The authorization server issues the registered client a **client identifier** (`client_id`), which is not secret. This process is not relevant to our research, as all of our testing will happen after registration has already been done.

2.2.5.1 Client authentication

If the client type is confidential, the client and authorization server establish a client authentication method. This can vary per implementation, and can for example be a password (`client_secret`) or public/private key pair. For a password, clients may use the HTTP Basic authentication scheme. Alternatively, the authorization server may support including the client credentials (`client_id` and `client_secret`) in the request-body.

2.2.6 Protocol endpoints

The authorization process uses three endpoints in total:

- Two *authorization server* endpoints: the authorization endpoint and the token endpoint
- One *client* endpoint: the redirection endpoint.

2.2.6.1 Authorization endpoint (e.g. `authserver.com/authorize`)

The authorization endpoint is used to interact with the resource owner and obtain an authorization grant. The authorization server must verify the identity of the resource owner. The client informs the authorization server of the desired grant type using the required response type (`response_type`) parameter, either "code" for the authorization code grant or "token" for the implicit grant.

In step A from Figure 2.1, a request is sent to the authorization endpoint.

2.2.6.2 Redirection endpoint (e.g. `client.com/callback`)

After verifying the identity of the resource owner, the authorization server directs the resource owner back to the redirection endpoint of the client, either established during client registration or with the authorization request (`redirect_uri`). The redirection endpoint should be registered prior to utilizing the authorization endpoint.

In step C from Figure 2.1, a request is sent to the redirection endpoint.

2.2.6.3 Token endpoint (e.g. `authserver.com/token`)

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token, except for the implicit grant type, since then an access token is issued directly. Clients with client credentials (e.g. confidential clients) must authenticate with the authorization server when making requests to the token endpoint, e.g. with the `client_id` and `client_secret` variables. This binds the refresh tokens and authorization codes to the client they were issued to, and allows disabling of compromised clients

In step D from Figure 2.1, a request is sent to the token endpoint.

2.2.6.4 Access Token Scope

The authorization and token endpoints allow the client to specify the scope of the access request using the `scope` request parameter. In turn, the authorization server uses the `scope` response parameter to inform the client of the scope of the access token issued.

2.2.7 Authorization Flows

In this section, we discuss two authorization flows in detail, including their HTTP requests and parameters. We only discuss the authorization code

grant and implicit grant, as these are the only relevant grants for the rest of the thesis.

OAuth supports four authorization flows. These are expressed in the form of an authorization grant, using one of the grant types from Section 2.2.3. OAuth can also be extended to support additional grant types.

2.2.7.1 Authorization Code Grant

The authorization code grant type is the grant used in our example in Figure 2.1. It is used to obtain both access tokens and refresh tokens and is optimized for confidential clients, e.g. a web application. This is a redirection-based flow.

The authorization request to the authorization server (A) contains the following query parameters:

- `response_type`: response type, set to "code" to indicate authorization code grant.
- `client_id`: the client identifier as described in Section 2.2.5.
- `redirect_uri`: optional redirection URI containing the redirection endpoint, has to conform to registered value during client registration in Section 2.2.5.
- `scope`: optional scope of the access request, see Section 2.2.6.4.
- `state`: recommended parameter used by the client to maintain state between the request and callback, reused in the response. Should be used to prevent Cross-Site Request Forgery.

Example:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcallback HTTP/1.1
```

The authorization response after user authorization to the client (C) redirects the user-agent to the validated redirection URI and contains the following query parameters:

- `code`: the authorization code generated by the authorization server, used to request an access token. The authorization code must expire shortly after it is issued (recommended maximum of 10 minutes), must not be reused, and is bound to the client identifier and redirection URI.
- `state`: exact state value received from the client in the authorization request.

Example:

```
HTTP/1.1 302 Found
```

```
Location: https://client.example.com/callback?
code=SplxlOBeZQQYbYS6WxSbIA&state=xyz
```

In case of an error, e.g. an invalid redirection URI, an error response is returned instead of the authorization response.

We do not discuss the access token request to the authorization server (D) and its response (E) in detail, as we do not use this in the rest of our thesis.

2.2.7.2 Implicit Grant

An example of the use of the implicit grant is shown in Figure 2.2. The implicit grant type does not support refresh tokens and is optimized for public clients, e.g. clients implemented in a browser using JavaScript. The client does not need an authorization code and gets the access token as result of the authorization request. The implicit grant type does not support client authentication. Because the access token is encoded into the redirection URI, it may be exposed.

The authorization request (A) sent to the authorization server is similar to the authorization request of the authorization code grant, and includes the query parameters `response_type` (this time set to "token"), `client_id`, `redirect_uri`, `scope` and `state`.

After user authorization, the authorization server responds with the access token response (C), a redirect to the redirection URI. The access token response contains "application/x-www-form-urlencoded" parameters in the fragment component of the redirection URI:

- `access_token`: the access token issued by the authorization server.
- `token_type`: the type of token issued, e.g. "bearer".
- `expires_in`: recommended lifetime in seconds of the access token, e.g. "3600" for an hour.
- `scope`: optional if identical to the scope requested by the client, otherwise required.
- `state`: the exact state value received from the client.

Example:

```
HTTP/1.1 302 Found
Location: http://example.com/callback#access_token=2YotnFZFEjr1zCsicMWpAA
          state=xyz&token_type=example&expires_in=3600
```

A client-side script (like JavaScript) has to be used to extract the access token (F) from the fragment component and to send it to the client (G).

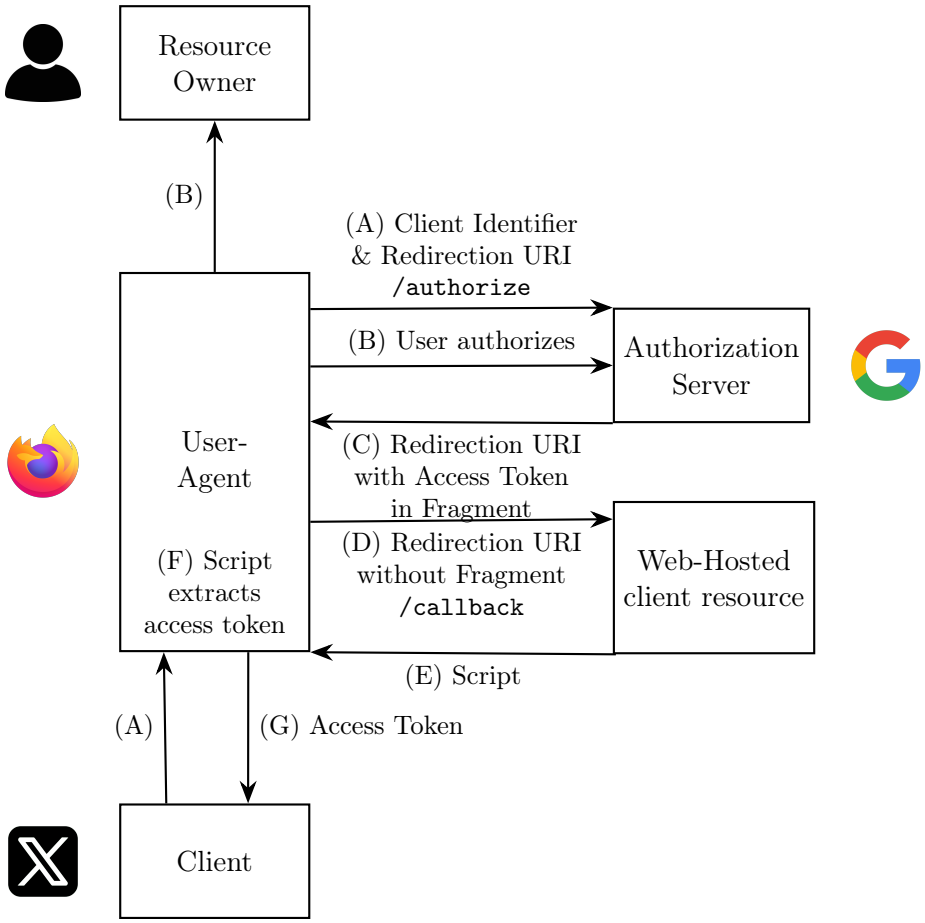


Figure 2.2: Implicit Flow

2.3 JWT

JSON Web Tokens (JWTs) allow parties to exchange JSON data ("claims") in a compact, URL-safe way [18]. JWTs can be authenticated with a signature using JWS (JSON Web Signature) [17] and encrypted with JWE (JSON Web Encryption) [19]. The ID token in OpenID Connect from Section 2.4.1 is a JWT.

Let's create a JWT together, an ID token. A JWT consists of three components: the header, the payload, and the signature. We start with a JSON header:

```
{
    "alg": "RS256"
}
```

This header specifies the used signature algorithm (RS256, an RSA variant). Next, we define the payload, which contains the data ("claims") we want to sign:

```
{
    "iss": "https://server.example.com",
    "sub": "248289761001",
    "aud": "s6BhdRkqt3",
    "nonce": "n-0S6_WzA2Mj",
    "exp": 1311281970,
    "iat": 1311280970,
    "name": "Jane Doe",
    "given_name": "Jane",
    "family_name": "Doe",
    "gender": "female",
    "birthdate": "0000-10-31",
    "email": "janedoe@example.com",
    "picture": "http://example.com/janedoe/me.jpg"
}
```

We can see all types of information in this payload, like the name and birth date of an end-user. For more information on these different claims, see Section 2.4.1.

Now we encode both the header and payload with base64url encoding, strip the padding, and put a "." as separator in between:

```
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL3NlcnZlci5leGFtcGxlLm
NvbSIsInN1YiI6IjI0ODI4OTc2MTAwMSIsImF1ZCI6InM2QmhkUmtxdDMiLCJub
25jZSI6Im4tMFM2X1d6QTJNaiIsImV4cCI6MTMxMTI4MTk3MCwiaWF0IjoxMzEx
MjgwOTcwLCJuYW1lIjoiSmFuZSBEb2UiLCJnaXZlbl9uYW1lIjoiSmFuZSIsImZ
hbWlseV9uYW1lIjoiRG9lIiwiZ2VuZGVyIjoiZmVtYWxlIiwiYmlydGhkYXRlIj
```

oiMDAwMC0xMC0zMSIsImVtYWlsIjoiamFuZWRvZUBleGFtcGxlLmNvbSIsInBpY
3R1cmUiOiJodHRwOi8vZXhhbXBsZS5jb20vamFuZWRvZS9tZS5qcGcifQ

We now take our private RSA key and sign this combination of header and payload with RS256. We take this signature, base64url encode it, strip the padding, and append it after a "." as separator to our JWT:

eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL3NlcnZlci5leGFtcGxlLm
NvbSIsInN1YiI6IjI0ODI4OTc2MTAwMSIsImF1ZCI6InM2QmhkUmtxdDMiLCJub
25jZSI6Im4tMFM2X1d6QTJNaiIsImV4cCI6MTMxMTI4MTk3MCwiaWF0IjoxMzEx
MjgwOTcwLCJuYW1lIjoiSmFuZSBEb2UiLCJnaXZlbl9uYW1lIjoiSmFuZSIsImZ
hbWlseV9uYW1lIjoiRG9lIiwiZ2VuZGVyIjoiZmVtYWxlIiwiYmlydGhkYXRlIj
oiMDAwMC0xMC0zMSIsImVtYWlsIjoiamFuZWRvZUBleGFtcGxlLmNvbSIsInBpY
3R1cmUiOiJodHRwOi8vZXhhbXBsZS5jb20vamFuZWRvZS9tZS5qcGcifQ.fc9bt
pCu7SFrEyF6LbpXjI_6UTqD0bvvg6Mgy6hpk4APLpj7gL8ubf4NK_rfkADRveFD
6hNp1UBhGJwIyTTrvU9PabSAG5Cp9YxyYI9Y-7p8kaLTC5mM40cv6tXHlidEK2_
OuO-O6cCh5pIKcKjvO2EpFztyYKEMMwWd1PhY_YQcQ3Sl0LbEEdgXXWXRlgeasD
iSSBSH-yKF3TxbZktvMgc9BOatzNXkY9BTwQf3W4760SmVlPou2TvaiAeckszUF
X_JZPKAhFrGEOfG-v-bEmGgIOWQMzGn85vKujxGrG6eVgIEqdmCS1_fEATZW5bY
Sh9jLe744CGO5aVEuUqYKg

We now have a valid JWT we can use! Anyone with access to our public RSA key can now verify the signature.

We discuss multiple vulnerabilities that can occur in JWTs in Section 4.2.2.3.

2.4 OpenID Connect

OpenID Connect (OIDC) is "a simple identity layer on top of the OAuth 2.0 protocol" [37]. Although it is meant to be simple, we don't find OIDC to be a simple protocol in our opinion, because it is a large protocol: it has a lot of (sometimes optional) features, and the OIDC Protocol Suite spans across multiple specifications and has a lot of underpinnings, as shown in Figure 2.3. We only discuss the relevant parts of the specification for this thesis. For more information on OIDC, please consult the specification [37].

OIDC allows for authentication of the end-user, and provides methods to obtain basic profile information about the end-user. Use of OpenID Connect is specified by clients with the `openid` scope in the authorization request. Information about the authentication of the end-user is returned in a JSON Web Token (JWT) called an ID token.

There exist multiple OpenID Connect specifications in the OpenID Connect Protocol Suite, as shown in Figure 2.3. We mainly focus on OpenID Connect Core in this thesis, but we also discuss the form post response mode in Section 9.3.1.

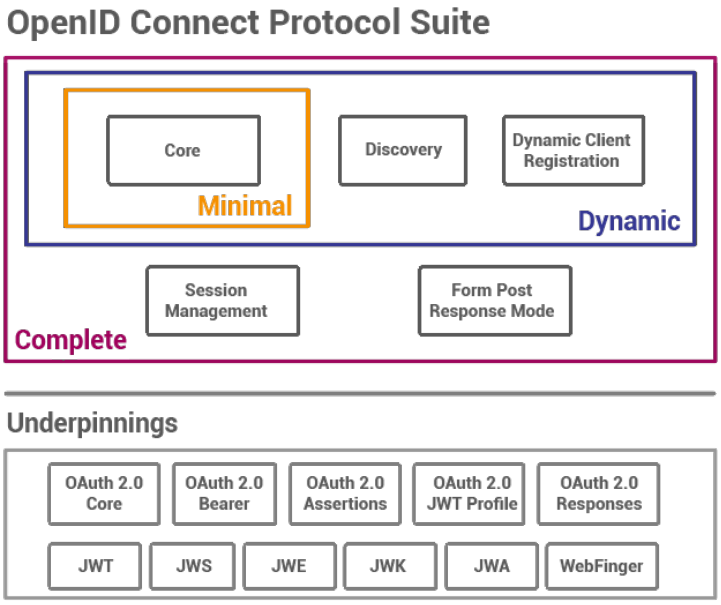


Figure 2.3: All parts of the OpenID Connect Protocol Suite. We focus on OpenID Connect Core (Minimal). Taken from `https://openid.net/developers/how-connect-works/`.

OAuth 2.0 authorization servers that implement OIDC are called OpenID Providers (OPs). OAuth 2.0 clients using OIDC are called Relying Parties (RPs).

OIDC has been proven to be secure by Fett et al. [10], but is still vulnerable to browser-swapping attacks as described in Chapter 9.

2.4.1 ID token

ID tokens are the primary extension to OAuth 2.0 that enable authentication of the end-user. An ID token is a JWT that contains claims, asserted pieced of information, about an end-user. For more information about claims, see Section 2.4.3.
We focus on the following claims that are used within the ID token:

- `iss`: Required issuer identifier, i.e. identifier of the OP.
- `sub`: Required subject identifier, i.e. identifier of the end-user, locally unique and never reassigned, intended to be used by the RP.
- `aud`: Required audience(s) that the ID token is intended for. It must contain the OAuth 2.0 `client_id` of the RP, but may also contain identifiers for other audiences.

Property	Authorization Code Flow	Implicit Flow	Hybrid Flow
All tokens returned from Authorization Endpoint	no	yes	no
All tokens returned from Token Endpoint	yes	no	no
Tokens not revealed to User Agent	yes	no	no
Client can be authenticated	yes	no	yes
Refresh Token possible	yes	no	yes
Communication in one round trip	no	yes	no
Most communication server-to-server	yes	no	varies

Figure 2.4: OpenID Connect Authentication Flows. Taken from the specification [37].

- `exp`: Required expiration time on or after which the ID token must not be accepted by the RP. The expiration time is represented in Unix time as a number.
- `iat`: Required Unix time the JWT was issued at.
- `nonce`: Case-sensitive string used to associate a client session with an ID token, and to mitigate replay attacks. Copied unmodified from the authentication request.

For information on other claims, namely `auth_time`, `acr`, `amr`, `azp`, `at_hash`, `c_hash`, or others, please consult the specification [37].

The ID token is signed using JWS [17] and optionally encrypted (after signing) using JWE [19].

An example of a set of Claims of an ID token is:

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

2.4.2 Authentication

OpenID Connect performs authentication of the end-user. The authentication result is returned in an ID token. Authentication can follow one of three paths: the authorization code flow, the implicit flow, or the hybrid flow. The table in Figure 2.4 summarizes the characteristics of the flows.

"response_type" value	Flow
code	Authorization Code Flow
id_token	Implicit Flow
id_token token	Implicit Flow
code id_token	Hybrid Flow
code token	Hybrid Flow
code id_token token	Hybrid Flow

Figure 2.5: OpenID Connect “response_type" Values. Taken from the specification [37].

The flow used is determined by the `response_type` value in the authorization request, see Figure 2.5

2.4.2.1 Authorization Code Flow

The authorization code flow in OpenID Connect is very similar to the authorization code flow in OAuth 2.0, see Section 2.2.7.1.

The authentication request is an OAuth 2.0 authorization request that requests that the end-user be authenticated. OPs must support the use of both HTTP GET and POST methods at the authentication endpoint. The authentication request contains the same request parameters as in OAuth 2.0 in Section 2.2.7.1, with the following additions among others:

- `scope`: must contain `openid`
- `response_mode`: response mode, optional parameter specifying how parameters are to be returned by the OP. Possible values include [6] [20]:
  - `query`: return parameters in query parameters, e.g. `client.example.com/callback?code=XYZ`
  - `fragment`: return parameters in a URL fragment, e.g. `client.example.com/callback#code=XYZ`
  - `form_post`: return an automatically submitting form to the user-agent that submits the parameters with a POST request to the client. For more information, see Section 9.3.1.
- `nonce`: optional string that associates a client session with an ID token, mitigates replay attacks. Passed unmodified to the ID token.

For information on the `display`, `prompt`, `max_age`, `ui_locales`, `id_token_hint`, `login_hint`, and `acr_values` parameters, please consult the specification [37].

The following is an example of an authentication request:

```
GET /authorize?
  response_type=code
  &scope=openid%20profile%20email
  &client_id=s6BhdRkqt3
  &state=af0ifjsldkj
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb HTTP/1.1
Host: server.example.com
```

If the authentication request is valid, the OP attempts to authenticate the end-user. After authentication, the OP must obtain an authorization decision (i.e. consent) before releasing information to the RP.

The authentication response to the redirection endpoint is the same as in OAuth 2.0, for example:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
  code=SplxlOBeZQQYbYS6WxSbIA
  &state=af0ifjsldkj
```

To obtain an access token, an ID token, and optionally a refresh token, the RP sends a token request to the token endpoint of the OP. We do not discuss this request and response to it in detail, as it is not relevant for this thesis.

The RP then validates the ID token in the token response, which includes the following steps among others:

1. The issuer identifier for the OP must exactly match the value of the `iss` claim.
2. The RP must validate that the `aud` claim contains its `client_id`.
3. If the ID token is received via direct communication between the RP and the token endpoint (which it is in this flow), the RP may use TLS server validation instead of checking the token signature.
4. The current time must be before the time represented by the `exp` claim.
5. The `iat` claim can be used to reject tokens that were issued too far away from the current time.
6. If a nonce value was sent, a `nonce` claim must be present and match the one sent to prevent replay attacks.

The `azp`, `acr` and `auth_time` claims can also be validated. For more information, please see the specification [37].

If the ID token contains an `at_hash` claim, the RP may use it to validate the access token. This can be done by hashing the ASCII representation of the access token, taking the left-most half, base64url encoding it and then checking if it matches the `at_hash` value.

2.4.2.2 Implicit Flow

The implicit flow is also very similar to the OAuth 2.0 implicit flow. We will only comment on their differences.

The authentication request is the same as for the OIDC authorization code flow, except for a few differences:

- The `response_type` value is equal to `id_token token` (both an ID token and access token are returned) or `id_token` (only an ID token is returned).
- The `nonce` value is now required.

An example of an authentication request in the implicit flow:

```
GET /authorize?
  response_type=id_token%20token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile
  &state=af0ifjsldkj
  &nonce=n-0S6_WzA2Mj HTTP/1.1
Host: server.example.com
```

The OP then validates the authentication request, authenticates the end-user and obtains end-user consent/authorization in the same manner as with the authorization code flow.

The authentication response is the same as for the authorization code flow, except for the following differences:

- All response parameters are added to the fragment component of the redirection URI, unless a different response mode was specified.
- An `access_token` parameter is returned, containing the OAuth 2.0 access token if the `response_type` included `token`.
- A `token_type` parameter is returned.
- An `id_token` parameter is returned with the ID token.
- Optionally, the `expires_in` parameter is returned with the expiration time of the access token in seconds since the response was generated.
- No `code` parameter is returned.

An example of the authentication response in the implicit flow is as follows:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  access_token=SlAV32hkKG
  &token_type=bearer
  &id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
  &expires_in=3600
  &state=af0ifjsldkj
```

As with OAuth 2.0, the fragment component needs to be extracted by the user-agent.

The authentication response is validated in the same way as with the authorization code flow, except that the access token should also be validated by the RP using the `at_hash` value in the ID token.

In case an ID token returned in the authentication response, it is required to include the `nonce` parameter, and also the `at_hash` parameter in case an access token was requested.

The validation of the ID token must be done in the same manner as with the authorization code flow, except for two differences:

1. The RP must validate the signature of the ID token according to JWS using the algorithm specified in the `alg` header parameter.

2. The value of the `nonce` claim must be checked to verify that it is the same value as the one that was sent in the authentication request, and should be checked for replay attacks.

2.4.2.3 Hybrid Flow

The hybrid flow allows RPs to have immediate access to an ID token, while optionally ensuring secure retrieval of access and refresh tokens [37]. It is a combination of the authorization code and implicit flows.

The authentication request is the same as in the authorization code flow, except for these differences:

- The `response_type` parameter is equal to either `code id_token`, `code token` or `code id_token token`.

- The `nonce` parameter is required if the response type includes `id_token`.

An example of the authentication request in the hybrid flow is:

```
GET /authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
```

```
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com
```

The OP validates the authentication request, authenticates the end-user and obtains end-user consent or authorization in the same way as in the authorization code flow.

The authentication response is the same as for the implicit flow, with these exceptions:

- The `access_token` parameter is only returned when the response type includes `token`.
- The `id_token` parameter is only returned when the response type includes `id_token`
- The `code` parameter with the authorization code is always returned.

An example of the authentication response in the hybrid flow is:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  code=SplxlOBeZQQYbYS6WxSbIA
  &id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
  &state=af0ifjsldkj
```

To validate the authorization code with an ID token, the RP should use the `c_hash` claim in the ID token if it is present. This is done in the same way as with the `at_hash` claim, see Section 2.4.2.1

The contents of the ID token returned by the authorization endpoint in the hybrid flow must include:

- The `nonce` claim.
- The `at_hash` access token hash value when the response type is `code id_token token`.
- The `c_hash` authorization code hash value when the response type is `code id_token` or `code id_token token`.

2.4.3 Claims

A claim is a piece of information that is asserted about an entity (e.g. end-user). Predefined sets of claims can be requested using specific scope values, while individual claims can be requested using the `claims` request parameter. They can be requested to be returned either in the UserInfo response or in the ID token. Standard claims include values like `name`, `email`, and `phone_number`.

The UserInfo endpoint (e.g. `https://server.example.com/userinfo`) is an OAuth 2.0 protected resource that returns claims about the authenticated end-user if presented with an access token. For more information, please consult the specification [37].

For OpenID Connect, scopes can be used to request that specific sets of information be made available as claim values.

OIDC also defines the optionally supported `claims` parameter in the authorization request to request specific claims to be returned. For more information, please see the specification [37].

The combination of the `sub` and `iss` claims from the ID token are the only claims that can be used as a stable identifier. Other Claims such as email, phone_number, preferred_username, and name must not be used as unique identifiers for the end-user, whether obtained from the ID token or the UserInfo endpoint.

2.4.4 Passing Request Parameters as JWTs

OIDC defines the following authorization request parameters:

- `request`: optional request object, enables OIDC requests to be passed in a single, self-contained JWT parameter to be optionally signed and/or encrypted.
- `request_uri`: optional, enables OIDC requests to be passed by reference, rather than be value. It contains a URL referencing a resource containing a request object value. Valid values may be preregistered.

When the `request` parameter is used, this supersedes the parameters used in the normal request syntax. `request` and `request_uri` parameters must not be included in request objects.

An example of the request object before base64url-encoding and signing is:

```
{
 "iss": "s6BhdRkqt3",
 "aud": "https://server.example.com",
 "response_type": "code id_token",
 "client_id": "s6BhdRkqt3",
 "redirect_uri": "https://client.example.org/cb",
 "scope": "openid",
 "state": "af0ifjsldkj",
 "nonce": "n-0S6_WzA2Mj",
 "max_age": 86400,
}
```

2.5 Conclusion

This chapter provided the required knowledge on OAuth 2.0, JWT and OIDC to understand the rest of the chapters. OIDC is an extension of OAuth 2.0 that adds authentication via the ID token, which is a JWT. Although OAuth 2.0 and OIDC are proven secure, they are still vulnerable to browser-swapping attacks as described in Chapter 9.

The specifications include a lot of terminology and concepts which we tried to explain in this chapter. If you forget a term or concept while reading the next chapters, you can use the glossary at the start of this thesis as a reference.

We have only summarized the relevant parts of the OAuth 2.0, JWT and OIDC specification for our thesis. The OIDC specification includes a lot more, such as:

- More claims
- The UserInfo endpoint
- Aggregated and Distributed claims
- Self-issued OPs
- New client authentication methods
- Detailed rules for signatures and encryption

Please consult the specification [37] if you are interested in any of these topics.

Chapter 3

Related Work on Fuzzing OIDC implementations

There has not been prior research on fuzzing OpenID Connect implementations specifically. However, Yang et al. have used fuzzing with adaptive model-based testing to test OAuth 2.0 implementations for vulnerabilities [41]. Their testing framework, OAuthTester, uses a manually defined system model based on the OAuth specification. It automatically generates test cases (HTTP requests) that cover every execution path in the model, determines a parameter to be fuzzed and fuzzes it. This fuzzing is done either with predetermined properties of the parameter (which can also be learned for unknown parameters), or with specific domain knowledge. For example, the `redirect_uri` parameter is carefully crafted with domain knowledge of possible misconfigurations. The responses to the test cases are compared against the expected behavior defined in the system model. In case of non-conformance, some security properties are checked to determine if there is a vulnerability present. We use a similar approach with properties of parameters for our fuzzer in Section 7.4.

Aichernig et al. use **learning-based fuzzing**, a technique that combines automata learning and fuzz testing to test black-box systems, to test implementations of the MQTT protocol [1]. This differs from the adaptive model-based testing approach by Yang et al. in the way the state machine (model) is obtained: Yang et al. define the state machine manually, while with learning-based fuzzing, the state machine is learned with active automata learning. Learning-based fuzzing works as follows:

1. We define a coarse, abstract input and output alphabet for the state machine of the black-box system. For example, the input alphabet of the state machine of an OAuth 2.0 implementation could contain "authorize". This alphabet should be comprehensive enough to capture the core components of the system, but small enough to make state machine learning doable, as the alphabet-size strongly influences the

performance of active automata learning [1]. We also create abstract inputs for interactions that should cause errors, for example, "authorize_invalid" with invalid OAuth 2.0 parameter values.

2. We introduce a mapper that maps abstract input and outputs to concrete ones. We need this mapper, as the System Under Learning (SUL) does not understand the abstract input alphabet: we need to translate it to concrete interactions (e.g. HTTP requests) with the system. Moreover, the SUL's responses to these interactions (e.g. HTTP responses) might contain details like random values that will clutter the learning process, which should be abstracted away. This mapper could for example map the "authorize" input letter to the authorization request of the OAuth 2.0 protocol.

3. We start the *learning phase*: we use a learning algorithm to learn the abstract model of the SUL, for example an OAuth 2.0 implementation we learn.

4. We create a fuzzing mapper that maps abstract inputs to concrete fuzzed inputs, and concrete outputs back to abstract outputs. Only the abstract inputs that should trigger an error, e.g. "authorize_invalid", are mapped to fuzzed inputs, as fuzzed inputs are supposed to be invalid and to cause errors.

5. We start the *fuzzing phase*: we choose a System Under Test (SUT), either the same as the SUL or a different implementation, and test conformance with the learned model using the fuzzing mapper. Abstract inputs are mapped to fuzzed inputs by the fuzzing mapper and passed to the SUT. The SUT produces an output (e.g. an HTTP response), which is mapped to an abstract output by the fuzzing mapper. If this abstract output differs from the expected output in the abstract model, for example, when a fuzzed token does not cause an error but is accepted, we have found non-conformance to our learned model and a potential vulnerability.

We use learning-based fuzzing to fuzz OIDC implementations in Chapter 7 and improve upon it in Section 7.4.1.

In parallel to our research, Koper et al. researched stateful learning of OAuth implementations [21]. Their research is more focused on automata learning than ours and deals with more advanced issues like nondeterminism. Our research focuses more on fuzzing *using* prior stateful learning.

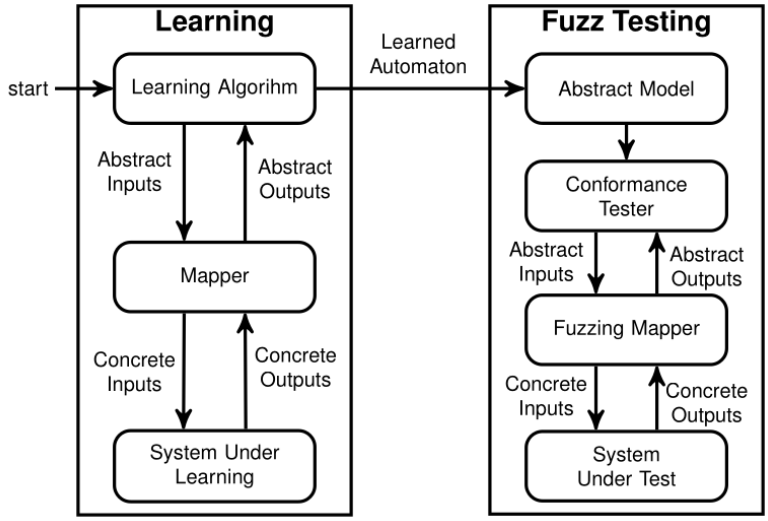


Figure 3.1: Learning-based fuzzing framework, taken from [1] (Figure 3).

Chapter 4

Security Requirements & Vulnerabilities in OIDC implementations

In this chapter, we provide an overview of known vulnerabilities in both OAuth 2.0 (Section 4.2.1) and OpenID Connect (Section 4.2.2) implementations. For more information about these protocols, see Chapter 2. We use these vulnerabilities in Section 6.2 to decide which fuzzing technique to use. We also collected security requirements for these protocols (Section 4.1 and Appendix A), which we have obtained from their specifications.

There are quite a lot of security requirements for both OAuth 2.0 and OIDC and also numerous potential vulnerabilities. While the protocols are now proven to be be secure by Fett et al. [9] [10], as mentioned in Sections 2.2 and 2.4, our impression is that it is rather easy to make a significant security mistake in the implementation of these protocols. For the vulnerabilities in Sections 4.2.1 and 4.2.2, we mention which vulnerabilities our fuzzer from Chapter 7 should be able to find.

4.1 Security Requirements

While reading the OAuth 2.0 [13] and OIDC [37] specifications, we kept lists of security requirements for both protocols. We also made sure to look for key words like “MUST”, “MUST NOT” and “SHOULD”. These lists often contain copied sentences from the specifications and are not exhaustive as they do not include any security requirements from other specifications like RFC 9700 “Best Current Practice for OAuth 2.0 Security” [24].

Although creating these lists gave us insight in the protocols, we did not end up using them for any other part of our research. We still include the lists because they might be valuable for future work. We give some ideas for this future work in Section 10.4.

To avoid cluttering this thesis, the list of security requirements can be found in Appendix A.

4.2 Vulnerabilities

In this section, we discuss a selection of vulnerabilities in OIDC implementations. We first discuss vulnerabilities in OAuth 2.0 implementations in Section 4.2.1, which in this case also apply to OIDC implementations. Then we continue by discussing vulnerabilities in OIDC implementations in Section 4.2.2. We discuss how these vulnerabilities may be found with fuzzing in Section 6.2.

We have obtained these vulnerabilities by searching for articles, blog posts, bug bounty reports, and other resources on the internet. Our collection of vulnerabilities is far from exhaustive. We have focused on vulnerabilities that are well-documented. We do not know which of vulnerabilities are most common, or if we may have missed vulnerabilities that are more common. It may be interesting to investigate which vulnerabilities are most common by, for example, analyzing CVEs as mentioned in Section 10.4.

4.2.1 OAuth 2.0 Vulnerabilities

In this section, we discuss some possible vulnerabilities in OAuth 2.0 implementations. For more information about OAuth 2.0, see Section 2.2.

Cross-Site Request Forgery (CSRF) is an attack in which a malicious web application tricks the victim's browser into making an authenticated request to an honest web application [2]. An example is that a malicious website may redirect the victim to a URL that transfers money to the attacker, e.g. `https://bank.com/transfer?amount=1000&to=attacker`.

4.2.1.1 Flawed `redirect_uri` validation

If the redirection URI (`redirect_uri`) is not properly validated, an attacker may execute a Cross-Site Request Forgery-like attack, by sending the victim an OAuth authorization link with an invalid, attacker controlled `redirect_uri`. If the victim (the resource owner) gives authorization, the authorization server will send the access token or authorization code to the attacker controlled `redirect_uri`. The attacker could abuse this token or code to achieve account takeover [32].

For example, the attacker could trick the victim to click the following link for account takeover:
`https://authserver.example.com/authorize?client_id=1234&redirect_uri=https://evil.com/pwn&response_type=code`

If the victim clicks the link, they might have to login for the code to be sent. If they already have an active session, this step might be skipped and

the redirect could occur immediately.

The attacker receives a request on their server: `GET /pwn?code=...` and can abuse the code for account takeover at the client, e.g. `https://client.example.com/callback?code=...`

To make the link less suspicious, the attacker can embed the link in an iframe within another page:

```
<html>
    <body>
        <iframe src="https://authserver.example.com/authorize?
    client_id=1234&redirect_uri=https://evil.com/pwn&
    response_type=code"></iframe>
    </body>
</html>
```

Ideally, the authorization server should keep a strict allow-list of allowed `redirect_uri` values per client. Sometimes, improper validation can be bypassed:

- If the authorization server checks if the `redirect_uri` starts with `https://client.example.com`, an attacker could abuse this by appending an "@" or ".", e.g. `https://client.example.com@evil.com` or `https://client.example.com.evil.com`
- Sometimes differences in URL parses can be abused, e.g. `https://client.example.com&@evil.com#@evil.com/`
- An open redirect vulnerability at the client could be abused, e.g. `https://client.example.com/callback/../redir?url=https://evil.com`
- Sometimes parameter pollution could work: `redirect_uri=https://client.example.com/callback&redirect_uri=https://evil.com`
- There might be an allow-list for certain development URLs like localhost: `localhost.evil.com`
- Validation via regular expressions could be implemented incorrectly. For example, `https://client.example.com/callback` could be interpreted as a regular expression, allowing random characters in the place of the dots [12]: `https://clientXexample.com/callback`.

Our fuzzer tests the `redirect_uri` for flawed validation in the fuzzing phase, as described in Section 7.4.

4.2.1.2 Flawed CSRF protection: missing `state`

The `state` parameter, discussed in Section 2.2.7.1, is recommended, but not required. It is used to prevent a CSRF attack. If it is omitted, the attacker could start an OAuth login flow and send his `redirect_uri` link to the victim, e.g. `https://client.example.com/callback?code=xyz`. If the victim clicks this link, or visits a page that embeds this link via e.g. an iframe, the victim will be logged in to the account of the attacker. This variant of CSRF is called *login CSRF* [2]. This is not always a security issue, but it could be in some cases. For example, if the client provides a way to link accounts via OAuth, the victim's account might be linked to the attackers account [32]. Or if the victim thinks they are logged into their own account, they might enter confidential information that can later be accessed by the attacker.

Our fuzzer tests whether the `state` parameter can be omitted and is bound to a specific session in the fuzzing phase in Section 7.4.

4.2.1.3 Flawed `scope` validation

The `scope` parameter can be used to restrict access of an access token. Sometimes, the scope is not validated correctly, and can be upgraded to include more restricted resources. For example, if the attacker controls a malicious client, they could send an access token request and add a scope parameter. Although this is not officially supported, the authorization server might use the scope parameter and update the scope of the access token. During implicit flow, the attacker does not even need to have a malicious client [32].

Our fuzzer does not assume the role of a client or Relying Party, as this is not in our attacker model as described in Section 7.1. Our fuzzer also does not include checks for scope upgrades, as it does not know the available scopes.

4.2.1.4 Token and code vulnerabilities

Access tokens, refresh tokens, but also authorization codes, need to be random and unguessable. If these values are generated insecurely, they might be guessable, which could allow for account takeover. In the case of signed tokens like JWTs, an attacker could try to change the data in the token and forge a signature. Tokens should also have a relatively short expiry time. Sometimes authorization codes can also be used more than once.

Our fuzzer checks whether authorization codes and access tokens can be used more than once, see Section 7.4. It does not detect whether these are insecurely generated, as this is a task better suited for Static Application Security Testing.

4.2.1.5 URL leaks

The redirection URI contains either the authorization code or the access token in a query string or URL fragment. If this URI gets leaked somehow, the code or access token can be stolen and abused for account takeover. This can for example happen if the redirection URI has a XSS vulnerability, or if it supports a postMessage that leaks `location.href` [36]. In the case of a stolen authorization code, this code must not have been used, as it can only be used once. To prevent the authorization code from being used, an attacker can try to trigger an "unhappy" flow, a flow that ends in an error and thus does not use the authorization code, but does include a valid authorization code in its URL. This technique is called "dirty dancing" [36].

Our fuzzer does not detect URL leaks, as finding XSS vulnerabilities or vulnerable postMessages are complex problems on their own and therefore outside of the scope of our fuzzer.

4.2.2 OpenID Connect Vulnerabilities

All OAuth 2.0 vulnerabilities from Section 4.2.1 can also occur in OpenID Connect implementations, as OpenID Connect is an extension of OAuth 2.0[1]. For more information on OIDC, see Section 2.4.

Server-Side Request Forgery (SSRF) is an attack in which an attacker can create HTTP requests from a vulnerable web application server to the internet or its local network [25]. For example, an attacker might trick an OIDC implementation into sending a request to `http://192.168.0.1/shutdown`, which might shut down a server in the local network with IP 192.168.0.1.

4.2.2.1 Unprotected dynamic client registration

OpenID Connect specifies a standard way for clients to dynamically register themselves at the OpenID provider, by sending a POST request to a registration endpoint with information about itself in JSON format. This registration should require client authentication, however, some providers allow registration without it. This allows an attacker to register their own malicious client, which could have security consequences [33]. For example, the OpenID provider might reach out to some of the URIs provided by the attacker during registration, leading to a Server-Side Request Forgery (SSRF) vulnerability.

This attack vector is not in scope for our fuzzer, as client registration is not part of our attack model, see Section 7.1.

[1] This does not apply to all OAuth 2.0 vulnerabilities in general, as some OAuth 2.0 vulnerabilities are mitigated by features of OIDC. For example, the nonce value can prevent some replay attacks when the state parameter is not used [10]

4.2.2.2 `request` and `request_uri` vulnerabilities

The `request` and `request_uri` parameters allow a relying party to pass authorization request parameters as a JWT, see Section 2.4.4. Some implementations might fail to validate the parameters in the JWT in the same manner as in the query string [33].

If arbitrary values for the `request_uri` parameter are allowed, than this could lead to a Server-Side Request Forgery (SSRF) vulnerability when the server reaches out to this URI.

Our fuzzer tests the `request` object for vulnerabilities in the fuzzing phase in Section 7.4, but does not test the `request_uri`.

4.2.2.3 JWT vulnerabilities in ID token

The ID token in OIDC is a JSON Web Token (JWT) as described in Section 2.4.1. For more information on JWTs, see Section 2.3. There are a lot of mistakes an implementation can make with a JWT. These mistakes can lead to serious security issues, as the claims in an ID token are trusted by the relying party.
Some attacks on JWTs include [40]:

- Changing the `alg` value in the header to `none`. If this is accepted by the RP, the signature is not validated, which allows an attacker to inject arbitrary claims.
- Changing the `alg` value in the header from `RS256` to `HS256`. This might make the RP validate the signature using the public key as the HMAC shared secret. An attacker could then forge a signature by signing their own ID token using the public key as the shared secret.
- Cracking the signing key of a JWT, in case a weak key is used.
- The JWT might not be verified correctly. Some implementations might accept empty signatures or any signature [31].
- Some relying parties might accept the parameters `jwk` or `jku`. This parameters specify which keys should be used for signature validation. If these parameters are allowed and not properly validated, an attacker could trick the server into validating the ID token with their own key [31].

There already exist tools for testing JWT vulnerabilities, for example, `jwt_tool`[2].

Our fuzzer tests for a few JWT vulnerabilities in the ID token and access token in the fuzzing phase in Section 7.4, like the none algorithm.

[2]GitHub: `https://github.com/ticarpi/jwt_tool`

4.2.2.4 `nonce` vulnerabilities

The `nonce` parameter is an optional parameter that is intended to prevent replay attacks [37], as discussed in Section 2.4.2.1. If this parameter is not used or not validated correctly, an attacker might be able to perform a replay or session fixation attack against the relying party, similar to the `state` parameter in OAuth 2.0, as discussed in Section 4.2.1.2.

Our fuzzer tests the `nonce` parameter in the fuzzing phase in Section 7.4, for example whether it is bound to a specific session.

Chapter 5

OAuth CTF: Custom vulnerable OAuth 2.0 implementation

We have created an OAuth 2.0 Capture The Flag (CTF) with 5 different challenges. The goal of a Capture The Flag is to exploit an application to obtain a protected string, the flag. This flag can then be handed in for points. The player has to exploit a different vulnerability for each challenge to get the corresponding flag. The CTF is open source and can be found on GitHub[1], although the installation process is currently undocumented.

McDaniel et al. found that CTFs are widely successful at introducing players to technologies, but also at motivating continual learning [26]. In our personal experience, creating the CTF also helped us gain better understanding of OAuth. Additionally, it provided us with a great target for our fuzzer during development and testing. We show which of the OAuth CTF challenges our fuzzer is able to solve in Section 8.1. Note that this chapter *includes the solutions to the challenges*, so if you want to try them yourself, feel free to skip this chapter.

5.1 Challenges

There are five challenges in total, consisting of three redirection URI vulnerabilities (Section 4.2.1.1), one scope vulnerability (Section 4.2.1.3), and one URL leak vulnerability (Section 4.2.1.5). We limited the challenges to these classes of vulnerabilities out of time constraints and personal interest for these vulnerabilities. Specifically, the flawed Cross-Site Request Forgery protection from Section 4.2.1.2 appeared difficult to implement, because it required setting up a scenario in which this CSRF had security impact (e.g.

[1] `https://github.com/Harm-r/oauth-ctf`

with account-linking functionality). Furthermore, we found vulnerabilities in tokens and codes from Section 4.2.1.4 to be a bit stale and less interesting.

It would be possible to extend this CTF to also include each OIDC vulnerability from Section 4.2.2, see Section 10.5.

5.1.1 Basic redirect_uri

In this challenge, the player has to modify the `redirect_uri` parameter to direct the target to their own malicious server[2]. By changing the redirect_uri parameter to their own server, and sending the authorization request link to the target with the admin bot, they will receive an authorization code at their web server. They can exchange this code for a session at the redirection endpoint, and obtain the flag with the admin session.

5.1.2 Validation bypass

This time, the application checks that the `redirect_uri` starts with the client URL, e.g. `https://client.example.com`. However, by appending "@evil.com", the URL becomes `https://client.example.com@evil.com`, and the user is redirected to evil.com. Using this trick, the player can once again redirect the admin bot to their malicious server, steal the authorization code and get the flag.

5.1.3 Regex & Headers

The application checks that the `redirect_uri` matches a regular expression. However, this regular expression does not end in "$", allowing for a newline character to escape the regular expression. Moreover, the application processes the `redirect_uri` in such a manner that it allows for header injection. Using a `redirect_uri` like "https://client.example.com\r\nLocation: evil.com", the player can bypass the regular expression and inject a Location header to redirect to the malicious web server, obtain an authorization code and get the flag.

5.1.4 Basic scope

There is no scope validation in this challenge: the player can obtain arbitrary scope by appending a "scope=read:flag" to their authorization request. With this scope, the player can read the flag.

[2]The player can use a service like https://requestrepo.com to avoid having to host a web server themselves

5.1.5 Error

This is the most advanced challenge. The player needs to realize that they can cause an error in the authorization request to get on a non-happy path: an authorization code is sent to the redirection endpoint, but it is not consumed. By specifying a custom error, the player can exploit a cross-site scripting (XSS) vulnerability in the redirection endpoint. By sending the admin bot to the authorization endpoint with this cross-site scripting payload and non-happy path in the `redirect_uri`, they can extract the authorization code with cross-site scripting and obtain the flag.

Chapter 6

Fuzzing techniques

There are a lot of different ways to fuzz. This chapter shows how we decided which fuzzing technique to use. We first give some background information on fuzzing in Section 6.1, in which we discuss multiple fuzzing techniques. Then, we discuss how fuzzing might find the vulnerabilities from Chapter 4 in Section 6.2. Finally, we decide to use learning-based fuzzing as defined by Aichernig et al. [1] in Section 6.3.

We have decided to focus on fuzzing techniques over other types of testing techniques, like SAST or DAST. We chose fuzzing because of our interest in it, but also because it is widely regarded as one of the most effective and scalable vulnerability discovery techniques [23]. Additionally, SAST and other DAST techniques come with some trade-offs:

SAST (Static Application Security Testing) is a testing technique in which source code is scanned for vulnerabilities, without actually executing the program. SAST requires source code, but we would also like to be able to test closed-source OIDC implementations. Therefore, SAST is not well-suited for our purposes. SAST is also prone to false-positives due to its lack of a vulnerability detection model [23]. However, SAST might be better at uncovering certain types of vulnerabilities like predictably generated tokens from Section 4.2.1.4, where knowledge of the source code makes detection a lot easier.

DAST (Dynamic Application Security Testing) is a type of testing that tests an application at runtime [23]. Fuzzing is a type of DAST: it also tests an application at runtime. DAST can either be done manually or automatically (e.g. fuzzing). We focus on automated testing over manual testing, as manual testing is less efficient [23]. However, manual testing might outperform automated testing in some scenarios, for example, when a vulnerability is hard to detect, like the URL leak vulnerability from Section 4.2.1.5. This vulnerability requires multiple conditions to be satisfied, and some conditions like e.g. an XSS vulnerability are already difficult to automatically detect on their own due to their complexity. Creating a collection of manual test cases

might be interesting future work, see Section 10.4.

6.1 Background on Fuzzing

Fuzzing is the process of feeding random input into a program in the hope of uncovering failures [42], which may result in vulnerabilities. The origin of the word stems from a programming exercise Miller gave to his students, in which the students fed unpredictable input into Unix utilities to find bugs [27].

The program or system that we test is called the *system under test (SUT)*. We can fuzz either stateless or stateful systems. A stateless system takes only a single input. A *stateful system* takes a sequence of messages as input, where inputs might change some internal state of the system [5]. The format of the input to a stateful system consists of the format of the individual messages (*message format*, described by, for example, a *context-free grammar*), and the format describing the *sequences of the messages* (described by a *state model* or *(protocol) state machine*). A *grammar* describes the input to a stateful system, both the message format and sequence of messages. We focus on stateful fuzzing in this thesis, because most functionality in OIDC can only be reached after some initial sequence of messages.

We define a *trace* to a stateful system as a sequence of messages, e.g. trace ABC is a sequence of messages A, B, and C. We say that we are fuzzing the message format, when we change the format of at least one message in the trace, e.g. trace ABC becomes $AB'C$, where B' is a mutated variant of B. We say that we are fuzzing the sequence of messages when we do not change the format of any message in a trace, but only the order in which the messages are sent, e.g. trace ABC becomes ACB.

We can categorize fuzzers in multiple ways. One way is to differentiate between how the fuzzer observes the SUT [5]:

- *Black-box fuzzers* only observe the SUT from the outside. They require knowledge of the message format, either via:
  - a grammar of the message format and possibly the sequence of messages: *generation-based* or *grammar-based* fuzzing
  - sample traces which the fuzzer mutates: *mutation-based* fuzzing
- *White-box fuzzers* have access to the program code and generate inputs using analysis of this code, using for example symbolic execution.
- *Gray-box fuzzers* live somewhere in the middle. They can observe parts of the SUT and use these observations as feedback. They are also known as *evolutionary fuzzers*. *Coverage-guided gray-box fuzzers* monitor which code branches are taken in the execution of the SUT for a trace, and use these as feedback. This often requires some instrumentation of the code or requires running the code in an emulator.

A fuzzer requires some mechanism to mutate a message. Generally, this is done randomly with e.g. bit flips, or according to a grammar. We define *ad hoc mutations* as mutations specifically designed for a target or vulnerability. These mutations are generally more effective at uncovering a specific vulnerability, but are less likely to find novel vulnerabilities and are not generally applicable to other targets or protocols.

A fuzzer also requires some mechanism to detect if a trace triggered a bug in the SUT, for example a program crash or a deviation from the state machine.

Daniele et al. define seven categories of fuzzers for stateful systems [5]:

1. **Grammar-based fuzzers**: fuzzers that use a user-provided grammar to fuzz. The grammar has to describe both the message format and the protocol state machine. The requirement of a grammar can be a downside, as producing an accurate grammar can be challenging and error-prone [5].

2. **Grammar-learner fuzzers**: fuzzers that extract a grammar from traces or interaction with the SUT, which they then use to fuzz.

3. **Evolutionary fuzzers**: fuzzers that mutate sample traces using a feedback system to steer the mutation. This feedback can use different types of observations, like the response of the SUT, the branch coverage using e.g. instrumentation, manual branch annotation, values of program variables, or memory segments.

4. **Evolutionary grammar-based fuzzers**: fuzzers that use a grammar to generate traces and an evolution mechanism to mutate them.

5. **Evolutionary grammar-learner fuzzers**: fuzzers that use two feedback mechanisms to both mutate individual messages and also mutate sequences to infer a protocol state machine.

6. **Man-in-the-Middle fuzzers**: fuzzers that sit between the SUT and a program interacting with it and modify messages going to the SUT. These fuzzers have the limitation that the sequence of the messages can only be modified in a limited way.

7. **Machine learning fuzzers**: fuzzers that use a machine learning model trained on a large set of input traces to generate mutated traces. These have a hard time generating new traces that are not in the training data.

We use these categories to narrow down our preferred fuzzing technique in Section 6.3.

6.2 Fuzzing for OIDC vulnerabilities

In this section, we go over the OAuth 2.0 and OIDC vulnerabilities from Chapter 4 and describe how we think these vulnerabilities could be found with fuzzing. Note that the vulnerabilities in our OAuth CTF of Chapter 5 are also included in Chapter 4. For each vulnerability, we discuss two main components of fuzzing: mutation, how to mutate a message to trigger the vulnerability, and detection, how to detect whether the vulnerability was triggered.

6.2.1 OAuth 2.0 vulnerabilities

For an explanation of each OAuth 2.0 vulnerability, see Section 4.2.1.

- **Flawed `redirect_uri` validation** (Section 4.2.1.1):
  - Mutation: needs to be intelligent to find specific bypasses or flaws in URL parsing, as random mutation is very unlikely to trigger this vulnerability. We think a grammar-based URL fuzzer or ad hoc mutations using specific bypass techniques would be best.
  - Detection: we need to detect an unexpected redirect. This can be done with either a specific detection rule or by detecting a deviation from a state machine.
- **Flawed CSRF protection: missing `state`** (Section 4.2.1.2):
  - Mutation: omit `state` parameter, or maybe substitute with e.g. value from a different session, which would require the fuzzer to keep track of values from a different session.
  - Detection: detecting a missing or invalid `state` requires either a specific detection rule or a deviation from a state machine.
- **Flawed `scope` validation** (Section 4.2.1.3):
  - Mutation: try each of the available scopes, these can be found in the OP configuration document[1].
  - Detection: difficult, you need to somehow know which scopes are allowed and which are not. Maybe these scopes could be added manually as input to the fuzzer.
- **Token and code vulnerabilities** (Section 4.2.1.4):

[1]This is a file located at `/.well-known/openid-configuration` at the OP containing information like the supported scopes. For more information, see the OpenID Connect Discovery specification: `https://openid.net/specs/openid-connect-discovery-1_0.html`

  - Mutation: to test for weakly generated tokens and codes via mutation, you would have to guess a large amount of tokens and codes. You can check for one-time use via replays.
  - Detection: maybe detect short lengths or weak entropy without mutation at all. SAST seems better suited for this.

- **URL Leaks** (Section 4.2.1.5): infeasible, requires analysis of client-side code, which is a large problem on its own and outside of the scope of our fuzzer. It should however be possible to find non-happy flows by combining parameter values in an unexpected way, which we could flag as differences with a reference flow. This generally just involves manipulating messages (parameters), not the sequence of messages.

6.2.2 OIDC vulnerabilities

For an explanation of each OIDC vulnerability, see Section 4.2.2.

- **Unprotected dynamic client registration** (Section 4.2.2.1): out of scope for our fuzzer, see Section 7.1.
- `request` **and** `request_uri` **vulnerabilities** (Section 4.2.2.2):
  - Mutation: for `request_uri`: similar to `redirect_uri`, grammar-based URL fuzzer or ad hoc mutations. For `request`: test whether each parameter within the request object is properly validated. For example, mutate the `redirect_uri` in the `request` object.
  - Detection: for `request_uri`: test whether your own server or web hook gets a request from the target OP. For `request`: use detection strategies of parameters contained in the `request` object.
- **JWT vulnerabilities in ID token** (Section 4.2.2.3):
  - Mutation: grammar-based JWT mutation or ad hoc mutations testing for specific vulnerabilities. Random mutations like bit flips are unlikely to create a valid JWT.
  - Detection: specific detection rule to detect acceptance of fuzzed JWTs or a deviation from a state machine.
- `nonce` **vulnerabilities** (Section 4.2.2.4): same as `state` parameter
  - Mutation: omit `nonce` parameter, or maybe substitute with e.g. value from different session.
  - Detection: either a specific detection rule for the acceptance of an invalid or missing nonce or a deviation from a state machine again.

6.2.3 Conclusion

Our selection of vulnerabilities seems to suggest that issues are mostly found in the message format, not in the sequence of messages. For the difference between the message format and the sequence of messages, see Section 6.1. We described how we selected these vulnerabilities in Section 4.2. Because we did not research which vulnerabilities are most common, this is not an academic claim by any means, but rather an impression from our work. It may also be the case that changing the sequence of messages is done less often during testing of OIDC, causing vulnerabilities to be less often found this way. Because of this, we still decide to change the sequence of messages during fuzzing in Section 6.3. Researching which OIDC vulnerabilities are most common is interesting future work as described in Section 10.4. Our fuzzer tests the message format in the fuzzing phase in Section 7.4.

Mutations differ a lot per parameter and per vulnerability. Because of this, we think ad hoc mutations are more effective at finding these vulnerabilities and easier to implement than grammars for each parameter. This does come with the trade-off that ad hoc mutations are less likely to find other vulnerabilities than the ones they test for.

Detecting vulnerabilities in OIDC implementations is a lot more difficult than with binary fuzzing, as we cannot just detect crashes or exceptions. Most vulnerabilities are actually in cases where input traces should trigger an exception or error, but do not. Detecting vulnerabilities as deviations from expected behavior in a state machine seems more promising to us than writing detection rules for each security boundary.

We do not expect our fuzzer to find all types of vulnerabilities. For example, finding URL leaks from Section 4.2.1.5 seems impossible, as you need to find a client-side web attack like an XSS, which our fuzzer does not search for. We believe that, in general, automated testing should be supplemented with manual testing, as automated testing might miss some vulnerabilities.

6.3 Choosing a fuzzing technique

We review the categories of stateful fuzzers from Section 6.1 to see which category fits our use case best. To recall, Daniele et al. define seven categories of fuzzers for stateful systems [5], **Grammar-based fuzzers**, **Grammar-learner fuzzers**, **Evolutionary fuzzers**, **Evolutionary grammar-based fuzzers**, **Evolutionary grammar-learner fuzzers**, **Man-in-the-Middle fuzzers** and **Machine learning fuzzers**.

From Section 6.2, we know that most vulnerabilities can be found with modification of just message format, not the sequence of the messages. However, we would still like our fuzzer to have the option of changing the sequence of messages, as this might help us uncover additional vulnerabilities. There-

fore, man-in-the-middle fuzzers and machine learning fuzzers do not fit our use case, as these categories of fuzzers often do not support changing the sequence of messages or only in a limited way.

Now we have another decision to make: do we prefer a black-box or gray-box/white-box fuzzer, for example, evolutionary fuzzers? Evolutionary fuzzers often require instrumentation, which we cannot do from a black-box perspective. Because we would like to test multiple OIDC implementations, written in different programming languages, instrumentation is hard or sometimes impossible to do. We would also like to have the option to test OIDC implementations of which we do not have the source code or binary. Therefore, we reject evolutionary fuzzers and build a black-box fuzzer.

What remains are grammar-based fuzzers and grammar-learner fuzzers. This brings us to our next question: do we want to learn e.g. the OIDC state machine or grammar from an implementation, or do we want to provide it to the fuzzer ourselves? Learning a grammar has the advantage of being less time-consuming. It may also be the case that multiple state machines would conform to the specifications: the specifications do not define a strict state machine themselves. However, writing a grammar ourselves would probably result in a more complete and formal grammar, which would be a great academic contribution on its own. As explained in Section 6.1, a grammar has two levels: the definition of the message format and the definition of the sequence of messages (state machine). We decide to start by trying to learn the state machine first, because we expect that this saves us a significant amount of time, while we define mutations for the message format. If this does not work, we will write the state machine ourselves. Note that we will only try to learn the state machine, not the message format.

Another question is how we will detect errors in the implementations. Detecting errors in implementations of protocols is not as easy as looking for crashes: we need to find deviations of the original protocol specification. Because we want to avoid creating a state machine manually, a great alternative is comparing learned state machines of different implementations. Differences between implementations suggest that one of the implementations contains an error. We can also use the learned state machine to detect vulnerabilities during fuzzing, as deviations from the state machine might be caused by bugs.

How will we mutate messages? In Section 6.2, we argued that random message mutation has worse performance than using grammars or ad hoc mutations for specific parameter types, like URLs or JWTs. Furthermore, we think that using a specific grammar for each parameter type is infeasible for us due to time constraints. Therefore, we will mostly use ad hoc mutations tailored to each type of parameter and vulnerability.

Concretely, we will follow the learning-based fuzzing process from Aichernig et al. [1], as described in Chapter 3. Learning-based fuzzing fits our choices on types of fuzzing techniques perfectly, as it allows us to fuzz both

the sequence and format of messages. It is also a black-box fuzzing technique that learns the state machine, and allows us to use deviations from learned state machines to find vulnerabilities. It does not define a fixed way to mutate messages, so we can customize this to our liking.

To implement learning-based fuzzing, we will use the AALpy Python library[2] [28]. AALpy is an automata learning library which will allow us to learn the state machines of OIDC implementations, between which we can then find differences. It even includes a short example on learning-based fuzzing in the documentation[3]. AALpy is developed by DES Lab, which contains authors of the learning-based fuzzing method [1].

Learning-based fuzzing contains a learning phase, in which we learn the state machine of an implementation, and a fuzzing phase, in which we fuzz an implementation (either the same implementation, or a different one) using the state machine. We decided to fuzz the same implementation as for which we learned the state machine, because the state machines turned out to be too different in practice. We give more details on our setup of the learning phase in Section 7.3 and on the fuzzing phase in Section 7.4.

[2] `https://github.com/DES-Lab/AALpy`

[3] `https://github.com/DES-Lab/AALpy/wiki/Learning-Based-Fuzzing-with-AALpy`

Chapter 7

Fuzzing setup

In this chapter, we describe our fuzzing setup and process in detail. Our fuzzer is open-source and can be found on GitHub[1]. In Section 7.1, we describe our attacker model: the capabilities of our fuzzer. We shortly list our systems under test in Section 7.2.

As mentioned in Section 6.3, our fuzzer follows the learning-based fuzzing approach from Aichernig et al. [1], which we described in Chapter 3. This means that the fuzzer consists of two main phases:

1. The learning phase: learning the state machine of the SUT, which we describe in Section 7.3. We manually analyze the state machine afterwards. This process is called *protocol state fuzzing* by De Ruiter et al. [7], as it changes (fuzzes) the sequence of messages during learning.

2. The fuzzing phase: Fuzzing the SUT using the learned state machine, which we describe in Section 7.4. This step fuzzes the content of messages as well as their sequence.

7.1 Attacker model

Our attacker model is shown in Figure 7.1 and is defined as follows:

1. Our fuzzer assumes the role of the end-user: we allow it to modify traffic to the RP (Relying Party) and OP (OpenID Provider).

2. We assume that the attacker is not able to inspect or modify the traffic between the OP and RP: our trusted computing base (TCB).

3. Our fuzzer has credentials for two accounts[2].

[1] `https://github.com/Harm-r/oidc-learning-based-fuzzing`

[2] We require at least one account to complete a flow. The other account is used to test whether certain parameter values, like the authorization code, are bound to a specific user.

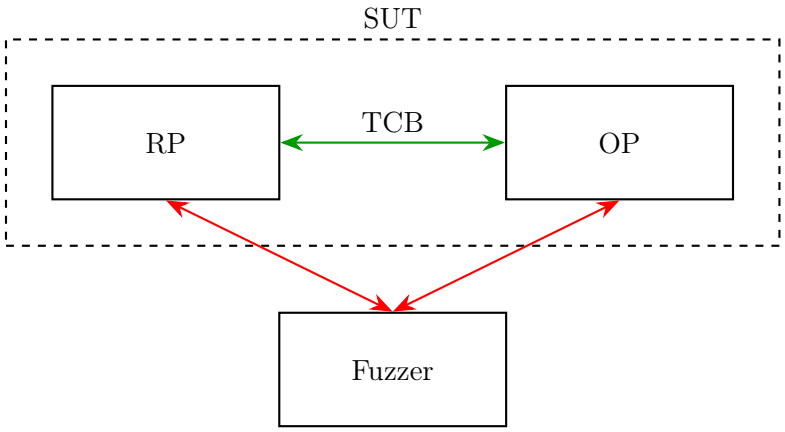


Figure 7.1: Our attacker model. Our fuzzer is positioned as an end-user, it can not modify traffic between the RP and OP (the trusted computing base, TCB), or assume the role of the RP or OP. The RP and OP together form the system under test (SUT).

Our fuzzer cannot assume the role of an RP or OP. The RP and OP together form the system under test (SUT). We do not include client registration as described in Section 2.2.5.

We chose this attacker model because it is the most realistic (an end-user becoming an attacker). In most cases, it is less feasible for an attacker to assume the role of an RP or OP in an already configured OIDC setup. Without valid credentials, our fuzzing would be less exhaustive: less states of the protocol would be fuzzed. However, other attacker models are still interesting future research, which we describe Section 10.3.

7.2 Systems under test

Our fuzzing setup requires both an OP (OpenID Provider) and an RP (Relying Party) in our system under test (SUT). We used three different combinations of OPs, RPs, and flows, as shown in Table 7.1.

The target implementations (the SimpleSAMLphp OIDC, SimpleSAMLphp AuthOAuth2, Shibboleth OIDC and Apache mod_auth_openidc modules) were chosen in consultation with the T&I Incubator according to their popularity in the NREN community. Because it was challenging to automate the login step at SUT 3, we enabled automatic login with HTTP Basic Auth, which is why there is no login transition in Figure 8.2b.

We only fuzz the authorization code and implicit flows, as specified in Table 7.1. No other parts of the protocols (no hybrid flow, no dynamic client registration, etc.) are fuzzed. We chose to only focus on the authorization

#	**OP**	**RP**	**Flow**	**Setup**
1	SimpleSAMLphp OIDC module	SimpleSAMLphp AuthOAuth2 module	Authorization code	Appendix B
2	SimpleSAMLphp OIDC module	SimpleSAMLphp AuthOAuth2 module and Apache mod_auth_openidc module	Authorization code and implicit	Appendix B + part of Appendix C
3	Shibboleth OIDC module	Apache mod_auth_openidc module	Authorization code	Appendix C

Table 7.1: Combinations of OPs, RPs, and flows that were fuzzed

code and implicit flows because they are part of the core specification[3], because the hybrid flow is logically just a combination of those flows, and because we did not have time to expand our fuzzing to other parts of the protocol.

To test both the authorization code flow (Section 2.4.2.1) and the implicit flow (Section 2.4.2.2) together, we use two different RPs in setup 2: the SimpleSAMLphp AuthOAuth2 module for the authorization code flow, and the Apache mod_auth_openidc module for the implicit flow. We require two different RPs, as the AuthOAuth2 module only supports the authorization code flow, and the mod_auth_openidc module can only be configured to use one flow.

7.3 Learning phase

Learning-based fuzzing starts with a learning phase. We have to define a mapper from abstract inputs to concrete inputs, and concrete outputs to abstract outputs, such that the learning algorithm can use this abstract alphabet to learn the state machine of the SUT effectively. We define the following abstract inputs and their concrete mappings:

- `client_sso_login`: Sends a request to the RP that starts the OIDC authentication flow.
- `client_callback`: Sends a request to the redirection endpoint of the RP with the stored authorization code and state.
- `client_callback_invalid`: Sends a request to the redirection endpoint of the RP with invalid `code` and `state` parameters.

[3]For an overview of all other specifications in the OpenID Connect Suite, see Figure 2.3 in Section 2.4.

- `client_callback_implicit`: Sends a request to the redirection endpoint of the Apache mod_auth_openidc module RP with the stored implicit flow parameters.
- `client_callback_implicit_invalid`: Sends a request to the redirection endpoint of the Apache mod_auth_openidc module RP with invalid implicit flow parameters.
- `authserver_authorize`: Sends a request to the authorization endpoint of the OP using the stored client identifier, redirection URI, `state` and `scope` using the authorization code flow. Randomly picks between providing the parameters in the query string or in a request object.
- `authserver_authorize_implicit`: Sends a request to the authorization endpoint of the OP using the stored client identifier, redirection URI, `state`, `nonce` and `scope` using the implicit flow. Randomly picks between providing the parameters in the query string or in a request object.
- `authserver_authorize_invalid`: Sends an invalid request to the authorization endpoint of the OP using hard coded invalid parameters.
- `authserver_login`: Sends a POST request to the login endpoint of OP, authenticating the user.
- `authserver_login_invalid`: Sends a POST request to the login endpoint of the OP, but with invalid credentials.

The requests are sent using the Python "requests" library[4]. This library sends HTTP requests, which is faster than browser-emulation. If an implementation requires browser-emulation, e.g. because they use browser-specific features like local storage, a library like Selenium[5] might be a viable alternative.

Most of these inputs parse the parameters in the response and save these for later use. We include invalid inputs as well, as we need to know how the SUT responds to invalid requests, as this is how we will expect it to respond to fuzzed requests in the fuzzing phase. It is important to keep the input alphabet as small as possible, as a large input alphabet will significantly increase learning time.

We also map concrete outputs to abstract outputs as follows:

- All errors, determined by response codes and error messages in the response, get mapped to generic "Error" strings

[4] `https://pypi.org/project/requests/`
[5] `https://www.selenium.dev/`

- Redirects get mapped to a tuple containing the response code and the location of the redirect. The parameters in the URL being redirected to are replaced with placeholders, such that new values for e.g. the code parameter are not interpreted as different abstract outputs.

Next, we run the L* algorithm with the abstract input alphabet and the mapper on the SUT. This is *active learning*, we learn the state machine by interacting with the implementation directly. We stick to the methodology of Aichernig et al. and also use the Random Walk equivalence oracle [1].

The result of the learning phase is a state machine consisting of transitions between our abstract inputs and abstract outputs. This is a Mealy machine: it contains states with transitions between them, and each transition has an input and an output. An example of such a transition is:

- Input: client_sso_login
- Output: (303, 'Location: https://example.com/simplesaml-op/module.php/oidc/authorization?<PARAMS>')

Note that learning the state machine of the SUT can also be considered a form of fuzzing, as during learning, we are changing the sequence of messages sent to the SUT. For example, a SUT might expect a trace of three messages A, B and C in order ABC, but will receive other orderings of those messages during learning like ACB. De Ruiter et. al define this as *protocol state fuzzing* [7]. We always manually evaluate the learned state machine for unexpected states or transitions that might be caused by bugs. Moreover, two OIDC implementations implement the same specification and should have the same state machines. We compare learned state machines and look for differences, as these might point to bugs one of the implementations.

7.4 Fuzzing phase

In the fuzzing phase, we try to find *counter examples*, traces that AALpy detects as contradictions to the learned state machine[6]. These counter examples might be caused by bugs in the implementation. To do this, we walk through the learned state machine, and then fuzz certain concrete inputs. We describe how we use the learned state machine in Section 7.4.1. We fuzz concrete inputs by mapping certain abstract inputs to fuzzed inputs instead. We only fuzz the `client_callback_invalid`, `client_callback_implicit_invalid` and `authserver_authorize_invalid` inputs, as these are expected to produce invalid results and trigger an error, just like fuzzed inputs. When a fuzzed input then does not trigger an error, this might point towards a bug

[6]We explain this a bit further in Section 7.4.1

like an authentication bypass, and will be detected as a deviation from the learned state machine.

A fuzzed concrete input is a valid concrete input but with one fuzzed parameter. We fuzz the parameters based on their expected properties, which we define ourselves based on the OIDC specification [37] and our knowledge of vulnerabilities from Chapter 4, while also including the properties from OAuthTester from Yang et al. [41]. A property of a parameter describes its expected, secure behavior. For example, a parameter might have the "mandatory" property, which means that in a secure implementation it is mandatory to include the parameter in a request. We define the following properties of parameters and their mutation strategies:

- **Constant**: constant parameters are mutated slightly by changing the last character to "X", e.g. `scope=openid` becomes `scope=openiX`.
- **Mandatory**: mandatory parameters are omitted.
- **Used once**: parameters that should only be used once are replaced with old values that have already been used.
- **User-specific**: user-specific parameters are replaced with a value belonging to a different user.
- **Session-specific**: session-specific parameters are replaced with a value from a fresh session.
- **Flow-specific**: a parameter value obtained from e.g. the implicit flow is used in the authorization code flow instead.
- **URL**: URLs are modified to a value that might bypass validation, using one of the mutations from the PortSwigger URL validation bypass cheat sheet[7].
- **Request Object**: Recall that the request object is a way to wrap other parameters in a JWT as described in Section 2.4.4. We use specific mutation strategies for the request object:
  - We mutate the parameters within the request object according to their own properties and provide them within a request object
  - We mutate the parameters and provide them within the request object, while also providing valid parameters in the query string
  - We provide a request object with valid parameters within a request object with invalid parameters, a nested request object

[7] `https://portswigger.net/web-security/ssrf/url-validation-bypass-cheat-sheet`

- **Token**: JWT tokens like the access token (in the SimpleSAMLphp OIDC module, the access token is a JWT) and the ID token are modified using some JWT mutation strategies:
  - We slightly mutate the signature to see if the signature is verified at all
  - We provide an empty signature (NULL signature) to see if this bypasses verification
  - We specify `{"alg":"none"}` in the JWT header to see if this is accepted, which would allow us to bypass signature validation

We also define some mutation strategies that apply to all parameters:

- We test for type juggling by converting parameters to arrays, e.g. `code=abc` becomes `code[]=abc`.
- We try to duplicate a parameter, by fuzzing it and also including a valid value in the same request, e.g. `code=abc&code=FUZZED`.
- We try valid parameter values of different parameters. For example, we might use a valid authorization code instead of an access token in the implicit flow.

We mostly use *ad hoc mutations*, instead of random mutations according to e.g. a grammar. For example, we mutate URLs according to the PortSwigger URL validation bypass cheat sheet, instead of randomly. Our conjecture is that ad hoc mutations perform better than grammar-based mutations for fuzzing OIDC implementations, because they are designed to test for specific vulnerabilities more efficiently. For example, we believe that the PortSwigger URL validation bypass cheat sheet has better mutation strategies than a grammar for URLs would. Ad hoc mutations do come with the trade-off that they are less likely to find vulnerabilities that they do not specifically test for. Maybe adding more random mutations would improve our fuzzer, which we also discuss in Section 10.2.

Next, we specify which properties hold for each parameter. These can be seen in Table 7.2. It is necessary to mutate parameters according to their expected properties, because otherwise, we would get a lot of false positives. For example, if we specify the `code` (authorization code) parameter to be session-specific while it is not required to be, the fuzzer would incorrectly detect this as a bug[8]. Note that we do not test all OIDC parameters due to time constraints.

We randomly select which parameter to fuzz, weighted by how many mutation strategies apply to it.

[8]This is exactly what we accidentally did and what let us to investigate browser-swapping attacks, see Section 9.4

Parameter	Properties
State	Mandatory, used once, user-specific, session-specific, flow-specific
Code	Mandatory, used once
Client identifier	Constant, mandatory, flow-specific
Redirection URI	Constant, URL, flow-specific
Response type	Constant, mandatory, flow-specific
Scope	Constant, mandatory
Request	Request object
Nonce	Used once, mandatory, user-specific, session-specific
Response mode	Constant, mandatory
Token type	Constant, mandatory
Expires in	Constant, mandatory
Access token	Used once, user-specific, session-specific, token
ID token	Used once, user-specific, session-specific, token

Table 7.2: Our defined properties of the OIDC parameters.

Note that this fuzzing strategy might not be sufficient to find all bugs in an implementation. There are, for example, more mutation strategies we could add that test for different vulnerabilities. We discuss this in more detail in Section 10.2.

Let us go over an example of a fuzzed request. Let's say that we are in the state after we have just initiated the authorization code flow, and are fuzzing an authorization request. We look at a fuzzed `authserver_authorize_invalid` request. Our fuzzer selects a random parameter to fuzz, weighted by the amount of mutation strategies each parameter has. The fuzzer picks to mutate the `redirect_uri` parameter according to the URL property. This mutation strategy tries a random bypass from the PortSwigger URL validation bypass cheat sheet, and modifies the stored `redirect_uri` to `://evil.com%09https%3A%2F%2Ffuzz1.incubator.geant.org%2Fsimplesaml-rp%2Fmodule.php%2Fauthoauth2%2Flinkback`. The fuzzed request becomes:

```
GET /simplesaml-op/module.php/oidc/authorization?client_id=
    _f7ce2e5905627dc39882b5b27a45041c0273b8528e&response_type=
    code&scope=openid%20profile&redirect_uri=://evil.com%09https
    %3A%2F%2Ffuzz1.incubator.geant.org%2Fsimplesaml-rp%2Fmodule.
    php%2Fauthoauth2%2Flinkback HTTP/1.1
```

The OP responds with a 400 Bad Request status code. This is categorized by the abstract output mapper to an "Error" response, which is what is to

be expected. This is not a deviation from our learned model. No bug is reported, and we continue fuzzing other requests and states.

7.4.1 State-by-state: improvement upon learning-based fuzzing

Our fuzzer supports two different fuzzing methods:

- We can fuzz by taking random walks through the state machine. This is similar to the learning-based fuzzing approach from Aichernig et al. [1], we fuzz with the State Prefix equivalence oracle[9], walking through states until a counter example to the learned model is found, i.e. when a fuzzed input does not return an error.

- We can fuzz state-by-state, our own method which is the default. This method iterates over all states, and fuzzes each state a set number of times per letter in the fuzzing mapper (`client_callback_invalid`, `client_callback_implicit_invalid` and `authserver_authorize_invalid`). Concretely, the fuzzer executes the prefix to reach the target state, and then sends the fuzzed message. Because fuzzed messages cause errors that almost never change the state, we can almost always keep fuzzing without having to walk back to our target state. We report a bug in case we receive a different output from the learned model than we expected.

Initially, our fuzzer only supported the random walks method as we closely followed Aichernig et al. [1]. Our state-by-state method was a later addition, after which we kept the random walks method for completeness and comparison.

Our state-by-state fuzzing method is significantly faster. This is because only a few messages, the ones that are expected to trigger an error (`client_callback_invalid`, `client_callback_implicit_invalid` and `authserver_authorize_invalid`), are fuzzed. With random walks, non-fuzzed messages can be selected randomly, while with the state-by-state method, non-fuzzed messages are only sent to reach a target state. This causes fuzzed messages to be sent a lot more often in the state-by-state method. However, if a fuzzed input triggers a bug that would only appear after a few other steps in the learned model, this bug would be missed by this method. We still believe[10] that our state-by-state fuzzing method is

[9] We use this oracle as it guarantees to explore each state and avoid excessive testing of initial states. More information can be found in the AALpy source code, we call the `find_cex` function: `https://github.com/DES-Lab/AALpy/blob/8d91b988e14c84db0c87ea98d9b22c0102743e1b/aalpy/oracles/StatePrefixEqOracle.py#L38`

[10] Note that this is not an academic claim, but rather an impression from our work. Comparing our state-by-state method with other fuzzing methods is interesting future work, see Section 10.2.

superior for our OIDC use case, as it is multitudes faster, and because all vulnerabilities that might be triggered by our mutations from Section 7.4 should cause a deviation from the learned state machine immediately.

7.5 Other features

The fuzzer includes some other features, mostly for convenience.

- It allows you to save a learned model to a `.dot` file
- It allows you to load a previously learned model from a `.dot` file
- It shows a visualization of the learned state machine, which is useful for manual inspection and comparison
- It allows you to specify a proxy server, which is very useful for debugging
- It appends a "FUZZED" header to fuzzed requests, showing which parameter was fuzzed using which strategy
- You can specify which parameters should be fuzzed and with which strategies from the command line, which is useful if you only want to fuzz with specific strategies

Chapter 8

Results of fuzzing

This chapter discusses the results found using the fuzzing setup from Chapter 7. We first show which vulnerabilities in our OAuth CTF from Chapter 5 our fuzzer was able to find in Section 8.1. Then we continue with the results from fuzzing the SUTs from Section 7.2. In Section 8.2, we discuss the results of the *learning phase*, as explained in Section 7.3, by looking into differences found in the state machines of different implementations. Then in Section 8.3, we go over the results of the *fuzzing phase*, as explained in Section 7.4, where we investigate why our fuzzer marked a feature as a bug.

Note that we can not confirm that these implementations conform to the OIDC specification [37], as this was not our goal: with fuzzing we tried to find bugs, not test conformance. An audit tool or the OpenID Conformance Suite [29] as described in Section 10.4 might be a better solution for this than fuzzing.

8.1 Fuzzing OAuth CTF

We ran our fuzzer on our OAuth CTF from Chapter 5, an intentionally vulnerable OAuth 2.0 implementation. Our fuzzer was able to successfully find the redirection URI validation bugs in the first two challenges: "Basic redirect_uri" from Section 5.1.1 and "Validation bypass" from Section 5.1.2. These challenges are the same except for an additional validation step in "Validation bypass", which is why they have the same state machine, shown in Figure 8.1. We found no security issues directly from this state machine, although it interestingly differs from the state machines we discuss in Section 8.2 (Figure 8.2).

Our fuzzer even found some additional, unintended bugs in these challenges:

- The scope value is allowed to be anything. This was not a security issue because these two challenges did not use the scope in any way.

In	Meaning	Comment
S	Start: `client_sso_login`	Starts the OIDC authorization code flow
SI	Start implicit: `client_sso_login_implicit`	Start the OIDC implicit flow
A	Authorize: `authserver_authorize`	Hits the authorization endpoint at the OP with the stored authorization code flow parameters
AI	Authorize implicit: `authserver_authorize_implicit`	Hit the authorization endpoint at the OP with the stored implicit flow parameters
L	Login: `authserver_login`	Logs in at OP with credentials. Note that SUT 3 does not have this: here, we log in automatically with HTTP Basic Auth.
C	Callback: `client_callback`	Hits the redirection endpoint at the RP with the stored authorization code flow parameters
CI	Callback implicit: `client_callback_implicit`	Hits the redirection endpoint at the RP with the stored implicit flow parameters

Out	Meaning	Comment
A	Authorize	Redirect to authorization endpoint at OP
L	Login	Redirect to login page at OP
C	Callback	Redirect to the redirection endpoint at the RP
S	Start	Redirect to the starting page of the authorization code flow
SI	Start implicit	Redirect to the starting page of the implicit flow
E	Error	Some kind of error was triggered

Table 8.1: Legend of input and output letters from Figures 8.1, 8.2 and 8.3

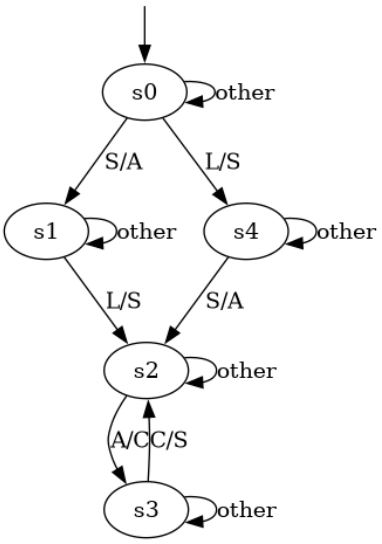


Figure 8.1: State machine of OAuth CTF challenges "Basic redirect_uri" (Section 5.1.1) and "Validation bypass" (5.1.2). See Table 8.1 for the meaning of each letter.

- The state value can be omitted at the redirection URI (`/callback`) if the user does not have a session. This allows for a Login CSRF attack as described in Section 4.2.1.2. This is caused by the following comparison:

  ```
  if state != session.get('state'):
      ...
      return "Invalid state", 400
  ```

  When the `state` value does not exist in the session, `session.get('state')` returns `None`, the same as the omitted `state` parameter.

- Sending a request to the redirection URI (`/callback`) with an invalid code still returns a "Login successful!" message, although a valid session is not created.

It seems that we are even better at creating a vulnerable application than we thought...

We did not try our fuzzer on the other three challenges, because we know from the design of these challenges and our fuzzer that the fuzzer will not be able to find these vulnerabilities:

- "Regex & Headers" from Section 5.1.3 requires the fuzzer to inject the

Location header in the redirection URI, which is not in our ad hoc mutations[1].

- "Basic scope" from Section 5.1.4 requires us to know the list of available scopes during fuzzing. This may be added to the fuzzer in future work as described in Section 6.2.
- "Error" from Section 5.1.5 requires the fuzzer to detect an XSS vulnerability, which is out of scope for our fuzzer, as described in Section 6.2

8.2 Results of the learning phase

In the learning phase of fuzzing (Section 7.3), we learned the state machines of the systems under test (SUTs). By comparing these state machines, we found a difference between the implementations.
We can compare the following two state machines:

- SUT 1, Figure 8.2a. OP: SimpleSAMLphp OIDC module. RP: SimpleSAMLphp AuthOAuth2 module.
- SUT 3, Figure 8.2b. OP: Shibboleth OIDC module. RP: Apache mod_auth_openidc module.

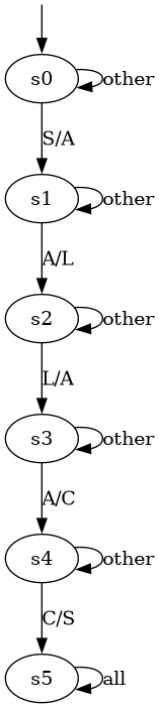
For the meaning of each letter in the state machines, see Table 8.1. As mention in Section 7.2, we automatically login with HTTP Basic Auth in SUT 3, which is why there is no login transition in Figure 8.2b.

Apart from this, these state machines are still clearly different. SUT 1 has a very linear progression, while SUT 3 includes an "error state": s4 is a state that does not lead to the end of the flow without returning back to s1 by sending the initial message.

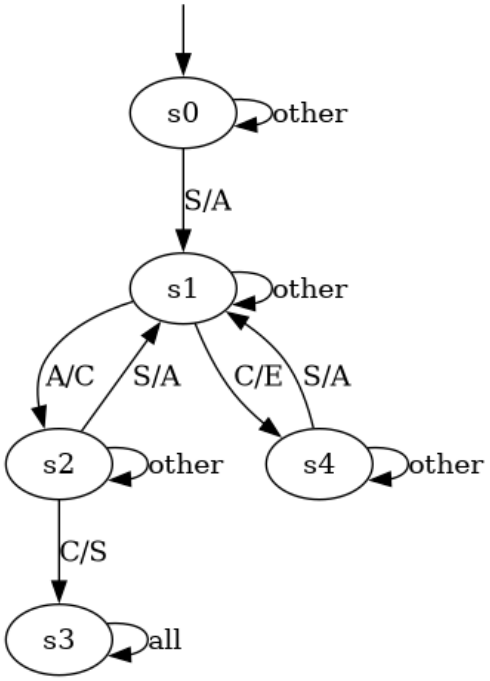
This error state is reached as follows (see Figure 8.2b):

1. We start in s0, the initial state.
2. We start the OIDC flow with the `client_sso_login` input (letter S). We get redirected to the authorization endpoint (letter A).
3. We hit the redirection URI with the stored parameters with the `client_callback` input (input letter C). There is no valid authorization code in this part of the protocol yet, so we get an error (output letter E). We now reach the error state s4. In SUT 1, we would have remained in the same state (s1 in Figure 8.2a).

[1]Note that a random mutation would also be unlikely to find this vulnerability, as e.g. a URL grammar would be unlikely to include the Location header



(a) SUT 1. OP: SimpleSAMLphp OIDC. RP: SimpleSAMLphp AuthOAuth2.



(b) SUT 3. OP: Shibboleth OIDC. RP: Apache mod_auth_openidc.

Figure 8.2: Simplified state machines of SUT 1 and SUT 3. Simplified inputs and outputs for clarity, see Table 8.1 for the meaning of each letter.

4. From this error state, we need to restart the OIDC flow to get back to s1 with the `client_sso_login` input (letter S).

We have investigated this difference and found the root cause in the RP implementations. Apache mod_auth_openidc, the RP of SUT 3, invalidates the state parameter on an invalid callback to the redirection URI, while SimpleSAMLphp AuthOAuth2, the RP of SUT 1, does not. Invalidating the state on an invalid callback is recommended in RFC 9700 "Best Current Practice for OAuth 2.0 Security" [24]: "The state value SHOULD be invalidated by the client after its first use at the redirection endpoint". This eliminates the risk of the state parameter leaking at the redirection endpoint. Because SimpleSAMLphp AuthOAuth2 does not follow this best practice on error, it is vulnerable to the State Leak attack from Fett et al. [9]. For the state leak attack to work, the redirection URI page at SimpleSAMLphp AuthOAuth2 would have to include a link to the attacker's website or some resource located at the attacker's website. When this link is clicked or this resource is loaded, the attacker receives the state value via

the victim's browser's "Referrer" header. The attacker can then perform a (login) CSRF attack on the victim as described in Section 4.2.1.2.

Another difference between the state machines is that SUT 3 has transitions back to s1 when a user revisits the starting page, while SUT 1 does not. This does not impact security in any way.

Interestingly, these two state machines also differ from the state machine of our OAuth CTF in Figure 8.1. It may be the case that there are multiple ways to implement the specification while still conforming to it.

We also looked at interaction between the authorization code and implicit flows in SUT 2. The learned state machine can be seen in Figure 8.3. For the meaning of each letter in the state machine, see Table 8.1.

This state machine might look interesting, but in reality, it is not. The authorization code and implicit flows are completely independent from each other and do not interact. The left-most path (s0 to s14 to s21) starts with the authorization code flow (s0 to s14, colored blue), then the implicit flow (s14 to s21, colored red), and finishes in the final state s21. The right-most path (s0 to s15 to s21) is the opposite: it starts with the implicit flow (s0 to s15, colored red), then the authorization code flow (s15 to s21, colored blue), and finishes in the final state s21. All states in between are basically different combinations of the two flows (it is a cartesian product): e.g. one input for the authorization code flow, then one for the implicit flow, etc. Apart from the self-looping transitions and the cycles between s16-s19, s12-s17, s9-s13, and s8-s11, the state machine is a directed acyclic graph (DAG), meaning that there is no way to return back to previous states. There are no interesting cross-flow interactions.

Note that Figure 8.3 of SUT 2 does not contain the error state (s4) of SUT 3 in Figure 8.2b. We suspect this is the case because mod_auth_openidc gives an error on a different parameter than the authorization code in the implicit flow, for example the invalid `access_token`, `token_type`, `expires_in` or `id_token`. An error in these parameters does not cause the session to be invalidated, but returns a 400 Bad Request status code instead.

8.3 Results of the fuzzing phase

In the fuzzing phase, we fuzz the learned state machines using mutations described in Section 7.4. We did not find any bugs in the OIDC implementations during fuzzing, only bugs in the fuzzer itself.

We did find one interesting feature of Apache `mod_auth_openidc` during fuzzing: browser-back detection. This feature detects whether the redirection URI was hit before with the same state parameter and session, e.g. by hitting the browser-back button. If it was, the end-user gets redirected to the original URL they tried to access. This feature is not mentioned in any

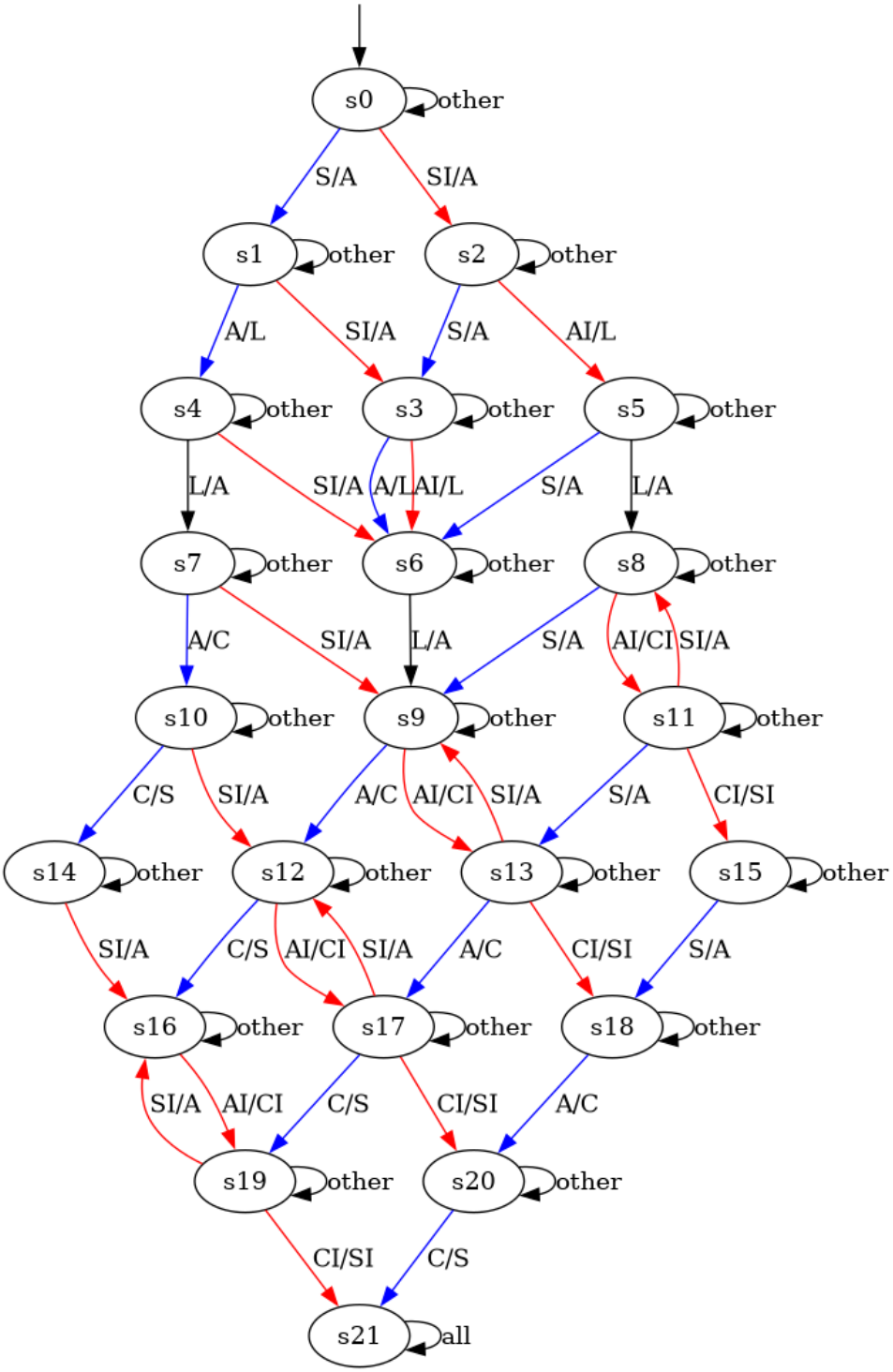


Figure 8.3: Combined state machine of authorization code and implicit flows of SUT 2. Authorization code flow transitions are colored blue, implicit flow transitions red. For the meaning of each letter, see Table 8.1.

OIDC or OAuth specification, it seems to be specific to Apache `mod_auth_openidc`. The relevant code is shown in Figure 8.4. Our fuzzer incorrectly reported this feature as a bug, because it allowed the authorization code to be invalid, as long as the `state` matched and the authentication flow was already completed. This is not a security issue, as in the case of browser-back detection, an authenticated session is already established.

```
/* handle the browser back on an authorization response */
static apr_byte_t oidc_response_browser_back(request_rec *r,
    const char *r_state, const oidc_session_t *session) {
        const char *s_state = NULL;
        const char *o_url = NULL;

        /*  see if we have an existing session and browser-back
     was used */
        if (session->remote_user == NULL)
                /* no session was established yet */
                return FALSE;

        s_state = oidc_session_get_request_state(r, session);
        if ((r_state == NULL) || (s_state == NULL) || (
    _oidc_strcmp(r_state, s_state) != 0))
                /*  state does not match with the state that
    was used to create  the session earlier, no replay is going
                 * on here */
                return FALSE;

        /* get the URL that was originally accessed by the user
     */
        o_url = oidc_session_get_original_url(r, session);
        /* log the browser back event detection */
        oidc_warn(r, "browser back detected, redirecting to
    original URL: %s", o_url);
        /* go back to the URL that he originally tried to
    access */
        oidc_http_hdr_out_location_set(r, o_url);

        /* signal that a browser back event was detected indeed
     and we handled this here */
        return TRUE;
}
```

Figure 8.4: Code of the browser-back behavior in Apache mod_auth_openidc. Taken from https://github.com/OpenIDC/mod_auth_openidc/blob/4c8077051b85066d4901cf1e7208cd9aae22f5bf/src/handle/response.c#L84

Chapter 9

Browser-swapping attacks

During our fuzzing, we came across browser-swapping attacks. In this chapter, we describe what browser-swapping attacks are in Section 9.1. We show a demo of a browser-swapping attack in Section 9.2 and discuss mitigation strategies in Section 9.3. We describe how to protect against browser-swapping attacks using the form post response mode in the SimpleSAMLphp and Shibboleth OIDC modules (OpenID Providers) in Section 9.3.1. One of our contributions is support for the form post response mode in the SimpleSAMLphp OIDC module. As a side note, we mention how fuzzing led us to browser-swapping attacks in Section 9.4.

This chapter assumes technical knowledge of the OAuth 2.0 protocol. For more information, see Section 2.2. We use OAuth 2.0 terminology: we refer to the authorization server and the client, but you can respectively read OP (OpenID Provider) and RP (Relying Party) for OIDC.

9.1 Background on browser-swapping attacks

A browser-swapping attack, as described by Jonas Primbs [34][1], is an attack on the OAuth 2.0 (and also OpenID Connect) protocol in which a victim's session is compromised via a stolen or leaked authorization code. The attacker generally gains access to this authorization code via social engineering (e.g. phishing), but might also through other ways, by e.g. viewing system logs. As it is an attack on the OAuth 2.0 protocol itself, the attack works on *all* OAuth 2.0 and OIDC implementations by default, unless they implement specific additional protections. The attack abuses two main properties of the OAuth 2.0 protocol:

- If a victim uses a `state` parameter belonging to an attacker's session

[1]The attack was first discovered by the OpenID FAPI Working Group in 2022 [38], but was brought under the attention in 2025 by Jonas Primbs. The FAPI working group excluded browser-swapping attacks from their threat model by reducing their attacker model [11].

in the authorization code flow, the victim gets an error, but also a *valid* authorization code.

- This valid authorization code is returned in the URL. The URL is now a target for social engineering (e.g. phishing), which a victim might not expect ("why would a URL be sensitive?"[2]). Also, the URL might leak through e.g. system logs.

Browser-swapping attacks require the authorization code in the URL to be leaked, for example via phishing. This is more feasible than it might initially seem. For example, the ConsentFix attack on Azure [16] was built around this exact attack vector: attackers tricked the users into sharing the final URL that contained the unconsumed[3] authorization code (Figure 9.1).

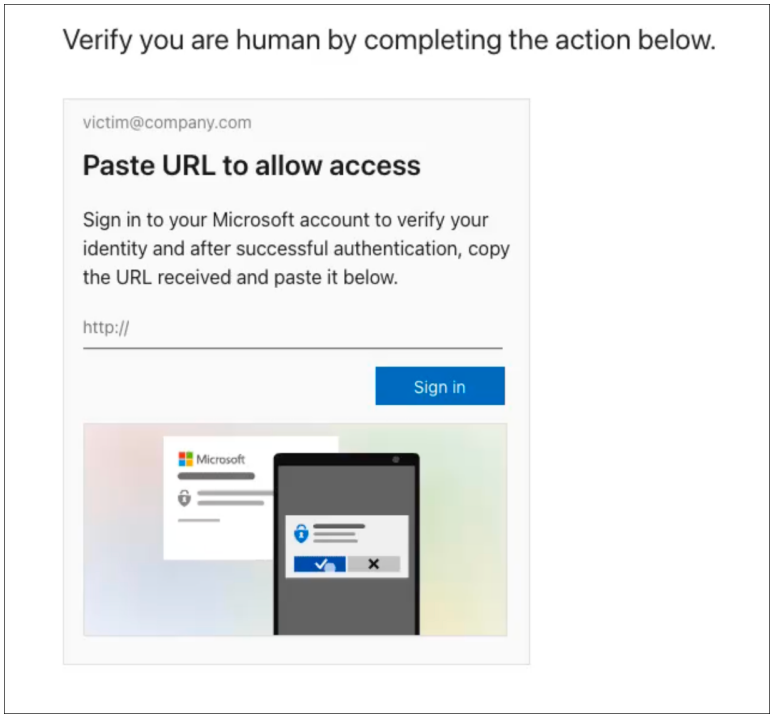


Figure 9.1: A ConsentFix phishing page, taken from [16]

The attack is shown in detail in Figures 9.2 and 9.3. The main steps are:

1. The attacker obtains a valid Authorization Request URL with a `state` value belonging to their session, e.g. `https://as.example.com/authorize?response_type=code&redirect_uri=https%3A%2F%2Fclient.example.`

[2]There are indeed scenario's in which a URL is sensitive, for example, a file sharing link, but the victim might not expect that this is the case for an OAuth 2.0 or OIDC flow.

[3]Meaning: unused, not exchanged for an access token. An authorization code is one-time use.

`com%2Fcallback&client_id=123&state=zyx` (steps 1 & 2 in Figure 9.2)

2. The victim clicks this link and logs in as usual. Note that this is a legitimate link at a legitimate authorization server. The attacker tricks the victim via social engineering, e.g. phishing (steps 3 & 4 in Figure 9.2)

3. The victim gets an error after logging in due to a `state` mismatch, as the `state` parameter belongs to the attacker's session. The URL contains the attacker's `state` and a valid authorization code belonging to the victim, e.g. `https://client.example.com/callback?code=ABC&state=zyx` (steps 6 & 7 in Figure 9.2)

4. The attacker obtains this URL, via e.g. social engineering or viewing system logs.

5. The attacker opens the URL in their own browser and gets logged in as the victim (steps 6 & 7 in Figure 9.3)

Security analyses of OpenID Connect and OAuth 2.0 have "missed" browser-swapping attacks [9] [10]. This is because the ways to leak the authorization code that Primbs describes, e.g. social engineering, are not in the assumed capabilities of the attacker in these analyses. We do still believe that this is a legitimate risk because of the similarity to the ConsentFix attack [16].

9.2 Demo

We created a demo of the browser-swapping attack, based on a GÉANT login page as shown in Figure 9.4. In the T&I Incubator, we deemed this a valid attack worth protecting against. We did this by implementing the form post response mode as explained in Section 9.3.1.

9.3 Mitigations

Jonas Primbs describes multiple mitigation strategies for browser-swapping attacks [34] [35].

For the authorization server:

1. Do not share the authorization code in a query parameter. This can be configured with alternative response modes, mainly the form post [20]

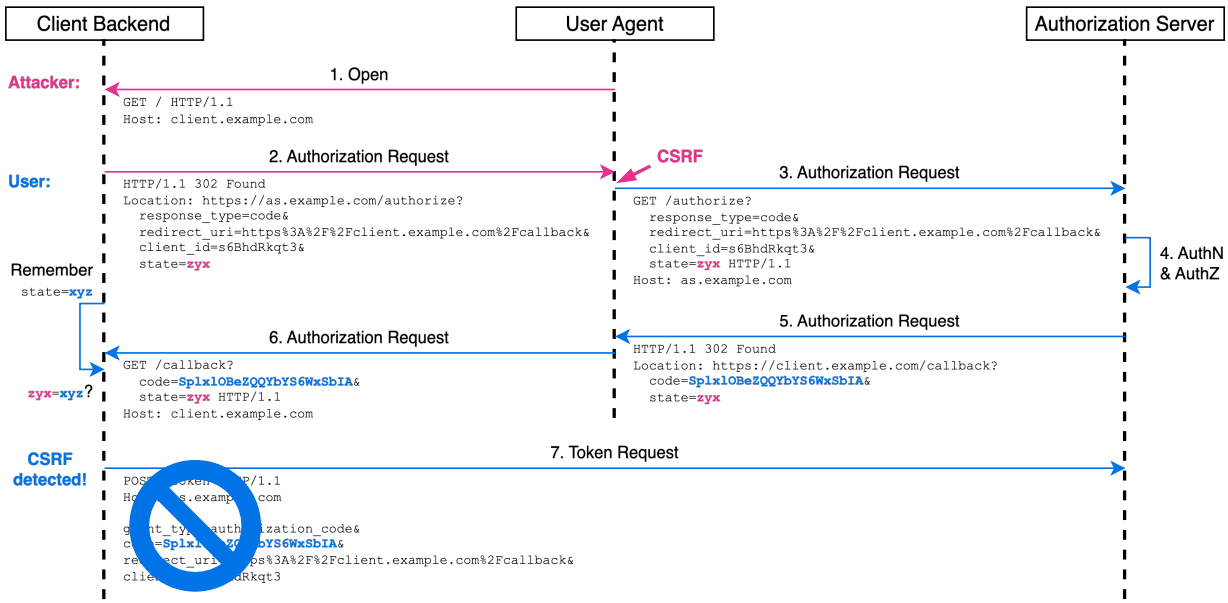


Figure 9.2: First stage: end-user uses attacker's `state`, gets an error but with a valid authorization code. Note that the Client and Authorization Server are the RP and OP in OIDC respectively. Taken from [34].

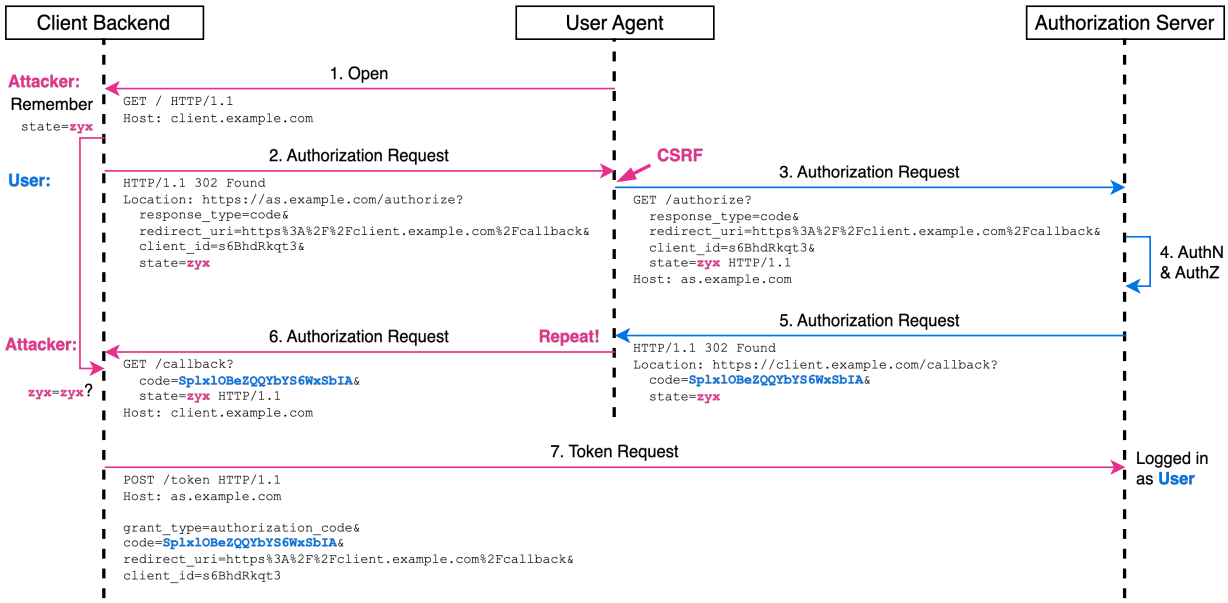
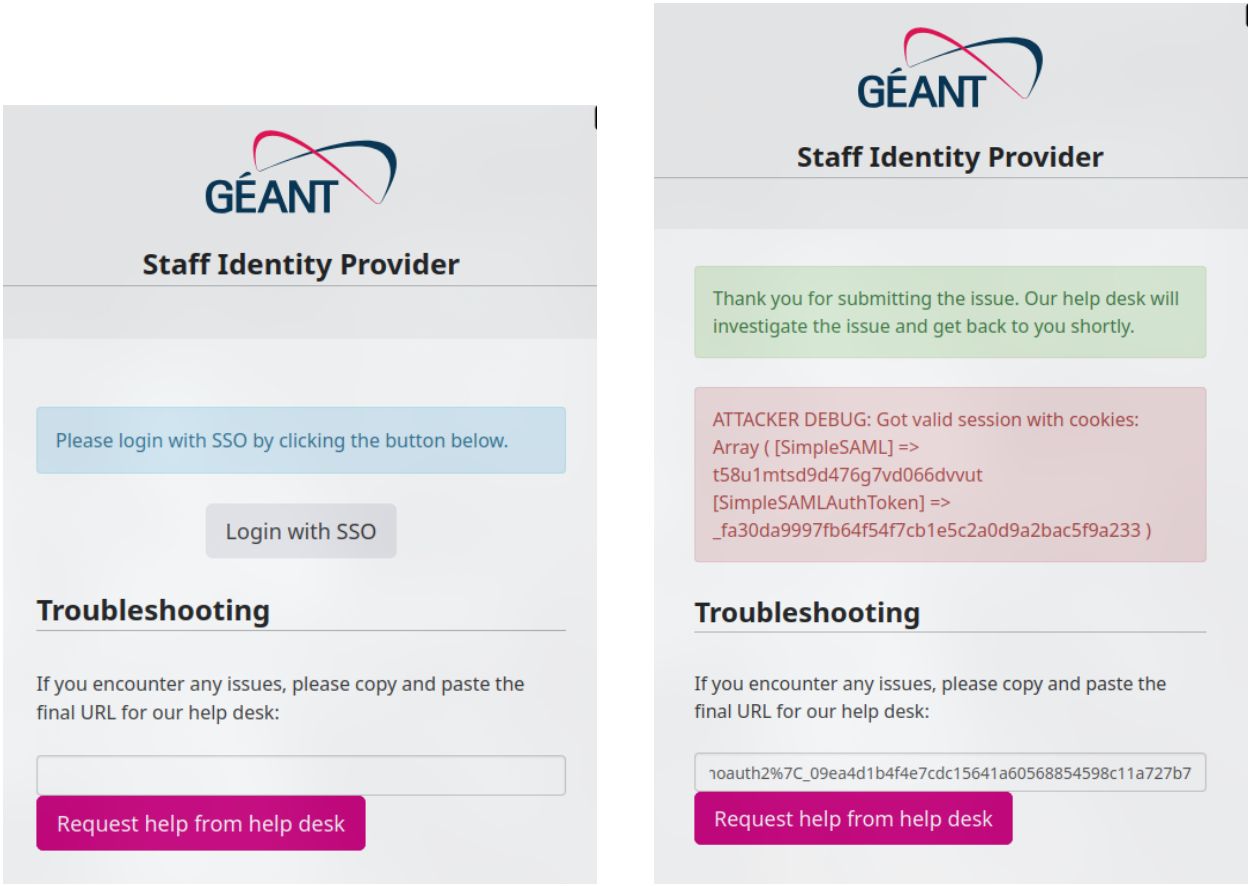


Figure 9.3: Third stage: attacker replays the victim's stolen URL (step 6) and gets authorized as the victim. Note that the Client and Authorization Server are the RP and OP in OIDC respectively. Taken from [34].



(a) The victim is presented with a normal-looking login page, and logs in via OIDC.

(b) The victim pastes the final URL. The attacker obtains a session as the victim.

Figure 9.4: Browser-swapping attack demo.

response mode[4]. This response mode makes the authorization server respond with an auto-submitting HTML form in step 5 in Figures 9.2 and 9.3. By doing this, the authorization code and `state` parameters will be sent with a POST request to the client in step 5 in Figures 9.2 and 9.3. This means that the URL will not contain these parameters, which mitigates the social engineering risk.

An attacker can specify the preferred response mode with e.g. `response_mode=query` in the authorization request, which could bypass

[4]Jonas Primbs also mentions the fragment response_mode as an option, which returns the parameters in a URL fragment (e.g. `https://client.example.com/callback#code=ABC&state=zyx`) instead. As the fragment component is never sent to the client (fragment components are kept in the User-Agent), this prevents leakage through server system logs. However, it does not prevent leakage through the URL by social engineering, which is in our opinion the main threat.

the form post mitigation. Therefore, the authorization server needs to be able to *enforce* the form post response mode as well. Not all clients support the form post response mode, so this needs to be configurable on a client basis, or enforced by the specification in a future version of OAuth.

2. Ask the user to consent before returning the authorization code, so that the user knows what access they grant.

3. Reduce the authorization code validity as much as possible, e.g. to 5 seconds, such that the victim can not get tricked into sharing the authorization code in such a short time frame. This is the easiest mitigation, but might reduce usability for users with e.g. bad internet.

For the client:

1. Invalidate the authorization code on error by sending it to the authorization server. There is not really a standardized method for this. One possible method is to send the authorization code to the authorization server with a wrong `redirect_uri` parameter, but not all authorization servers invalidate the authorization code when this happens.

2. On error, remove the authorization code from the URL bar, by e.g. redirecting to an error page.

Note that browser-swapping attacks are not mitigated by PKCE (Proof Key for Code Exchange). PKCE is an optional OAuth 2.0 extension that is meant to prevent authorization code interception attacks [39]. PKCE is mandated in OAuth 2.1 [14]. It specifies a way for a client to bind the authorization code to a specific authorization request. However, this protects the *attacker* instead of the victim, as the attacker initiates the authorization code flow at the start of the attack.

Also note that browser-swapping attacks are not mitigated by DPoP (Demonstrating Proof of Possession). DPoP is an optional OAuth 2.0 extension that binds access tokens to the client [8]. DPoP also defines the `dpop_jkt` parameter, which can bind the authorization code to an authorization request if implemented as such. In this case, it has the same effect as PKCE: it binds the authorization code to the attacker's authorization request, and therefore, protects the attacker.

9.3.1 Form post response mode in the SimpleSAMLphp and Shibboleth OIDC modules

One mitigation to browser-swapping attacks that an OpenID Provider can provide is support for the `form_post` response mode, as described in Section 9.3. With the form post response mode, the URL does not contain the

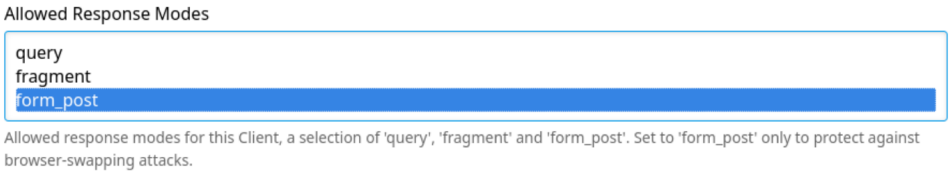


Figure 9.5: How to enforce the form post response mode in the SimpleSAMLphp OIDC module.

authorization code, which makes the attack infeasible. The response mode can, however, be specified by the attacker. Therefore, the authorization server should also support a configuration option to enforce the form post response mode.

To allow the SimpleSAMLphp OIDC module to protect against browser-swapping attacks, we contributed an implementation of the form post response mode and a configuration option to enforce this. This required us to essentially implement part of the specification "OAuth 2.0 Multiple Response Type Encoding Practices" [6] and the entire "OAuth 2.0 Form Post Response Mode" [20] specification.

The pull request can be found on GitHub at `https://github.com/simplesamlphp/simplesamlphp-module-oidc/pull/337`. It required quite some modifications and resulted in over a 1000 changed lines of code. This mitigation (and feature) will be released in version 7.

To configure your SimpleSAMLphp OIDC OP to enforce the form post response mode, go to the Admin interface, click on "OIDC OP", go to "Client Registry" and add or edit a client. Then, select "form_post" as the only allowed response mode as shown in Figure 9.5.

The Shibboleth OIDC module already supported enforcement of the form post response mode. To configure this, change your `conf/relying-party.xml` file to include the code snippet shown in Figure 9.6.

9.4 How fuzzing led us to browser-swapping

Our fuzzer tests specific properties of parameters during fuzzing, as described in Section 7.4. We made some mistakes when configuring the security properties of the parameters during development. One mistake we made was to configure the `code` parameter, the authorization code, to be session-specific. It turns out that the `code` parameter is not required to be session-specific by the OIDC and OAuth 2.0 specifications.

This led us to investigate attacks in which the authorization code is leaked or stolen. One of these types of attacks is a browser-swapping attack.

```
<bean id="shibboleth.DefaultRelyingParty" parent="RelyingParty">
    ...SNIP...
    <bean parent="OIDC.SSO">
        <property name="responseModes">
            <set>
                <value>form_post</value>
            </set>
        </property>
    </bean>
    ...SNIP...
</bean>
```

Figure 9.6: How to enforce the form post response mode in the Shibboleth OIDC module. Put this configuration snippet in your `conf/relying-party.xml` file.

Chapter 10

Future Work

During this thesis, we gained lots of interesting research ideas, but had to constrain ourselves due to time. This chapter presents those interesting ideas that are in our opinion great for future research, while therefore also defining the limitations of our work. We separate future work in categories: different targets in Section 10.1, improvements to our fuzzer in Section 10.2, different fuzzing techniques in Section 10.3, alternatives to fuzzing in Section 10.4, future work on our OAuth CTF in Section 10.5, and finally future work in browser-swapping attacks in Section 10.6.

10.1 Different targets

We chose to fuzz OIDC implementations using learning-based fuzzing as described in Section 6.3. Learning-based fuzzing could similarly be applied to other SSO protocols like SAML or LDAP, or non-SSO protocols.

It would also be interesting to see how our fuzzer performs on different OIDC implementations, like Keycloak[1].

10.2 Improvements to our fuzzer

Our fuzzer has some limitations due to time constraints. It would be interesting to see if our fuzzer could find more bugs when improvements are made to it. For example:

- Add more mutations types than the ones described in Section 7.4, for example by injecting JWTs in the ID token via the jwk parameter as described in Section 4.2.2.3. Chapter 4 could serve as a reference for this, the lists of requirements as well as the vulnerabilities. In Sections 4.2.1 and 4.2.2, we list some vulnerabilities and whether our fuzzer would be able to find them.

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

- Fuzz more parameters, like the `id_token_hint` as briefly mentioned in Section 2.4.2.1.
- Fuzz more parts of the OIDC protocol, like the hybrid flow as described in Section 2.4.2.3, but also e.g. PKCE[2]. Our fuzzer can only fuzz the authorization code and implicit flows.

The speed of our fuzzer can also likely be improved, for example with multi-threading.

Our conjecture in Section 7.4 was that ad hoc mutations perform better than random mutations with e.g. a grammar, but maybe this conjecture is incorrect. One could try to improve our fuzzer by adding grammar-based fuzzers for e.g. JWTs and URLs. When to use ad hoc mutations over random mutations is an interesting subject to further explore.

We believe that our state-by-state fuzzing approach is an improvement upon the learning-based fuzzing approach from Aichernig et al. [1] as discussed in Section 7.4.1. It would be interesting to do a proper academic comparison between our state-by-state method and the random walks method from learning-based fuzzing for different use cases.

10.3 Different fuzzing technique

We chose learning-based fuzzing as our fuzzing technique in Section 6.3. In hindsight, this might not have been the correct choice. We chose learning-based fuzzing because it is a stateful and black-box approach, but maybe gray-box or white-box would have been better. For example, coverage-guided fuzzing might be more effective, although this is difficult to apply to different implementations in different programming languages. Maybe a combination of coverage-guided fuzzing with our learning-based fuzzing approach is possible. A white-box fuzzing approach might reach more code branches by using e.g. symbolic execution, or with just string literals from the source code.

With learning-based fuzzing, we use active automata learning to obtain the state machines of the implementations. We could have also created a state machine according to the specification ourselves, and see if the other implementations conform to it. However, implementations might be able to have different state machines, while still conforming to the specification, as the specification does not define a state machine. This could be an interesting topic to further explore.

One could also separate functions of an implementation like Apache mod_auth_openidc, which is written in C, and fuzz these with a coverage-guided fuzzer like AFL++. Or even look at individual parsers of e.g. JWTs and URLs that are used in OIDC implementations, and fuzz these separately.

[2]PKCE is shortly described in Section 9.3

The library that we used for learning-based fuzzing, AALpy [28], also supports passive-learning from traces. It could be interesting to experiment with passive learning instead of our active learning approach from Section 7.3.

Future research could also use a different attacker model than the one we chose in Section 7.1. For example, maybe a fuzzer that assumes the role of a malicious OP might find security issues in OIDC implementations, like the Identity Provider Confusion attack described by Mainka et al. [25].

10.4 Alternatives to fuzzing

We chose to use fuzzing to test the security of OIDC implementations. There are lots of alternatives to fuzzing that may be used for the same purpose.

Fuzzing is a form of automated testing. Manual testing might outperform automated testing in certain cases. It would be a nice addition to write an extensive manual security testing guide for OIDC, for example by expanding on the OWASP WSTG[3]. It is also possible to compare automated testing with manual testing for e.g. OIDC.

Fuzzing is also a form of dynamic application security testing (DAST). Static application security testing (SAST) does not dynamically interact with an application and examines the source code instead. This might be more effective to uncover certain types of bugs like weak secrets, for example with Semgrep[4] or CodeQL[5]. Our list of requirements in Appendices A.1 and A.2 might be convertible to specific SAST test cases.

One could also use an "audit" tool based on our list of requirements in Appendices A.1 and A.2, although this might be similar to the OpenID Conformance Suite [29]. This tool could dynamically test an implementation by sending requests, or statically by examining the source code. It might also be able to test if the OIDC implementation conforms to the specification, which could be another interesting result on its own.

LLM-based vulnerability research has gained a lot of popularity while this thesis was written. LLMs are quite effective at finding vulnerabilities, already without extensive prompt-engineering [4]. It might also be possible to use LLMs to find vulnerabilities in OIDC implementations, especially open-source implementations of which the code can be given to the LLM. This might also be improved with a specific prompt or harness. Chapter 4 could be used as a starting point for the prompt, for example to see if the source code adheres to the lists of security requirements.

[3] `https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/05-Testing_for_OAuth_Weaknesses`

[4] `https://semgrep.dev/`

[5] `https://codeql.github.com/`

We did not discuss CVEs related to OIDC in this thesis. Investigating CVEs, building testcases for these CVEs, and then applying these tests to other implementations might help to uncover unknown vulnerabilities. By investigating CVEs, one might also be able to learn which vulnerabilities are most common, and use this to focus their effort. Researching which OIDC vulnerabilities are most common is an interesting research topic on its own.

10.5 OAuth CTF

We created an OAuth Capture The Flag in Chapter 5 which can be found on GitHub[6]. We believe that playing this CTF is a great way to learn about some OAuth 2.0 vulnerabilities. The CTF currently has 5 challenges. It could be improved by adding more challenges, e.g. by adding challenges based on OIDC vulnerabilities from Section 4.2.2. The installation process of the CTF is currently undocumented, it would be great if the CTF was easy to setup.

10.6 Browser-swapping attacks

We documented browser-swapping attacks as described by Jonas Primbs [34] in Chapter 9, and deemed them a valid risk worth protecting against due to their similarity to the ConsentFix attack [16]. We implemented a protection against browser-swapping attacks in the SimpleSAMLphp OIDC module in Section 9.3.1, but many OIDC implementations remain vulnerable by default. Auditing other popular OIDC implementations to see if they are vulnerable to browser-swapping attacks would be a great contribution. It may be possible to automate this auditing process.

[6] `https://github.com/Harm-r/oauth-ctf`

Chapter 11

Conclusions

In this chapter, we summarize the main conclusions of our research. We shortly list our contributions in Section 11.1 and reflect on our fuzzing results in Section 11.2.

11.1 Our contributions

We have created an **OAuth 2.0 Capture The Flag (CTF)** with 5 challenges in Chapter 5. CTFs are widely successful at introducing players to technologies, but also at motivating continual learning [26]. This also provided us with a great target for our fuzzer, and helped us to learn about OAuth vulnerabilities ourselves. The CTF can be found on GitHub[1].

Learning-based fuzzing by Aichernig et al. is a technique that combines automata learning (in the learning phase) with fuzzing (in the fuzzing phase) to test black-box systems [1]. **We improve upon learning-based fuzzing by fuzzing state-by-state**, which increases the speed of fuzzing, as described in Section 7.4.1. Our fuzzer can be found on GitHub[2]. We fuzzed four OIDC implementations: the SimpleSAMLphp OIDC, the SimpleSAMLphp AuthOAuth2, the Shibboleth OIDC and the Apache mod_auth_openidc modules. **We found that the SimpleSAMLphp AuthOAuth2 module does not follow the OAuth 2.0 best practices** [24] by not invalidating the state at the redirection endpoint on error and might therefore be vulnerable to a state leak attack as defined by Fett et al. [9] in some configurations. We also discovered the browser-back detection feature from the Apache mod_auth_openidc module, but we did not find any bugs here.

We believe that browser-swapping attacks are a legitimate threat due to their similarity to the ConsentFix attacks, as described in Chapter 9. We

[1] `https://github.com/Harm-r/oauth-ctf`
[2] `https://github.com/Harm-r/oidc-learning-based-fuzzing`

created a **demo of browser-swapping attacks** on the tested implementations with a phishing page, which showed that the attack is feasible. We implemented a **protection against browser-swapping attacks in the SimpleSAMLphp OIDC module** in Section 9.3.1, namely the form post response mode, for which the pull request can be found on GitHub[3]. This contribution required more work than we initially anticipated, and resulted in a pull request that changed over a 1000 lines of code.

11.2 Reflections on our fuzzing

We decided to use learning-based fuzzing as defined by Aichernig et al. [1] in Section 6.3, because it is a back-box fuzzing technique that allows us to fuzz message format as well as the sequence of messages, while it also defines a clear way to detect vulnerabilities. With learning-based fuzzing, we found some differences in the tested implementations in Chapter 8. Our main finding was that the SimpleSAMLphp AuthOAuth2 module does not adhere to the OAuth 2.0 best practice [24], as it does not invalidate the state on first use at the redirection endpoint, and might therefore be vulnerable to a state leak attack as defined by Fett et al. [9] in some configurations. Apart from this, we found *no security issues*. This begs the question: why did we not? We distinguish between two main possible causes.

First of all, the tested OIDC implementations might *just not contain more security issues*. We looked at the SimpleSAMLphp OIDC module, SimpleSAMLphp AuthOAuth2 module, Shibboleth OIDC module and Apache mod_auth_openidc, because they are widely used in the Trust & Identity community of GÉANT. This makes them interesting targets with high impact, but also suggests that they have already been tested and examined by others. The SimpleSAMLphp OIDC module runs SonarQube static analysis[4]. Apache mod_auth_openidc also does this[5], and even runs CodeQL static analysis[6]. It would be interesting to see how our fuzzer performs on different OIDC implementations, see Chapter 10.

Secondly, security issues might still exist in the tested implementations, but our fuzzer is unable to find them. This could have multiple causes:
A) Our test case generation may not be good enough:

[3] `https://github.com/simplesamlphp/simplesamlphp-module-oidc/pull/337`

[4] `https://github.com/simplesamlphp/simplesamlphp-module-oidc/blob/master/.github/workflows/sonar.yml`

[5] `https://github.com/OpenIDC/mod_auth_openidc/blob/master/.github/workflows/sonarqube.yml`

[6] `https://github.com/OpenIDC/mod_auth_openidc/blob/master/.github/workflows/codeql-analysis.yml`

- Security issues might exist in *untested functionality* of the implementations for which we do not generate test cases, e.g. in the PKCE[7] implementation.

- It may not be feasible to find existing security issues *with our specific fuzzing methodology*, i.e. black-box, learning-based fuzzing. It would be interesting to see if, for example, coverage-guided fuzzing would be able to find bugs, which we discuss in Chapter 10.

- Our *learning phase* from Section 7.3 is insufficient: we may not learn the state machine of the implementation correctly. For example, the state machine might be too abstract to capture small differences.

- Our *fuzzing phase* from Section 7.4 is insufficient: we may not mutate the tested parameters in a way that detects existing security issues. For example, we do not test for algorithm confusion in the ID token, as described in Section 4.2.2.3. This could be solved by adding more mutation strategies. It could also be the case that ad hoc mutation strategies for e.g. the ID token are not the correct approach to mutation strategies for fuzzing, but rather, e.g. a grammar should be used instead. Our conjecture in Section 7.4 was that ad hoc mutation strategies for certain parameter types would be best, as they are designed to test for specific vulnerabilities, but maybe this conjecture was incorrect.

B) Our detection of bugs may not be good enough:

- We may have missed security issues in the *learning phase* during our manual analysis.

- We may have missed security issues in the *fuzzing phase*. The learning-based fuzzing approach from Chapter 7 should detect any deviations from the learned state machine. However, a bug may not be visible in the HTTP response of the implementation, or if it is, it may get mapped to an incorrect abstract output. It may also not be feasible to find existing security issues *with fuzzing*. For example, an implementation may generate authorization codes in a predictable way, as described in Section 4.2.1.4. SAST tools would be able to find this, but fuzzing would not. Maybe SAST or manual testing would have found security issues.

We cannot determine the exact cause, but our believe is that the functionality *which we tested* of the implementations does not contain more security issues, i.e. the authorization code and implicit flows. Bugs might still reside in untested functionality.

[7]For a short description of PKCE, see Section 9.3. For the specification, see [39].

In hindsight, we do believe that it would have been better to use some form of coverage-guided fuzzing over our current learning-based fuzzing approach, or maybe a combination of the two (e.g. coverage-guided mutations in the fuzzing phase, detecting bugs using the learned state machine), as also mentioned in Section 10.3. However, we decided to use a black-box approach, as we wanted our fuzzer to work for multiple implementations in different programming languages, which we decided in Section 6.3. Coverage-guided fuzzing is difficult to setup for multiple implementations with different programming languages. Also, if we have issues with detecting bugs (B), coverage-guided fuzzing would not change anything about this.

On a positive note: we did find the best practice deviation in the SimpleSAMLphp AuthOAuth2 module. Also, our fuzzing research did lead us to browser-swapping attacks, which in turn led to a nice contribution to the SimpleSAMLphp OIDC module, which we are quite pleased with. In the end, this is just the nature of security research: sometimes you find many bugs, sometimes you don't, and sometimes it puts you on an unexpected path and you end up implementing a specification instead.

Bibliography

[1] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-Based Fuzzing of IoT Message Brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58, 2021. URL: https://doi.org/10.1109/ICST49551.2021.00017.

[2] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, 2008. URL: https://doi.org/10.1145/1455770.1455782.

[3] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997. URL: https://www.rfc-editor.org/info/rfc2119/.

[4] Nicholas Carlini. Black-hat LLMs. Conference talk at [un]prompted 2026, YouTube video, 2026. URL: https://www.youtube.com/watch?v=1sd26pWhfmg.

[5] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. *ACM Comput. Surv.*, 56(9), April 2024. URL: https://doi.org/10.1145/3648468.

[6] Breno de Medeiros, Marius Scurtescu, Paul Tarjan, and Michael B. Jones. OAuth 2.0 Multiple Response Type Encoding Practices, February 2014. Final specification, OpenID Foundation. URL: https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html.

[7] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-de-ruiter.pdf.

[8] Daniel Fett, Brian Campbell, John Bradley, Torsten Lodderstedt, Michael B. Jones, and David Waite. OAuth 2.0 Demonstrating Proof

of Possession (DPoP). RFC 9449, September 2023. URL: https://www.rfc-editor.org/info/rfc9449.

[9] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1204–1215, 2016. URL: https://doi.org/10.1145/2976749.2978385.

[10] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web SSO standard OpenID Connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202. IEEE, 2017. URL: https://doi.org/10.1109/CSF.2017.20.

[11] Daniel Fett, Dave Tonge, and Joseph Heenan. FAPI 2.0 Security Profile, February 2025. Final specification, OpenID Foundation. Section 6.5: "Browser-swapping attacks". URL: https://openid.net/specs/fapi-security-profile-2_0-final.html#name-browser-swapping-attacks.

[12] Pontus Hanssen. Writeup: Account Takeover in Authentik due to Insecure Redirect URIs (CVE-2024-52289). Omegapoint Security Blog, January 2025. URL: https://securityblog.omegapoint.se/en/writeup-authentik-cve-2024-52289/. Accessed: 2025-12-02.

[13] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL: https://www.rfc-editor.org/info/rfc6749.

[14] Dick Hardt, Aaron Parecki, and Torsten Lodderstedt. The OAuth 2.1 Authorization Framework. Internet-Draft draft-ietf-oauth-v2-1-15, Internet Engineering Task Force, March 2026. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/.

[15] Jannett, Louis and Westers, Maximilian and Wich, Tobias and Mainka, Christian and Mayer, Andreas and Mladenov, Vladislav. SoK: SSO-Monitor - The Current State and Future Research Directions in Single Sign-On Security Measurements. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, 2024. URL: https://sso-monitor.me/statistics.

[16] Luke Jennings. ConsentFix: Analyzing a browser-native ClickFix-style attack that hijacks OAuth consent grants, December 2025. Push Security Blog. URL: https://pushsecurity.com/blog/consentfix.

[17] M. Jones, J. Bradley, and N. Sakimura. JSON Web Signature (JWS). RFC 7515, May 2015. URL: https://www.rfc-editor.org/info/rfc7515/.

[18] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL: https://www.rfc-editor.org/info/rfc7519/.

[19] M. Jones and J. Hildebrand. JSON Web Encryption (JWE). RFC 7516, May 2015. URL: https://www.rfc-editor.org/info/rfc7516/.

[20] Michael B. Jones and Brian Campbell. OAuth 2.0 Form Post Response Mode, April 2015. Final specification, OpenID Foundation. URL: https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html.

[21] M. M. M. Koper. System Under Oath: Learning Microservice Architectures. Master's thesis, Delft University of Technology, May 2026. URL: https://resolver.tudelft.nl/uuid:18f2d338-d10b-47fd-a387-22c2148779bf.

[22] Austin Larsen, Matt Lin, Tyler McLellan, and Omar ElAhdan. Widespread Data Theft Targets Salesforce Instances via Salesloft Drift. Google Threat Intelligence Group and Mandiant, August 2025. URL: https://cloud.google.com/blog/topics/threat-intelligence/data-theft-salesforce-instances-via-salesloft-drift. Accessed: 2025-12-02.

[23] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018. URL: https://doi.org/10.1186/s42400-018-0002-y.

[24] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. Best Current Practice for OAuth 2.0 Security. RFC 9700, January 2025. URL: https://www.rfc-editor.org/info/rfc9700.

[25] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. SoK: single sign-on security — an evaluation of OpenID Connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 251–266. IEEE, 2017. URL: https://doi.org/10.1109/EuroSP.2017.32.

[26] Lucas McDaniel, Erik Talvi, and Brian Hay. Capture the flag as cyber security introduction. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5479–5486, 2016. URL: https://doi.org/10.1109/HICSS.2016.677.

[27] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990. URL: https://doi.org/10.1145/96267.96279.

[28] Edi Muškardin, Bernhard Aichernig, Andrea Pferscher, and Martin Tappler. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 18:1–10, 03 2022. URL: https://doi.org/10.1007/s11334-022-00449-3.

[29] OpenID Foundation. About the conformance suite. Accessed: 2026-08-18. URL: https://openid.net/certification/about-conformance-suite/.

[30] Andreas Pashalidis and Chris J. Mitchell. A taxonomy of single sign-on systems. In *Australasian conference on information security and privacy*, pages 249–264. Springer, 2003. URL: doi.org/10.1007/3-540-45067-X_22.

[31] Portswigger. JWT attacks. Web Security Academy. URL: https://portswigger.net/web-security/jwt. Accessed: 2025-12-02.

[32] Portswigger. OAuth 2.0 authentication vulnerabilities. Web Security Academy. URL: https://portswigger.net/web-security/oauth. Accessed: 2025-12-02.

[33] Portswigger. OpenID Connect. Web Security Academy. URL: https://portswigger.net/web-security/oauth/openid. Accessed: 2025-12-02.

[34] Jonas Primbs. OAuth 2.0 Browser Swapping Attacks, November 2025. SySS Tech Blog. URL: https://blog.syss.com/posts/browser_swapping/.

[35] Jonas Primbs. Browser Swapping – How to Hack & How to Fix? OAuth Security workshop presentation, 2026. SySS GmbH / University of Tübingen. URL: https://uni-tuebingen.de/en/fakultaeten/mathematisch-naturwissenschaftliche-fakultaet/fachbereiche/informatik/lehrstuehle/kommunikationsnetze/staff/jonas-primbs/.

[36] Frans Rosén. Account hijacking using "dirty dancing" in sign-in OAuth-flows. Detectify Labs, July 2022. URL: https://labs.detectify.com/writeups/account-hijacking-using-dirty-dancing-in-sign-in-oauth-flows/. Accessed: 2025-12-02.

[37] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 2. Technical report, OpenID Foundation, December 2023. URL: https://openid.net/specs/openid-connect-core-1_0.htm.

[38] Nat Sakimura. Browser swap attack explained on 2022-09-28. Bitbucket issue, 2022. OpenID FAPI Working Group issue 543. URL: https://bitbucket.org/openid/fapi/issues/543.

[39] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, September 2015. URL: https://www.rfc-editor.org/info/rfc7636.

[40] Yaron Sheffer, Dick Hardt, and Michael B. Jones. JSON Web Token Best Current Practices. RFC 8725, February 2020. URL: https://www.rfc-editor.org/info/rfc8725.

[41] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on OAuth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 651–662, 2016. URL: https://doi.org/10.1145/2897845.2897874.

[42] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Fuzzing: Breaking things with random inputs. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/html/Fuzzer.html. Accessed: 2025-11-24.

Appendix A

OAuth 2.0 and OIDC Security Requirements

For more information on these requirements, see Section 4.1.

Capitalized key words like "MUST" are to be interpreted as described in RFC 2119 [3]. To summarize:

- "MUST", "REQUIRED", or "SHALL" specify absolute requirements of the specification.
- Items with "SHOULD" or "RECOMMENDED" may be ignored with valid reasons, but the implications must be understood and carefully weighed.
- "MAY" or "OPTIONAL" are truly optional and can be ignored freely.

A.1 OAuth 2.0

Security requirements of the authorization and resource server:

1. The client identifier, a unique string representing the registration information provided by the client, MUST NOT be used alone for client authentication, as it is exposed to the resource owner.
2. The authorization server MUST use TLS.
3. The authorization server MUST validate the redirection URI securely, no unintended redirection URIs should pass validation.
4. The authorization server MUST ensure that both redirection URIs are identical in authorization code flow.
5. The authorization endpoint URI MUST NOT include a fragment component

6. Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once.

7. Lack of a redirection URI registration requirement can enable an attacker to use the authorization endpoint as an open redirector.

8. When a redirection URI is included in an authorization request, the authorization server MUST compare and match the value received against at least one of the registered redirection URIs (or URI components) if any redirection URIs were registered. If the client registration included the full redirection URI, the authorization server MUST compare the two URIs using simple string comparison.

9. If an authorization request fails validation due to a missing, invalid, or mismatching redirection URI, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

10. Issued authorization codes SHOULD expire in 10 minutes.

11. Authorization codes MUST NOT be reusable. Reuse of authorization codes SHOULD cause revocation of all issued tokens based on this code.

12. The authorization server MUST NOT automatically redirect the user-agent to an invalid redirection URI.

13. The `redirect_uri` parameter in the access token request MUST match the `redirect_uri` in the authorization request.

14. Client authentication MUST be required for confidential clients.

15. The authorization server MUST ensure that the authorization code was issued to the client identifier used.

16. The authorization code MUST be verified.

17. Issued access tokens MUST expire.

18. The authorization server MUST protect the access token request endpoint against brute force attack if it supports the resource owner password credentials grant.

19. The authorization server MUST verify the password correctly with the resource owner password credentials grant.

20. The authorization server MUST authenticate the client in case of the client credentials grant.

21. The authorization server MUST include appropriate anti-caching response headers, otherwise causing access tokens to be cached.

22. The resource server MUST validate the access token and ensure that it has not expired and that its scope covers the requested resource.

23. The authorization server MUST NOT issue client passwords or other client credentials to native application or user-agent-based application clients for the purpose of client authentication.

24. When client authentication is not possible, the authorization server SHOULD employ other means to validate the client's identity – for example, by requiring the registration of the client redirection URI for enlisting the resource owner to confirm identity.

25. It SHOULD NOT be possible to generate, modify or guess valid access or refresh tokens.

26. The authorization server MUST maintain the binding between a refresh token and the client to whom it was issued.

27. The authorization server MUST sanitize (and validate when possible) any value received, in particular the value of the "state" and "redirect_uri" parameters.

Security requirements of the client:

1. The client SHOULD NOT register a wrong redirection URI, allowing for more redirection URIs to be used.

2. The client SHOULD NOT include any third-party scripts in the redirection endpoint response, which could allow these scripts to steal credentials.

3. The client MUST ensure that used redirection URIs are identical.

4. The redirection endpoint URI MUST be an absolute URI. The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted query component, which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

5. The redirection endpoint SHOULD require the use of TLS when the requested response type is "code" or "token".

6. The client MUST use the HTTP "POST" method when making access token requests.
7. The state parameter of the authorization request SHOULD be used to prevent cross site request forgery.
8. The client SHOULD discard the credentials obtained from the resource owner password credentials grant.
9. The client credentials grant MUST only be used by confidential clients.
10. The client SHOULD use the state parameter to protect against CSRF. This value must be non-guessable.
11. The client SHOULD prevent clickjacking attacks using the “x-frame-options” header.
12. The client MUST sanitize (and validate when possible) any value received, in particular the value of the “state” and “redirect_uri” parameters.

A.2 OpenID Connect

Security requirements of the OP:

1. The Subject Identifier MUST NOT exceed 255 ASCII characters in length.
2. ID Tokens MUST NOT use `none` as the `alg` value unless the response type used returns no ID Token from the authorization endpoint and the client explicitly requested the use of `none` at registration time.
3. Communication with the authorization endpoint MUST utilize TLS.
4. Authorization servers MUST support the use of the HTTP GET and POST methods at the authorization endpoint.
5. `redirect_uri` MUST match one of the redirection URI values for the client preregistered at the OP using simple string comparison.
6. If the `sub` claim is requested with a specific value of the ID Token, the OP MUST NOT reply with an ID Token or Access Token for a different user than the one authenticated, even if they have an active session.
7. When an `id_token_hint` is present, the OP MUST validate that it was the issuer of the ID Token.

8. When interacting with the end-user, the OP MUST employ appropriate measures against CSRF and Clickjacking attacks.

9. After authentication, the OP MUST obtain an authorization decision (i.e. consent) before releasing information to the RP.

10. Requests to the token endpoint MUST use TLS.

11. In the implicit flow, the nonce claim MUST be included in the returned ID Token.

12. In the implicit flow, the ID Token MUST include the `at_hash` claim.

13. If signed, the UserInfo response MUST contain the `iss` and `aud` claims, where the `iss` value MUST be the OP's issuer identifier URL and the `aud` value MUST be or include the RP's client identifier.

14. With the claims parameter in the authorization request, if a specific value is requested, the value MUST be valid. If the claim was `sub`, a mismatch MUST cause the authentication to fail.

15. If the acr Claim is requested as an Essential Claim for the ID Token with a value or values parameter requesting specific Authentication Context Class Reference values and the implementation supports the claims parameter, the Authorization Server MUST return an acr Claim Value that matches one of the requested values. The Authorization Server MAY ask the End-User to re-authenticate with additional factors to meet this requirement.

16. In an aggregated claim, the JWT SHOULD NOT contain a sub (subject) Claim unless its value is an identifier for the End-User at the Claims Provider (and not for the OpenID Provider or another party); this typically means that a sub Claim SHOULD NOT be provided.

17. The `request_uri` parameter MUST use the https scheme unless the target Request Object is signed in a way that is verifiable by the OP.

18. The parameters in the `request` parameter need to be validated the same as in the authentication request.

19. The request and request_uri parameters MUST NOT be included in Request Objects.

20. If the request or request_uri parameter is used in combination with normal authorization request syntax parameters, the values in the request parameter supersede them.

21. If the client requests the acr Claim using both the acr_values request parameter and an individual acr Claim request for the ID Token listing specific requested values, the resulting behavior is unspecified.

22. When the message is both signed and encrypted, it MUST be signed first and then encrypted.

23. The signing party MUST select a signature algorithm based on the algorithms supported by the recipient.

24. Asymmetric signatures: the `alg` header MUST be set to an appropriate algorithm. The private key used to sign the content MUST be associated with a public key used for signature verification published by the sender in its JWK Set document. If there are multiple keys, a `kid` value MUST be provided in the header. The key usage MUST support signing. Symmetric signatures: the `alg` header MUST be set to a MAC algorithm. The `client_secret` is used as the MAC key and must have enough entropy. Symmetric signatures MUST NOT be used by public clients, as they can not keep secrets.

25. The encrypting party MUST select an encryption algorithm based on the algorithms supported by the recipient.

26. Asymmetric encryption with RSA: the public key MUST be a public key used for encryption in the JWK set document of the recipient. If there are multiple keys, a `kid` value MUST be provided in the header. The key usage MUST include encryption. Asymmetric encryption with elliptic curves: create an ephemeral elliptic curve public key for the `epk` element in the header. The other public key used for the key agreement MUST be a public key published by the recipient in its JWK set document. If there are multiple keys, a `kid` value MUST be provided in the header. The key usage MUST include encryption. Symmetric encryption: the symmetric encryption key is derived from the `client_secret` by truncating its SHA-2 hash. Symmetric encryption MUST NOT be used by a public client as they can not keep secrets.

27. When offline access is requested, a `prompt` parameter value of `consent` MUST be used unless other conditions for processing the request permitting offline access to the requested resources are in place. The request MUST be ignored unless an authorization code is to be returned.

28. If an ID token is returned as a result of a token refresh request, the following requirements apply:

    - The `iss`, `sub` and `aud` claim values MUST be the same as in the ID token issued when the original authentication occurred

- The `iat` claim MUST represent the time that the new ID token is issued
- If the ID token contains an `auth_time` claim, its value MUST represent the time of the original authentication

29. Unicode Normalization MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.

30. the Access Token SHOULD be audience and scope restricted.

31. Implementations SHOULD NOT terminate the validation process at the instant of the finding an error but SHOULD continue running until all the octets have been processed to avoid timing attacks.

32. HTTP 307 redirects send a POST request to the party being redirected to that contains all the form data from the previous request. This can leak credentials intended for the OpenID Provider to the Relying Party. Therefore, HTTP 307 redirects MUST NOT be used when redirecting to the Redirection URI. Likewise, while HTTP 302 redirects are typically implemented in a way that does not do this, the use of HTTP 303 redirect is preferable, as it is defined not to do this.

Security requirements of the RP:

1. The ID Token MUST NOT be accepted after the expiration time, apart from some small leeway.

2. RPs must verify that the nonce claim equals the nonce parameter sent in the authentication request.

3. ID Tokens MUST NOT use `none` as the `alg` value unless the response type used returns no ID Token from the authorization endpoint and the client explicitly requested the use of `none` at registration time.

4. The scope parameter in the authentication request MUST contain "openid".

5. Sufficient entropy MUST be present in the nonce values used to prevent attackers from guessing values.

6. RP SHOULD reject the ID Token if encryption was negotiated but the ID Token is not encrypted.

7. The issuer identifier for the OP MUST exactly match the value of the `iss` claim.

8. The RP MUST validate that the `aud` claim contains its `client_id`.

9. If the ID Token is received via direct communication between the RP and the token endpoint, the RP MAY use TLS server validation instead of checking the token signature.

10. The `alg` value SHOULD be the default of `RS256` or the algorithm sent by the RP during registration.

11. If the JWT `alg` uses a MAC based algorithm such as `HS256`, `HS384`, or `HS512`, the octets of the UTF-8 representation of the `client_secret` corresponding to the `client_id` are used as the key to validate the signature.

12. The current time MUST be before the time represented by the `exp` claim.

13. If a nonce value was sent, a `nonce` claim MUST be present and match the one sent to prevent replay attacks.

14. If the `acr` claim was requested, the RP SHOULD check if the claim value is appropriate.

15. If the `auth_time` claim was requested, the RP SHOULD check its value and potentially request re-authentication if too much time has elapsed.

16. In the implicit flow, the nonce parameter MUST be included.

17. In the implicit flow, the access token SHOULD also be validated by the RP using the `at_hash` value in the ID Token.

18. In the implicit flow, the RP MUST validate the signature of the ID Token according to JWS using the algorithm specified in the `alg` header parameter.

19. In the implicit flow, the value of the `nonce` claim MUST be checked to verify that it is the same value as the one that was sent in the authentication request, and SHOULD be checked for replay attacks.

20. RPs MUST verify the `target_link_uri` value at the login initiation endpoint to prevent it from being used as an open redirector to external sites.

21. RPs SHOULD employ techniques to prevent end-users from being logged in by third party sites without their knowledge through attacking like Clickjacking.

22. The sub Claim in the UserInfo Response MUST be verified to exactly match the sub Claim in the ID Token; if they do not match, the UserInfo Response values MUST NOT be used, due to the possibility of token substitution attacks.

23. If signed, the UserInfo response MUST contain the `iss` and `aud` claims, where the `iss` value MUSt be the OP's issuer identifier URL and the `aud` value MUST be or include the RP's client identifier.

24. The combination of the `sub` and `iss` claims from the ID token are the only claims that can be used as a stable identifier. Other Claims such as email, phone_number, preferred_username, and name MUST NOT be used as unique identifiers for the End-User, whether obtained from the ID Token or the UserInfo Endpoint.

25. If the Request Object pointed to by the request_uri includes requested values for Claims, it MUST NOT be revealed to anybody but the Authorization Server.

26. The signing party MUST select a signature algorithm based on the algorithms supported by the recipient.

27. Asymmetric signatures: the `alg` header MUST be set to an appropriate algorithm. The private key used to sign the content MUST be associated with a public key used for signature verification published by the sender in its JWK Set document. If there are multiple keys, a `kid` value MUST be provided in the header. The key usage MUST support signing. Symmetric signatures: the `alg` header MUST be set to a MAC algorithm. The `client_secret` is used as the MAC key and must have enough entropy. Symmetric signatures MUST NOT be used by public clients, as they can not keep secrets.

28. The encrypting party MUST select an encryption algorithm based on the algorithms supported by the recipient.

29. Asymmetric encryption with RSA: the public key MUST be a public key used for encryption in the JWK set document of the recipient. If there are multiple keys, a `kid` value MUST be provided in the header. The key usage MUST include encryption. Asymmetric encryption with elliptic curves: create an ephemeral elliptic curve public key for the `epk` element in the header. The other public key used for the key agreement MUST be a public key published by the recipient in its JWK set document. If there are multiple keys, a `kid` value MUST be provided in the header. The key usage MUST include encryption. Symmetric encryption: the symmetric encryption key is derived from the `client_secret` by truncating its SHA-2 hash. Symmetric encryption MUST NOT be used by a public client as they can not keep secrets.

30. Unicode Normalization MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.

Appendix B

SimpleSAMLphp OIDC RP and OP setup

This appendix describes the setup of the SimpleSAMLphp OIDC RP and OP that was used during fuzzing, SUT 1 as described in Section 7.2. We used SimpleSAMLphp version 2.4.3, SimpleSAMLphp OIDC module version 6.2.1, and SimpleSAMLphp AuthOAuth2 module version 5.1.0 on PHP 8.3.6 on Ubuntu 24.04.1 LTS. We have set up a SimpleSAMLphp OIDC RP and OP on the same host for fuzzing. We have used multiple sources of documentation for this setup:

- T&I Incubator Documentation: `https://wiki.geant.org/spaces/G52W5/pages/1135542803/Step-by-step+guide+to+set+up+a+minimal+OpenID+Federation+with+gorp-offa+apache+simplesamlphp+and+lighthouse`
- SimpleSAMLphp AuthOAuth2 module (RP) documentation: `https://github.com/cirrusidentity/simplesamlphp-module-authoauth2`
- SimpleSAMLphp documentation: `https://simplesamlphp.org/docs/stable/index.html`
- SimpleSAMLphp OIDC module (OP) documentation: `https://github.com/simplesamlphp/simplesamlphp-module-oidc/tree/master/docs`

Our setup can be reproduced with the following steps:

1. Install necessary packages: `apt install apache2 libapache2-mod-php mariadb-server php-mysql`
2. Setup SSL with certbot:

   ```
   sudo snap install --classic certbot
   sudo ln -s /snap/bin/certbot /usr/bin/certbot
   sudo certbot --apache
   ```

3. Follow the SimpleSAMLphp docs on installing SimpleSAMLphp in `/var/simplesamlphp-op`. Make sure to install the necessary PHP extensions like `php-xml` with `apt`, and to give the www-data ownership of the folder: `chown -R www-data /var/simplesamlphp-op`.

4. Repeat the process for the RP: install SimpleSAMLphp in `/var/simplesamlphp-rp`.

5. Edit `/etc/apache2/sites-enabled/000-default.conf` to redirect to HTTPS:

```
<VirtualHost *:80>
...
    RewriteEngine on
    RewriteCond %{SERVER_NAME} =fuzz1.incubator.geant.org
    RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI} [
   END,NE,R=permanent]
</VirtualHost>
```

6. Edit `/etc/apache2/sites-enabled/000-default-le-ssl.conf` to add SimpleSAMLphp routes:

```
<VirtualHost *:443>
    ...
    # SimpleSAMLphp config
    Alias /simplesaml-op /var/simplesamlphp-op/public

    <Directory /var/simplesamlphp-op/public>
            Require all granted
    </Directory>

    Alias /simplesaml-rp /var/simplesamlphp-rp/public

    <Directory /var/simplesamlphp-rp/public>
            Require all granted
    </Directory>
    ...
</VirtualHost>
```

   Enable rewrite and reload afterwards: `sudo a2enmod rewrite && sudo systemctl reload apache2`

7. Edit the following values in `/var/simplesamlphp-op/config/config.php`:

```
...
$config = [
    ...
    'baseurlpath' => 'https://fuzz1.incubator.geant.org/
    simplesaml-op',
    ...
    'secretsalt' => '<random secret salt>',
    'auth.adminpassword' => '<secure admin password>',
    ...
    'database.dsn' => 'mysql:host=localhost;dbname=
    simplesamlphp',
    'database.username' => 'simplesamlphp',
    'database.password' => 'simplesamlphp',
    ...
    'module.enable' => [
            'exampleauth' => true,
            'core' => true,
            'admin' => true,
            'saml' => true,
            'oidc' => true,
    ],
    ...
    'session.phpsession.cookiename' => 'SimpleSAML',
    'session.authtoken.cookiename' => 'SimpleSAMLAuthToken
    '
    ...
];
```

8. Edit the following values in `/var/simplesamlphp-rp/config/config.php`:

```
...
$config = [
    ...
    'baseurlpath' => 'https://fuzz1.incubator.geant.org/
    simplesaml-rp',
    ...
    'secretsalt' => '<random secret salt>',
    'auth.adminpassword' => '<secure admin password>',
    ...
    'database.dsn' => 'mysql:host=localhost;dbname=
    simplesamlphprp',
    'database.username' => 'simplesamlphp',
    'database.password' => 'simplesamlphp',
```

```
    ...
    'module.enable' => [
            'exampleauth' => true,
            'core' => true,
            'admin' => true,
            'saml' => true,
            'authoauth2' => true,
    ],
    ...
    'session.phpsession.cookiename' => 'SimpleSAMLrp',
    'session.authtoken.cookiename' => '
   SimpleSAMLrpAuthToken',
    ...
];
```

9. Setup database:

   (a) Run `mysql_secure_installation`

   (b) Connect to mariadb and create the databases:

```
CREATE DATABASE simplesamlphp CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;
CREATE USER 'simplesamlphp'@'localhost' IDENTIFIED BY
    'simplesamlphp';
GRANT ALL PRIVILEGES ON simplesamlphp.* TO '
    simplesamlphp'@'localhost';
FLUSH PRIVILEGES;

CREATE DATABASE simplesamlphprp CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;
GRANT ALL PRIVILEGES ON simplesamlphprp.* TO '
    simplesamlphp'@'localhost';
FLUSH PRIVILEGES;
```

10. Setup example auth in `/var/simplesamlphp-op/config/authsources.php`

```
...
$config = [
    ...
    'example-userpass' => [
        'exampleauth:UserPass',
```

```
        // Give the user an option to save their username
    for future login attempts
        // And when enabled, what should the default be,
    to save the username or not
        //'remember.username.enabled' => false,
        //'remember.username.checked' => false,

        'users' => [
            'student:studentpass' => [
                'uid' => ['laura123'],
                'eduPersonAffiliation' => ['member', '
    student'],
                'cn' => ['Laura Erasmus'],
                'mail' => ['laura@example.org'],
            ],
            'employee:employeepass' => [
                'uid' => ['employee'],
                'eduPersonAffiliation' => ['member', '
    employee'],
            ],
        ],
    ],
    ...
];
```

11. Install the OIDC OP module in `/var/simplesamlphp-op` by following: `https://github.com/simplesamlphp/simplesamlphp-module-oidc/blob/master/docs/2-oidc-installation.md`

12. Install the OIDC RP module with `composer require cirrusidentity/simplesamlphp-module-authoauth2` in `/var/simplesamlphp-rp`

13. Register a client in the OP admin interface at `https://fuzz1.incubator.geant.org/simplesaml-op/admin` > OIDC > Client Registry > Add Client. Use Authentication Source "example-userpass", Redirect URI "https://fuzz1.incubator.geant.org/simplesaml-rp/module.php/authoauth2/linkback", Post-logout Redirect URI "https://fuzz1.incubator.geant.org/simplesaml-rp/module.php/authoauth2/loggedout". Copy the resulting clientId and clientSecret.

14. Add the OP to the RP configuration in `/var/simplesamlphp-rp/config/authsources.php`

```
$config = [
    'openidconnect' => array(
```

```
        // Must install correct provider with: composer
    require patrickbussmann/oauth2-apple
        'authoauth2:OpenIDConnect',
         // *** Required for all integrations ***
        'issuer' => 'https://fuzz1.incubator.geant.org', #
     e.g https://accounts.google.com
        'clientId' => '<your clientId>',
        'clientSecret' => '<your clientSecret>',

        // Most Optional settings for OAuth2 above can be
    used
        // *** Optional ***
        // Customize post logout redirect, if you don't
    want to use the standard /module.php/authoauth2/
    loggedout.php
        // 'postLogoutRedirectUri' => 'https://myapp.
    example.com/loggedout'

        // Set a specific discovery url. Default is
    $issuer/.well-known/openid-configuration
        'discoveryUrl' => 'https://fuzz1.incubator.geant.
    org/simplesaml-op/module.php/oidc/.well-known/openid-
    configuration',
        // Check if the issuer in the ID token matches the
     one from discovery. Default true. For some multi-
    tenant
        // applications (for example cross tenant Azure
    logins) the token issuer varies with tenant
        'validateIssuer' => true,

        // Earlier version OpenIDConnect authsource doesn'
    t support using `scopes` for overriding scope
        //'urlAuthorizeOptions' => [
        //    'scope' => 'openid'
        //]
    ),
    ...
];
```

15. Test the login flow at `https://fuzz1.incubator.geant.org/simplesaml-rp/admin/test` > openidconnect > Login with example-userpass (student:studentpass).

Appendix C

Shibboleth OIDC OP setup with Apache `mod_auth_openidc`

This appendix describes the setup of the Shibboleth OIDC OP with Apache `mod_auth_openidc` as RP, SUT 3 as described in Section 7.2. Note that we do not provide documentation for the setup of SUT 2. This setup can be reproduced by installing Apache `mod_auth_openidc` on SUT 1. Both Shibboleth OIDC and Apache `mod_auth_openidc` run on the same host. We used Apache `mod_auth_openidc` 2.4.19.2, OpenJDK 17.0.19, Jetty 12.1.8, Shibboleth 5.2.1, with Shibboleth plugins versions:

```
net.shibboleth.oidc.common Version: 3.4.0
net.shibboleth.idp.plugin.oidc.op Version: 4.4.0
net.shibboleth.idp.plugin.oidc.config Version: 3.1.0
```

on Ubuntu 24.04.1 LTS. We have used multiple sources of documentation for this setup:

- Shibboleth OIDC OP documentation: `https://shibboleth.atlassian.net/wiki/spaces/IDPPLUGINS/pages/1376878976/OIDC+OP`
- Apache `mod_auth_openidc` documentation: `https://github.com/OpenIDC/mod_auth_openidc/wiki`

Our setup for SUT 3 can be reproduced with the following steps:

1. Setup Apache with certbot as described in Appendix B.
2. Install the required packages:

   ```
   wget https://github.com/OpenIDC/mod_auth_openidc/releases/
       download/v2.4.19.2/libapache2-mod-auth-openidc_2
       .4.19.2-1.noble_amd64.deb
   sudo dpkg -i libapache2-mod-auth-openidc_2.4.19.2-1.
       noble_amd64.deb
   ```

3. Then, configure your apache as follows:

   /etc/apache2/sites-enabled/default-ssl.conf:

```
<IfModule mod_ssl.c>
<VirtualHost *:443>
        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/html

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined

        # Proxy /idp to Shibboleth IdP on Jetty
        ProxyPreserveHost On
        ProxyPass        /idp  http://127.0.0.1:8080/idp
        ProxyPassReverse /idp  http://127.0.0.1:8080/idp

        # Tell Shibboleth the request arrived over HTTPS
        RequestHeader set X-Forwarded-Proto "https"
        RequestHeader set X-Forwarded-For "%{REMOTE_ADDR}s
    "

        # Mod Auth OpenIDC config
        OIDCProviderMetadataURL https://fuzz2.incubator.
    geant.org/idp/profile/oidc/configuration
        OIDCClientID demo_rp
        OIDCClientSecret topsecret
        OIDCRedirectURI https://fuzz2.incubator.geant.org/
    mod_auth_openidc/redirect_uri
        OIDCCryptoPassphrase a-long-random-secret-here
        OIDCPKCEMethod none

        <Location /mod_auth_openidc>
                AuthType openid-connect
                Require valid-user
        </Location>

        ServerName fuzz2.incubator.geant.org
        SSLCertificateFile /etc/letsencrypt/live/fuzz2.
    incubator.geant.org/fullchain.pem
        SSLCertificateKeyFile /etc/letsencrypt/live/fuzz2.
    incubator.geant.org/privkey.pem
        Include /etc/letsencrypt/options-ssl-apache.conf
</VirtualHost>
</IfModule>
```

We disable PKCE to simplify the flow.

4. Now we continue with Shibboleth. First, install java: `sudo apt install openjdk-17-jdk`

5. Download and extract Shibboleth 5.2.1 (you might need to change the shibboleth download link to the latest version):

```
wget https://shibboleth.net/downloads/identity-provider/
    latest5/shibboleth-identity-provider-5.2.1.tar.gz
tar -xf shibboleth-identity-provider-5.2.1.tar.gz
cd shibboleth-identity-provider-5.2.1
```

6. Run the installer: `sudo bin/install.sh`, select `/opt/shibboleth-idp` as the installation directory.

7. Download and extract Jetty 12.1.8:

```
wget https://repo1.maven.org/maven2/org/eclipse/jetty/
    jetty-home/12.1.8/jetty-home-12.1.8.tar.gz
tar -xf jetty-home-12.1.8.tar.gz
mv jetty-home-12.1.8 /opt/jetty-home-12.1.8
```

8. Get Shibboleth's Jetty base and extract it:

```
git clone https://git.shibboleth.net/git/java-idp-jetty-
    base
cd java-idp-jetty-base
git checkout 12.1
cp -r src/main/resources/net/shibboleth/idp/module/jetty/
    jetty-base /opt/jetty-base
```

9. Install and enable the Shibboleth OIDC OP plugin with:

```
cd /opt/shibboleth-idp
bin/plugin.sh -I net.shibboleth.oidc.common
bin/plugin.sh -I net.shibboleth.idp.plugin.oidc.config
bin/plugin.sh -I net.shibboleth.idp.plugin.oidc.op
```

   Add an import statement to `/opt/shibboleth-idp/conf/credentials.xml`:

```
<!-- OIDC extension default credential definitions -->
<import resource="oidc-credentials.xml" />
```

   Enable automatic property discovery in `/opt/shibboleth-idp/conf/idp.properties`:

```
idp.searchForProperties = true
```

   Generate the required keys:

```
cd /opt/shibboleth-idp
bin/jwtgen.sh -t RSA -s 2048 -u sig -i defaultRSASign |
    tail -n +2 > credentials/idp-signing-rs.jwk
bin/jwtgen.sh   -t EC -c P-256 -u sig -i defaultECSign |
    tail -n +2 > credentials/idp-signing-es.jwk
bin/jwtgen.sh   -t RSA -s 2048 -u enc -i defaultRSAEnc |
    tail -n +2 > credentials/idp-encryption-rsa.jwk
```

10. /opt now contains jetty-base, jetty-home-12.1.8 and shibboleth-idp. Configure them as follows:
/opt/jetty-base/start.d/shibboleth.ini:

```
# Setup common for running an IdP or a Hub (for SP4)

# Enable our "comprehensive" module
# We disable this to disable TLS
# --module=shibboleth

# Include all modules manually
--module=ee11-annotations
--module=ee11-deploy
--module=ee11-jsp
--module=ee11-jstl
--module=ee11-plus
--module=ee11-servlets
--module=ee11-webapp
--module=resources
--module=server
--module=ext
--module=http

# Support X-Forwarded-For as we're behind Apache
--module=forwarded

# Route access logging through standard SLF4J logging API
etc/jetty-requestlog.xml

# Do not expose contexts to web.
jetty.server.default.showContexts=false

###################################
## Network/Host/Port configuration
###################################
```

```
# Non-TLS host and port to bind to
jetty.http.host=127.0.0.1
jetty.http.port=8080

# Suppress node name in JSESSIONID values
# This assumes a non-clustered Jetty deploy
jetty.sessionIdManager.workerName=
```

/opt/jetty-base/webapps/idp.xml:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "
    https://www.eclipse.org/jetty/configure_10_0.dtd">
<!--
    ===============================================================
     -->
<!-- Configure the Shibboleth IdP webapp
                -->
<!--
    ===============================================================
     -->
<Configure class="org.eclipse.jetty.ee11.webapp.
    WebAppContext">
  <Set name="war"><SystemProperty name="idp.home" default
    ="/opt/shibboleth-idp" />/war/idp.war</Set>
  <Set name="contextPath"><SystemProperty name="idp.
    context.path" default="/idp" /></Set>
  <Set name="extractWAR">false</Set>
  <Set name="copyWebDir">false</Set>
  <Set name="copyWebInf">true</Set>
</Configure>
```

/etc/systemd/system/jetty.service:

```
[Unit]
Documentation=https://www.eclipse.org/jetty/documentation/
Description=Jetty Server
After=network-online.target
Wants=network-online.target

[Service]
# To use a shell script to run Jetty, change to "forking
    "...
Type=exec

# Set as desired to limit privileges.
```

```
User=ubuntu
Group=ubuntu

# Allow use of port80/443.
AmbientCapabilities=CAP_NET_BIND_SERVICE

Restart=no
TimeoutSec=45
KillMode=process

WorkingDirectory=/opt/jetty-base
# Change every time you upgrade Jetty or you could use a
    symlink, etc...
Environment=JETTY_HOME=/opt/jetty-home-12.1.8
Environment=JETTY_BASE=/opt/jetty-base

# Checks that start configuration exists ad we're not
    trying to start jetty without having that configured
ExecStartPre=/usr/bin/test -d /opt/jetty-base/start.d
ExecStartPre=/usr/bin/test -f /opt/jetty-base/start.d/
    shibboleth.ini

ExecStart=/usr/bin/java -Djetty.home=/opt/jetty-home
    -12.1.8 -Djetty.base=/opt/jetty-base -jar /opt/jetty-
    home-12.1.8/start.jar

ExecStop=/bin/kill ${MAINPID}

SuccessExitStatus=130 143
```

Make sure to enable the service.

`/opt/shibboleth-idp/conf/oidc-attribute-resolver.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<AttributeResolver xmlns="urn:mace:shibboleth:2.0:resolver
    "
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:oidc="urn:mace:shibboleth:2.0:resolver:oidc"
    xsi:schemaLocation="urn:mace:shibboleth:2.0:resolver
    http://shibboleth.net/schema/idp/shibboleth-attribute-
    resolver.xsd
                        urn:mace:shibboleth:2.0:resolver:
    oidc http://shibboleth.net/schema/oidc/shibboleth-
    attribute-encoder-oidc.xsd">
```

```
<!--
Some of the examples assume conf/attributes/oidc-claim
-rules.xml is
loaded into the registry service.

Additionally, the "sub" claim examples below all
assume that the file
including them is also loaded into that service to
process the
AttributeEncoder elements shown in these examples.
-->

<!--
Exactly one attribute needs to supply the "sub" claim,
 but this can't be
fully standardized. There are suggested approaches
below. Using the Scoped
definition is recommended to ensure intrinsic
uniqueness.
-->

<!--
This example (the data connector in particular) will
generate public or
pairwise values depending on client registration. The
public values will
depend on, but not expose, an underlying value, which
is again set to "uid"
for simplicity, but this is not a good strategy unless
 "uid" itself is a
stable, managed value.
-->
<AttributeDefinition id="subject" xsi:type="Scoped"
scope="%{idp.scope}"
        activationConditionRef="shibboleth.oidc.
Conditions.SubjectRequired">
    <InputDataConnector ref="computedSubjectId"
attributeNames="subjectId"/>
    <AttributeEncoder xsi:type="oidc:OIDCScopedString"
 name="sub" />
</AttributeDefinition>

<!--
```

```
 The EPPN is the most common federated username in
higher education.
 For guidelines on the implementation of this attribute
, refer to eduPerson
 and/or federation documentation. Above all, do not
expose a value for this
 attribute without considering the long term
implications.
 -->
 <AttributeDefinition id="eduPersonPrincipalName" xsi:
type="Scoped" scope="%{idp.scope}">
     <InputAttributeDefinition ref="uid" />
     <AttributeEncoder xsi:type="oidc:OIDCScopedString"
  name="eppn" />
 </AttributeDefinition>

 <!--
 The uid is the closest thing to a "standard" LDAP
attribute representing
 a local username, but you should generally *never*
expose uid to federated
 services, as it is rarely globally unique. The default
  mapping for OIDC is
 to preferred_username, which seems suitably
meaningless.
 -->
 <AttributeDefinition id="uid" xsi:type="PrincipalName
">
 </AttributeDefinition>

 <!-- This is just for illustrative purposes given that
  the connector is static.  -->
 <AttributeDefinition id="mail" xsi:type="Template">
     <InputAttributeDefinition ref="uid" />
     <Template><![CDATA[
            ${uid}@%{idp.scope}
       ]]></Template>
 </AttributeDefinition>

 <!--
 Start of static attributes. In actual deployment you
would use a real source
 for most of these attributes/claims.
 -->
```

```
<AttributeDefinition id="eduPersonScopedAffiliation"
xsi:type="Scoped" scope="%{idp.scope}">
    <InputDataConnector ref="staticAttributes"
attributeNames="affiliation" />
</AttributeDefinition>

<!--
This demonstrates a complex claim constructed by
forming a JSON structure.
The default transcoding rule for the address claim
expects such a structure.
-->
<AttributeDefinition id="address" xsi:type="Template">
    <InputDataConnector ref="staticAttributes"
attributeNames="street_address locality region
postal_code country"/>
    <Template><![CDATA[
        {"street_address":"${street_address}", "
locality":"${locality}","region":"${region}","
postal_code":"${postal_code}","country":"${country}" }
    ]]></Template>
</AttributeDefinition>

<!--
Data Connector for generating 'sub' claim. It may be
used to generate both
public and pairwise subject values because it
recognizes the OIDC sector_id
if used during client registration.
-->
<DataConnector id="computedSubjectId" xsi:type="
ComputedId"
        generatedAttributeID="subjectId"
        salt="%{idp.oidc.subject.salt}"
        algorithm="%{idp.oidc.subject.algorithm:SHA}"
        encoding="BASE32">
    <InputAttributeDefinition ref="%{idp.oidc.subject.
sourceAttribute}"/>
</DataConnector>

<!--
Static example to populate default claims. Most of
these are directly exposed and
```

```
 handled by the default set of transcoding rules
provided for optional inclusion.
 -->
 <DataConnector id="staticAttributes" xsi:type="Static"
         exportAttributes="telephoneNumber
phone_number_verified email_verified displayName sn
givenName middle_name eduPersonNickname profile picture
 website gender birthdate zoneinfo preferredLanguage
updated_at">
     <Attribute id="affiliation">
         <Value>member</Value>
         <Value>staff</Value>
     </Attribute>
     <Attribute id="telephoneNumber">
         <Value>+1 (604) 555-1234;ext=5678</Value>
     </Attribute>
     <Attribute id="phone_number_verified">
         <Value>true</Value>
     </Attribute>
     <Attribute id="email_verified">
         <Value>false</Value>
     </Attribute>
     <Attribute id="displayName">
         <Value>Mr.Teppo Matias Testaaja</Value>
     </Attribute>
     <Attribute id="sn">
         <Value>Testaaja</Value>
     </Attribute>
     <Attribute id="givenName">
         <Value>Teppo Matias</Value>
     </Attribute>
     <Attribute id="middle_name">
         <Value>Matias</Value>
     </Attribute>
     <Attribute id="eduPersonNickname">
         <Value>TT</Value>
     </Attribute>
     <Attribute id="profile">
         <Value>https://fi.wikipedia.org/wiki/
Tom_Cruise</Value>
     </Attribute>
     <Attribute id="picture">
         <Value>https://pixabay.com/fi/pentu-kissa-
kukka-potin-tabby-pentu-2766820/</Value>
```

```
        </Attribute>
        <Attribute id="website">
            <Value>https://www.facebook.com/
    officialtomcruise/</Value>
        </Attribute>
        <Attribute id="gender">
            <Value>male</Value>
        </Attribute>
        <Attribute id="birthdate">
            <Value>1969-07-20</Value>
        </Attribute>
        <Attribute id="zoneinfo">
            <Value>America/Los_Angeles</Value>
        </Attribute>
        <Attribute id="preferredLanguage">
            <Value>en-US</Value>
        </Attribute>
        <Attribute id="updated_at">
            <Value>1509450347</Value>
        </Attribute>
        <Attribute id="street_address">
            <Value>234 Hollywood Blvd.</Value>
        </Attribute>
        <Attribute id="locality">
            <Value>Los Angeles</Value>
        </Attribute>
        <Attribute id="region">
            <Value>CA</Value>
        </Attribute>
        <Attribute id="postal_code">
            <Value>90210</Value>
        </Attribute>
        <Attribute id="country">
            <Value>US</Value>
        </Attribute>
    </DataConnector>

</AttributeResolver>
```

/opt/shibboleth-idp/conf/services.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/
    context"
    xmlns:util="http://www.springframework.org/schema/util
```

```
" xmlns:p="http://www.springframework.org/schema/p"
 xmlns:c="http://www.springframework.org/schema/c"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/
schema/beans http://www.springframework.org/schema/
beans/spring-beans.xsd
                        http://www.springframework.org/
schema/context http://www.springframework.org/schema/
context/spring-context.xsd
                        http://www.springframework.org/
schema/util http://www.springframework.org/schema/util/
spring-util.xsd"

 default-init-method="initialize"
 default-destroy-method="destroy">

 <!-- By default we look at resources whose names are
derived from %{idp.home}. -->

 <util:list id="shibboleth.
RelyingPartyResolverResources">
     <value>%{idp.home}/conf/relying-party.xml</value>
     <value>%{idp.home}/conf/credentials.xml</value>
 </util:list>

 <util:list id="shibboleth.MetadataResolverResources">
     <value>%{idp.home}/conf/metadata-providers.xml</
value>
 </util:list>

 <util:list id ="shibboleth.AttributeResolverResources
">
     <value>%{idp.home}/conf/attribute-resolver.xml</
value>
     <value>%{idp.home}/conf/oidc-attribute-resolver.
xml</value>
 </util:list>

 <!--
 This is suitable for new installs but will usually
produce duplicate Attribute
 output if a legacy resolver file is used that contains
  AttributeEncoders.
 -->
```

```
    <util:list id ="shibboleth.AttributeRegistryResources
    ">
        <value>%{idp.home}/conf/attribute-registry.xml</
    value>
        <value>%{idp.home}/conf/attributes/default-rules.
    xml</value>
        <value>%{idp.home}/conf/attribute-resolver.xml</
    value>
        <value>%{idp.home}/conf/attributes/oidc-claim-
    rules.xml</value>
        <value>%{idp.home}/conf/oidc-attribute-resolver.
    xml</value>
    </util:list>

    <util:list id ="shibboleth.AttributeFilterResources">
        <value>%{idp.home}/conf/attribute-filter.xml</
    value>
        <value>%{idp.home}/conf/oidc-attribute-filter.xml
    </value>
    </util:list>

    <util:list id ="shibboleth.
    NameIdentifierGenerationResources">
        <value>%{idp.home}/conf/saml-nameid.xml</value>
    </util:list>

    <util:list id="shibboleth.AccessControlResources">
        <value>%{idp.home}/conf/access-control.xml</value>
    </util:list>

    <!--
    This collection of resources differs slightly in that
    it should not include the file extension.
    Message sources are internationalized, and Spring will
     search for a compatible language extension
    and fall back to one with only a .properties extension
    .
    -->
    <util:list id="shibboleth.MessageSourceResources">
        <value>%{idp.home}/messages/messages</value>
    </util:list>

</beans>
```

/opt/shibboleth-idp/conf/metadata/oidc-client.json:

```
[
  {
    "scope":"openid email",
    "redirect_uris":["https://fuzz2.incubator.geant.org/
   mod_auth_openidc/redirect_uri"],
    "client_id":"demo_rp",
    "client_secret":"topsecret",
    "response_types":["code"],
    "grant_types":["authorization_code"],
    "subject_type":"public"
  }
]
```

/opt/shibboleth-idp/conf/oidc-clientinfo-resolvers.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/
   schema/context"
       xmlns:util="http://www.springframework.org/schema/
   util"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance"
       xsi:schemaLocation="http://www.springframework.org/
   schema/beans http://www.springframework.org/schema/
   beans/spring-beans.xsd
                           http://www.springframework.org/
   schema/context http://www.springframework.org/schema/
   context/spring-context.xsd
                           http://www.springframework.org/
   schema/util http://www.springframework.org/schema/util/
   spring-util.xsd"

       default-init-method="initialize"
       default-destroy-method="destroy"
       default-lazy-init="true">

    <!--
    The following example contains two OIDC client
    information resolvers:
        - first one reading a single client's information
```

```
    from a JSON file
        - second one fetching client information from a
    configured StorageService
     -->

     <util:list id="shibboleth.oidc.
    ClientInformationResolvers">
        <ref bean="ExampleFileResolver" />
        <ref bean="ExampleStorageClientInformationResolver
    " />
     </util:list>


     <bean id="ExampleFileResolver" parent="shibboleth.oidc
    .FilesystemClientInformationResolver"
        c:metadata="%{idp.home}/metadata/oidc-client.json"
     />


     <bean id="ExampleStorageClientInformationResolver"
    parent="shibboleth.oidc.
    StorageClientInformationResolver"
        p:storageService-ref="#{'%{idp.oidc.dynreg.
    StorageService:shibboleth.StorageService}'.trim()}" />

</beans>
```

/opt/shibboleth-idp/conf/authn/password-authn-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/
    schema/context"
       xmlns:util="http://www.springframework.org/schema/
    util"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
       xsi:schemaLocation="http://www.springframework.org/
    schema/beans http://www.springframework.org/schema/
    beans/spring-beans.xsd
                           http://www.springframework.org/
    schema/context http://www.springframework.org/schema/
    context/spring-context.xsd
```

```
                        http://www.springframework.org/
schema/util http://www.springframework.org/schema/util/
spring-util.xsd"

    default-init-method="initialize"
    default-destroy-method="destroy">

<!--
 Ordered list of CredentialValidators to apply to a
request.

 The four supplied variants are shown below; the
HTPasswd option
 is an OOB default for demo account purposes, and you
will
 want to remove it after initial install and testing.
 -->
 <util:list id="shibboleth.authn.Password.Validators">
     <!-- <ref bean="shibboleth.LDAPValidator" /> -->
     <!-- <ref bean="shibboleth.KerberosValidator" />
-->
     <!-- <ref bean="shibboleth.JAASValidator" /> -->
     <!-- <bean parent="shibboleth.HTPasswdValidator" p
:resource="%{idp.home}/credentials/demo.htpasswd" />
-->
     <ref bean="shibboleth.FileCredentialValidator"/>
 </util:list>

 <!-- Apply any regular expression replacement pairs to
  username before validation. -->
 <util:list id="shibboleth.authn.Password.Transforms">
     <!--
     <bean parent="shibboleth.Pair" p:first="^(.+)
@example\.org$" p:second="$1" />
     -->
 </util:list>

 <bean id="shibboleth.FileCredentialValidator"
   parent="shibboleth.HTPasswdCredentialValidator"
   p:resource="%{idp.home}/credentials/demo.htpasswd"/>

 <!-- Uncomment to configure account lockout backed by
in-memory storage. -->
 <!--
```

```
<bean id="shibboleth.authn.Password.
AccountLockoutManager"
    parent="shibboleth.
StorageBackedAccountLockoutManager"
    p:maxAttempts="5"
    p:counterInterval="PT5M"
    p:lockoutDuration="PT5M"
    p:extendLockoutDuration="false" />
-->

<!--
Define entries here to map error messages detected by
validation actions and classify them as particular
kinds of errors for use in your templates and as
events in flows.

Keys are events to signal, values are error codes.
-->
<util:map id="shibboleth.authn.Password.
ClassifiedMessageMap">
    <entry key="UnknownUsername">
        <list>
            <value>NoCredentials</value>
            <value>CLIENT_NOT_FOUND</value>
            <value>Client not found</value>
            <value>Cannot get kdc for realm</value>
            <value>Client not found in Kerberos
database</value>
            <value>DN_RESOLUTION_FAILURE</value>
            <value>Cannot authenticate dn, invalid dn
</value>
            <value>Cannot authenticate dn, invalid
credential</value>
            <value>AcceptSecurityContext error, data
525</value>
        </list>
    </entry>
    <entry key="InvalidPassword">
        <list>
            <value>InvalidCredentials</value>
            <value>PREAUTH_FAILED</value>
            <value>INVALID_CREDENTIALS</value>
            <value>Checksum failed</value>
            <value>Integrity check on decrypted field
```

```
failed</value>
            <value>Pre-authentication information was
invalid</value>
            <value>Key bytes cannot be null</value>
            <value>AcceptSecurityContext error, data
52e</value>
        </list>
    </entry>
    <entry key="AccountLocked">
        <list>
            <value>Clients credentials have been
revoked</value>
            <value>AcceptSecurityContext error, data
775</value>
        </list>
    </entry>
    <entry key="AccountDisabled">
        <list>
            <value>AcceptSecurityContext error, data
533</value>
        </list>
    </entry>
    <entry key="ExpiredPassword">
        <list>
            <value>PASSWORD_EXPIRED</value>
            <value>CLIENT KEY EXPIRED</value>
            <value>AcceptSecurityContext error, data
532</value>
            <value>AcceptSecurityContext error, data
773</value>
            <value>AcceptSecurityContext error, data
701</value>
        </list>
    </entry>
    <entry key="ExpiringPassword">
        <list>
            <value>ACCOUNT_WARNING</value>
        </list>
    </entry>
 </util:map>

</beans>
```

11. Add users:

```
htpasswd -cs /opt/shibboleth-idp/credentials/demo.htpasswd
     student
htpasswd -s /opt/shibboleth-idp/credentials/demo.htpasswd
    employee
```

12. /opt/shibboleth-idp/conf/relying-party.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/
    schema/context"
       xmlns:util="http://www.springframework.org/schema/
    util"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
       xsi:schemaLocation="http://www.springframework.org/
    schema/beans http://www.springframework.org/schema/
    beans/spring-beans.xsd
                           http://www.springframework.org/
    schema/context http://www.springframework.org/schema/
    context/spring-context.xsd
                           http://www.springframework.org/
    schema/util http://www.springframework.org/schema/util/
    spring-util.xsd"

       default-init-method="initialize"
       default-destroy-method="destroy">

    <!--
    Unverified RP configuration, defaults to no support
    for any profiles. Add <ref> elements to the list
    to enable specific default profile settings (as below)
    , or create new beans inline to override defaults.

    "Unverified" typically means the IdP has no metadata,
    or equivalent way of assuring the identity and
    legitimacy of a requesting system. To run an "open"
    IdP, you can enable profiles here.
    -->
    <bean id="shibboleth.UnverifiedRelyingParty" parent="
    RelyingParty">
        <property name="profileConfigurations">
```

```
        <list>
        <!-- <bean parent="SAML2.SSO" p:
encryptAssertions="false" /> -->
            <ref bean="OIDC.Keyset" />
            <bean parent="OIDC.Configuration" />
        </list>
    </property>
</bean>

<!--
Default configuration, with default settings applied
for all profiles.

Take care with any defaults you apply at this level
because you will have to create
overrides or apply metadata tags for every single SP
that requires a different setting.
Changed defaults should be things you really do want
to apply to nearly every SP.
-->
<bean id="shibboleth.DefaultRelyingParty" parent="
RelyingParty">
    <property name="profileConfigurations">
        <list>
            <!-- SAML 1.1 and SAML 2.0 AttributeQuery
are disabled by default. -->
            <!--
            <ref bean="Shibboleth.SSO" />
            <ref bean="SAML1.AttributeQuery" />
            <ref bean="SAML1.ArtifactResolution" />
            -->
            <!--<ref bean="SAML2.SSO" />
            <ref bean="SAML2.ECP" />
            <ref bean="SAML2.Logout" />
            -->
            <!--
            <ref bean="SAML2.AttributeQuery" />
            -->
            <!--<ref bean="SAML2.ArtifactResolution"
/>
            -->
            <ref bean="OIDC.SSO" />
            <ref bean="OIDC.UserInfo"/>
            <ref bean="OAUTH2.Token"/>
```

```
                <ref bean="OAUTH2.Revocation"/>
                <ref bean="OAUTH2.Introspection" />
            </list>
        </property>
    </bean>

    <!-- Container for any overrides you want to add. -->

    <util:list id="shibboleth.RelyingPartyOverrides">

        <!--
        Override example that identifies a single RP by
    name and configures it
        for SAML 2 SSO without encryption. This is a
    common "vendor" scenario.
        -->
        <!--
        <bean id="ExampleSP" parent="RelyingPartyByName" c
    :relyingPartyIds="https://sp.example.org">
            <property name="profileConfigurations">
                <list>
                    <bean parent="SAML2.SSO" p:
    encryptAssertions="false" />
                </list>
            </property>
        </bean>
        -->

    </util:list>

</beans>
```

`/opt/shibboleth-idp/conf/oidc.properties` (only the changed lines are shown):

```
# Set the Open ID Connect Issuer value
idp.oidc.issuer = https://fuzz2.incubator.geant.org

# Signing keys for id tokens / userinfo response
idp.signing.oidc.rs.key = %{idp.home}/credentials/idp-
    signing-rs.jwk
idp.signing.oidc.es.key = %{idp.home}/credentials/idp-
    signing-es.jwk
# Request object decryption key
idp.signing.oidc.rsa.enc.key = %{idp.home}/credentials/idp
```

```
    -encryption-rsa.jwk

# The source attribute used in generating the sub claim
idp.oidc.subject.sourceAttribute = uid

idp.oidc.subject.salt = <randomly generated salt>

# Settings for the configuration flow
# Flow is available at /oidc/configuration, usually it
    should be wired from /.well-known/openid-configuration
idp.oidc.discovery.template = %{idp.home}/static/openid-
    configuration.json
idp.oidc.discovery.resolver = shibboleth.oidc.
    DefaultOpenIdConfigurationResolver
idp.oidc.discovery.resolver.values = shibboleth.oidc.
    discovery.DefaultDynamicValueResolvers
```