RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Security Analysis of Real-Life OpenID Connect Implementations

Christian Fries

Master's Thesis – December 20, 2020.
Chair for Network and Data Security.

Supervisor: Prof. Dr. Jörg Schwenk
Advisor: Dr. -Ing. Christian Mainka, Dr. -Ing. Vladislav Mladenov

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

Single Sign-On with OpenID Connect is a widely adopted delegated authentication protocol. It is a layer above OAuth 2.0 which provides delegated authorization. This protocol framework allows users to connect several Service Providers with their account, identified from a single Identity Provider. The development of several extensions and additional features is still in progress. Previous work has revealed that not all implementations strictly follow the OpenID Connect specification.

This master's thesis has aimed to unveil security flaws in OpenID Connect Certified implementations with well-known attack methods. For this purpose, we present a novel and sustainable lab environment based on Docker, which offers an expandable platform. We intend this lab for developers and penetration-testers to test Service Providers and Identity Providers in a real-world scenario. It aims to lower the initial effort to analyze the implementations automatically and manually. Therefore, we included the tool PrOfESSOS for automatic tests. Together with MitMProxy, we supplied a debugging interface and created a Command-line Interface to perform manual tests with support of PrOfESSOS.

In summary, we have selected six Identity Provider and eight Service Provider with support of Implicit Flow and Hybrid Flow. For a comprehensive security analysis, we tested them against eleven Service Provider attacks and seven Identity Provider attacks in different variations. We have disclosed twelve implementation flaws and reported them to the developers in a responsible disclosure process.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Date	Christian Fries

Erklärung

Ich erkläre mich damit einverstanden, dass meine Masterarbeit am Lehrstuhl NDS dauerhaft in elektronischer und gedruckter Form aufbewahrt wird und dass die Ergebnisse aus dieser Arbeit unter Einhaltung guter wissenschaftlicher Praxis in der Forschung weiter verwendet werden dürfen.

Date	Christian Fries

Contents

Acronyms

AMC Authentication Method Confusion.

CAB Client Authentication Bypass.

CLI Command-line Interface.

CSRF Cross Side Request Forgery.

DoS Denial of Service.

EaaS Evaluation as a Service.

IC Issuer Confusion.

IdP Identity Provider.

IDPC IdP Confusion.

IDS ID Spoofing.

jku JWK Set URL.

JWE JSON Web Encryption.

jwk JSON Web Key.

JWKS JSON Web Key Set.

JWS JSON Web Signature.

JWT JSON Web Token.

KC Key Confusion.

LFI Local File Inclusion.

MAC Message Authentication Code.

MEA Malicious Endpoint Attack.

MFC Message Flow Confusion.

MitM Man-in-the-Middle.

MitMProxy Man-in-the-Middle Proxy.

OIDC OpenID Connect.

OP OpenID Provider.

OR Open Redirector.

PDA PKCE Downgrade Attack.

PKCE Proof Key for Code Exchange.

PrOfESSOS Practical Offensive Evaluation of Single Sign-On Services.

RA Replay Attack.

RFI Remote File Inclusion.

RP Relying Party.

RQ Research Question.

RUM Redirect URI Manipulation.

SaaS Software as a Service.

SCS Sub Claim Spoofing.

SM Signature Manipulation.

SO Session Overwriting.

SP Service Provider.

SPA Single-Page Application.

SSO Single Sign-On.

SSRF Server Side Request Forgery.

TRC Token Recipient Confusion.

TS Token Substitution.

x5c X.509 Certificate Chain.

x5u X.509 URL.

XSS Cross-Site-Scripting.

1 Introduction

With the growing usage of cloud services and web-based mobile applications, the Internet has grown over the last decade. More and more micro services with dedicated functionalities exist. Platform developers allow others to use their services. It opened a completely new market with Software as a Service (SaaS) delivery model. Nearly all leading enterprise software companies are offering their business applications for web browsers or have announced to step into this market. These are, for example, services for collaboration, IT management, enterprise resource planning, human resource management, messaging software or office suites.

It requires access control and authorization to enable these interactions between multiple micro-services and platforms. In addition, End-Users should be identified to perform tasks on their behalf. A standard protocol framework is OAuth 2.0 for authorization delegation that has been established since 2012. Two years later and on top of this protocol, OpenID Connect (OIDC) was designed to authenticate End-Users. An essential part of the protocol is token-based authorization (OAuth 2.0) and authentication (OIDC). End-Users are no more forced to give their credentials directly to third-party applications to access resources of another service.

Leading companies use OpenID Connect, for example, Amazon, Microsoft, Google, and Facebook. The Single Sign-On (SSO) service they provide allows users to interact with different services without a complete registration process. It is sufficient for an End-User, to login at the Identity Provider (IdP) and allow a Service Provider (SP) to access data to identify them.

Due to complexity and security requirements, OAuth 2.0 and OpenID Connect are interesting for web security researcher. The Identity Provider manages various user accounts and creates tokens to access web services. Users can register unknown and potential malicious SP clients to the IdP. On the opposite site, a SP must be able to verify the exchanged tokens and provide user access to granted resources.

1.1 Motivation

This thesis focus is on real-life OpenID Connect implementations with support of Implicit Flow and Hybrid Flow, motivated through the following three Research Questions (RQs).

RQ I: How secure are OpenID Certified implementations and are they fully compliant to the standard specification? On the one side, the OpenID homepage [6] lists several SP and IdP implementations, which are *OpenID Certified*. This grants libraries and services a higher level of trust, than other implementations in the wild without a certification. On the other side, new extensions are introduced since OIDC was published and developers implement these additional features. Towards the upcoming OAuth 2.1 version, there are preparations to support the required changes. There are various security recommendations and best practice guides for developers. If they follow these references and the specification, then the implementations are supposed to be very secure.

An up-to-date overview about known security vulnerabilities in the implementations is useful to identify common issues. This can be a hint of missing awareness to newer attack vectors and can be used to distribute this information to developers. To close knowledge gaps between developers and security researchers, the following two RQ should be answered.

RQ II: How can a lab environment be created, to reduce the initial implementation and setup effort? Before a security researcher can start the analysis, a target implementation must be setup. It is a time-consuming task to search for sample implementations, required configurations, and software dependencies. For a library-only implementation, an example must be written, first. In this case, a researcher must have a detailed understanding of the programming language, the basic framework, and the libraries API.

A lab environment intended for security analysis presents a promising improvement. Reproducible results and a maintainable structure are expected to save a valuable amount of time. If there is a debug functionality to visualize protocol issues, it might attract developers to use this environment, too. The attraction arises because counterparts for testing of different IdPs or SPs exists.

RQ III: How can manual and automatic security tests be performed in a lab environment? A lab environment can run locally or in a data center. Thus, the security evaluation tools should run as a part of the lab. This also reduces the initial setup effort. TLS certificates must be rolled out and several implementations must be configured first, to use the security tools.

If easy to use, automatic security tools exist, developers can discover security issues on their own and before a new version is released. In a collaboration between developers and security researchers, the penetration testers can concentrate on implementing new security relevant checks to their tools and perform manual penetration testing.

1.2 Related Work

The security properties of OAuth 2.0 and OpenID Connect have been analyzed using formal methods [4]. Fett, Küsters and Schmitz [4, 5] have analyzed the security of OAuth 2.0 and OIDC using their web model. They found eleven OIDC formal attack vectors and provide a security guideline, therefore.

In a large-scale practical analysis of Google, OpenID Connect implementations Li Wanpeng et al. [14] have shown that several flaws could be revealed in implementation of SPs and IdPs. They have also discovered some SP custom Hybrid Flow implementations, that are not conform to the OIDC specification. This customization was implemented to provide an improved user experience. The OpenID Foundation provides, until now, only a basic and an Implicit Flow [7] development guide.

With their work on second-order vulnerabilities Vladislav Mladenov and Christian Mainka [23, 24] have unveiled various specification and implementation flaws since 2014. Towards automated testing of SSO implementations Yuchen Zhou [33] has published a tool named SSOScan. This tool can analyze the security of OAuth implementations. Based on this previous work and in cooperation with Tobias Wich [19] the project Practical Offensive Evaluation of Single Sign-On Services (PrOfESSOS) has been created. It is an open-source implementation for fully automated Evaluation-as-a-Service for SSO.

1.3 Thesis Contribution and Organization

This master's thesis contributes towards a state-of-the-art security analysis of several OpenID Connect implementations. Therefore, we have selected a comprehensive number of IdPs and SPs and included them into a lab environment. We have improved our tool PrOfESSOS for automatic analysis and presenting a method to allow manual or semi-automatic penetration tests.

Chapter 2 introduces the basics of SSO protocols based on OAuth 2.0 for End-User authorization and OpenID Connect on top as identity layer. The front and back-channel communication between SP and IdP endpoints are explained, along with the different token usage.

Chapter 3 gives an overview about attacker goals and capabilities. We introduce seven known Single-Phase and four Cross-Phase attacks on SPs. In addition, we explain seven attack vectors against IdPs. All attacks are with different variations implemented in PrOfESSOS.

Chapter 4 explains the base concept of the docker lab environment. It should require only low effort for developers, to integrate and test their IdP or SP services automatically with PrOfESSOS. For developers and penetration testers, we have added Man-in-the-Middle Proxy (MitMProxy) to analyze the TLS web traffic in front and back-channel communication. With the MitMProxy script interface, we could enhance the PrOfESSOS capabilities.

Chapter 5 presents the findings from our security analysis with PrOfESSOS. Moreover, we introduce an idea of how manual scripting tests could be supported by PrOfESSOS.

Chapter 6 and 7 describe the responsible disclosure process. Finally, we conclude this work and give further suggestions for the future.

2 Single Sign-On with OpenID Connect

This chapter provides the basic SSO expertise, required to follow the later explained and used attacks in this thesis. The basic concept of most SSO specifications is that an End-User has a single account registered to a trusted IdP. After the user has authenticated against this IdP, a token is generated, which could be used to access any allowed resource from a SP. Only precondition is that SP must be known to the IdP and SP can validate the token. The advantage of SSO for an End-User is, only one account is required to access several SP and he must remember only one password. If the SSO protocol provides user authentication information for SP, this can be used to synchronize profile information like address, email, or phone number. Disadvantage is that IdP must be secured against several threats, and End-Users must trust this service. In cases IdP is not reachable or the service is discontinued the SP cannot be accessed.

2.1 OAuth 2.0

The OpenID Connect (OIDC) Core specification describes itself as a simple identity layer on top of the OAuth 2.0 protocol [29]. This means both protocols share many similarities and changes to OAuth protocol can affect the OIDC protocol. OAuth is a complex protocol with the ability to provide an authorization layer. It defines four different core grant types *Authorization Code*, *Implicit*, *Resource Owner Password Credentials* and *Client Credentials*, which defines different protocol processes. Only Authorization Code Flow and Implicit Flow are part of OIDC. In addition, OIDC introduces Hybrid Flow as a third mode.

Two basic types of web application clients are defined. First, a confidential client which can store `client_id` and `client_secret` confidential. Second, a public client which cannot hide client credentials from an End-User, for example, a native application or a Single-Page Application (SPA) executed in a web browser. For the second client type, the simpler Implicit Flow is often used.

The protocol communication defines several endpoints. A critical detail of the OAuth communication flow is the redirect URI, to connect different endpoints. OAuth 2.0 defines an Authorization Endpoint and Token Endpoint that represents one major part of an IdP. Furthermore, a Redirect Endpoint, which represents a web client application. Beyond these core features, there are several OAuth 2.0 protocol extensions. The Dynamic Client Registration is one of them and is also used with OIDC. It introduces an additional Registration Endpoint and a few more protocol steps. All of these previous steps are required, to generate an Access Token and using the token to access protected REST API resources.

2.2 Protocol Communication

OpenID Connect refers the OAuth 2.0 authentication servers (IdP) also as OpenID Providers (OPs) and the clients (SPs) as Relying Parties (RPs) [29]. The third protocol participant is the End-User using a User-Agent, in most cases a web browser for user interactions. OP is a trusted authentication server, which manages all user identity data. After an End-User is registered to this platform, he can use any connected and authorized RP with the OP. Neither OAuth 2.0 nor OpenID Connect specify how an End-User could authenticate to the OP. For example, a simple username and password login or an additional, more complex web authentication protocol can be used. The RP is a web or native application, which can use the tokens to identify the user and to access further resources from another web server with the Access Token. This communication is performed via several endpoints. The related communication channels explained as follows:

Back-channel communication uses normal HTTP request and response formats. This can be HTTP headers, CRUD methods, query, post data and JSON objects. In general, this appears outside the purview of a User-Agent directly between a RP and OP [25]. The client can access a resource server with the token, using a back-channel communication.

Front-channel communication is the method used to allow indirect communication between client and server, over an End-Users browser [25]. Usually, RP and OP are located on different domains. A browser isolates these sessions for security reasons. Each session has an own cookie, local and session storage per security domain. This means the RP client could not directly receive a token after an End-User has logged into an OP. Through redirection, it is possible to attach parameters to an URL. The receiving party must parse this information. In general, front-channel communication is the End-User observable segment, within the OpenID Connect protocol.

2.2.1 Discovery Endpoint

The Discovery Endpoint offers OpenID Provider Issuer Discovery and Provider Configuration with well-known locations. Issuer Discovery is an optional feature, to determine the location of an OpenID Provider. With this mechanism, a RP can detect where an End-User account is located. Furthermore, a RP can register to a new unknown OP with the Dynamic Registration feature explained in section 2.2.3. To start discovery an End-User supplies an Identifier to the RP [27]. The Relying Party parses the Identifier information and requests with the specified WebFinger [13] URI, the OP location (Figure 2.1). For example, WebFinger specification defines email address syntax, which can figure out domain or realm of an OP. Alternatively, without Issuer Discovery, a RP requires out-of-band configuration to determine the OP location, and the End-User can only select predefined providers.

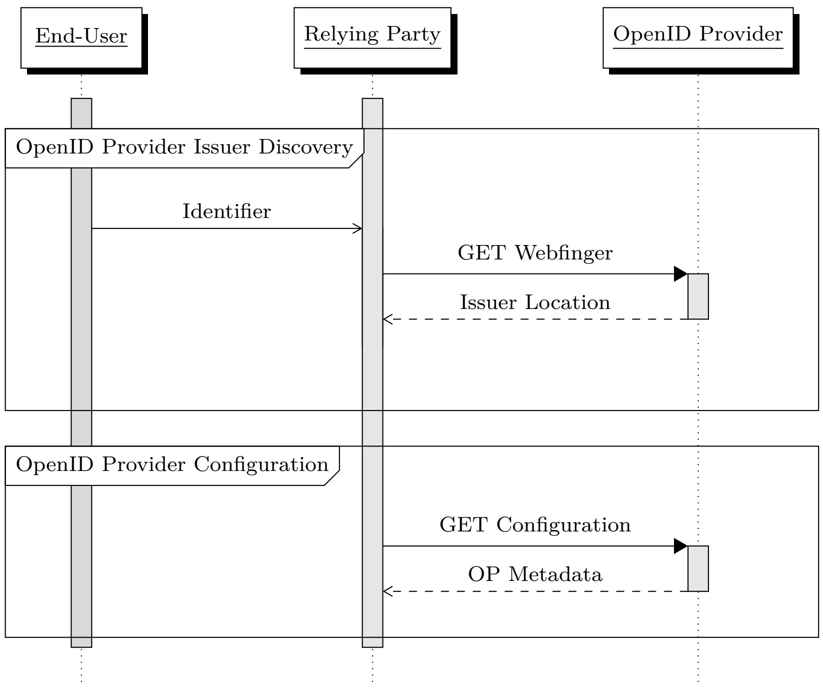


Figure 2.1: Abstract Issuer Discovery and Provider Configuration sequence diagram.

The Provider Configuration provides metadata about the OP. Configuration document [27] is a defined JSON object, including all endpoints, public key location and supported configurations. Thus, clients which want to communicate with the OP, can read endpoint locations from Provider Configuration. In Listing 2.2, for instance, the specified `registration_endpoint` is set to "https://honest.com/register". This URL route can be different per OP.

Clients can access this data during registration to determine if they are properly configured to support OP features. For example, a client configured for Implicit Flow cannot use the OP, if it is not listed in `grant_types_supported`. OP and RP must find a consent. The `issuer` claim is a mandatory part, which must match with configuration URL and ID Token provided later.

```
{
  "issuer": "https://honest.com/",
  "authorization_endpoint": "https://honest.com/auth-req",
  "token_endpoint": "https://honest.com/token-req",
  "token_endpoint_auth_methods_supported": [
    "client_secret_basic"
  ],
  "jwks_uri": "https://honest.com/jwks",
  "registration_endpoint": "https://honest.com/register",
  "userinfo_endpoint": "https://honest.com/user-info"

  "grant_types_supported": [
    "authorization_code",
    "implicit"
  ],
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "response_modes_supported": [
    "query",
    "fragment"
  ],
  "response_types_supported": [
    "code",
    "id_token"
  ],
  "subject_types_supported": [
    "public"
  ],
  "scopes_supported": [
    "openid",
    "name",
    "preferred_username",
    "email"
  ]
}
```

Listing 2.2: JSON metadata from Provider Configuration.

2.2.2 JSON Web Key Set Endpoint

JSON Web Key Set (JWKS) [10] is a JSON structure to represent cryptographic keys. For OIDC it contains keys used to verify or encrypt the ID Token. Several algorithms are supported, for instance, RSA, elliptic curves, or symmetric keys.

The public RSA key in Listing 2.3 represents an example structure. Parameter `key type (kty)` is used to identify the cryptographic algorithm. Furthermore, `public key use (use)` to determine if this key is used for data encryption (`enc`) or to verify a signature (`sig`). The `algorithm (alg)`, in this case, defines the used hashing algorithm to generate the signature. A `Key ID (kid)` can identify a specific key in the set. For RSA, the public `modulus n` and `exponent e` is exposed.

```
{
  "keys": [
    {
      "kty":"RSA",
      "alg":"RS256",
      "use":"sig",
      "n": "KwWjsaK",
      "e": "AQAB"
      "kid":"my-unique-key-id"
    }
  ]
}
```

Listing 2.3: JWKS for an RSA key.

2.2.3 Client Registration Endpoint

A RP client can be configured out-of-band with a manual registration step in the OP. The Client Registration Endpoint offers another mechanism with Dynamic Client Registration. This allows to register a RP to an OP. The OpenID Provider can require an initial Access Token to allow registration or using a rate-limit to prevent Denial of Service (DoS) attacks [28].

A RP sends an OP, JSON data to register with the preferred settings and redirect URIs. The OP responses with a newly created `client_id`. In addition, confidential clients receiving a `client_secret`. The OP can change or reject unsupported client settings.

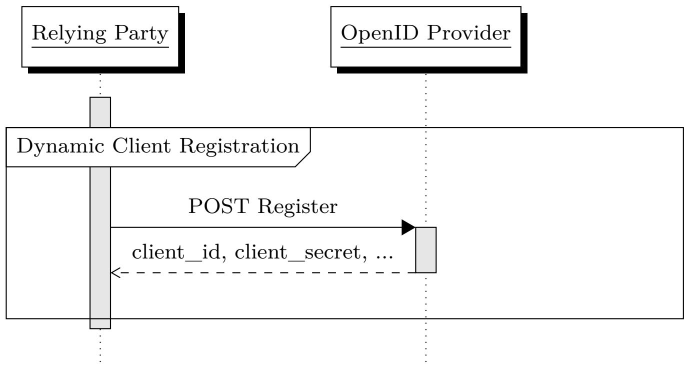


Figure 2.4: Abstract Dynamic Client Registration protocol extension.

2.2.4 Authorization Endpoint

The End-User is authenticated through the Authorization Endpoint [29, Section 3.1.2]. Therefore, RP redirects the User-Agent with a HTTP 302 redirect response to the OP. Following parameters are required.

response_type is used to determine the flow regarding Table 2.6. OP must validate the values with previous client registration data.

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

Table 2.6: OIDC response_type values per flow [29].

scope defines the information, presented as content in the ID Token and UserInfo Endpoint. For OpenID Connect it must be at least `openid`, some additional standard claims are `name`, `profile`, or `email` [29, Section 5.1]. An OP can define own additional scopes, for example, user role or read permissions.

client_id must match a valid previously registered client identifier.

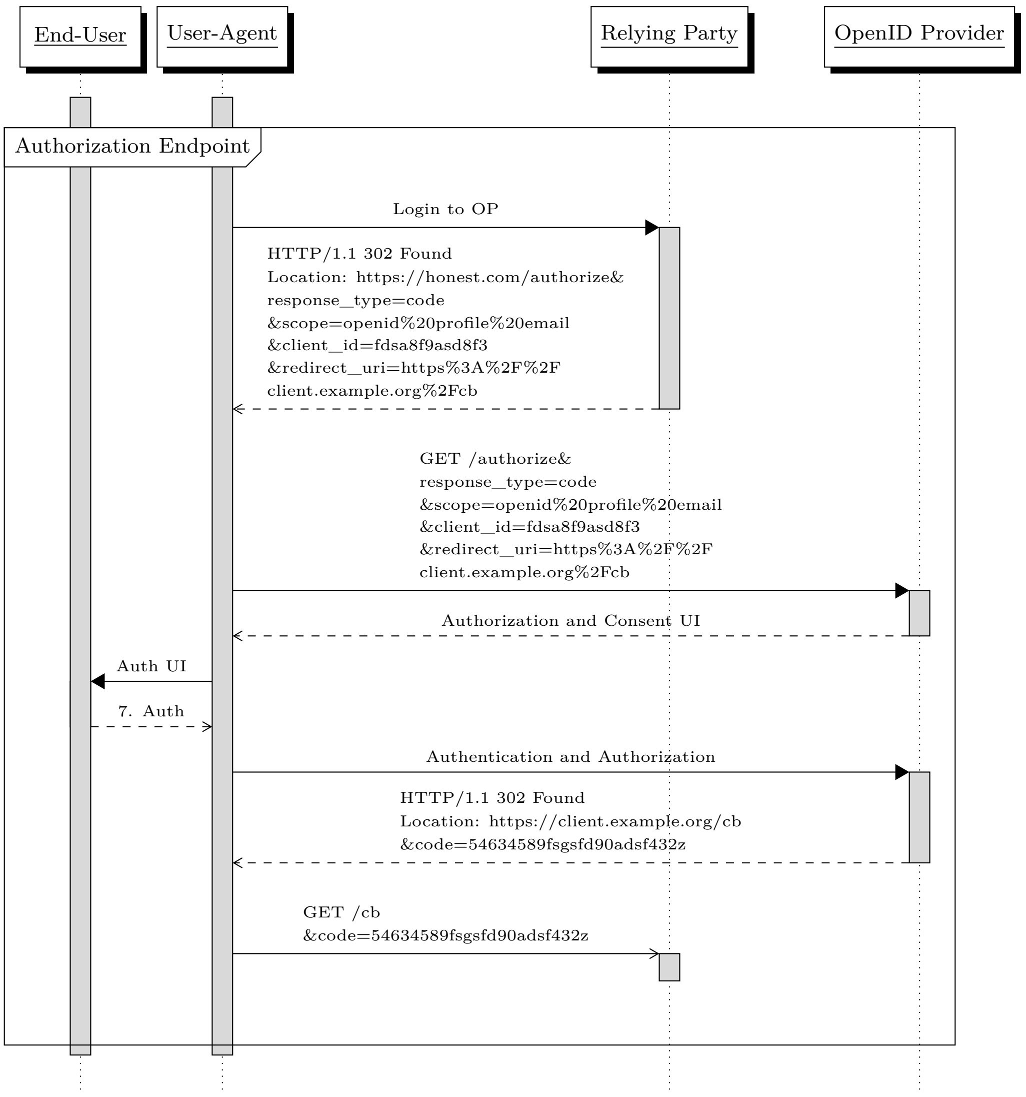

End-User
User-Agent
Relying Party
OpenID Provider
Authorization Endpoint
Login to OP
HTTP/1.1 302 Found
Location: https://honest.com/authorize&
response_type=code
&scope=openid%20profile%20email
&client_id=fdsa8f9asd8f3
&redirect_uri=https%3A%2F%2F
client.example.org%2Fcb
GET /authorize&
response_type=code
&scope=openid%20profile%20email
&client_id=fdsa8f9asd8f3
&redirect_uri=https%3A%2F%2F
client.example.org%2Fcb
Authorization and Consent UI
Auth UI
7. Auth
Authentication and Authorization
HTTP/1.1 302 Found
Location: https://client.example.org/cb
&code=54634589fsgsfd90adsf432z
GET /cb
&code=54634589fsgsfd90adsf432z

Figure 2.5: Login to an Authorization Endpoint with Authorization Code Flow.

redirect_uri is the location where Authorization Endpoint is supposed to redirect End-Users after the login process. This must strictly match one of the registered client's URIs.

state is used to mitigate a Cross Side Request Forgery (CSRF) attack. It must be bound to an End-User session.

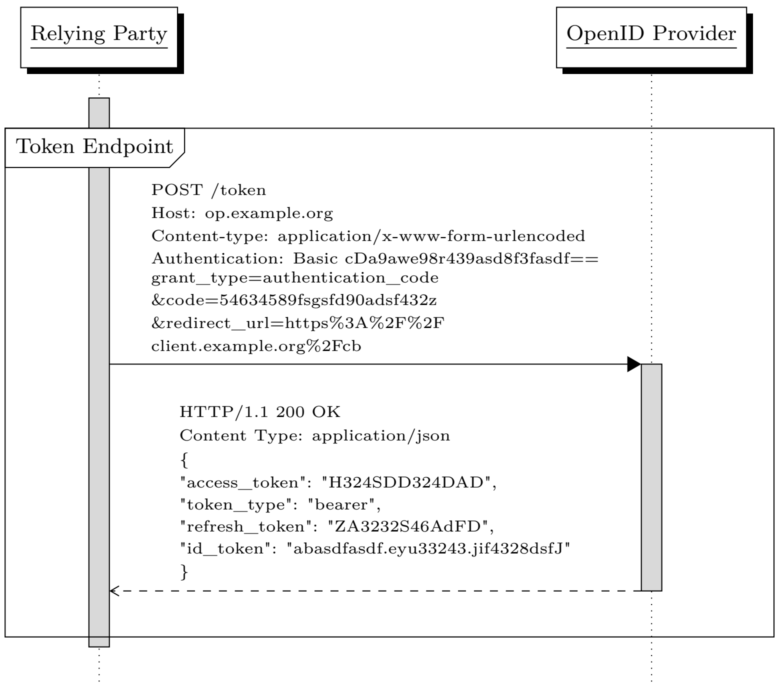
response_mode is an optional parameter to select if the Authentication Response parameters should be in the `query` string or part of the `fragment` string. The OP should send parameters in `fragment` only in Implicit Flow and `query` for Authorization Code Flow.

nonce is freshness parameter to mitigate token replay attacks. This parameter is recommended or required depending on the selected flow.

If an End-User is authenticated and authorized at the Authorization Endpoint, it is possible to accept the set scopes on an optional consent page. In the last step, OP redirects the User-Agent back to the RP, with a previously chosen token. For Authorization Code Flow the Authorization Endpoint returns only an Authorization Code.

2.2.5 Token Endpoint

In Authorization Code Flow and Hybrid Flow, the RP receives an Authorization Code from the Authorization Endpoint. To redeem this code, the RP sends a request to the Token Endpoint, as shown in Figure 2.7. A confidential client must authenticate with the registered authentication method. With a successful response, the RP obtains an Access Token, an ID Token and optional a Refresh Token [29, Section 3.1.3].



```
POST /token
Host: op.example.org
Content-type: application/x-www-form-urlencoded
Authentication: Basic cDa9awe98r439asd8f3fasdf==
grant_type=authentication_code
&code=54634589fsgsfd90adsf432z
&redirect_url=https%3A%2F%2F
client.example.org%2Fcb
```

```
HTTP/1.1 200 OK
Content Type: application/json
{
"access_token": "H324SDD324DAD",
"token_type": "bearer",
"refresh_token": "ZA3232S46AdFD",
"id_token": "abasdfasdf.eyu33243.jif4328dsfJ"
}
```

Figure 2.7: Token Endpoint sequence diagram.

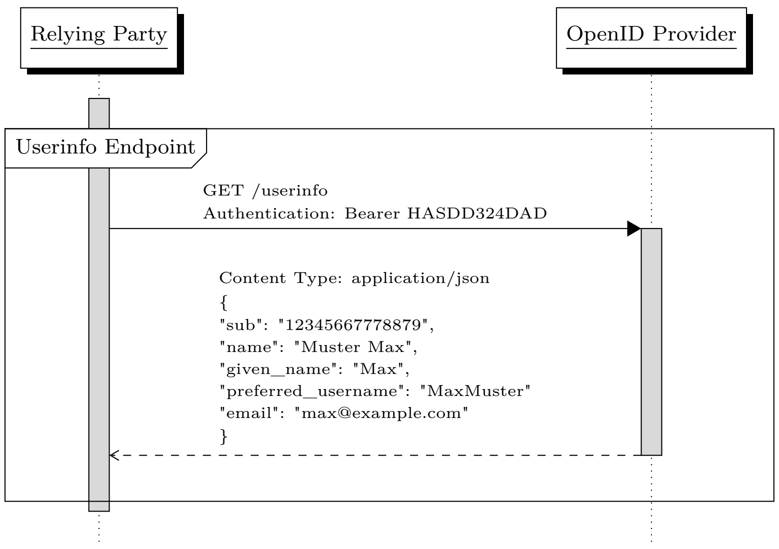
Authentication Method is used to authenticate a RP against the Token Endpoint. Per default HTTP basic authentication scheme is allowed with base64 encoded `client_id` and `client_secret` in HTTP header (Figure 2.7). A HTTP Post method is permitted, where client credentials are part of the request body. In addition, two JSON Web Token (JWT) variants exist, one with `client_secret` as a shared key and one with a private key.

grant_type is set to `authorization_code` to redeem the Authorization Code in `code` parameter. Optionally, to use a Refresh Token it can be set to `refresh_token`.

redirect_uri is the destination to where Token Endpoint sends its response. It must be identical to the redirect URI used at Authorization Endpoint.

2.2.6 UserInfo Endpoint

The UserInfo Endpoint returns an End-User requested claim with the associated Access Token [29, Section 5.3]. UserInfo response can be a JSON object or a JWT. At least, the sub claim must be issued to RP and verified against the sub claim in the ID Token. Other claims depend on requested scope during the Authentication Request.



```
GET /userinfo
Authentication: Bearer HASDD324DAD
```

```
Content Type: application/json
{
"sub": "12345667778879",
"name": "Muster Max",
"given_name": "Max",
"preferred_username": "MaxMuster"
"email": "max@example.com"
}
```

Figure 2.8: Abstract UserInfo Endpoint protocol flow.

For clients using Implicit Flow with only ID Token, all claims are stored in the ID Token, since no Access Token is available to access the UserInfo Endpoint.

2.3 Token Types and Structure

In this section the OIDC tokens and their structures are explained. OAuth 2.0 itself defines Authorization Code and Access Token with unspecified content to access server resources. The ID Token is an additional token, specified in OpenID Connect for End-User authentication.

2.3.1 Authorization Code

The Authorization Endpoint issues an Authorization Code in Hybrid Flow and Authorization Code Flow. Afterwards, the Authorization Code is used to obtain an Access Token and ID Token from a Token Endpoint.

Since Authorization Code is received over the User-Agent via redirection in the front-channel, it is the only disclosed secret. Therefore, it is designed as a short lived and single-use secret [9]. An OP must ensure that this token is unguessable and is not issued to another RP. The format for this token is not defined and must only be understood by the OP. Generally, it is only a cryptographically secure pseudo-random number.

For Implicit Flow, where all tokens are potentially and intentionally revealed to the User-Agent, an Authorization Code is not utilized.

2.3.2 Access Token

To access a protected resource from RP server, an Access Token could be used as credential [9]. The Bearer Token authentication method is used accordingly. The Access Token is an opaque string for the RP client in OpenID Connect. Different Access Token can be created for several endpoints and access privileges [29]. With an expiration time, an Access Token has a limited lifetime. The period is configurable.

2.3.3 ID Token

The ID Token contains claims to authenticate an End-User. This is the primary extension of OpenID Connect, to provide a standardized data structure for authentication, together with the defined UserInfo Endpoint. It is allowed for the OP developers to add additional claims. The RP must know these claims to use them or ignores the additional information. JSON Web Token format is used to represent an ID Token [29].

The JWT object is represented with a JOSE header, to describe the used cryptographic operations. JSON Web Signature (JWS) is used to sign a payload to provide authentication, integrity, and non-repudiation (Listing 2.9). For confidentiality JSON Web Encryption (JWE) format shall be used. Thus, ID Token must be signed and afterwards encrypted to create a nested JWT [12, 29].

```
header = {
  "typ": "JWT"
  "alg": "RS256",
}

payload = {
  "iss": "https://honest.example.com",
  "sub": "BgDltc1AfF223sc3BCc4",
  "aud": "honest-client-id",
  "iat": 1492728260,
  "exp": 1492728320,
  "nonce": "i87yhOySCQCGYQwXbFKK7aDhqDIyERHOG2AfDaWSCYg",
  "at_hash": "BGkvIFNRqH6YJr7u4g3NEA"
  "c_hash": "7u88ogKxDy6V2XxEZntg8g",
  "name": "Honest User",
  "preferred_username": "honest-user-name",
  "email": "user@honest.com"
}

signature = RSA-SHA256(
              private-key,
              base64urlEncoding(header) + '.' +
              base64urlEncoding(payload))

ID Token = base64urlEncoding(header) + '.' +
           base64urlEncoding(payload) + '.' +
           base64urlEncoding(signature)

// ID Token = eyJ0BeXAiO.eyJhdFSoYXNoIjoia.GSoMTGzQQ
```

Listing 2.9: Pseudo-Code to generate an ID Token.

The algorithm (alg) value, defined in the header, is used to determine which cryptographic algorithm is applied to sign or encrypt a JWT. Only secure algorithms should be allowed. All other insecure, broken algorithms and the special *none* value should be blocked by default. These algorithms should only be explicitly allowed, in cases the RP requested them during client registration [29].

Regarding Listing 2.9, the following claims are important for the later executed security analysis and the used OIDC flows. It is to ensure that claims are validated upon RP receives the ID Token and the JWT is signed with a valid signature.

iss Issuer Identifier must contain the complete URL of an OP and must match the issuer in Provider Configuration as described in Section 2.2.1.

sub Subject Identifier is used to assign an unique and not reused identity per OP, to an End-User. The combination of `iss` and `sub` claim should be used to identify an End-User.

aud Audience claim must contain the `client_id` from the RP, for which the ID Token is intended for. The RP is required to ensure that the ID Token is not issued for another client.

iat Issued at Time is a token freshness claim. The RP must validate at time of receiving, if ID Token is issued in an appropriate time frame. A clock skew tolerance should be provided for not exactly synchronized systems and network latency.

exp Expiration time after an ID Token becomes invalid. A RP must reject old, expired ID Tokens.

nonce is another freshness parameter to mitigate replay attacks. RP sends it with the Authentication Request and the OP adds it unmodified to the ID Token. The RP must ensure that the sent `nonce` is equal to `nonce` in the ID Token. This claim is required for Implicit Flow and Hybrid Flow.

at_hash is the left-most half of an Access Token hashed with algorithm used in JWT `alg` header parameter. For Hybrid Flow and Implicit Flow, it is mandatory to validate the Access Token, which is received over the front-channel.

c_hash is the left-most half of an Authorization Code hashed with algorithm used in JWT `alg` header parameter. This is a mandatory hash for Hybrid Flow, which must be validated before the code is used by the intended RP part.

standard claims typical examples are name, email, preferred_username or address. For Implicit Flow, these arguments are included in the ID Token, since only an Access Token grants access to an UserInfo Endpoint to retrieve the claims.

2.4 OpenID Connect Flows

All previously described components can be combined, in subsequent explained, OpenID Connect flows. For the flows shown in this section, it is assuming that Relying Party is already registered to the OpenID Provider. A Discovery Endpoint is not depicted. The End-User is intended to login to a predefined OP and Provider Configuration is retrieved whenever it is required.

2.4.1 Authorization Code Flow

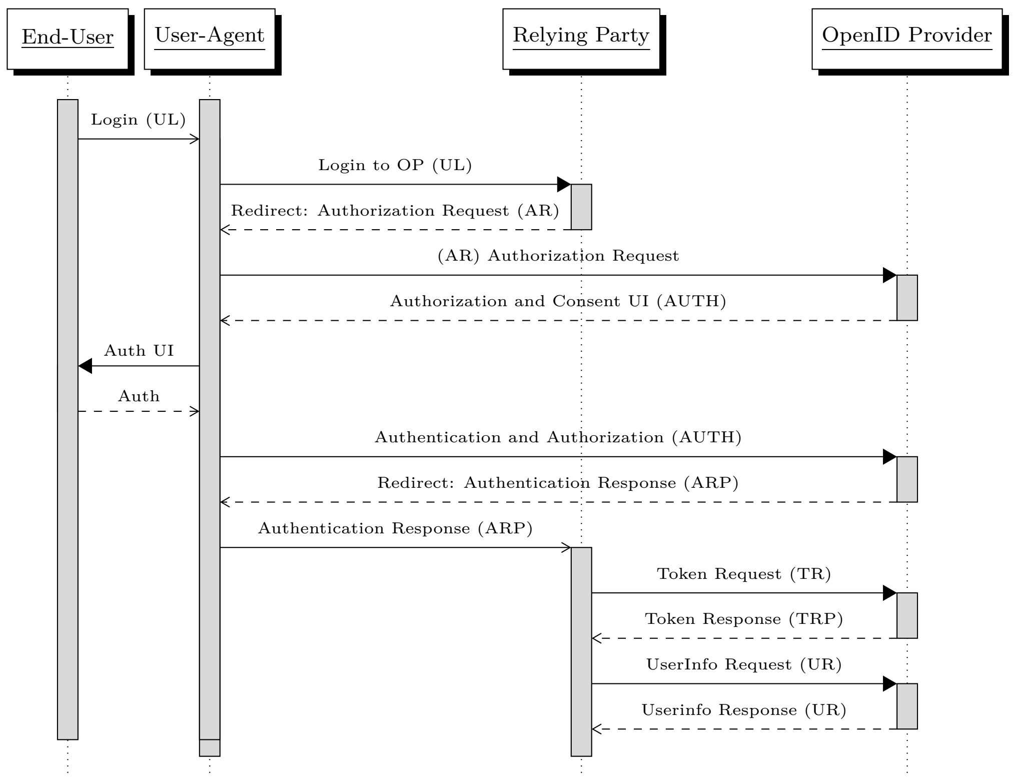

End-User
User-Agent
Relying Party
OpenID Provider
Login (UL)
Login to OP (UL)
Redirect: Authorization Request (AR)
(AR) Authorization Request
Authorization and Consent UI (AUTH)
Auth UI
Auth
Authentication and Authorization (AUTH)
Redirect: Authentication Response (ARP)
Authentication Response (ARP)
Token Request (TR)
Token Response (TRP)
UserInfo Request (UR)
Userinfo Response (UR)

Figure 2.10: OpenID Connect Authorization Code Flow sequence diagram.

The Authorization Code Flow is intended for confidential RP clients, running on server-side [29, Section 3.1]. Only the Authorization Code is revealed to the End-User. Access Token and ID Token are known by RP and OP exclusively.

For public clients, vulnerable to the Access Token Substitution attack, the extension Authorization Code Flow with Proof Key for Code Exchange (PKCE) exists as a mitigation [26]. This attack is feasible considering all client credentials are known by an attacker, because they are shipped in a binary application or with a client-side web application. In this scenario an attacker must be able to intercept the Authentication Response and obtain the Authorization Code. To prevent this, a proof of possession key is used, solely known by the initial honest client. Figure 2.10 illustrates the communication between all parties for both variants.

User Login Usually an End-User can choose from a list with different "Login with OpenID Provider" buttons. If only one provider is configured by the RP, it could be just a login button or a link. With support of Issuer Discovery, regarding subsection 2.2.1, End-User can enter a well-known OP location. Afterwards, the browser (User-Agent) sends this request to the RP.

Authentication Request The RP requests browser to redirect to selected OP as explained in Section 2.2.4 and shown in Figure 2.5. Redirect request must contain the previously registered redirect URI, the `client_id`, and all required user scopes. For Authorization Code Flow, the `response_type` value must be set to `code`. An optional `state` parameter can be added, to mitigate CSRF attacks against the User-Agent. This must be bound to an End-User session, for example, a cookie [29, Section 3.2.2.5]. The OP must store the issued `code` internally and add a reference to End-User and RP.

With PKCE the additional parameter `code_challenge` and optional a `code_challenge_method` is issued to the Authentication Request. The method is used for the Code Verifier transformation. Values are either `plain` (Plaintext) or `S256` (SHA256). It is recommended to register the client particularly for SHA256 and the OP must solely allow this method. The `code_verifier` is generated as a cryptographical random number. Subsequently, it is stored on client-side and transformed with the chosen method. The OP must associate the Code Challenge and used method with the later issued Authorization Code.

```
# Plain
code_challenge = code_verifier
# S256
code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
```

Listing 2.11: Pseudo-Code to generate a Code Challenge.

Auth The End-User it then required to login to the OP. OpenID Connect does not specify this process and the illustrated steps vary per OP. Typically the End-User logs in with credentials and afterwards accepts or denies the requested scopes on an optional consent page. If the user is already logged-in and client was previously accepted, the OP can skip the Authentication process and send the Authentication Response directly.

Authentication Response When OP has successfully validated the End-User login attempt, the User-Agent receives another HTTP redirect response. The browser sends the End-User back to referred callback, previously addressed with the redirect URI in Authentication Request. RP receives the mandatory Authorization Code as a query parameter. The optional `state` parameter can be validated to detect a CSRF attack. In case of missing or mismatching `state`, the client must not proceed.

Token Request To redeem an Authorization Code at the Token Endpoint, the RP sends a Token Request. The `grant_type` parameter must be set to `authorization_code` and the received Authorization Code must be included in `code` parameter. A valid registered redirect URI location is required, where OP should send the Access Token and ID Token afterwards. The RP uses client credentials to authenticate at the Token Endpoint with methods described in Section 2.2.5.

The OP is obligated to check several security aspects [29, Section 3.1.3.2]. First, it is essential that client credentials are valid. Second, a valid code was not used previously and is issued to the requesting RP through an Authentication Response. Third, the redirect URI must be identical to the URI used in Authentication Request.

With PKCE the RP sends an Authorization Code along with the Code Verifier to the Token Endpoint. The OP must apply the transformation and verify the solution regarding Listing 2.12.

```
# Plain
code_verifier == code_challenge
# S256
BASE64URL-ENCODE(SHA256(ASCII(code_verifier))) == code_challenge
```

Listing 2.12: Pseudo-Code to verify a Code Challenge.

Token Response The client receives an Access Token and an ID Token from the Token Response. The client must validate the signature and claims in the ID Token regarding Section 2.3.3.

UserInfo The Relying Party is supposed to collect additional claims about the End-User. Accordingly, these issued claims must correspond to the previously requested scope. After issuing the Access Token to UserInfo Endpoint, a corresponding UserInfo Response is received.

2.4.2 Implicit Flow

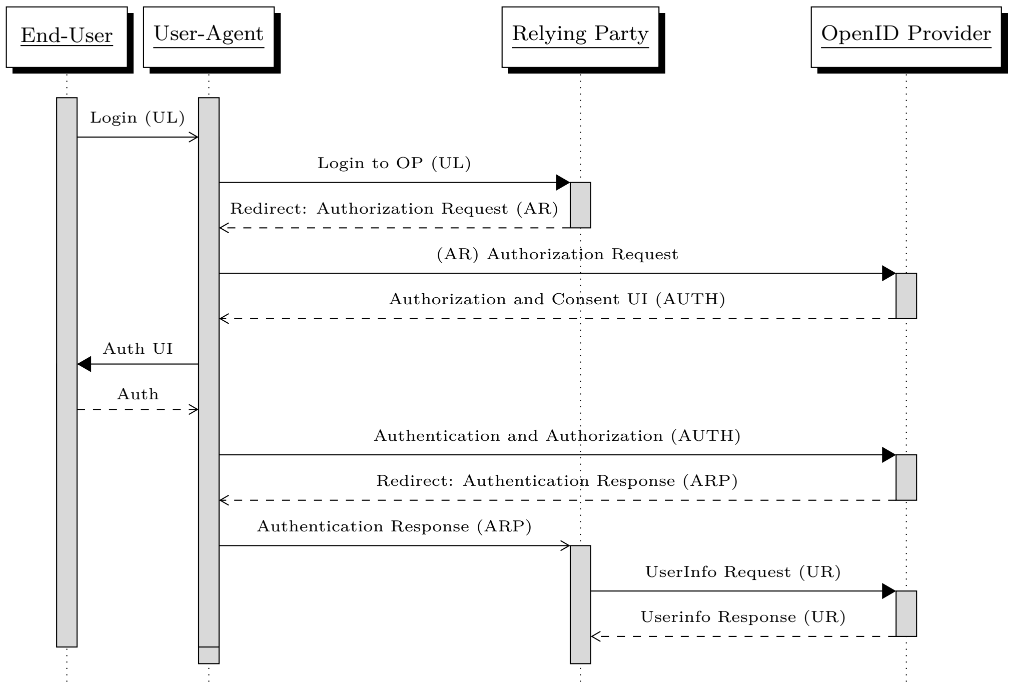

End-User
User-Agent
Relying Party
OpenID Provider
Login (UL)
Login to OP (UL)
Redirect: Authorization Request (AR)
(AR) Authorization Request
Authorization and Consent UI (AUTH)
Auth UI
Auth
Authentication and Authorization (AUTH)
Redirect: Authentication Response (ARP)
Authentication Response (ARP)
UserInfo Request (UR)
Userinfo Response (UR)

Figure 2.13: OpenID Connect Implicit Flow sequence diagram.

The Implicit Flow is intended, for public clients, implemented in a scripting language and executed in a browser. Relying Party depicted in Figure 2.13 runs in the User-Agent. An End-User and the User-Agent have access to Access Token and ID Token. In Implicit Flow all tokens are returned from the Authorization Endpoint [29, Section 3.2]. Token Endpoint is not used in this flow. It is recommended to use Authorization Code Flow with PKCE instead of Implicit Flow [15].

User Login, Authentication and Authorization protocol steps are equal to Authorization Code Flow 2.4.1. Other steps change as follows.

Authentication Request Regarding the Authorization Code Flow the following parameters are changed in the Authentication Request (Listing 2.14). The parameter `response_type` is `id_token` or `id_token token`, as explained in Table 2.6.

In addition, the `nonce` parameter is required and checked during ID Token validation.

```
GET /authorize?
  response_type=id_token%20token
  &client_id=s897dsafA
  &redirect_uri=https%3A%2F%2Fclient.honest.org%2Fcallback
  &scope=openid%20profile
  &state=dasfSdasf532
  &nonce=n-asdfWombd HTTP/1.1
Host: honest.com
```

Listing 2.14: Non-normative example for Authentication Request in Implicit Flow.

Authentication Response According to Listing 2.15 the ID Token and if requested an Access Token is returned as a fragment component. The client is required to validate the signature and claims in the ID Token. If an Access Token was requested, the `at_hash` must be verified. The `nonce` in ID Token should be available and equal to parameter in the Authentication Request.

```
  HTTP/1.1 302 Found
Location: https://client.honest.org/callback#
access_token=SlAV32hkKG
&token_type=bearer
&id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
&expires_in=3600
&state=dasfSdasf532
```

Listing 2.15: Non-normative example for Authentication Response in Implicit Flow.

UserInfo These protocol steps are equal to Authorization Code Flow 2.4.1, however it is only available, with a retrieved Access Token.

2.4.3 Hybrid Flow

The Hybrid Flow is a mixture between Implicit Flow and Authorization Code Flow. It is used in native applications or mobile applications. For example, a WebKit-based web browser for the frontend and a bundled backend service. The frontend application requires an ID Token to identify the user, as long as the token is not expired. The backend service can try to load new content, when mobile connection is available. In Figure 2.16 the horizontally dashed line between Authorization Endpoint and Token Endpoint indicates, where a frontend service can transfer tokens intended for the backend service. If the application is merely one client, no transfer is required.

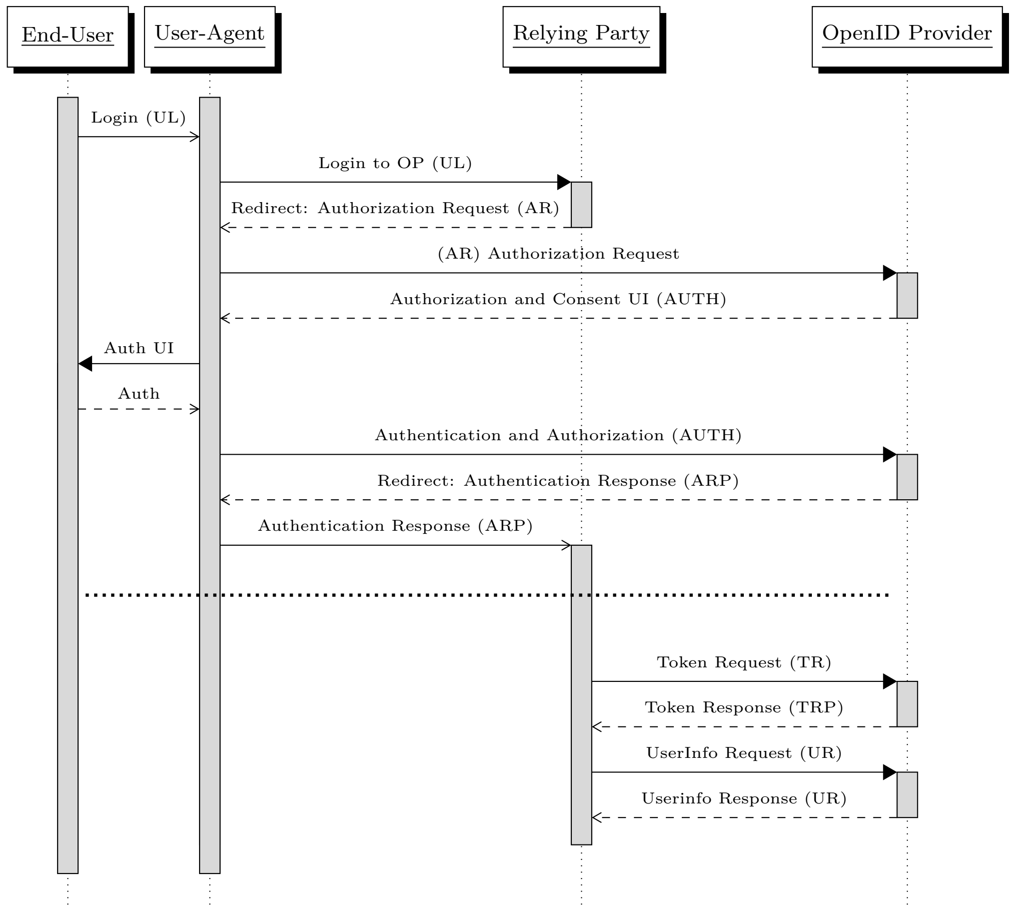

End-User
User-Agent
Relying Party
OpenID Provider
Login (UL)
Login to OP (UL)
Redirect: Authorization Request (AR)
(AR) Authorization Request
Authorization and Consent UI (AUTH)
Auth UI
Auth
Authentication and Authorization (AUTH)
Redirect: Authentication Response (ARP)
Authentication Response (ARP)
Token Request (TR)
Token Response (TRP)
UserInfo Request (UR)
Userinfo Response (UR)

Figure 2.16: OpenID Connect Hybrid Flow sequence diagram.

User Login, UserInfo, Authentication and Authorization protocol steps are equal to Authorization Code Flow 2.4.1. Other steps change as follows.

Authentication Request Analogous to Implicit Flow a `nonce` is required. Regarding Table 2.6 three combinations are available for `response_type`.

Authentication Response Regarding the requested `response_type` an Authorization Code and at least one of Access Token or ID Token is returned from Authorization Endpoint. For variants with ID Token, it is demanded that the client validates its signature and claims. If an Access Token is requested, the `at_hash` must be verified. An Authorization Code must be verified with `c_hash`. The `nonce` in ID Token should be available and equal to parameter in the Authentication Request.

3 Security Framework

In this chapter, we define our framework for the following security analysis. Starting with introducing the attacker's abilities and goals, we specify our security model. Previously related efforts have revealed several well-known attack vectors and weaknesses. We gathered these attacks in Practical Offensive Evaluation of Single Sign-On Services and considered them in manual analysis steps.

3.1 Attacker Capabilities and Goals

For the security analysis and the presented attacks in this chapter, we use the web attacker model [1]. Thus, the attacker can deploy a malicious web service with a valid TLS certificate to https://attacker.com. Attacker has full control over this service without the ability to manipulate the network traffic. Hence, the malicious web service can only send HTTP requests from an attacker-controlled network to an honest server and answer with a HTTP response. A victim must visit the attacker's site with the browser. From this point, the attacker can call browser API functionalities to load sites or open new windows. In addition, he can generate and send cross-origin HTTP requests.

The attacker and victim in our OIDC lab environment can create new accounts on every OP service. On the one side, an attacker can register his malicious RP to an honest OP manually or with Dynamic Client Registration. On the other side, he can manage to register an honest RP to his malicious OP by using the discovery extension or offer his free OP service to administrators. Further he can try a URL hijacking attack or takeover a forgotten sub-domain from a cloud service where previously an honest OP was running.

After fulfilling these preconditions, an attacker can reach several goals. One goal is to steal a valid token and redeem it at his own discretion. Therefore, an honest user must be lured to use a service under attacker's control. For example, this can be accomplished by sending a malicious link to the victim or introducing a Cross-Site-Scripting (XSS) attack and victim uses the malicious RP service to authenticate to an honest OP. Hence, an attacker can steal the token and impersonate this user. An attacker can try to login with his account to an honest OP. Afterwards by using ID Token, he tampers data in the browser to bypass security checks on an honest RP. This allows the attacker to impersonate any user. If the used digital signature, which protects an ID Token is bypassed or is not correctly validated this ID Token can be abused. An attacker might use a known key set, for example, from a malicious OP. Another goal is to bypass client authentication or to steal a client secret from a web service. This allows an attacker to use an honest OP with any malicious RP on behalf of the honest RP. Another attacker's goal might be executing arbitrary code or preparing an advanced attack to gain direct access into an honest web service.

3.2 Common Web Vulnerabilities

The following basic attacks represent an overview of attack vectors in the context of OpenID Connect. These can be applied or serve as a foundation for attacks in this chapter. OWASP Top 10 [8] lists several of these attacks. This indicates that they should be taken seriously.

Broken Authentication An attacker can directly try a brute force attack or a dictionary attack against the OP to login with credentials from an End-User. The other possibility is to steal an OIDC token or to forge a new token which an OP or RP accept.

Man-in-the-Middle (MitM) This is an attack, where an attacker places himself into the communication channel between two trusting parties. Implicit Flow and Hybrid Flow are more vulnerable to this type of attack. For example, a mobile application accesses a public WLAN Hotspot, and an attacker can manipulate the ID Token, if it is not secured properly.

Cross-Site-Scripting (XSS) Is a web application vulnerability, which tries to execute an attacker script in the victim's browser. Once executed, it steals data from local browser storage or from cookies. Furthermore, an attacker can redirect an End-User to a malicious website.

Local File Inclusion (LFI) An attacker tries to load a local file which is executed by the web server. On the one side, this discloses information if the file content is displayed. On the other side, it might execute commands or prepares a XSS attack. A similar attack is called Remote File Inclusion (RFI), where the victim server loads an external malicious resource.

Denial of Service (DoS) This attack aims to produce a high amount of load on server side and denies End-User from using a service. An attacker can try to generate huge log files or reference large files which the victim server downloads. In addition, cryptographic operations can also generate a high server load.

Server Side Request Forgery (SSRF) An attacker transforms an URL or a resource file which is indented for internal usage or communication between server. An attacker gains direct access to internal resources or prepare an advanced attack. In addition, he can perform requests towards background services that should not be exposed externally.

3.3 Single-Phase Attacks on Relying Parties

The following explained attacks are executed with modifications in a single step. One case is that developers did not adhere to recommendations in the specification, or an attacker abuses parameters to change the RP internal code flow. Another case is that the developers have overseen a flow or extension specific requirement change. If RPs do not handle parameters or error paths correctly, these attacks succeed.

3.3.1 Cross Side Request Forgery (CSRF)

The CSRF is an attack used to send malicious requests from an authenticated End-User. An attacker obtains authorization to a protected resource without the consent of the End-User [17].

Attack An attacker creates an account on behalf of the victim. Afterwards he starts the Authentication Request and intercepts the Authentication Response. The victim user must be lured to click on a callback link with the obtained Authorization Code. Without a protection, the RP follow this link and redeems the Authorization Code. Victim gains access to RP with an account owned by the attacker.

Countermeasure The RP should use `state` parameter to link an Authentication Request with the redirect URI. Therefore, the `state` parameter is bound to an End-User session. If a wrong or missing `state` is returned in the Authentication Response, the obtained tokens should not be used.

3.3.2 Token Substitution (TS)

The Token Substitution (TS) attack aims to swap tokens from an account under attacker's control to the victim's session. Tokens that are exposed in the front-channel and available in the browser can be manipulated, easily. This is known as the "cut and paste" attack [29, Section 16.11.].

Attack There are two variants for this attack.

Variant 1: An attacker eavesdrops on an Authorization Code or Access Token from another session of a victim user. Thus, the attacker swaps the received token with his session tokens. If the RP cannot verify this token, an attacker can access the victim's account functions.

Variant 2: An attacker can login to a malicious account and swaps his obtained Access Token within the victim's browser. The victim has access to the malicious account without noticing it. For example, the End-User may upload confidential files to the malicious account.

Countermeasure For Implicit Flow and Hybrid Flow the ID Token is signed. In subsection 2.3.3 the `at_hash` is described, which prevents an Access Token substitution. For, Hybrid Flow the `at_hash` should be verified to prevent an Authorization Code substitution. In addition, the RP should validate other identifiers, for example, `aud`, `iss`, and `sub`.

3.3.3 ID Spoofing (IDS)

The ID Spoofing (IDS) attack targets identity related information [18, 19]. A malicious OP tries to impersonate an End-User which is registered to another OP. Regarding subsection 2.3.3, the combination of `iss` and `sub` is used to identify an End-User.

Requirement RP is registered to honest and malicious OP.

Attack There are two variants for this attack.

Variant 1: With a malicious OP key, the attacker signs an ID Token with identifiers of an End-User registered to honest OP.

Variant 2: An UserInfo is requested by the RP and malicious OP provides identifiers of an End-User registered to honest OP. This variant bypass the signature check of an ID Token. If the RP overwrites ID Token information with data obtained from UserInfo in JSON format, this attack succeeds.

Accordingly, how a RP identifies an End-User in the database, different attack scenarios are likely to use these variants. For example, in Listing 3.1 `iss` and `sub` from an honest OP are issued in the UserInfo. Another example is to provide only `iss` and additional claim information without a `sub`.

Besides the specified identifiers, the RP might use any additional claims from the ID Token or UserInfo for End-User identification. The attack is successful, if the RP accepts provided identifiers and the attacker gained access to the honest End-User account.

```json
{
  "sub": "honest-op-user",
  "iss": "https://honest-op",
  "name": "Honest User",
  "email": "user@honest.com",
  "preferred_username": "evil-user-name"
}
```

Listing 3.1: IDS (sub+iss) with ID Token including `sub` and `iss` from honest OP.

Countermeasure A countermeasure is to strictly match the required `iss` and `sub` claim from ID Token to identify an End-User. When an UserInfo is retrieved, the `iss` and `sub` from ID Token must match inevitably. If the ID Token is signed, it should be verified that the signature key matches the corresponding issuer from Provider Configuration. Additional claims, for example, `name` and `email` should not be referenced as replacement for a missing `sub` claim to identify an End-User.

3.3.4 Replay Attack (RA)

The Replay Attack (RA) is a well-known attack vector on freshness parameters from a protocol. An attacker circumvents the time restrictions and reuses the same ID Token multiple times [19]. For example, if the `iss` is not validated appropriately (IDS), an attacker can replay a leaked ID Token with a malicious OP from another honest OP.

Countermeasure For an ID Token, the OpenID Connect specification defines three freshness claims. These are the `exp` and `iat` which are always required as explained in subsection 2.3.3. For clock sync differences between RP and OP a few minutes fault-tolerance are appropriate. This time offset must be considered for past and future.

The third claim is the `nonce` value, which must exactly match the `nonce` parameter provided in the Authentication Request. Developers must be aware that an attacker can exclude the `nonce` parameter in the Authentication Request. Thus, `nonce` in ID Token should be used as indicator for the use of this freshness parameter. For Implicit Flow and Hybrid Flow, the `nonce` is required. With a `nonce`, the one-time use requirement against replay attacks can be solved.

3.3.5 Signature Manipulation (SM)

The Signature Manipulation (SM) attack targets signature validation of an ID Token [3]. Tim McLean [20] has discovered that many libraries are allowing to bypass the Signature check. The called *Unsecured JWT* [12] grants an attacker to use the *none* algorithm. If the library supports *none* algorithm or incorrectly validates the signature, an attacker can manipulate the claims inside an ID Token. Highest risk applies to Authorization Code Flow with PKCE, Hybrid Flow, and Implicit Flow which exchange tokens via the front-channel or in an insecure public network.

Attack In Listing 3.2, the attacker creates an ID Token with the *none* algorithm to bypass the signature validation. If this attack succeeds, the attacker can impersonate any End-User from each OP known to the RP.

```
{
  "typ": "JWT",
  "alg": "none"
}
```

Listing 3.2: SM attack with `none` algorithm in JWT header.

Countermeasure A RP service must detect an invalid signature, signed with an unknown key. The *none* and any insecure cryptographic algorithm should be blocked per default. They should solely be whitelisted explicitly [21].

3.3.6 Key Confusion (KC)

Key Confusion (KC) is a further attack to break the ID Token signature validation. The attacker's goal is to use a key of his choice to verify the ID Token [18]. This attack is also known as Algorithm Substitution [3].

Attack The KC attack considers two variants. To bypass the digital signatures or using unexpected Message Authentication Codes (MACs).

Variant 1: Attacker includes an own key to the JOSE header. The JWS [11] defines fields to add a public key directly JSON Web Key (jwk) or as reference to a JWK Set URL (jku). Corresponding for X.509 public key certificates the fields X.509 Certificate Chain (x5c) and X.509 URL (x5u) exists. For example, in Listing 3.3 the attacker includes a jku and signs the ID Token with this key. If the RP requests the referenced URL and uses it for verification, this attack succeeds.

```
{
  "jku":  "https://attacker-op/untrusted-key",
  "kid": "1kb1aIINeOSs",
  "typ": "JWT",
  "alg": "RS256"
}
```

Listing 3.3: KC attack with untrusted key in `jku` inside JWT header.

Variant 2: Tim McLean [20] has investigated several libraries. He observed that the implementations using same verification function for symmetric and asymmetric operations as shown in Listing 3.4. An attacker can abuse this functionality by using the OP's public key as a shared key. The attacker can reference other keys with the method from *Variant 1*.

```
# HMAC verification
verify(clientToken, serverHMACSecretKey)
# RSA verification
verify(clientToken, serverRSAPublicKey)
```

Listing 3.4: Same verification function call for RSA and HMAC.

Countermeasure Only keys and algorithms from Provider Configuration and negotiated during Registration should be used. The Header fields in the ID Token should not use additional referenced key material for JWS or JWE [29]. RP should verify that the referenced key belongs to the OP.

3.3.7 Token Recipient Confusion (TRC)

The ID Token is intended for a specific client. With a successful Token Recipient Confusion (TRC) attack, an attacker reuses this ID Token for different clients [18, 24].

Attack The attacker creates a malicious RP. A victim must be lured to authenticate at the RP and the attacker reuses the obtained ID Token for a different client.

Countermeasure The `aud` claim contains the `client_id`. A RP must validate that this claim exists, and the `client_id` is correct. Since an OP is used for various RPs, it is mandatory that a RP client verifies the recipient information [19].

3.4 Cross-Phase Attacks on Relying Parties

This section describes a more complex attack scenario. In a Cross-Phase attack, several steps are required to start a successful attack. The following Malicious Endpoint Attack (MEA) and IdP Confusion (IDPC) attack using specification flaws. They abuse a missing connection between Authorization Endpoint and Authorization Code redemption to the Token Endpoint. Therefore, these attacks cannot be fixed without a protocol specification change. The other two Cross-Phase attacks are implementation flaws.

3.4.1 Malicious Endpoint Attack (MEA)

The idea of a MEA is to confuse and enforce the RP to send its client credentials together with a valid Authorization Code to an OP under attacker's control [19]. Implicit Flow is not affected from this attack.

Requirement A successful attack requires support of Provider Configuration and Dynamic Client Registration. With Issuer Discovery, it offers an even wider attack surface.

Attack The following steps explain the attack in Figure 3.5.

Step 1: The victim must be registered to honest OP and is tricked to login with *alice@attack-op* instead of *alice@honest-op* on the RP website.

Step 2: Regarding Listing 3.6, Registration Endpoint and Authorization Endpoint from honest OP is referenced in $Metadata_{Malicious}$.

Step 3: RP starts the registration automatically. The attack-op is a new issuer for the RP.

Step 4: RP redirects End-User to a trusted login page of the honest OP.

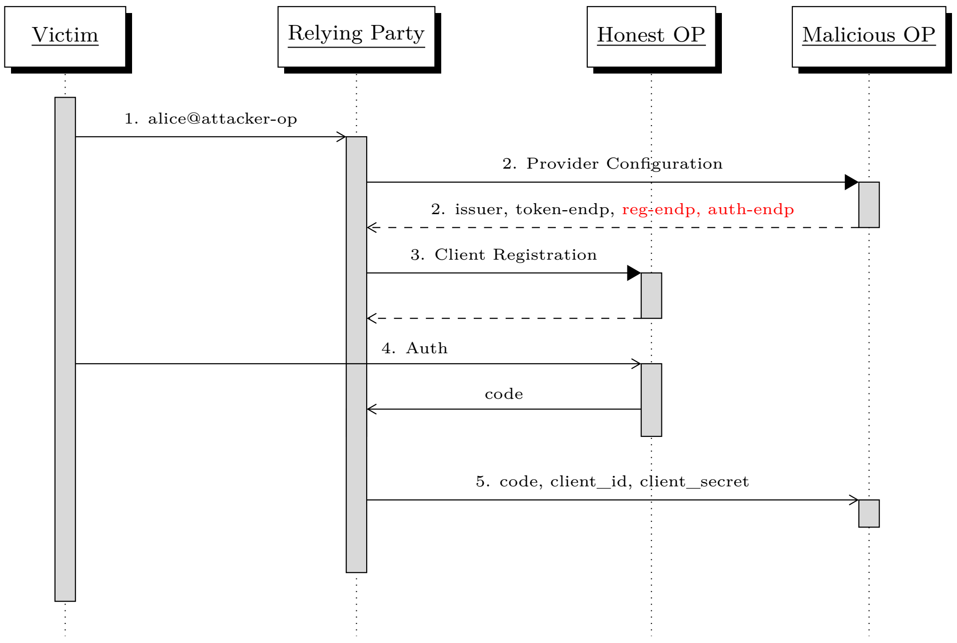

Victim
Relying Party
Honest OP
Malicious OP
1. alice@attacker-op
2. Provider Configuration
2. issuer, token-endp, reg-endp, auth-endp
3. Client Registration
4. Auth
code
5. code, client_id, client_secret

Figure 3.5: Malicious Endpoint Attack manipulates Provider Configuration.

Step 5: RP sends the Authorization Code with client_id and client_secret to the Token Endpoint, which is owned by the malicious OP.

Step 6: The malicious OP redeems the Authorization Code together with client credentials at the Token Endpoint of honest OP. Attacker impersonates the victim.

```
"issuer": "https://attack-op/",
"jwks_uri": "https://attack-op/jwks",
"token_endpoint": "https://attack-op/token",
"authorization_endpoint": "https://honest-op/authorization",
"registration_endpoint": "https://honest-op/register",
```

Listing 3.6: Discovery metadata for a MEA.

Countermeasure The countermeasure for this attack is currently in discussion and work in progress [30]. The base idea is to provide the RP information about received Authorization Code in the Authentication Response. In Listing 3.7, one can see that an additional issuer is provided in the Authentication Response. The RP should use this information as a sanity check.

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
  code=Qcb090450349sdafgAsDF-fdfsa34zdfdf5ksad890fGa8dsg
  &state=4387f0asd8435jilhkjhlkjdsaf
  &iss=https://honest-op
```

Listing 3.7: Concept to encounter a mix up attack with `iss` in Authentication Response.

3.4.2 IdP Confusion (IDPC)

IDPC is an attack which is described in previous work [19]. It uses the missing linkage between the issued Authorization Code from an honest OP and the Token Endpoint where it is supposed to be redeemed. Implicit Flow is not affected from this attack.

Requirement RP is registered at malicious and honest OP or it supports Issuer Discovery. The `client_id` must be the same for both OPs.

Attack The following steps explain the attack in Figure 3.8.

Step 1: The victim End-User visits a website under attacker's control or clicks on a malicious link. This starts a login attempt to account alice@attacker-op controlled by the attacker.

Step 2: If Provider Configuration is supported by RP, it retrieves metadata from malicious OP.

Step 3: RP redirects to the Authorization Endpoint from the malicious OP.

Step 4: Malicious OP redirects the End-User again to the honest OP but replaces `nonce` parameter.

Step 5: Victim End-User must login to his account at honest OP or is signed in automatically through previous login.

Step 6: Honest OP response with a valid Authorization Code and `state` parameter to the RP.

Step 7: With `state` parameter, RP expects that it communicates with the malicious OP. RP redeems the Authorization Code together with client credentials to the malicious Token Endpoint.

Step 8: Attacker can use the Authorization Code and client credentials to access victim's account functionalities at the honest OP.

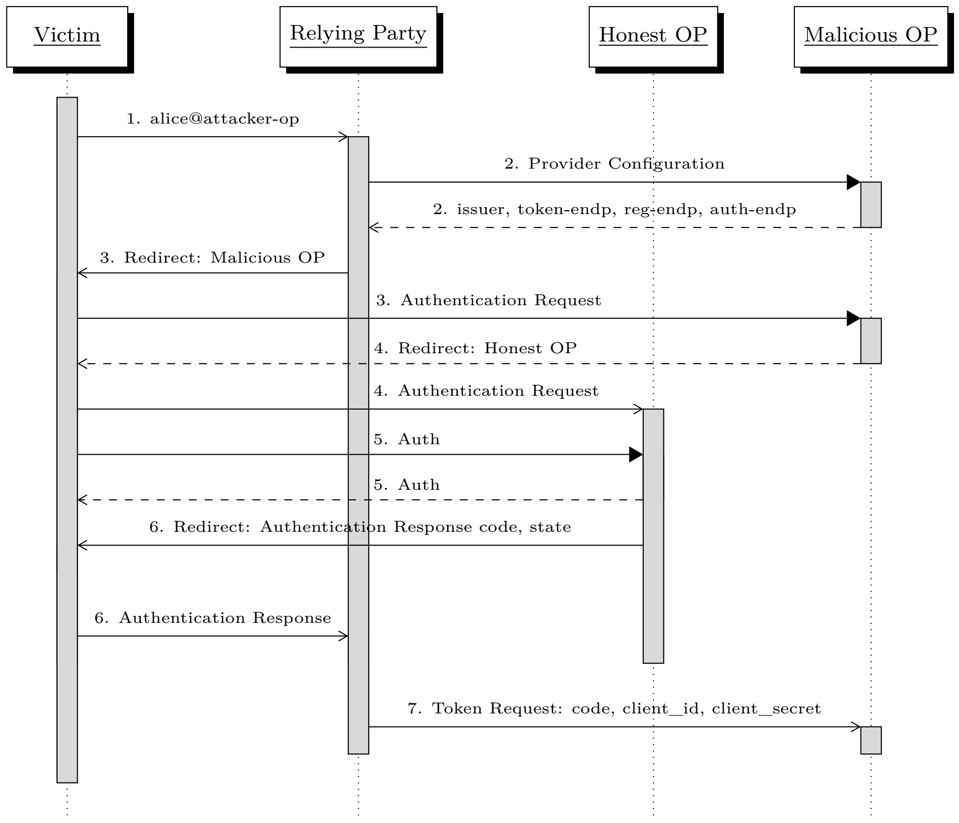

Victim
Relying Party
Honest OP
Malicious OP
1. alice@attacker-op
2. Provider Configuration
2. issuer, token-endp, reg-endp, auth-endp
3. Redirect: Malicious OP
3. Authentication Request
4. Redirect: Honest OP
4. Authentication Request
5. Auth
5. Auth
6. Redirect: Authentication Response code, state
6. Authentication Response
7. Token Request: code, client_id, client_secret

Figure 3.8: IdP Confusion attack flow diagram.

Countermeasure The countermeasure is the same mitigation as proposed in subsection 3.4.1.

3.4.3 Session Overwriting (SO)

A web application requires storing some objects between subsequent requests. The Session Overwriting (SO) targets the End-User session storage [24]. It overwrites objects stored in the browser session. The target is to manipulate the session Endpoint metadata received from Provider Configuration. An attacker abuses the session storage mechanism to redeem tokens at malicious OP instead of intended honest OP. The attacker confuses the RP to use an ID Token signed with a malicious OP key in the name of honest OP.

Requirement RP is registered at malicious and honest OP or RP supports Issuer Discovery.

Attack The SO attack considers two variants.

Variant 1: The following steps explain the SO attack in Figure 3.9.

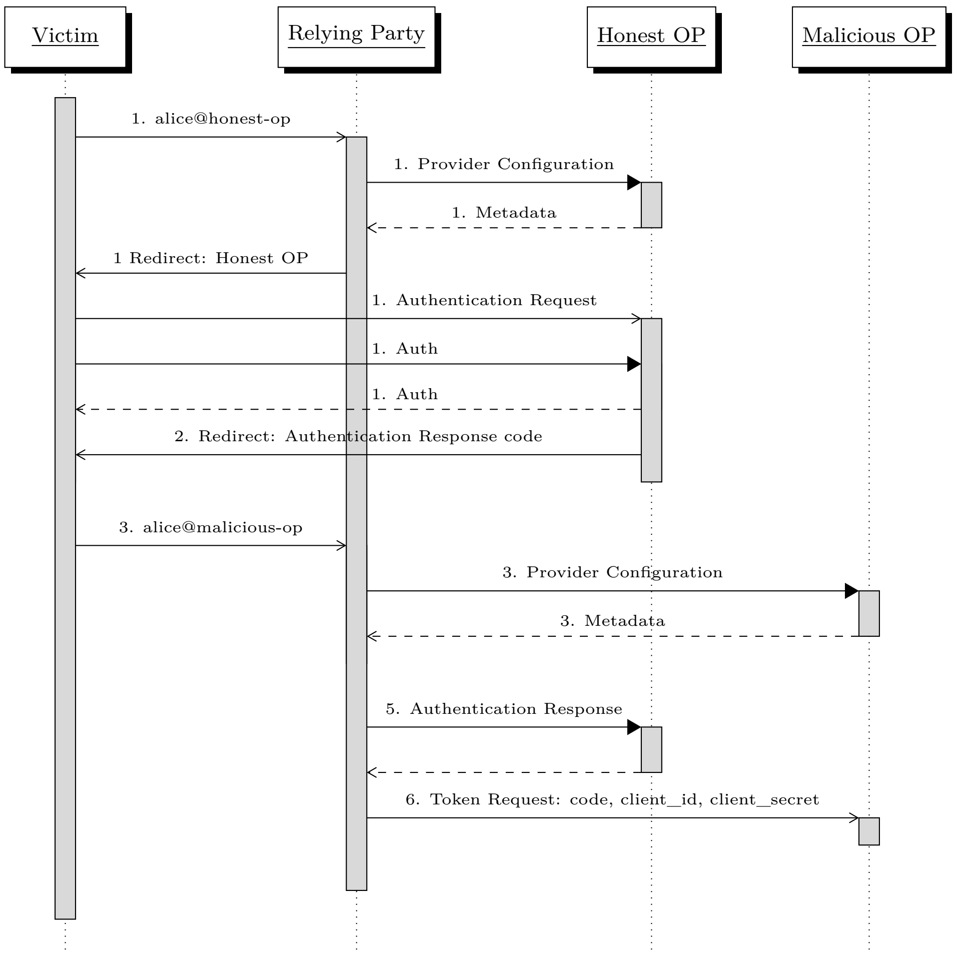

Victim
Relying Party
Honest OP
Malicious OP
1. alice@honest-op
1. Provider Configuration
1. Metadata
1 Redirect: Honest OP
1. Authentication Request
1. Auth
1. Auth
2. Redirect: Authentication Response code
3. alice@malicious-op
3. Provider Configuration
3. Metadata
5. Authentication Response
6. Token Request: code, client_id, client_secret

Figure 3.9: Session Overwriting attack flow diagram.

Step 1: The victim End-User visits a website under attacker's control or clicks on a malicious link. This starts a login attempt to account alice@honest-op and stores the $Metadata_{Honest}$ in the current session. The End-User authenticates himself at the honest Authorization Endpoint.

Step 2: The attacker intercepts Authentication Response in the browser.

Step 3: A second OP discovery is started. This time with alice@malicious-op and overwrites the current session with $Metadata_{Malicious}$.

Step 4: The attacker stops the Authentication Request to the malicious OP.

Step 5: Attacker resumes the intercepted Authentication Response to the honest OP.

Step 6: For Authorization Code Flow and Hybrid Flow, the Token Endpoint stored in the session is overwritten. The RP sends Authorization Code together with client credentials to the malicious OP. In Implicit Flow the UserInfo Endpoint can be used in case RP requests further information and uses the Access Token to access this data. For each case, the attacker gains access to the victim End-User account.

Variant 2: This combines the Key Confusion attack with Session Overwriting. The `client_id` must be the same for both OP. In Figure 3.10 the steps for Implicit Flow are depicted.

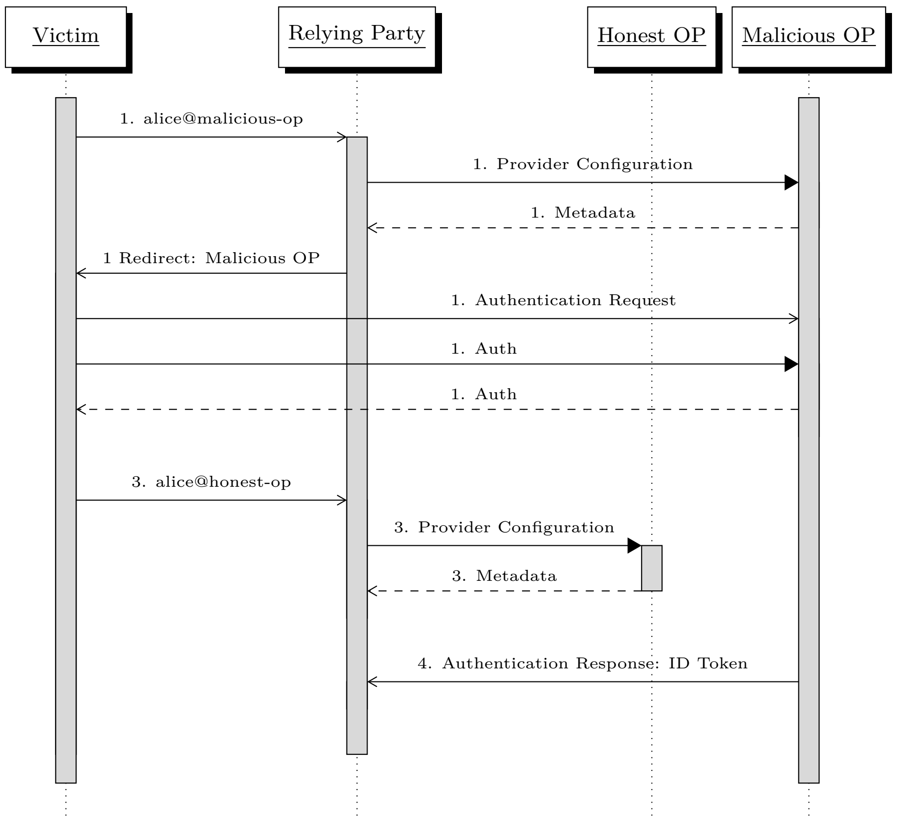


Figure 3.10: Key Confusion with Session Overwriting attack flow diagram.

Step 1: The victim End-User visits a website under attacker's control or clicks on a malicious link. This starts a login attempt to account alice@malicious-op and stores the $Metadata_{Malicious}$ in the current session.

Step 2: The attacker stops the Authentication Response from malicious OP.

Step 3: A second OP discovery is started for alice@honest-op to overwrite the session with $Metadata_{Honest}$.

Step 4: Attacker resumes the Authentication Response in step 2 with an ID Token signed with the malicious key. The ID Token is issued with `iss` and `sub` from the honest OP which are overwritten in the session through step 3. If the RP has retrieved the key material in step 1 and bound it to the session, this attack succeeds.

Countermeasure For mitigation the `state` parameter should be used. It should be bound to the session and the OP. Other appropriate CSRF countermeasures can mitigate these attacks, too. The strong authentication methods client_secret_jwt and private_key_jwt can also prevent an attack.

3.4.4 Issuer Confusion (IC)

Issuer Confusion (IC) [19] aims to confuse the RP with an ID Token and a manipulated Provider Configuration. Therefore, the attacker signs an ID Token while RP assumes it has been issued by honest OP.

Requirement RP is registered at malicious and honest OP and support Issuer Discovery.

Attack The victim starts login to alice@malicious-op. The malicious OP provides in the discovery metadata the issuer parameter `https://honest-op`. Regarding the flow, it proceeds all other steps. Finally, the malicious OP signs an ID Token with `iss` claim for `https://honest-op`.

Countermeasure The RP must verify that the issuer parameter received in the metadata matches the called URL in order to retrieve the metadata [29, Section 4.3].

3.5 Attacks on OpenID Provider

In this section we introduce seven attack vectors. These attacks abuse implementation flaws. Incorrect validation and verification steps are also a threat for Relying Parties. The most critical step is the redirect URI verification. If it is not handled correctly, the redirection can be abused to prepare several attack scenarios.

3.5.1 Open Redirector (OR)

This attack is well-known for various protocols and is described in the OAuth 2.0 framework specification [9, Section 10.15.]. The Open Redirector (OR) is using a redirect URI parameter to redirect an End-User automatically to the specified location without any validation. An attacker can abuse this for phishing attacks or to redirect End-Users to a malicious website. Furthermore, if the redirection is not validated by the OP, tokens can be leaked to a website under attacker's control.

Attack The attacker includes invalid scopes to the Authentication Request and a redirect URI of attacker's choice. This is demonstrated in Listing 3.11. Another option is to abuse a wildcard redirection to honest RP into a location where an attacker controls the site or tokens are displayed.

```
GET /authorize?
response_type=code
&scope=openid+Invalid+Scope
&client_id=s897dsafA
&redirect_uri=https%3A%2F%2Fevil.attacker.org%2Fcallback HTTP/1.1
Host: honest.com
```

Listing 3.11: Authentication Request with invalid scopes.

Countermeasure The OP should only automatically redirect to trusted redirect URI [16]. The wildcard URI should be considered as less trusted. An automatically redirection to untrusted should not be performed. The End-User may be informed if a redirection to the provided URI is intended. In an error case, the OP must validate the redirect URI. This mandatory check should not be skipped.

3.5.2 Redirect URI Manipulation (RUM)

The Redirect URI Manipulation (RUM) attack aims to lure a victim to login at an honest RP client, with a manipulated redirect URI [9, Section 10.6.]. It uses the fact that several OPs allow to register redirect URI with patterns instead of strictly matching URI. Insufficient pattern matching and wrong URI decoding enables the attacker to abuse the redirect URI. OP sends User-Agent with obtained tokens depending on the flow and endpoint to an URI under attacker's control [16].

In a mass evaluation, Lau et al. [31] have analyzed several OAuth implementations. They determined that several implementations allow domain whitelisting, prefix matching, or arbitrary schemes. Together with classic unicode attacks on URI encoding and decoding summarized in a *Unicode Security Guide* [32] they have found several flows. An attacker can try to use a redirect URI registered for other RP clients.

```
# Over-consumption
https://attacker%ff@honest.com -> https://attackernest.com
https://attacker%ff.honest.com -> https://attackernest.com
# Scheme Manipulation
attacker.com://honest.com -> https://attacker.com://honest.com
# Append TLD
https://honest.evil-tld/callback/
https://honest.evil-tld.com/callback/
# Subdomain
https://evil-sub.honest.com/callback/
# Path
https://rp.honest.com/callback/additional-path
# Userinfo
https://rp.honest.com/callback:pass@rp.evil.com/callback -> rp.evil.com/callback
https://rp.honest.com:pass@rp.evil.com/callback/callback -> rp.evil.com/callback
# Parameter Pollution
redirect_uri=https://rp.honest.com/callback&redirect_uri=https://rp.attacker.com/callback
```

Listing 3.12: Overview of several RUM variants using different manipulations.

Attack There are the following two variants of this attack.

Variant 1: In a RUM on Authorization Code Flow, the attacker sends the victim an email with the manipulated Authentication Request. Therefore, the attacker can use a pattern shown in Listing 3.12. If an attacker manipulates the Token Request, RUM can be applied also against the Token Endpoint.

Variant 2: RUM on Implicit Flow can be applied against the Authorization Endpoint, as in Variant 1. In addition, a further attack can be utilized with the special fragment handling. It assumes the registered URI pattern *client.honest.com/cb?**. If this callback supports a *redirect_to* as Open Redirector, an attacker can abuse it. First, the victim follows the link shown in Listing 3.13.

```
GET /authorize?response_type=token&state=9ad67f13
  &client_id=honest-id
  &redirect_uri=https://rp.honest.com/cb?redirect_to=https://attacker.com/
```

Listing 3.13: RUM in Implicit Flow, for readability without HTTP encoding.

Second regarding Listing 3.14, the OP responses with the Open Redirector URI. Finally, the RP redirects to the attacker's site and can get the Access Token and ID Token from the fragment.

```
https://rp.honest.com/cb?
  redirect_to%3Dhttps%3A%2F%2Fattacker.com%2Fcb
  #access_token=a78a9sdf987asdR&id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeDt5Qu ... ZDso
```

Listing 3.14: Authentication Response with *redirect_to*.

Countermeasure The OP should only allow strict matching redirect URIs, and wildcard redirects should be avoided. RP clients should not expose an OpenID Provider at callback address [16]. The only exception can be port numbers required for native apps which are using a *localhost* address [2].

3.5.3 Authorization Code Reuse and Substitution

This attack validates the one-time usage of the Authorization Code. It checks the link between client and Authorization Code [29, Section 16.9, 16.11].

Authorization Code Reuse: The attacker reuses a previously obtained Authorization Code. For example, this can be an Authorization Code from server logs or browser history.

Authorization Code Substitution: Regarding TS attack the attacker swaps the Authorization Code.

Countermeasure An Authorization Code must be used only once for a specific client. In case of an error, the Authorization Code must be revoked. Previously issued ID Token and Access Token can be revoked. This should be considered for Hybrid Flow.

3.5.4 Client Authentication Bypass (CAB)

The Client Authentication Bypass (CAB) is a preparation to redeem an Authorization Code at the Token Endpoint. If an attacker steals an Authorization Code, it is required to present the client credentials an OP. This attack aims to bypass or downgrade the authentication method. The attacker's goal is to redeem the Authorization Code with malicious RP.

Requirement Malicious and honest RP are registered to the same OP.

Variant 1: The authentication methods `client_secret_basic` and `client_secret_post` hold the same security property. An attacker issues invalid or missing values for `client_id` and `client_secret` to both authentication methods.

Variant 2: If the JWT JOSE header is not properly secured, the attacker can start the same bypass attack as described in subsection 3.3.5. For example, the attacker applies algorithm `none` to bypass the signature validation.

Variant 3: In an Authentication Method Confusion (AMC) the attacker tries to use another client authentication, which is less secure. The attacker redeems Authorization Code with Basic Authentication instead of a JWT Authentication Method.

Countermeasure An OP should not allow to use authentication methods with different security properties. OP should only allow the registered methods, and error paths must be handled properly. It is recommended to use asymmetric methods for client authentication [16].

3.5.5 Message Flow Confusion (MFC)

The `response_type` and the optional `response_mode` parameter exists for the Authentication Request. An attacker manipulates these parameters for a Message Flow Confusion (MFC) attack. The attacker obtains tokens not intended for the specific flow or RP cannot handle the mode properly. This attack can reveal sensitive data or bypass other security features. For instance, PKCE cannot be applied in an enforced Implicit Flow.

Variant 1: An attacker uses other `response_type` values as negotiated during the client registration. If it is not properly handled by the OP, an attacker changes from Authorization Code Flow to Implicit Flow. Here, the Access Token and ID Token are exposed in the Authentication Response.

Variant 2: The parameter `response_mode` is used to change the mode between `query`, `form_post` and `fragment`. In Implicit Flow, the attacker uses `query` to expose the tokens into query parameters. The attacker can directly read these tokens in the browser history.

Attack The following variants can be combined to expose tokens in a channel under the supervision of an attacker. To attack a RP registered with Authorization Code Flow, the attacker creates a link with parameters in Listing 3.15.

```
GET /authorize?
  response_type=id_token%20token
  &response_mode=query
  &client_id=s897dsafA
  &redirect_uri=https%3A%2F%2Fclient.honest.org%2Fcallback
  &scope=openid%20profile HTTP/1.1
  Host: honest.com
```

Listing 3.15: Authentication Request link preparation to change to Implicit Flow and expose tokens as query parameters.

If this attack is successful, the OP responses with Listing 3.16. One can see `query` (?) mode is used instead of `fragment` (#). The Access Token and ID Token are provided instead of the Authorization Code.

```
HTTP/1.1 302 Found
Location: https://client.honest.org/callback?
  token_type=bearer
  &access_token=SlAV32hkKG
  &id_token=eyJ0 ...  NiJ9.eyJ1c ...  I6IjIifX0.DeXT4Qu ...  ZXso
  &expires_in=3600
```

Listing 3.16: Successful MFC attack with ID Token and Access Token in query parameter.

Countermeasure An OP should allow solely one flow per client instance. A client registered for Authorization Code Flow cannot request a `response_type` other than `code`. The parameter `response_mode=query` is not allowed for Implicit Flow [22]. Access Token and ID Token must not pass in a URI query [16]. The `form_post` can be used alternatively.

3.5.6 PKCE Downgrade Attack (PDA)

The PKCE is an additional security feature, introduced in subsection 2.4.1. PKCE Downgrade Attack (PDA) can remove this feature or allow the attacker to use the simpler *Plaintext* method. An attacker manipulates the flags `code_challenge` and `code_challenge_method` in the Authentication Request [16]. Both parameters are available in the front-channel with purpose to signal that PKCE is supposed to be used. For mobile application which moved from Implicit Flow to Authorization Code Flow with PKCE this means no additional security is provided.

Attack The attacker removes `code_challenge` from Authentication Request. In case, OP does not validate that `code_verifier` is issued in Token Request this attack succeeds. If this is not possible, the attacker can change `code_challenge_method` to *Plaintext* and store the `code_challenge`. Afterwards the attacker intercepts the Token Request of honest RP or being faster in redeeming the Authorization Code. With a malicious RP, the attacker can redeem the Authorization Code and issues the stored Code Challenge as Code Verifier.

Countermeasure The OP must detect a missing `code_challenge` parameter in the Authentication Request, if a `code_verifier` is supplied in the Token Request [16]. The Code Challenge should be bound to the issued Authorization Code. As a protection against eavedroppers, `code_challenge_method` should not allow downgrade to Plaintext mode [26].

3.5.7 Sub Claim Spoofing (SCS)

The attack Sub Claim Spoofing (SCS) abuses optional `claims` parameter in the Authentication Request. It aims to pass additional claims identifying an End-User to the requested ID Token. If an OP is not handling the `sub` claim correctly, an attacker can impersonate an arbitrary End-User [24].

Attack The attacker requests in the `claims` parameter `sub` of a victim shown in Listing 3.17. An attacker includes the `sub` as an array with a malicious and honest user to bypass the validation. Other claims might be also used for this attack to overwrite claims in the ID Token.

```
{
  "userinfo":
  {
    "given_name": {"essential": true},
    "nickname": null,
    "email": {"essential": true},
    "email_verified": {"essential": true},
  },
  "id_token":
  {
    "sub":  "value":  "Victims-Sub"
  }
}
```

Listing 3.17: SCS example with victim sub in claims request.

Countermeasure Misunderstood claim members must be ignored. The OP should not add additional claims to the ID Token without verification that the claim is valid in this request scope [29, Section 5.5].

4 Lab Environment

For the security analysis, several RP and OP services should run in a secured lab environment. A containerization technology offers the opportunity to isolate each service. It also reduces issues with different build dependency requirements. Due to current popularity Docker is used for OS-level virtualization. The docker containers provides reproducible results and a maintainable test environment for the following security analysis. Docker is well documented, stable, and examples are available for several RP and OP services.

The created lab framework must serve several preconditions. It must provide a possibility to execute manual security analysis and automatic analysis with PrOfESSOS. The docker server runs either on a developer machine or on a server in a data center. Services inside the containers must be reachable without a complicated client setup for each lab user. Different service configuration and target versions should run in the lab environment. This is archived with build and run-time environment variables. In addition, it should be possible to reset every service to an initial state, to ensure there are no errors from previous tests.

The following sections explain the lab structure and the purpose of each component.

4.1 Selection of OpenID Connect Implementations

We have selected *OpenID Certified* implementations, which claims to support Implicit Flow, Hybrid Flow, or both flows. This reduces the amount of valid target implementations to the scope of this thesis. It provides an up-to-date overview about certified implementations that delivers a higher trust relationship.

Another limitation is that the services must run On-Premise and without a license fee. Several OP implementations offer a Software as a Service platform and cannot be integrated into a lab environment.

The most RP and OP services can be deployed on Linux based operation systems. Therefore, Windows exclusive implementations are out of scope to reduce initial lab complexity.

4.1.1 Selected Relying Parties

The listed certified RPs [6] providing libraries or example implementations. We selected the RPs in Table 4.1, which fulfills the previously defined selection criteria. For local development and testing, we used Okta as a SSO provider service with OIDC support. This gives the required stability and reduced the overhead to test the implementations.

				Flow		
Name	Language	Version	Discovery / Dynamic Registration	Code	Implicit	Hybrid
angular-auth-oidc-client	Angular	10.0.15	no / no	yes[a]	yes	no
angular-oauth2-oidc	Angular	9.2.2	no / no	yes[a]	yes	no
express-openid-client (node openid-client)	Java-Script	1.0.1 (3.14.1)	no / no (yes / yes)	yes (yes)	yes (yes)	yes (yes)
MITREid Connect	Java	1.3.1	yes / yes	yes	no	no
mod_auth_openidc	C	2.4.2.1	no / yes	yes	yes	yes
phpOIDC	PHP	rev166	yes / yes	yes	yes	yes
pyoidc	Python	1.1.2	no / yes	yes	yes	yes
OidcRP	Python	0.6.10	no / yes	yes	yes	yes

Table 4.1: Overview about selected Relying Parties.

[a]Authorization Code Flow with PKCE is supported.

In particular, both angular libraries offer documentation to create a web application. The developers still work on a first release with full Authorization Code Flow and PKCE support. They are not certification ready, yet. Programmers already recommend using this flow in the future, instead of insecure Implicit Flow. The client is configured in both implementations during compile time and limited to one OP.

The *node oidc-provider* implementation cannot be used directly. An application developer must write the security related checks. Therefore, we decided to use *express-openid-client* as a Middleware, that is based on *node oidc-provider*. The downside of this decision is the missing support of Issuer Discovery and Dynamic Client Registration. The service provides client configuration in a static built-in configuration file. We configured the client to use Implicit Flow (ID Token).

Apache server module *mod auth openidc* can be installed without issues. It provides user claims in the header information after a successful login. We implemented a custom minimal PHP page to retrieve and display this data.

The *pyoidc* provided example implementation is outdated. It cannot be used in its current state. Instead, we use a self-written flask example, with a dependency module *Flask-pyoidc* which is based on *pyoidc*. A static configuration for different OPs is possible. The Dynamic Client Registration in this flask module has a bug in the current version. It requires a workaround to support Dynamic Client Registration in the flask module implementation. The redirect URI registered during registration phase is inherited from the Provider Configuration response. Hence, there is no validation anymore. This change creates a security flaw which must be considered in the security analysis. Another bug occurs during the client registration. The client always registers with Authorization Code Flow. Subsequently, the intended flow must be corrected on server side.

Developers of *OidcRP* provide a running client sample. Implicit Flow and Hybrid Flow are both not supported. The developers partially implemented it. An essential part for fragment parsing and different token handling is not implemented. Consequently, the client is solely configured for Authorization Code Flow.

The *phpOIDC* client implementation runs out-of-the-box. If an OP supports Issuer Discovery, it can be tested with this implementation. The client can be registered and used with all flows.

MITREid Connect client is a reference implementation to test PrOfESSOS. It does only support Authorization Code Flow. In addition, the client supports Issuer Discovery and Dynamic Client Registration. We use this stable client to test the other OPs.

4.1.2 Selected OpenID Providers

An OP service requires implemented user and client management in order to carry out a comprehensive security analysis. In most cases this demands a database or at least several configuration files. A service implementation should provide this among the required OIDC features. Through the requirements and that they are often acting in the background for normal users, there are more full featured implementations available than library only implementations.

Nine implementations meet the requirements. Among them, five services can run in the lab environment with no time-consuming adaptions. The *MITREid Connect* server considered as reference implementation to test PrOfESSOS is included. *Gluu Server* and *OIDC OP Overlay for Shibboleth IdP* plugin are not included, due to complex configuration requirements. *SimpleIdentityServer* is deprecated and the new fork is currently not certified. The new client cannot be compiled anymore with a Linux based docker container. Documentation mentions a Windows VM in the Azure Cloud. Although *pyoidc OP* is not included. The example server is outdated and is not running in the current major version.

All listed implementations in Table 4.2 supports Provider Configuration and Authorization Code Flow with PKCE.

				Flow		
Name	Language	Version	Dynamic Client Registration	Code	Implicit	Hybrid
IdentityServer4	C#	3.1.3	no	yes	yes	yes
Gravitee.io Access Management	Java	3.0.2	yes	yes	yes	yes
Keycloak	Java	10.0.2	yes	yes	yes	yes
MITREid Connect	Java	1.3.3	yes	yes	no	no
node oidc-provider	Java Script	6.17.3	yes	yes	yes	yes
phpOIDC	PHP	rev166	yes	yes	yes	yes

Table 4.2: Overview about selected OpenID Providers.

IdentityServer4 offers a lightweight framework to create an OP service. It provides example implementations alongside the source code. Clients and users are stored directly in source files and creates an In-Memory-Database. Implicit Flow and Hybrid Flow can be configured. The required functionality is not included in the example implementation.

The full featured production ready OP platforms *Gravitee.io Access Management* and *Keycloak* are shipped as docker images. Both use databases to store persistent data about users and clients. They offer an administrative dashboard to configure allowed flows and security specific options. These configurations can be enrolled for every client or as a template for newly registered clients if Dynamic Client Registration is allowed.

Node oidc-provider offers an example OP server. User accounts are configured in the source code. A real password check is currently not implemented into this server. It simply checks whether a user exists, but a provided password is not validated.

Finally, the *phpOIDC* server is an example implementation. We configure it to allow client registrations and added pre-defined users in a MySQL database. Initial server settings are configured in a PHP include file.

4.2 Docker Test-Environment

The OP implementations are relevant for the structure of our docker lab environment. Production ready containers offer load balancing and are distributed in several sub services. The *Gluu Server*, *OIDC OP Overlay for Shibboleth IdP* plugin, and *Gravitee.io Access Management* server are such OP services. In the *Gravitee.io* nginx reverse proxy configuration in Listing A.1, one can see that it involves three web services. To interact with this server, it is required to add DNS entries in the local host file or DNS server. These entries must match the reverse proxy settings. Both DNS setting actions require administrative privileges. Configuration of further DNS entries are limited to one lab server. In order to run different configurations on several servers, the DNS names requires to be changed in every configuration file. A security aspect must be considered. With the lab environment, potentially insecure configured applications can run in the institution or company's network.

4.2.1 Network Topology and Configuration

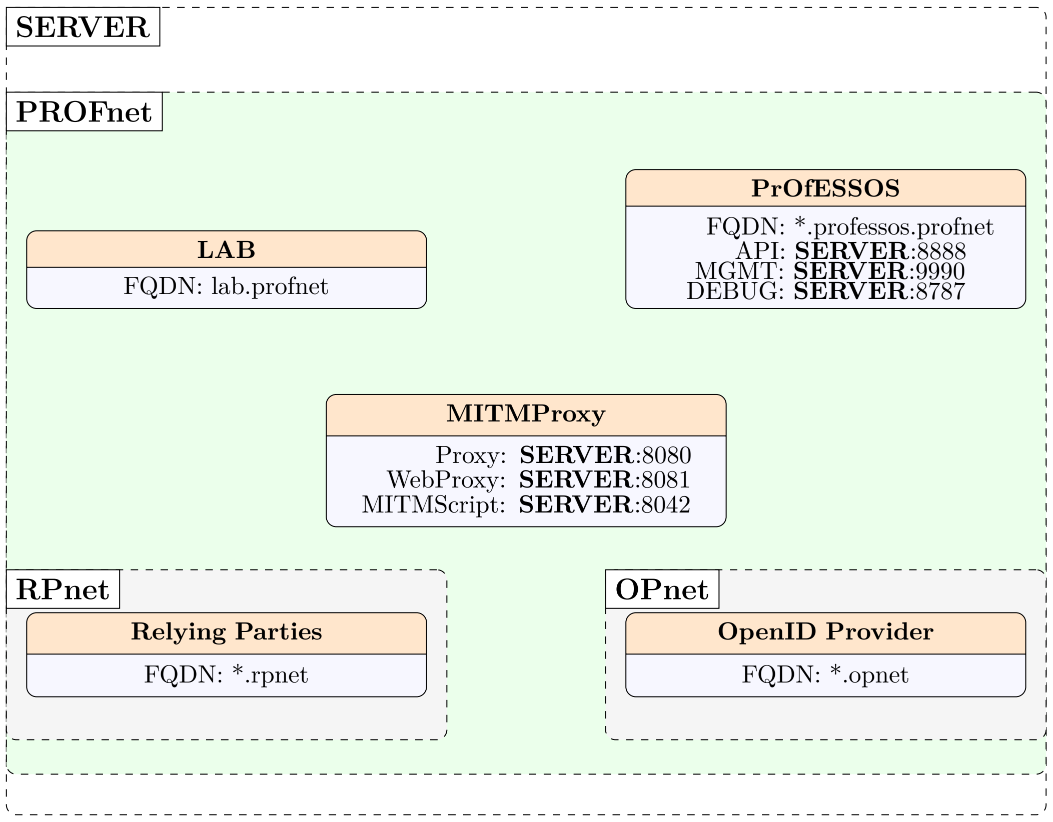
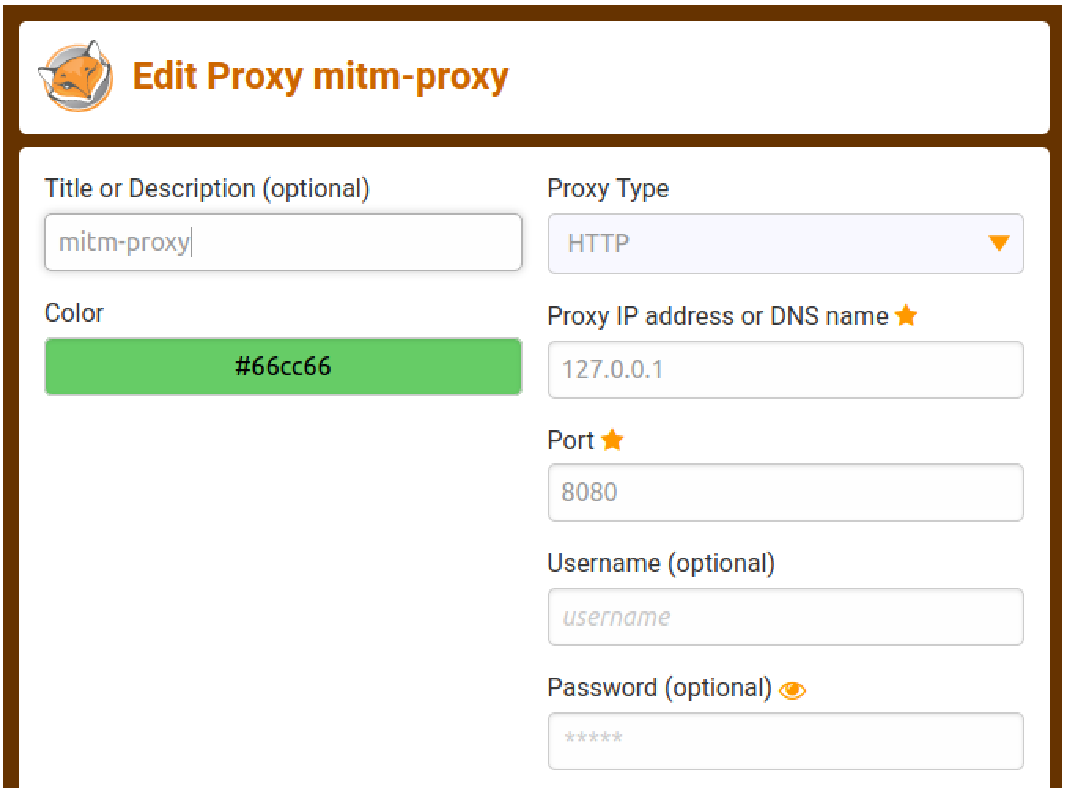


Figure 4.3: Schematic docker-compose overview.

We created an encapsulated environment to simulate a complete OpenID Connect network. Therefore, a docker-compose file defines three main networks depicted in Figure 4.3.

First network is *PROFnet*, intending to provide all non OIDC services like PrOfESSOS and the lab landing page. Web services running in this network can expose ports to the server through a bridged network interface. The network intefaces for the second *RPnet* and third *OPnet* are configured as internal networks. With these internal network interfaces, RPs and OPs are completely separated. The MitMProxy service acts as a proxy between all three networks, as explained in subsection 4.2.4. It is the preferred gateway to interact with the lab as an external user. As a requirement, the browser must be configured to use a web proxy. For example, the Firefox plugin FoxyProxy can be installed and parameterized as shown in Figure 4.4. Subsequently, all three networks communicate via this proxy and a user can access all web services with a web browser.


Edit Proxy mitm-proxy

Title or Description (optional): mitm-proxy

Color: #66cc66

Proxy Type: HTTP

Proxy IP address or DNS name: 127.0.0.1

Port: 8080

Username (optional): username

Password (optional): *****

Figure 4.4: FoxyProxy browser plugin configuration for MitMProxy.

These measures reduce the initial setup overhead for all users. A similar configuration can be used in a data center and for a development machine. A restricted server access can be realized through a VPN gateway and additional firewall port rules.

4.2.2 Service Integration

All services are gathered in a single *docker-compose.yml* file. The docker-compose allows to control which services is supposed to run. Without specifying a service, the complete lab starts.

```
$> # start complete lab
$> docker-compose up -d
$> # start only mitmproxy, professos and mitreid-server
$> docker-compose up -d mitmproxy professos mitreid-server mitreid-client
```

Listing 4.5: Commands to start lab with docker-compose.

If a working client or server is provided, an exact version can be referenced. This is configured by argument `BRANCH`. Alternatively, without a version tag, the revision `HASH` can be used. Another option is to set the docker image version for specific images. These images can be retrieved via docker hub or any other docker registry.

```yaml
version: "3.6"

services:
  rp-example-client:
    build:
      context: ./rp/rp-example-client
      args:
        ISSUER: ${HONEST_OP_HOST}
        BRANCH: master
        HASH: 7f4819b7107b069b50bb1b2f2ff2acb5e296c2d9
        CONTROLLER_URL: ${CONTROLLER_HOST}
        CLIENT_HOST: ${UNIQUE_CLIENT_HOST}
    depends_on:
      - certs
    volumes:
      - certs:/certs:ro
    env_file:
      - .proxy_env
    environment:
      CA_DIR: ${CA_DIR}
      CA_CERT: ${CA_CERT}
      VIRTUAL_HOST: ${UNIQUE_CLIENT_HOST}
    networks:
      - rpnet
```

Listing 4.6: Example structure of a docker-compose file.

Another crucial step is shown in Listing 4.6. The SSL certificate required for the communication is generated automatically. A *certs* service runs in the background and listens on the docker socket. The environment variable `VIRTUAL_HOST` is the indicator to create new certificates. The self-generated Root-CA file from the certs volume is installed into every service. To use this proxy as user without self-signed warnings, it is recommended to install the Root-CA into local browser store. Per default, MitMProxy is enabled with the environment file *.proxy_env* in every service. If a service cannot handle proxy settings or the feature is not demanded, it is possible to remove the file. It is solely required to attach the opposite party network to allow direct communication.

We tried to deliver our provided Dockerfiles in a similar generic structure, as shown in Listing A.2. Every service provides its own reverse proxy and is configured to use the Root-CA file. On the one side, this grants the flexibility to start certain services or change the network configuration as previously described. On the other side, it simplifies the integration of further implementations. The build and run steps are in most cases the same for each programming language.

To test different configurations, it is possible to change the configuration files and build the lab again. Another option is to set up different container configurations with environment variables. In Listing 4.7 this option is demonstrated. With the `ISSUER` environment variable, the used OP can be changed. Through the build argument, the Dockerfile selects a respective configuration file.

```
express-openid-client:
  build:
    context: rp/express-openid-client
  args:
    CONTROLLER_URL: ${CONTROLLER_HOST}
    CLIENT_HOST: ${EXPRESS_OPENID_CLIENT_HOST}

express-openid-client-prof:
  build:
    context: rp/express-openid-client
  args:
    ISSUER: ${EVIL_OP_HOST}
    CONTROLLER_URL: ${CONTROLLER_HOST}
    CLIENT_HOST: ${EXPRESS_OPENID_CLIENT_PROF_HOST}
```

Listing 4.7: Express openid client configured for two different OPs.

4.2.3 Practical Offensive Evaluation of Single Sign-On Services

PrOfESSOS is an Evaluation as a Service (EaaS) security tool [19]. It offers a web interface to start the evaluation of a targeted RP or OP. For tests with RPs, it provides an honest OP and an attacker OP. To test an OP, PrOfESSOS starts two RPs analogously. With a selenium script, user interactions are simulated to trigger a login attempt. The created OP or RP instances are configured per test case through a compiled-in test plan.

The lab environment provides the opportunity to test and improve PrOfESSOS. Through the different OP and RP implementations, authentication methods, and flow variants, it assists to identify edge cases which are rarely tested. For debugging and hot-swapping of PrOfESSOS it is possible to expose the required ports 9990 and

8787 (s.a. Figure 4.3) to a development machine. Most PrOfESSOS changes are related to non-specification compliant service implementations. These are required fields or parameters which are not set by applications. The others are optional fields and parameter which are expected to be set. We have assessed that some fallback mechanisms need to be added for these cases. Another mandatory change was to support the proxy settings to use MitMProxy.

During tests with PrOfESSOS and RP services, we have discovered that the following three libraries require an available route to the configured OP metadata endpoint. The *pyoidc* implementation requests the Provider Configuration on app start. Both angular applications require the information, while the browser loads the respective content. Automatic tests for PrOfESSOS are configured to check the consent flag first. Afterwards, it loads the login page and starts the simulation OPs for a specific test. This does not work for the three mentioned RP web applications. Therefore, as an enhancement, we implemented the API call *expose*. A test specific OP simulation is loaded through the REST API call and the Provider Configuration is provided. Eventually, an automatic test can be started without an issue. With this improvement, it is possible to manually test a loaded Single-Phase attack. Therefore, custom test cases are included in the test plans.

4.2.4 Man-in-the-Middle Proxy

We use the MitMProxy as a default gateway for users and to connect all services. Primarily it is intended to use this proxy for debugging and analytical purposes. The Mitmweb package allows to display and decode every web traffic exchanged between all connected networks and the browser. User interactions in front-channel and communication between services in back-channel can be inspected, modified, and intercepted. The web interface is accessible on server port 8081.

Through the possibility to intercept and modify the traffic, it is possible to fix minor handling issues without being forced to alter service implementations. MitMProxy provides a python API as scripting interface. We added a script (Listing B.1) to change requests and responses. This script allows to perform manual tests with support of PrOfESSOS. This enhancement script for PrOfESSOS is a TCP server which is loaded by MitMProxy and listens on port 8042. It is controlled with a client included in the following described Command-line Interface (CLI) tool.

4.2.5 Command-line Interface Tool

The REST API interface of PrOfESSOS provides additional configuration options. For this purpose, we created a python based CLI tool. It can store configuration file used to configure PrOfESSOS via REST API. This configuration file defines a target page and a selenium script to login and display the profile page. It provides the same JSON format as the Web-UI and can be used interchangeably. One can see, the base structure in Listing 4.8 for an RP. The CLI replaces the placeholder `CHANGE_TEST_ID` in case of a dynamic test id. In the Web-UI it must be replaced manually.

```
{
  "UrlClientTarget": "https://target-url/",
  "InputFieldName": "identifier",
  "SeleniumScript": "",
  "FinalValidUrl": "https://target-url",
  "HonestUserNeedle": "{sub=honest-op-test-subject, iss=https://
     honest-idp.professos/CHANGE_TEST_ID}",
  "EvilUserNeedle": "{sub=evil-op-test-subject, iss=https://attack-
     idp.professos/CHANGE_TEST_ID}",
  "ProfileUrl": "https://target-url/profile"
}
```

Listing 4.8: Example test configuration for RP required by PrOfESSOS.

The CLI has additional logic to alter the REST API control flow and to set additional settings. It is implemented to store optional configurations per project. A static `test id` can be set. This is required when a RP or an OP cannot handle a unique dynamic id. For example, the *mod auth openidc* supports Dynamic Client Registration. This client requires storing the metadata from a pre-configured OP in the Apache2 plugin directory. With this limitation, the Dynamic Client Registration can be used if the `test id` is static.

Before a test starts, the `pre_expose` switch in Listing 4.9 can expose the Provider Configuration. The comma separated list `skip_tests` allows to disable specific test steps.

```
{
  "test_id": "yourStaticTestId",
  "skip_tests": "2,3,5",
  "discovery": false,
  "dynamic": true,
  "pre_expose": false
}
```

Listing 4.9: Optional settings file for command-line interface.

For all these stored settings, automated tests for all services can be executed. A full test is started with these commands:

```
$ ./cli.py
$> load rp mitreid-client
$> full_test
```

After all tests are accomplished, a HTML report is generated. The report contains all findings, a description, and detailed step-by-step information, including screenshots.

Instead of a full test run, a single test run can be prepared or executed. This is essential for manual penetration tests. A specific RP test can be prepared in PrOfESSOS via the expose mechanism. Afterwards, the honest OP and attack OP can be used directly with the RP. Hence, a RP can be debugged and tested directly from a browser, with the limitation that only front-channel communication can be changed. Being aware which test is loaded, a MitMProxy script can create permutations of an attack. The client library and scripts to control the MitMProxy server are delivered as a part of this CLI tool. Command line steps to start semi-automated test with a report:

```
$ ./cli.py
$> load op server
$> create
$> learn
$> run_pyscript pentest/server-redirect.py
$> run 48
$> export
$> report
```

It must be considered, that PrOfESSOS itself is unaware of these manipulations. For instance, if a redirect URI is manipulated with a MitMProxy script, the PrOfESSOS report contains the original URI.

5 Security Analysis

Motivated by specification violations, missing examples, and documentation gaps encountered during integration into the lab environment; we identified various security flaws. These flaws are described in this chapter. The MitMProxy allowed us to read and verify front and back-channel communication. A check of any PrOfESSOS steps with manual test scripts becomes redundant. The new PrOfESSOS feature, to expose an OP configured for a specific test, supported us to perform comprehensive steps directly in the browser. The MitMProxy scripting interface, introduced in section 5.3, shows how automated tests with PrOfESSOS can be extended seamlessly by manual test steps.

5.1 Evaluation of Relying Parties

Despite the OpenID Certification of the used implementations, we discovered implementation flaws. Table 5.1 shows that five out of eight implementations were vulnerable against the attacks. We revealed two potential critical vulnerabilities, with Signature Manipulation and Key Confusion. These implementation flaws allow to take over every registered account at a RP. In the node client and both angular implementations, we could not detect any security issue. However, configuration is limited to one OP and these clients neither support, Issuer Discovery nor Dynamic Client Registration.

As expected, all five RPs were vulnerable against the Cross-Phase Attack IdP Confusion. In addition, the Malicious Endpoint Attack turned out as supposed for *MITREid Connect* and *phpOIDC*, which supports Issuer Discovery. The discovery feature is not supported by *mod auth openidc*. Therefore, it is solely vulnerable, if an administrator has explicitly added the attackers OP as trusted source.

No implementation was vulnerable against the Single-Phase attacks, Cross Side Request Forgery and Token Substitution. The Cross-Phase attacks, Session Overwriting and Issuer Confusion could also not be applied to any of them.

The IDS attack against *MITREid Connect*, *mod auth openidc*, and *phpOIDC* could be applied partially. All provided clients used the `sub` and `iss` from ID Token to lookup for an account. We could solely influence displayed name and email address with the manipulated UserInfo Endpoint. There is a risk, if an unaware application developer uses the email address or other claims instead of `sub` to create a database

Flaw	MITREid Connect	mod auth openidc	phpOIDC	pyoidc implicit / code	OidcRP
ID Spoofing Userinfo (sub+iss)	✓	✓	✗	✓/ ✓	✓
ID Spoofing Userinfo (name+username+email)	✗	✗	✗	✓/ ✓	✓
Key Confusion jku Spoofing	✓	✓	✗	✓/ ✓	✓
Key Confusion jku Spoofing [untrusted,tusted]	✓	✓	✗	✓/ ✓	✓
Key Confusion jku Spoofing [trusted,untusted]	✓	✓	✗	✓/ ✓	✓
IdP Confusion	✗	✗	✗	✓/ ✗	✗
Malicious Endpoint Attack	✗	✗	✗	✓	✓
Replay Attack id_token.iat: 1 day in future	✓	✓	✓	✗/✗	✗
Replay Attack id_token.iat: 1 year in future	✓	✓	✓	✗/✗	✗
Replay Attack id_token.nonce: invalid value	✓	✓	✓	✗/ ✓	✓
Replay Attack id_token.nonce: excluded	✓	✓	✓	✗/ ✓	✓
Signature Manipulation header.alg = none & invalid sig.	✓	✓	✓	✓/✗	✓
Signature Manipulation header.alg = none & no sig.	✓	✓	✓	✓/✗	✓
Token Recipient Confusion id_token.aud: invalid value	✓	✓	✓	✗/✗	✓
Token Recipient Confusion id_token.aud: otherOP.aud	✓	✓	✓	✗/✗	✓

✓: Attack failed/Secure; ✗: Attack successful/Insecure

Table 5.1: Discovered vulnerabilities in Relying Party implementations.

query. An attacker can abuse this. An email address might be a known factor, where a unique random `sub` is hard to predict.

The *phpOIDC* was vulnerable against a KC attack. This allowed us to inject a jku in the JOSE header and the signature is accepted. We could see in MitMProxy that the implementation retrieved this key from the referenced URL. We found the corresponding lines in the source code. There was no hint or comment that it was intended for debugging only.

We determined a different behavior in the *pyoidc* implementation as represented in Table 5.1. With a closer look inside the sources, we located a variable which is used to add some time to compensate a clock skew. It appears programmers forgot to set an appropriate time. This variable was set to zero and a conditional branch skipped the `iat` validation. The Signature Manipulation vulnerability in Authorization Code Flow is not generally forbidden by the OIDC specification. It is valid to set the `alg` to `none` for this flow. Recommendation was to let the application developer decide to allow this behavior explicitly, due to different use-cases and security requirements. Within the later responsible disclosure process the *pyoidc* developers informed us that this not validated `nonce` and TRC was an issue from our used Middleware. The flask module did not call a provided ID Token verification method.

In the source code of OidcRP, we noticed that developers considered the past to validate `iat`. Any future value was not checked.

5.2 Evaluation of OpenID Provider

Starting with default configurations in Table 5.2, we encountered implementation flaws and misbehavior in five out of six selected providers. The attacks Sub Claim Spoofing, Open Redirector, Authorization Code Reuse and Substitution cannot be observed.

The findings regarding Authentication Method Confusion required a closer investigation. All three affected implementations handled the authentication correct; regardless RP is configured for only one method. This potentially is a functionality for interoperability and fault tolerance towards not full specification conform RP implementations. If both basic authentication methods are implemented and handled correctly, it is not a security related issue. With manual tests, in Section 5.3.4, we ensured that no downgrade to less secure variants are possible.

Per default, *Gravitee.io Access Management* allows wildcard URIs. After enabling redirect URI strict matching in the administration console, these RUM vulnerabilities were mitigated. It is preferable that this option is enabled per default and categorized in a danger zone. Developers added hints in documentation entries, regarding the vulnerabilities of non-strict defined URIs.

The *MITREid Connect* server checks if a Code Challenge is included in the Authentication Request. If this is true in the Token Response, then it validates the Code Verifier. Therefore, it was vulnerable to the PKCE attack without a Code Challenge. During the code review, we discovered that it logs a wrong Code Challenge Method, but it throws no exception. We tested it with MitMProxy and we could bypass the PKCE. In addition, we observed a special error page during RUM and validated it manually in subsection 5.3.3.

Flaw	Identity Server4	Gravitee.io AM	Keycloak	MITREid Connect	node oidc	phpOIDC
Authentication Method Confusion (Basic)	✓	✓	✗	✗	✗	✓
Authentication Method Confusion (Post)	✓	✓	✗	✗	✗	✓
Redirect URI Validation (AR) - Path	✓	✗	✓	✓	✓	✓
Redirect URI Validation (AR) - Userinfo C	✓	✗	✓	✓	✓	✓
Redirect URI Validation (AR) - Userinfo D	✓	✗	✓	✓	✓	✓
Redirect URI Validation (TR) - Exclude URI	✓	✓	✓	✓	✗	✓
Message Flow Confusion Response Type	✓	✓	✓	✓	✓	✗
Message Flow Confusion Response Mode	✓	✓	✓	✓	✓	✗
PKCE Downgrade (AR) Exclude Challenge	✓	✓	✓	✗	✓	✓
PKCE Downgrade (AR) Invalid Method	✓	✓	✓	✗	✓	✓

✓: Attack failed/Secure; ✗: Attack successful/Insecure

Table 5.2: Discovered vulnerabilities in OpenID Provider implementations.

The *node oidc-provider* RUM with excluded URI in the Token Request is considered being a special use-case. The developer confirmed that the single registered redirect URI could be used in this case. With more than one registered URIs the excluded URI is detected as a fault.

In a first run, *phpOIDC* showed up a PKCE vulnerability. It was an issue of PrOfES-SOS, which not detected a disabled PKCE support. In the phpOIDC configuration we discovered a variable to enable PKCE. After enabling it no PKCE related vulnerabilities were discovered. The Message Flow Confusion vulnerabilities should be considered by the developers. Both parameters can be set in front-channel and might introduce risks to unaware RP implementations.

5.3 Manual Test Results

Besides the programming language independent tests offered by PrOfESSOS, we have carried out additional manual security evaluations. Our focus was to evaluate observations we made during tests with PrOfESSOS. On the one side, to verify the results, and on the other, to add some programming language specific tests.

5.3.1 Gravitee.io Access Management RUM Verification

There were two interesting aspects, which we verified regarding the Gravitee.io Access Management and the RUM vulnerability. First aspect was to verify, if the issued Authorization Code was revoked after a redirect URI was detected. This was required because we could not ensure that the same code path was used. For example, an exception could be raised which skipped the revocation. Second aspect to check, if the Authorization Code could be used with the manipulated URI.

#1 Check revoked code PrOfESSOS validates, if it is possible to redeem the Authorization Code with a Token Request and a correct registered URI. The Token Response (Listing 5.3) consequently returned an error 400.

```
400

Cache-Control no-store
Content-Type application/json; charset=UTF-8
Pragma no-cache

{
  "error_description": "Redirect URI mismatch.",
  "error": "invalid_grant"
}
```

Listing 5.3: PrOfESSOS redeems Authorization Code with correct URI.

To ensure the Authorization Code was revoked correctly, a curl command could be used. The curl command was issued by setting the web proxy accordingly.

```
export https_proxy=https://localhost:8080/
```

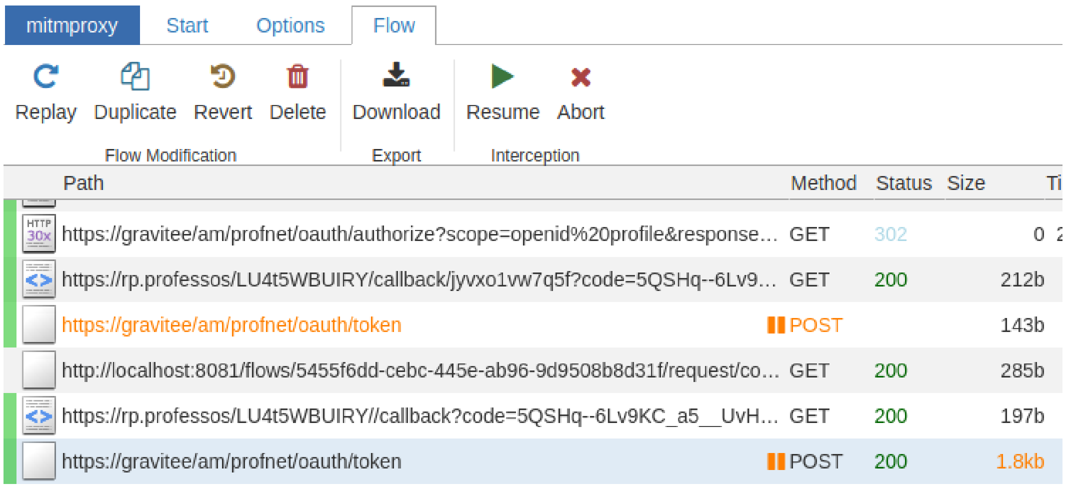
Afterwards, for RUM Path validation the curl command in Listing 5.4 was used.

```
curl -X POST -d "code=yoAXWs5M7P81Sxk7yXfYEGEExVdDeQndKnPX84tFXjE&redirect_uri=https%3A%2F%2
     Frp.professos%2Fwa9L9I1AB3Y%2Fcallback%2Fiivsyqm5ist2&grant_type=authorization_code" \
-H "X-Protocol-Scheme: https" \
-H "X-Protocol-Port: -1" \
-H "Authorization: Basic dmVYUUItSjJiTkNVVXVvMUN2cH..." \
-H "Content-Type: application/x-www-form-urlencoded; charset=UTF-8" \
-H "X-Prof-Sender: HONEST-RP" \
https://gravitee/am/profnet/oauth/token

{
"error" : "invalid_grant",
"error_description" : "The authorization code yoAXWs5M7P81Sxk7yXfYEGEExVdDeQndKnPX84tFXjE is
     invalid."
}
```

Listing 5.4: Check revoked Authorization Code after RUM detection.

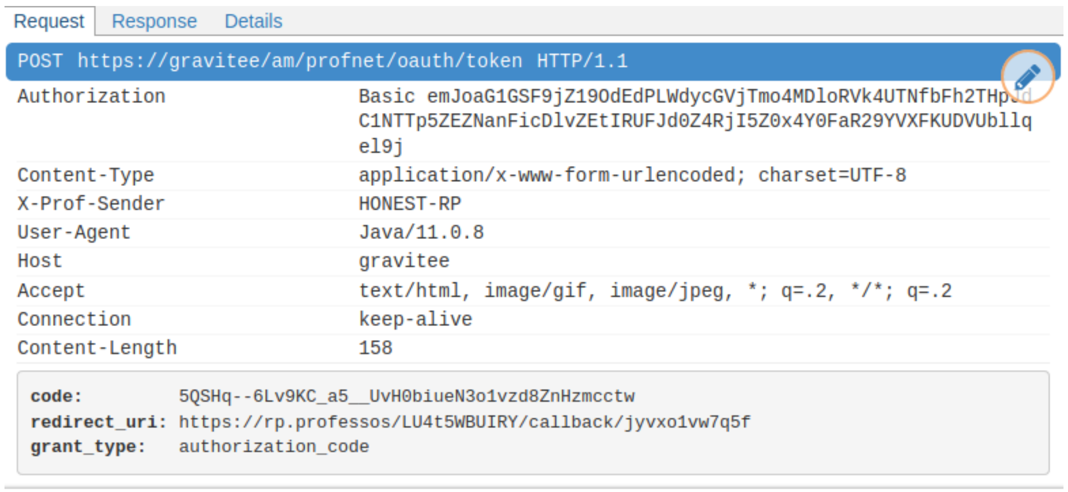
#2 Redeem code We received an error in the Token Response, as previously pointed out in Listing 5.3. For a countercheck we considered the MitMProxy. We added the familiar Token Endpoint address in the Web-UI as URI that was supposed to be intercepted.



Path	Method	Status	Size	Ti
https://gravitee/am/profnet/oauth/authorize?scope=openid%20profile&response…	GET	302	0	2
https://rp.professos/LU4t5WBUIRY/callback/jyvxo1vw7q5f?code=5QSHq--6Lv9…	GET	200	212b	
https://gravitee/am/profnet/oauth/token	POST		143b	
http://localhost:8081/flows/5455f6dd-cebc-445e-ab96-9d9508b8d31f/request/co…	GET	200	285b	
https://rp.professos/LU4t5WBUIRY//callback?code=5QSHq--6Lv9KC_a5__UvH…	GET	200	197b	
https://gravitee/am/profnet/oauth/token	POST	200	1.8kb	

Figure 5.5: MitMProxy intercepted Token Request.

Afterwards, we triggered the test in PrOfESSOS. The intercepted Post Method in Figure 5.5 was modified. Therefore, we appended path "jyvxo1w7q5f" to the redirect URI as depicted in Figure 5.6. Finally, we received the response Figure 5.7 with a successful Authorization Code redemption. We obtained an Access Token and an ID Token. With these steps, we validated that our attack was successful. Therefore, an additional test in PrOfESSOS become obsolete.

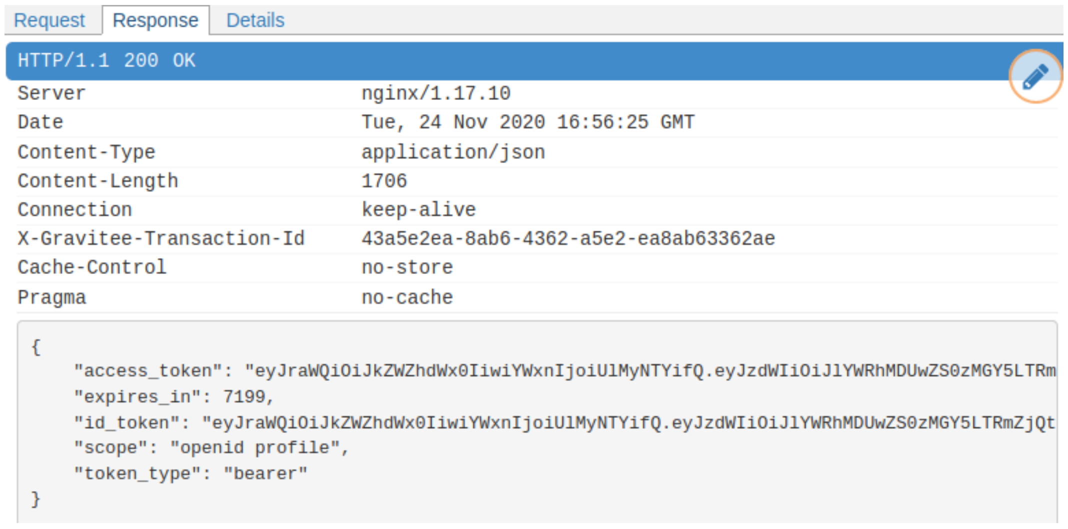


Request Response Details

```
POST https://gravitee/am/profnet/oauth/token HTTP/1.1
Authorization       Basic emJoaG1GSF9jZ19OdEdPLWdycGVjTmo4MDloRVk4UTNfbFh2THp
                    C1NTTp5ZEZNanFicDlvZEtIRUFJd0Z4RjI5Z0x4Y0FaR29YVXFKUDVUbllq
                    el9j
Content-Type        application/x-www-form-urlencoded; charset=UTF-8
X-Prof-Sender       HONEST-RP
User-Agent          Java/11.0.8
Host                gravitee
Accept              text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection          keep-alive
Content-Length      158

code:         5QSHq--6Lv9KC_a5__UvH0biueN3o1vzd8ZnHzmcctw
redirect_uri: https://rp.professos/LU4t5WBUIRY/callback/jyvxo1vw7q5f
grant_type:   authorization_code
```

Figure 5.6: MitMProxy modified Token Request.



Request Response Details

```
HTTP/1.1 200 OK
Server                      nginx/1.17.10
Date                        Tue, 24 Nov 2020 16:56:25 GMT
Content-Type                application/json
Content-Length              1706
Connection                  keep-alive
X-Gravitee-Transaction-Id   43a5e2ea-8ab6-4362-a5e2-ea8ab63362ae
Cache-Control               no-store
Pragma                      no-cache

{
    "access_token": "eyJraWQiOiJkZWZhdWx0IiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiJlYWRhMDUwZS0zMGY5LTRm
    "expires_in": 7199,
    "id_token": "eyJraWQiOiJkZWZhdWx0IiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiJlYWRhMDUwZS0zMGY5LTRmZjQt
    "scope": "openid profile",
    "token_type": "bearer"
}
```

Figure 5.7: Successful RUM Token Response.

5.3.2 JWKS Spoofing against mod auth openidc

The idea of a JWKS spoofing attack is to determine how many trusts an RP developer give to data, which is retrieved via back-channel communication. For example, a malicious OP can try to manipulate a key set to execute arbitrary code via SSRF. In addition, if JWKS size is not limited, it might be exploited to start a DoS attack. Drawback of this manual test is, that PrOfESSOS is limited to the old JWKS. The functionality can be retained by adding additional sets. In this test, mod auth openidc detected an error and thus disabled login to the OP.

```python
#!/usr/bin/env python3
from lib.attacks import *
from lib.client import MITMClient

if __name__ == "__main__":
    client = MITMClient()
    client.send(ClearCommand())

    jwks = JWKSSpoof()
    jwks.uri = "https://attack-idp.professos/modauthopenidc/jwks"

    # Test, if null is handled correctly. Jansson is used as JSON backend
    # Following message is displayed: \u0000 is not allowed without JSON_ALLOW_NUL error msg
        in apache2/errors
    # full jwk set is written to apache2 error log
    jwks.keys[0]["n"] = "\0<?php echo 'Hello' ?>"

    keys = jwks.keys
    client.send(jwks)
```

Listing 5.8: Script to test mod auth openidc.

In Listing 5.8, we checked the handling of special character `Null` in mod auth openidc. Therefore, we replaced the original modulus `n` parameter with a `Null` character and an additional code. This added code could be interpreted by PHP. After visiting the login page, this JWKS was retrieved with a curl easy function in mod auth openidc. The OP was not accessible anymore. An Apache2 server error was written to the log files, similar to Listing 5.9. It was obvious that the `Null` character was handled by Jansson as JSON backend. In addition, the malicious key set was written without escape characters to error log. Previous tests revealed we may write 4 KiB data into a JWKS.

```
oidc_util_decode_json_object: JSON parsing returned an error: \\u0000
     is not allowed without JSON_ALLOW_NUL (
{
"keys": [
{
"alg": "RS256",
"e": "AQAB",
"kid": "professos",
"kty": "RSA",
"n": "\\u0000<?php echo 'Hello' ?>",
"use": "sig",
"x5c": [ "MIIDQTCCAimgAwIBAgIBATAN..." ]
}
]
}), referer: https://mod-auth-openidc/protected
```

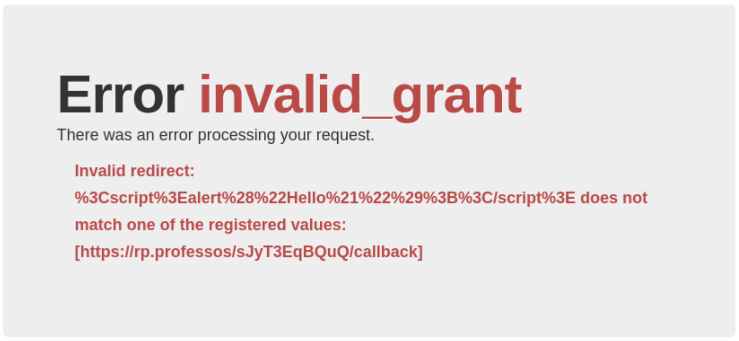
Listing 5.9: Apache2 error log while parsing Null in mod auth openidc.

An attack can be utilized, if the Apache2 error log is not protected correctly through file permissions. For example, with a well-known PHP Path Traversal attack, this code might be executed.

Regarding this issue, our recommendation was to suppress the JWKS dump.

5.3.3 Query Parameter Manipulations

During tests with PrOfESSOS, we investigated a special error page in MITREid Connect server. It was displayed only when the redirect URI partially did not match a registered URI. As shown in Figure 5.10 the `redirect_uri` query parameter was displayed with a correct escaped XSS attack.



Error invalid_grant

There was an error processing your request.

Invalid redirect: %3Cscript%3Ealert%28%22Hello%21%22%29%3B%3C/script%3E does not match one of the registered values: [https://rp.professos/sJyT3EqBQuQ/callback]

Figure 5.10: Manual XSS attack fails on MITREid Connect server.

This redirect URI manipulation could be altered in the browser or with a MitMProxy script. We prepared the depicted XSS with the script in Listing 5.11. It requires a valid URI, the query key and value which should be replaced. Afterwards, PrOfESSOS test was executed from the CLI tool.

```python
#!/usr/bin/env python3
from lib.attacks import *
from lib.client import MITMClient

if __name__ == "__main__":
    client = MITMClient()
    # cleanup previous mitm proxy hooks
    client.send(ClearCommand())
    # replace redirect_uri with a XSS check
    # mitreid-server escapes special characters before displaying it
    cmd = ReplaceCommand()
    cmd.uri = "mitreid-server/oidc-server/authorize"
    cmd.replaceKeyVal("redirect_uri", '<script>alert("Hello!");</script>')

    client.send(cmd)
```

Listing 5.11: Script to test MITREid Connect server against a XSS attack.

This kind of manipulation grants to test further special characters or implicit type conversions. For example, a Boolean value can be passed to a freshness parameter.

5.3.4 JSON Post Request Manipulations

For the evaluation of Authentication Method Confusion in Section 5.2, it was required to add a manipulation option for JSON Post requests. Direct integration into PrOfESSOS would be our preferred approach in the future. This demands additional time and routines are supposed to maintain full functionality.

The script in Listing 5.12 was implemented for a fast validation. It is necessary to find out the Registration Endpoint URI from Provider Configuration. The client authentication method in `token_endpoint_auth_method` was replaced accordingly. We manipulated registration with PrOfESSOS. Generally, PrOfESSOS was not aware of this and tried to authenticate with one of these basic methods at next test execution. This was the intention of our test. If a downgrade to less secure variants is feasible, PrOfESSOS can execute a complete test. Otherwise, an authentication method error is displayed.

```python
#!/usr/bin/env python3
from lib.attacks import *
from lib.client import MITMClient

if __name__ == "__main__":
    client = MITMClient()
    # cleanup previous mitm proxy hooks
    client.send(ClearCommand())
    # prepare attack to replace values in json post requests
    cmd = ReplacePostJsonCommand()

    # Set Registration Endpoint URI
    #cmd.uri = "keycloak/auth/realms/master/clients-registrations/openid-connect"
    cmd.uri = "node-oidc-provider/reg"
    #cmd.uri = "mitreid-server/oidc-server/register"

    # Change client authentication method
    #cmd.replaceKeyVal("token_endpoint_auth_method", 'Invalid')
    #cmd.replaceKeyVal("token_endpoint_auth_method", 'none')
    #cmd.replaceKeyVal("token_endpoint_auth_method", 'private_key_jwt')
    cmd.replaceKeyVal("token_endpoint_auth_method", 'client_secret_jwt')
    #cmd.replaceKeyVal("token_endpoint_auth_method", "NoNe")

    client.send(cmd)
```

Listing 5.12: Script to validate OP authentication methods.

Regarding the OpenID Connect specified known methods, it is possible to test additional invalid methods.

6 Responsible Disclosure

We disclosed the vulnerabilities responsible to the maintainers. All vulnerable implementations are open-source projects. They are hosted on Github. Except phpOIDC which is hosted on Bitbucket. The main challenge in this responsible disclosure process is to find the correct email addresses.

MITREid Connect provides a distribution group and on a closer look it is linked with a public mail archive. We decided to write an email directly to an explicit named maintainer. For all other libraries and services, we searched in git commits or on their profiles to find an email address. After we determined that we have found the right maintainer and email address, we submitted the report with recommendations. Most developers have responded within a day.

Name	Vulnerability	Reported	Fixed	CVE
MITREid Connect	PDA, IDS	✓	✦	
mod auth openidc	IDS, JWKS Spoofing	✓	✓	
node oidc-provider	RUM	✓	✗	
OidcRP	RA	✓	✦	
phpOIDC	MFC, IDS, KC	✓	✦	
pyoidc	RA, SM, TRC	✓	✓	CVE-2020-26244

✓: Reached/Solved; ✗: Rejected; ✦: Triaged

Table 6.1: Responsible disclosure status overview.

The pyoidc developers have fixed the reported vulnerabilities in consultation with the flask-pyoidc developers. With the Github Security Advisory feature, the developers created a CVE-2020-26244 and informed all related projects. The mod auth openidc created a bug fix which is part of new release version 2.4.5. The node oidc-provider developer explained that they are aware of the RUM issue. This special case for only one registered redirect URI is allowed. They keep the current implementation for interoperability and fault-tolerance with RPs which are using this functionality.

All other projects maintainer understands the vulnerabilities and will be working on a patch. The projects in Table 6.1 are therefore marked with a triage status.

7 Conclusion and Future Work

This thesis contributed to security analysis of real-life OpenID Connect implementations. We have analyzed OpenID Connect Certified Relying Parties and OpenID Providers with support of Implicit Flow and Hybrid Flow. Therefore, we introduced the required SSO protocol framework knowledge for OAuth 2.0 and OpenID Connect. With our security framework, we gave a state-of-the-art overview about known attack vectors and the capabilities of a web-attacker.

We developed a maintainable docker lab environment. It provides an extensible and stable test platform and ensures comprehensive test operations. We have integrated eight RP libraries and six OP services together with PrOfESSOS and MitMProxy. During the integration, we observed several minor non-specification conform behaviors. Developers are not providing required parameters, or consider optional parameters being required. They focus most on RP examples and documentation related to Authorization Code Flow. Implicit Flow and Hybrid Flow demanded more attention, to understand which development steps are required to use them. Partially, it was necessary to write minor patches that we could test these flows.

With MitMProxy, we could analyze and debug the protocol flow. This helped us to improve PrOfESSOS and verify the security test implementations. Together, they offer a comprehensive and fast way to perform security investigations within the lab environment. Developers and security researchers could have an out-of-the-box experience, after adding a new implementation. With the CLI tool and MitMProxy scripts, we have introduced an enhancement for PrOfESSOS. This allows manual and semi-automated security tests without reconfiguration of any RP or OP implementation. As an advantage, Relying Parties can be tested against a preferred test manually in the browser. PrOfESSOS provides this through an exposed attacker OP.

Finally, we analyzed the implementations and revealed several vulnerabilities. Five out of eight RPs have implementation flaws in six different attack types carried out with PrOfESSOS. We found one additional flaw during manual test analysis. OPs services appeared fewer flaws. Solely two out of six implementations were vulnerable against PrOfESSOS attacks. Gravitee.io Access Management in standard configurations revealed further weaknesses. We discovered a vulnerability in MITREid Connect server during code review and verified it with a MitMProxy changed PrOfESSOS test. In general, we have observed no common security hotspots other than the specification flaws. We reported all vulnerabilities in a responsible disclosure process.

Future Work might focus on extending the lab environment. A wider range of implementations should be considered. For example, with Authorization Code Flow and PKCE a newer extension is available, which is currently in development and designed to replace the Implicit Flow. Regarding the Relying Parties it would be beneficial if implementer validate the supported Code Challenge Methods metadata entry as an indicator that PKCE is supported. In case developers are not paying attention to this entry and foresee default values instead, it might be possible that an OP is used without PKCE support. If this remains unnoticed, the extension provides no more security than Implicit Flow.

The existing libraries should be tested in different configurations. With pyoidc we already proved that libraries could behave differently. This behavior is not limited to flows and response types. The implementations could be also configured with other authentication methods, response modes, or any valid configuration which could change the internal code flow.

We should add more flexibility to PrOfESSOS. First, an interface to change default values. On the one side, PrOfESSOS should detect non-specification conform settings with the defaults. On the other side, we need the capabilities to test the implementation in various configurations. Second, we should implement an interface for dynamic test configuration instead of compiled-in test plan. It is expected that these added features grant the flexibility to create language specific test cases.

The MitMProxy can be improved by adding more generic scripts for manipulations and concrete scripts for Single-Phase attacks. It is conceivable to use scripts to manipulate communication between honest RP and OP. This will add additional test variations. It should be also considered to add further SSO test suites. To have different test scenarios and ways of verifying test results against various tools.

List of Figures

List of Tables

List of Listings

Bibliography

[1] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song: *Towards a formal foundation of web security.* In *2010 23rd IEEE Computer Security Foundations Symposium*, pp. 290–304, 2010.

[2] W. Denniss and J. Bradley: *OAuth 2.0 for Native Apps.* RFC 8252, Oct. 2017. `https://rfc-editor.org/rfc/rfc8252.txt`.

[3] D. Detering, J. Somorovsky, C. Mainka, V. Mladenov, and J. Schwenk: *On the (in-)security of javascript object signing and encryption.* pp. 1–11, Nov. 2017, ISBN 978-1-4503-5321-2.

[4] D. Fett, R. Küsters, and G. Schmitz: *A comprehensive formal security analysis of oauth 2.0.* In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pp. 1204–1215, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4139-4. `http://doi.acm.org/10.1145/2976749.2978385`.

[5] D. Fett, R. Küsters, and G. Schmitz: *The web sso standard openid connect: In-depth formal security analysis and security guidelines.* In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pp. 189–202, Aug 2017.

[6] O. Foundation: *Certified openid connect implementations.* `https://openid.net/developers/certified/`, visited on 14-Sep-2020.

[7] O. Foundation: *Openid connect implicit client implementer's guide.* `https://openid.net/specs/openid-connect-implicit-1_0.html`, visited on 14-Sep-2020.

[8] O. Foundation: *Owasp top 10 - 2017*, 2017. `https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf`.

[9] D. Hardt: *The oauth 2.0 authorization framework.* `https://tools.ietf.org/html/rfc6749`.

[10] M. Jones: *Json web key (jwk)*, May 2015. `https://tools.ietf.org/html/rfc7517`.

[11] M. Jones, J. Bradley, and N. Sakimura: *JSON Web Signature (JWS).* RFC 7515, May 2015. `https://rfc-editor.org/rfc/rfc7515.txt`.

[12] M. Jones, J. Bradley, and N. Sakimura: *JSON Web Token (JWT).* RFC 7519, May 2015. `https://rfc-editor.org/rfc/rfc7519.txt`.

[13] P. Jones, G. Salgueiro, M. Jones, and J. Smarr: *WebFinger.* RFC 7033, Sept. 2013. `https://rfc-editor.org/rfc/rfc7033.txt`.

[14] W. Li and C. Mitchell: *Analysing the security of google's implementation of openid connect.* pp. 357–376, July 2016, ISBN 978-3-319-40666-4.

[15] T. Lodderstedt, J. Bradley, and A. Labunets: *Oauth 2.0 security best current practice.* `https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15`, visited on 14-Sep-2020.

[16] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, Oct 2020. `https://www.ietf.org/id/draft-ietf-oauth-security-topics-16.html`.

[17] T. Lodderstedt, M. McGloin, and P. Hunt: *Oauth 2.0 threat model and security considerations*, Jan 2013. `https://tools.ietf.org/html/rfc6819`.

[18] C. Mainka, V. Mladenov, and J. Schwenk: *Do not trust me: Using malicious idps for analyzing and attacking single sign-on.* Dec. 2014.

[19] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich: *Sok: Single sign-on security — an evaluation of openid connect.* In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 251–266, April 2017.

[20] T. McLean: *Blog: Critical vulnerabilities in json web token libraries*, Mar 2015. `https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html`.

[21] T. McLean: *Critical vulnerabilities in json web token libraries*, Aug 2020. `https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/`.

[22] B. de Medeiros, M. Scurtescu, P. Tarjan, and M. Jones: *Oauth 2.0 multiple response type encoding practices.* `https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html`, visited on 14-Sep-2020.

[23] V. Mladenov, C. Mainka, F. Feldmann, and J. Schwenk: *On the security of modern single sign-on protocols: Openid connect 1.0.* CoRR, abs/1508.04324, 2015. `http://arxiv.org/abs/1508.04324`.

[24] V. Mladenov, C. Mainka, and J. Krautwald: *Openid connect: Security considerations.* 2017. `https://www.nds.ruhr-uni-bochum.de/media/ei/veroeffentlichungen/2017/01/13/OIDCSecurity_1.pdf`.

[25] J. Richer and A. Sanso: *OAuth 2 in Action.* Manning Publications, Birmingham, 2017, ISBN 978-1-617-29327-6.

[26] N. Sakimura, J. Bradley, and N. Agarwal: *Proof key for code exchange by oauth public clients.* `https://tools.ietf.org/html/rfc7636`.

[27] N. Sakimura, J. Bradley, and M.B. Jones: *Openid connect discovery 1.0 incorporating errata set 1.* November 2014. https://openid.net/specs/openid-connect-discovery-1_0.html.

[28] N. Sakimura, J. Bradley, and M.B. Jones: *Openid connect dynamic client registration 1.0.* February 2014. https://openid.net/specs/openid-connect-registration-1_0-final.html.

[29] N. Sakimura, J. Bradley, M.B. Jones, B. de Medeiros, and C. Mortimore: *Openid connect core 1.0 incorporating errata set 1.* November 2014. https://openid.net/specs/openid-connect-core-1_0.html.

[30] K. Meyer zu Selhausen and D. Fett: *Oauth 2.0 authorization server issuer identifier in authorization response.* https://tools.ietf.org/html/draft-meyerzuselhausen-oauth-iss-auth-resp-01.html.

[31] X. Wang, W.C. Lau, R. Yang, and S. Shi: *Make Redirection Evil Again: URL Parser Issues in OAuth.* 2019. https://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Wang-Make-Redirection-Evil-Again-wp.pdf.

[32] C. Weber: *Unicode Security Guide*, 2017. https://websec.github.io/unicode-security-guide/character-transformations.

[33] Y. Zhou and D. Evans: *Ssoscan: Automated testing of web applications for single sign-on vulnerabilities.* In *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 495–510, San Diego, CA, Aug. 2014. USENIX Association, ISBN 978-1-931971-15-7. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou.

A Configuration Files

```
http {
        upstream gravitee-management {
            server gravitee-management:8093;
        }

        upstream gravitee-gateway {
            server gravitee-gateway:8092;
        }

        upstream gravitee-webui {
            server gravitee-webui:80;
        }

        server {
            listen 80;
            server_name SERVER_HOST;
            return 301 https://SERVER_HOST$request_uri;
        }

        server {
            listen 443 ssl http2;

            server_name SERVER_HOST;

            ssl_certificate /certs/SERVER_HOST/SERVER_HOST.crt;
            ssl_certificate_key /certs/SERVER_HOST/SERVER_HOST.key;

              location /am/ui/ {
                    proxy_pass http://gravitee-webui/;
                    proxy_set_header Host $host;
                    proxy_set_header X-Real-IP $remote_addr;
                    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                    proxy_set_header X-Forwarded-Host $server_name;
                    proxy_set_header X-Forwarded-Proto $scheme;
                    sub_filter '<base href="/"' '<base href="/am/ui/"';
                    sub_filter_once on;
              }

              location /am/management/ {
                    proxy_pass http://gravitee-management/management/;
                    proxy_redirect https://$host:$server_port/am/ui/ /am/ui/;
                    proxy_redirect https://$host:$server_port/management/ /am/management/;
                    proxy_cookie_path /management /am/management;
                    proxy_set_header Host $host;
                    proxy_set_header X-Real-IP $remote_addr;
                    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                    proxy_set_header X-Forwarded-Host $server_name;
                    proxy_set_header X-Forwarded-Proto $scheme;
                    proxy_set_header X-Forwarded-Prefix /am/management;
            }

            location /am/ {
                 proxy_pass http://gravitee-gateway/;
                 proxy_set_header Host $host;
                 proxy_set_header X-Real-IP $remote_addr;
                 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                 proxy_set_header X-Forwarded-Host $server_name;
                 proxy_set_header X-Forwarded-Prefix /am;
                 proxy_set_header X-Forwarded-Proto $scheme;
            }
       }
}
```

Listing A.1: Gravitee.io nginx configuration

```
FROM buildimage AS builder

ARG ISSUER
ARG CLIENT_HOST
ARG BRANCH
ARG HASH

RUN checkout
RUN buildsteps

################################################################################
# Image
################################################################################
FROM nginx:1.17.10-alpine

RUN apk update \
&& apk add ca-certificates \
&& rm -rf /var/cache/apk/*

COPY --from=builder /app/dist/client /usr/share/nginx/html

ARG CONTROLLER_URL
ARG CLIENT_HOST

ARG CA_DIR="/certs"
ARG CA_CERT="oidc-ca.crt"
VOLUME ["$CA_DIR"]

COPY config/default.conf /etc/nginx/conf.d/default.conf
RUN sed -i "s#CLIENT_HOST#$CLIENT_HOST#g" /etc/nginx/conf.d/default.conf

RUN echo "https://$CONTROLLER_URL" > /usr/share/nginx/html/.professos

EXPOSE 80
EXPOSE 443

COPY docker-entrypoint.sh /docker-entrypoint.sh

CMD ["/docker-entrypoint.sh"]
```

Listing A.2: Generic Dockerfile example

B Source Code

```python
from mitmproxy import ctx, http
import base64
import json
import threading
import socketserver

from urllib.parse import urlparse, quote


class CMDDef:
    TYPE_CLEAR = "clear"
    TYPE_REQUEST = "request"

    QUERY_SEARCH_REPLACE = "querySearchReplace"
    POST_JSON_SEARCH_REPLACE = "postJsonSearchReplace"
    JWKS_SPOOFING = "jwksSpoofing"


class Intercept:
    def __init__(self, search, replace):
        self.search = base64.b64decode(search).decode('ascii')
        self.replace = base64.b64decode(replace).decode('ascii')


class InterceptReplaceCommand:
    def __init__(self, uri, keyVal):
        self.type = CMDDef.TYPE_REQUEST
        self.action = CMDDef.QUERY_SEARCH_REPLACE
        self.uri = uri
        self.keyVal = keyVal

    def replace(self, requestUri):
        parse = urlparse(requestUri)

        # Check if url same
        if self.uri != parse.netloc+parse.path:
            return None

        querys = parse.query.split("&")
        new_query = []
        for query in querys:
            key, value = query.split('=')
            print(self.keyVal)
            if key in self.keyVal:
                print(key)
                value = self.keyVal.get(key)
            query = key + '=' + quote(value)
            new_query.append(query)

        new_query = "&".join(new_query)
        return parse._replace(query=new_query).geturl()


class InterceptPostJsonCommand:
    def __init__(self, uri, keyVal):
        self.type = CMDDef.TYPE_REQUEST
        self.action = CMDDef.POST_JSON_SEARCH_REPLACE
        self.uri = uri
        self.keyVal = keyVal

    def check(self, requestUri):
        parse = urlparse(requestUri)
        if self.uri != parse.netloc+parse.path:
            return False
        return True

    def replace(self, content):
        data = json.loads(content)
        for key, value in self.keyVal.items():
            data[key] = value
        return json.dumps(data).encode('utf-8')


class InterceptJWKSCommand:
    def __init__(self, uri, keys):
        self.type = CMDDef.TYPE_REQUEST
        self.action = CMDDef.JWKS_SPOOFING
        self.uri = uri
        self.keys = keys


class ThreadedTCPRequestHandler(socketserver.StreamRequestHandler):

    def handle(self):
        data = str(self.rfile.readline(), 'ascii') # recv until finish \n
        cmd = json.loads(data)
        if cmd.get("type") == CMDDef.TYPE_CLEAR:
            self.server.controller.clear()
        elif cmd.get("type") == CMDDef.TYPE_REQUEST:
            self.server.controller.requestInterceptor = self.request_selection(cmd)
        elif cmd.get("type") == 'response':
            intercept = Intercept(cmd.get('search'), cmd.get('replace'))
            self.server.controller.responseInterceptor = intercept

        response = bytes("OK", 'ascii')
        self.request.sendall(response)

    def request_selection(self, cmd):
        intercept = None
        if cmd.get("action") == CMDDef.QUERY_SEARCH_REPLACE:
            intercept = InterceptReplaceCommand(cmd.get('uri'), cmd.get('keyVal'))
        elif cmd.get("action") == CMDDef.POST_JSON_SEARCH_REPLACE:
            intercept = InterceptPostJsonCommand(cmd.get('uri'), cmd.get('keyVal'))
        elif cmd.get("action") == CMDDef.JWKS_SPOOFING:
            intercept = InterceptJWKSCommand(cmd.get('uri'), cmd.get('keys'))
        return intercept


class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    allow_reuse_address = True

    def __init__(self, host_port_tuple, streamhandler, controller):
        super().__init__(host_port_tuple, streamhandler)
        self.controller = controller


class Controller(object):
    def __init__(self):
        self.__requestInterceptor = []
        self.__responseInterceptor = []

    def clear(self):
        self.__requestInterceptor.clear()
        self.__responseInterceptor.clear()

    @property
    def requestInterceptor(self):
        return self.__requestInterceptor

    @requestInterceptor.setter
    def requestInterceptor(self, value):
        self.__requestInterceptor.append(value)

    @property
    def responseInterceptor(self):
        return self.__responseInterceptor

    @responseInterceptor.setter
    def responseInterceptor(self, value):
        self.__responseInterceptor.append(value)


class ProfessosEnhancer(object):

    def __init__(self) -> None:
        ctx.log.info("Init Server")
        self.controller = Controller()
        self.server = None

    def running(self):
        if self.server is not None:
            ctx.log.info("Server is already running")
            return
        HOST, PORT = "0.0.0.0", 8042
        self.server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler, self.
             controller)
        ip, port = self.server.server_address

        server_thread = threading.Thread(target=self.server.serve_forever)
        server_thread.daemon = True
        server_thread.start()
        ctx.log.info("Enhancer listens on {}:{}".format(ip,port))

    def request(self, flow: http.HTTPFlow) -> None:
        for intercept in self.controller.requestInterceptor:
            if intercept.action == CMDDef.QUERY_SEARCH_REPLACE:
                replaceUrl = intercept.replace(flow.request.pretty_url)
                if replaceUrl:
                    flow.request.url = replaceUrl
                    ctx.log.info("Request Replaced: {}".format(flow.request.pretty_url))
            elif intercept.action == CMDDef.POST_JSON_SEARCH_REPLACE:
                if intercept.check(flow.request.pretty_url):
                    content = intercept.replace(flow.request.content)
                    if content:
                        flow.request.content = content
                        ctx.log.info("Request Replaced: {}".format(flow.request.pretty_url))
            elif intercept.action == CMDDef.JWKS_SPOOFING:
                #ctx.log.info("Intercept URI: {} -> {}".format(flow.request.pretty_url,
                     intercept.uri))
                if flow.request.pretty_url == intercept.uri:
                    ctx.log.info("{}".format({"keys": intercept.keys}))

                    keys = {"keys": intercept.keys}
                    header = {
                        "Access-Control-Allow-Origin": "*",
                        "Access-Control-Allow-Credentials": "true",
                        "Access-Control-Allow-Methods": "*",
                        "Access-Control-Allow-Headers": "origin, content-type, accept,
                             authorization",
                        "Content-Type": "application/json;charset=UTF-8",
                    }

                    flow.response = http.HTTPResponse.make(
                        200,
                        json.dumps(keys, indent=4, ensure_ascii=False).encode('utf-8'),
                        header
                    )

    def response(self, flow: http.HTTPFlow) -> None:
        for intercept in self.controller.responseInterceptor:
            ctx.log.info("{}".format(intercept.search))
            if flow.request.pretty_url == intercept.search:
                #flow.response = http.HTTPResponse.make(status, content, header)
                pass

    def done(self):
        if self.server is not None:
            self.server.shutdown()
            self.server = None
        ctx.log.info("Finish")


addons = [ProfessosEnhancer()]
```

Listing B.1: Mitmproxy script to enhance PrOfESSOS