

Single Sign-On – OpenID Connect(ing) people

Security Analysis of the OpenID Connect Standard and its real-life Implementations

Master Thesis

Chair for Network and Data Security

**RUHR
UNIVERSITÄT
BOCHUM**

RUB

Submitted by:	Julian Krautwald
Master degree course:	IT-Security
First examiner:	Prof. Dr. Jörg Schwenk
Second examiner:	Vladislav Mladenov, Christian Mainka

© 2014

Abstract

OpenID Connect, as a combination of a Single Sign-On (SSO) and a delegated Authorization protocol, provides a highly security-critical service to be implemented by application developers. As, on the one hand, the open and decentralized structure of OpenID Connect brings flexibility and interoperability, it also makes verification of exchanged authentication and authorization tokens a non-trivial task. In this thesis we introduce five novel attacks on the protocol all resulting in unauthorized access of protected resources. All attacks described in this thesis target implementation flaws on either the Relying Party (RP) or the OpenID Provider (OP) side and are almost all applicable to other SSO systems. To demonstrate the real-life applicability of such attacks we also summarize the evaluation results of our security analysis of nine RP libraries, nine OP libraries as well as three OP implementations (running in open accessible productive systems) of the protocol. Associated to that, we introduce two self-developed proof-of-concept Java pentest applications for auditing OpenID Connect implementations.

Contents

List of Figures	IV
List of Tables	V
List of Listings	VI
1. Introduction	1
2. Foundations	3
2.1. The Concept of Single Sign-On	3
2.2. Original and Browser-Based Kerberos	4
2.3. OpenID 2.0	6
2.4. OAuth 2.0	8
2.4.1. Roles	9
2.4.2. Access Token	10
2.4.3. Protocol Endpoints	10
2.4.3.1. Authorization Endpoint	11
2.4.3.2. Redirection Endpoint	11
2.4.3.3. Token Endpoint	11
2.4.4. Abstract Protocol Flow	11
2.4.5. Authorization Grant Types	12
2.4.5.1. Authorization Code Grant Type	13
3. OpenID Connect	16
3.1. Preliminaries	16
3.2. Use Cases and Objectives	17
3.3. Roles	17
3.4. Protocol Endpoints	19
3.4.1. Authorization Endpoint	19
3.4.2. Redirection Endpoint	19
3.4.3. Token Endpoint	19
3.4.4. JSON Web Key Set Endpoint	20
3.4.5. UserInfo Endpoint	20
3.4.6. Dynamic Registration Endpoint	20

3.4.7. Discovery Endpoint(s)	20
3.5. OpenID Connect Core Specification	21
3.5.1. ID Token	21
3.5.2. Authentication using the Authorization Code Flow	22
3.5.3. Authentication using the Implicit Flow	26
3.5.4. Authentication using the Hybrid Flow	28
3.6. OpenID Connect Discovery Specification	29
3.7. OpenID Connect Dynamic Client Registration Specification	32
3.8. Validation Steps	33
3.8.1. Authentication Request Validation	34
3.8.2. Authentication Response Validation	34
3.8.3. Token Request Validation	35
3.8.4. Token Response Validation	35
4. Security Analysis	37
4.1. Security Model	37
4.1.1. Objectives of the Attacker	37
4.1.2. Assumptions	37
4.1.3. Capabilities of the Attacker	38
4.1.4. Behavior of the Victim	38
4.2. Related Work	39
4.3. OpenID Connect Pentest Applications	41
4.3.1. The Relying Party	42
4.3.2. The OpenID Provider	45
4.4. Attacks / Attack-Scenarios	48
4.4.1. ID Spoofing	48
4.4.2. Issuer Confusion	48
4.4.3. Signature Manipulation	49
4.4.4. Sub Claim Spoofing	49
4.4.5. Redirect URI Manipulation	50
4.5. Provider / Library Selection	53
4.5.1. Relying Party Implementations	53
4.5.2. OpenID Provider Implementations	54
4.6. Security Aspect Catalog	55
4.7. Practical Analysis	58
5. Conclusion	70
5.1. Summary	70
5.2. Further Studies	71

Bibliography	73
A. Appendix	i

List of Figures

2.1. Kerberos initial ticket exchange	5
2.2. Example of the browser-based Kerberos protocol	6
2.3. Abstract OpenID 2.0 Protocol Flow	7
2.4. Role Relationship within the OAuth protocol	9
2.5. OAuth 2.0 Abstract Protocol Flow	12
2.6. OAuth 2.0 Authorization Code Grant	13
3.1. Role Relationship within the OpenID Connect protocol	18
3.2. OpenID Connect Authorization Code Flow	23
3.3. OpenID Connect Implicit Flow	27
3.4. OpenID Connect Hybrid Flow	28
3.5. OpenID Connect Discovery Flow	30
3.6. OpenID Connect Dynamic Client Registration Flow	32
4.1. OpenID Connect Provider TestSuite - Provider EndPoints	43
4.2. OpenID Connect Provider TestSuite - Metadata Discovery	43
4.3. OpenID Connect Provider TestSuite - WebFinger Discovery	43
4.4. OpenID Connect Provider TestSuite - Dynamic Registration Request Parameters	44
4.5. OpenID Connect Provider TestSuite - Authorization Request Param- eters	44
4.6. OpenID Connect Provider TestSuite - Token Request Parameters	45
4.7. OpenID Connect Provider TestSuite - Parsed Token Response	45
4.8. OpenID Connect Client TestSuite - Accessible Endpoints	46
4.9. OpenID Connect Client TestSuite - Metadata Discovery Response Data	46
4.10. OpenID Connect Client TestSuite - Dynamic Client Registration Re- sponse Data	46
4.11. OpenID Connect Client TestSuite - Authentication Response Data	47
4.12. OpenID Connect Client TestSuite - Token Response Data	47
4.13. OpenID Connect Client TestSuite - Token Request-Response Pair	47
4.14. RUM using Authorization Code Flow	51
4.15. RUM using Implicit Flow	52

List of Tables

4.1. Relying Party Libraries	54
4.2. OpenID Provider Libraries	55
4.3. OpenID Provider Live Implementations	55
4.4. Security Aspect Catalog - General Information	56
4.5. Security Aspect Catalog - Validation	57
4.6. Relying Party Libraries - General Information	59
4.7. Relying Party Libraries - Validation	60
4.8. OpenID Provider Libraries - General Information	63
4.9. OpenID Provider Libraries - Validation	64
4.10. OpenID Provider Live Implementations - General Information	66
4.11. OpenID Provider Live Implementations - Validation	67
4.12. Relying Party Libraries - Applicable Attack-Scenarios	68
4.13. OpenID Provider Libraries - Applicable Attack-Scenarios	69
4.14. OpenID Provider Live Implementations - Applicable Attack-Scenarios	69

List of Listings

3.1. Non-Normative Example Authentication Request	24
3.2. Non-Normative Example Authentication Response	24
3.3. Non-Normative Example Token Request	25
3.4. Non-Normative Example Token Response	25
3.5. Non-Normative Example Issuer Discovery Request	30
3.6. Non-Normative Example Issuer Discovery Response	30
3.7. Non-Normative Example Configuration Discovery Response	31
3.8. Non-Normative Example Client Registration Request	33
3.9. Non-Normative Example Client Registration Response	33
4.1. Non-Normative Example Claims Request	49
4.2. Sub Claim Request	50

1. Introduction

Web-based client authentication, in the classical sense, serves the purpose for an End-User to prove her identity to a Service Provider (SP), for example, a social networking platform. To achieve this goal, username / password -based authentication processes are commonly utilized. To ensure that the user does not have to complete this procedure (with individual username and password submitting) for each and every service she is registered for, Single Sign-On (SSO) procedures have been established. SSO gives a user the opportunity to log in to a single system, the Identity Provider (IdP), that then provides access to various other SPs¹: The IdP takes over the authentication of the user to her RP. This relocation of the authentication process to a trusted third party together with partly complex used protocols (trying to provide confidentiality, authenticity as well as integrity of the exchanged information) are making SSO a security-critical topic. Despite of known security flaws and attack-scenarios [1, 2], due to partly poorly implemented protocols, a huge propagation and usage of SSO protocols like OpenID or SAML can be observed [3, 4]. OpenID Connect, developed by the OpenID Foundation, is a SSO protocol strictly based on the protocol principles of the authorization framework OAuth 2.0 [5] but extending these with several aspects and ideas of the SSO protocol OpenID 2.0 [6]. The OpenID Foundation calls it a "simple identity layer on top of the OAuth 2.0 protocol" [7]. The just recently (by the end of February 2014) published final specification of the protocol promises a raised interoperability compared to other SSO protocols like OpenID. In addition to this, the ease of deployment for developers, one the of defined goals for the protocol by the OpenID Foundation, apparently disembogues in a high willingness of huge companies like Google, Gakunin, Microsoft, Ping Identity, Nikkei Newspaper, Tokyu Corporation, mixi, Yahoo! Japan and Softbank to implement OpenID Connect within their production systems [8]. Despite the fact that there is an official feature test initiative of individual deployed OpenID Connect implementations [9], it lacks of security-based examinations of these.

Through an extensive literature research, this thesis will give a detailed overview of the topic Single Sign-On using OpenID Connect. For this purpose, we will at first outline technical and historical background information of SSO systems together

¹also named as Relying Parties (RPs)

1. Introduction

with a detailed description of the most important aspects of the protocols OpenID 2.0 and OAuth 2.0, as those two build the base on which OpenID Connect is constructed. Subsequently, we will give a detailed description of the "Core" protocol of OpenID Connect [10] plus its extensions "Discovery" [11] and "Dynamic Client Registration" [12]. When describing the Core protocol we will, among others, introduce its various fields of application together with its dedicated "Authentication Flows". With the help of abstract protocol flows we will then elaborate its underlying concepts. In addition to that, this thesis will be used to give a comprehensive practical security analysis of the protocol. Therefore, we define possible attack targets alongside conceivable attack scenarios and in the following apply the latter to several protocol frameworks / libraries as well as live implementations² of both, the Identity- and the Service Provider side. Associated to that, we will introduce two self-developed proof-of-concept Java pentest applications for auditing OpenID Connect implementations. We will then conclude with an overview of the carried out tests and thus make a statement about the application-security of each tested library / framework.

²meaning running in an open accessible productive system

2. Foundations

This chapter is intended to outline the general concept of Single Sign-On (SSO) as well as its applicability within a web-based environment. We therefore take a close look at the Kerberos protocol which can be utilized when realizing such a scheme. Furthermore, we will provide a detailed description of the most important aspects of the protocols OpenID 2.0 and OAuth 2.0 as these two protocols are building the groundwork for OpenID Connect.

2.1. The Concept of Single Sign-On

Since the term Web 2.0 [13] was initially used in April 1999, the World Wide Web has changed in many different ways. The Web is by no means only about retrieving information and accessing static websites. It is rather about the possibility to collaborate, interoperate and add value to existing content. With the advent of huge web-applications and the invention of the delivery of computing as a service (Cloud computing), users got the chance to use such services, looking and feeling like they are installed and executed on a local PC, only through their web browser. One of the many advantages of these concepts is the user's ability to store data and information in a centralized place instead of on a local hard drive. This is advantageous as the user can access this information and even applications from every web-enabled device in the world. All she needs to remember are her login credentials for the requested service. This is in fact the part where problems may arise. As most of the users do not only want to access a single, login protected application but often various, they have to remember a lot of authentication information. To identify the user, most application providers use password-based authentication schemes. Therefore, users may get their credentials, which commonly means a unique user ID, for example an Email-Address, plus a corresponding password. They can then use their credentials to authenticate themselves against the provider. When managing their credentials for various applications, users tend to get lazy and therefore choose weak but easy to remember passwords (In October 2013 a security breach at Adobe, with about 130 million stolen user accounts, revealed that the most used account password is

123456). They are also driven to use one and the same username and password combination for various providers. By brute-forcing such poorly chosen passwords, an adversary may be able to impersonate a user of a certain application. The impact of such an attack can be enormous. Single Sign-On schemes can be one solution to the above described problem. SSO provides the ability for users to log in once and gain access to various independent software systems without being prompted to log in at each of them separately. Thus it reduces the user's effort to remember multiple credentials. The authentication is generally handled via a central authentication system and is cross domain resistant. Reducing the amount of login credentials also has several security benefits. Strict password policies can be enforced because users only have to remember one password. Phishing success can be reduced because a single point of authentication is easier to remember and thus recognize. Furthermore, authentication information has to be transmitted only once and user change management can be centralized. However, the single point of failure architecture which SSO inherits brings a lot of risks, too. If the central authentication system can be compromised or if it is the target of a Denial-of-Service attack it could have severe implications for all users of the system. Furthermore, especially for small providers or in the early stages of an offered service it might often be much more complicated and thus expensive to deploy a SSO authentication system than an old fashioned password-based.

2.2. Original and Browser-Based Kerberos

Kerberos is a computer network authentication protocol [14] which can be used in SSO schemes. It was initially developed by the Massachusetts Institute of Technology within Project Athena and it was designed to mutually authenticate the involved parties in a client / server application over an unprotected network. It is based on the symmetric Needham-Schroeder protocol [15] and thus uses a trusted third party, named the Key-Distribution Center (KDC). The KDC can be seen as an Authentication Server and a Ticket Granting Server at the same time. Therefore, it builds the essential part for the authentication of clients and servers. For the sake of simplicity, the SSO implementation introduced in this thesis uses a simplified version of the original Kerberos protocol [16]. Kerberos authentication is ticket-based and consists in general of three distinct parties, called client, server, and Key-Distribution Center. The ticket is a time-limited token, issued by the KDC, which is used to prove the identity of the client to the server. The KDC also establishes a temporary encryption key (session key) which is used to securely communicate with each other. In addition to this session key, each party has to own a pre-shared

2. Foundations

key with the KDC to trust the authenticity of the tickets it receives. Added to the ticket, the client has to send an authenticator to the server in order to proof that she possesses the session key issued by the KDC. This is done to prevent session replay attacks. The flow of the initial ticket exchange between the three parties can be described as follows: The first message is sent by the client to the KDC and contains a unique identifier c , an identifier of the targeted server s , and a **nonce** n . The second message is sent by the KDC to the client and consists of a randomly chosen session key $K_{C,S}$, the received **nonce** and the newly generated ticket $T_{C,S}$. To protect the confidentiality of the transmitted information, $K_{C,S}$ and n are encrypted using the long-term key K_c of the client. For the same reason the ticket is encrypted using the long-term key K_s of the server. The final message is sent by the client to the server. This message contains the authenticator mentioned above, encrypted with the new session key $K_{C,S}$, and the encrypted ticket from (2). After receiving and validating (3), the server and the client are mutually authenticated. From now on they are able to communicate securely by using the session key $K_{C,S}$ for encryption. The described message flow is depicted in Figure 2.1.

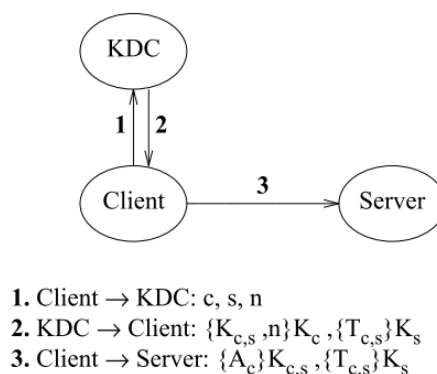


Figure 2.1.: Kerberos initial ticket exchange [16]

As Kerberos can easily be adapted to browser-based authentication schemes, we are using it to demonstrate the general concept of browser-based SSO schemes. To produce comprehensibility, we like to rename the participating entities of the authentication mechanism: From now on we refer to the client party as User Agent (UA), client or simply user; to the server as Service Provider (SP); to the KDC as Identity Provider (IdP). IdP and SP are commonly represented through web servers, whereas the UA is represented through the browser of the client. An example authentication flow, initiated by the UA, can be seen in Figure 2.2.

2. Foundations

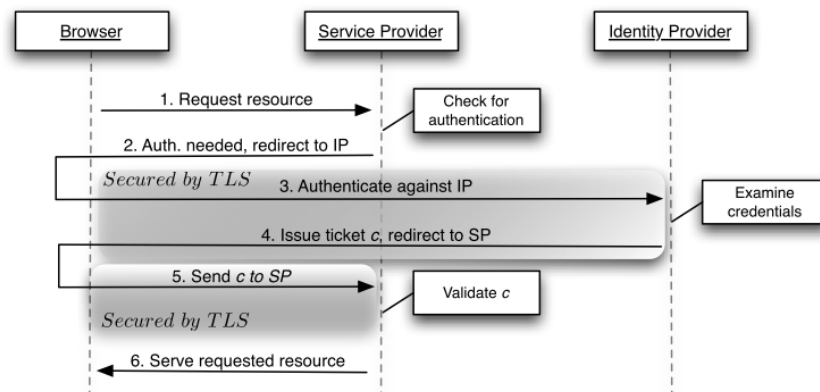


Figure 2.2.: Example of the browser-based Kerberos protocol [17]

Step (1.) shows the request of a protected resource from the SP, initiated by the user. Steps (2.) and (3.): As the user is not yet authenticated to the SP, she gets redirected to the IdP to request a ticket for authentication. Step (4.): After verifying the required credentials of the user, the IdP issues a ticket *c* and redirects the browser to the SP. Step (5.): By transmitting the provided ticket *c*, the user authenticates herself to the SP. Step (6.): After validating the received ticket, the SP serves the initially requested resource to the user. The browser-based Kerberos depicted above slightly differs from the original one. Because of the zero-footprint requirements of a browser, some concessions regarding the security of the scheme had to be done. As in the original Kerberos symmetric encryption keys are used to acquire the required trust between the different parties, in the browser-based variant this is usually done via Transport Layer Security (TLS) and server certificates.

2.3. OpenID 2.0

OpenID 2.0 is a decentralized, web-based SSO protocol. It is defined in an open standard which was finally specified in December 2007 by the OpenID Foundation. Its main goal is it to log in an End-User, represented by an Identifier, at a SP (in this context called the Relying Party (RP)) via an IdP called the OpenID Provider (OP). It thus provides a way to prove that an End-User controls an Identifier [6] without the RP having to store or verify any credentials of the End-User. Contrary to other SSO schemes, where a central authority is responsible for registering RPs or OPs, OpenID 2.0 is decentralized, meaning that literally everybody can become an OP and provide authentication services (Authentication-as-a-Service) for RPs. An OpenID Identifier, also simply called OpenID, of an End-User is an Uniform

2. Foundations

Resource Identifier (URI) usually containing the username of the End-User and the domain-name of the corresponding OP, for example:

- `username.myOpenIDProvider.com` or
- `myOpenIDProvider.com/username`

To get an OpenID, an End-User has to register with an OP (e.g. Google or Yahoo). Within the OpenID 2.0 protocol four different parties implying four different roles can be found:

1. End-User: A user, utilizing an User Agent (UA) to authenticate to a Relying Party.
2. Relying Party: A web-application providing a specific service where authentication of its users is mandatory.
3. OpenID Provider: An Authentication-as-a-Service web-application.
4. Identifier Host: A host, given an OpenID Identifier, responsible for resolving the identity of the corresponding OP.

Figure 2.3 depicts an abstract OpenID 2.0 protocol flow to demonstrate the interaction between the above described roles.

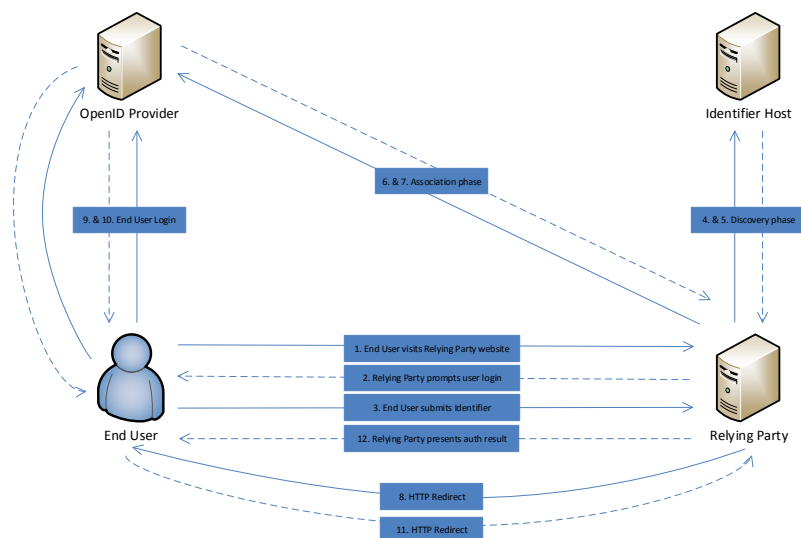


Figure 2.3.: Abstract OpenID 2.0 Protocol Flow

1. The OpenID 2.0 authentication process is initiated by the End-User requesting an authentication-required service of the RP.

2. Foundations

2. The RP prompts the End-User to submit her OpenID Identifier via a login page.
3. The End-User submits her Identifier.
4. The RP sends the received Identifier to the Identifier Host.
5. The Identifier Host resolves the received Identifier and responds with the identity of its corresponding OP together with additional metadata.
6. - 7. Within the association phase, the RP and the OP negotiate a shared secret (e.g. via Diffie-Hellman key exchange) for digitally signing and verifying the to-be-exchanged token.
8. The RP redirects the End-User to the OP.
9. - 10. The End-User authenticates herself to the OP.
11. The OP issues an authentication token, signed with the beforehand exchanged secret, redirects the End-User back to the RP and appends the token to the request.
12. The RP verifies the validity of the received token and presents the End-User the result of the authentication process.

2.4. OAuth 2.0

Contrary to the above mentioned protocols, OAuth is not a SSO protocol but a method to allow delegated access to protected resources. Version 1.0 of the protocol, developed in 2007 by the Internet Engineering Task Force (IETF) community, is an open standard giving a Client the possibility to access protected resources on behalf of an End-User (called the Resource Owner) without knowing her password [18]. The resources of the End-User are usually hosted on a different server called the Resource Server. A use case for this specific delegation could be a user who wants to access some of her resources at provider A (the Resource Server) via another provider B (the Client) without giving provider B her credentials to log in to provider A. As OAuth 1.0 required the use of signed access requests to provide authenticity of the Client, it was not easy to implement and thus not widely used by developers. This fact together with deficient scalability, due to excessive storage of partly redundant data [19], and a discovered session fixation vulnerability [20] led to the development of OAuth 2.0 where authenticity of requests and responses is ensured by the use of TLS. OAuth 2.0 is not downwards compatible to version 1.0 of the protocol [21].

2. Foundations

OAuth 2.0 was finally standardized within RFC 6749 [5] by the IETF in October 2012.

2.4.1. Roles

Within the OAuth protocol four different parties implying four different roles can be found. In a real-life implementation, one entity is not strictly bound to a single role but could also cover multiple, for example, the Authorization Server and the Resource Server, at once [21, 5]. The relationship between the different roles can be seen in Figure 2.4.

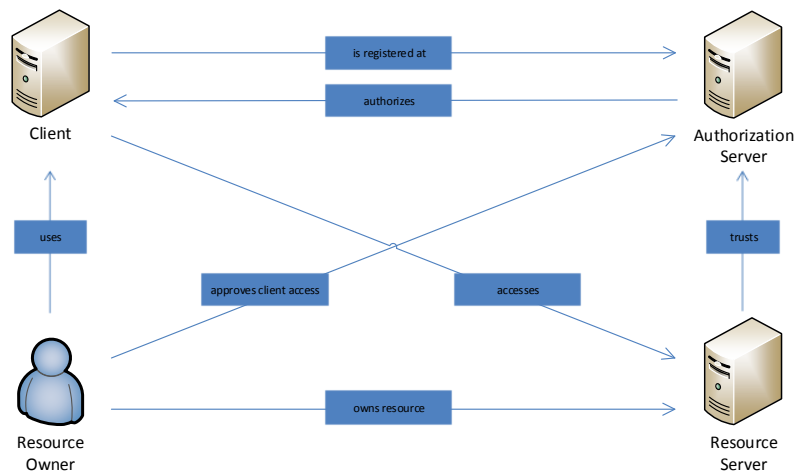


Figure 2.4.: Role Relationship within the OAuth protocol

- **Resource Owner:** The Resource Owner is an End-User trying to access her own resources at the Resource Server via a Client. Within this context the Resource Owner is able to delegate access to her resources to the Client (approve Client access).
- **Resource Server:** The Resource Server is responsible for hosting the resources of the Resource Owner. Access to protected resources is only granted when submitting valid credentials or an OAuth 2.0 Access Token (see Section 2.4.2) representing delegated authorization through the Resource Owner.
- **Authorization Server:** The Authorization Server issues an Access Token for the requested resources to the Client right after successful authorization by the Resource Owner. In order to do so the Resource Owner has to be authenticated to the Authorization Server.

- **Client:** The Client is an application which is authorized by the Resource Owner to access its resources. A Client can be run either on a dedicated webserver (server side web-application), in an End-User-sided application running within the browser of the Resource Owner (delivered for example via JavaScript or as browser extension) or a native application installed on the device of the Resource Owner. Depending on the application scenario the Client has the ability to authenticate itself to the Authorization Server.

2.4.2. Access Token

To access protected resources on behalf of a Resource Owner a Client must possess a so called Access Token. The Access Token is, according to the standard [5], an unspecified secret value which is entitled to grant access to a specific set of resources. To access the requested resources at the Resource Server the Client must transmit the Access Token in each request as Hypertext Transfer Protocol (HTTP) Authorization Header value, Uniform Resource Locator (URL) parameter or form-encoded body parameter. As the token is a secret value, also called Bearer Token [19], each transmission of it has to be cryptographically secured using TLS. Once a valid (not expired) Access Token is somehow disclosed to an entity other than the Client, this entity is, without any other security restraints, able to access the resources the token is entitled to access. Consequently the security of the Access Token and thus the security of the whole protocol is defined through the security of TLS. The to-be-received resources are defined via a parameter called **scope** which is exchanged in an earlier phase of the protocol than the actual Access Token allocation. The **scope** parameter can be used by both parties: the Client (to tell the Authorization Server which resources of the Resource Owner are being requested) and the Authorization Server (to give the Client feedback about which resources can really be accessed by the issued token). The lifetime of an Access Token is defined through another parameter, called **expires**, which is issued alongside the token itself. The value of this parameter can vary from several seconds to until the revocation of the token. In this context, an optional parameter, called Refresh Token, can also be used to regenerate an expired Access Token.

2.4.3. Protocol Endpoints

Within the OAuth 2.0 specification [5] the following endpoints are defined:

2.4.3.1. Authorization Endpoint

In order to consent the Authorization Request of the Client, the Resource Owner has to be redirected to the Authorization Endpoint of the Authorization Server. Prior to the proper consent, the Resource Owner has to authenticate herself to the Authorization Server. The actual authentication process is left unspecified within the specification. Once the Resource Owner has authenticated and consented the Authorization Request, the Authorization Server issues a so called Authorization Grant for the Client.

2.4.3.2. Redirection Endpoint

Each Client has to register one or more Redirect URI(s), building its Redirection Endpoint, with the Authorization Server it wants to communicate with. This registration process is left unspecified within the core specification but a proposed specification for dynamically registering OAuth 2.0 Clients with Authorization Servers can be found at [22]. To specify the to-be-used Redirect URI, the Client has to add the URI as request parameter to the initial Authorization Request. This specific URI has to match (to be validated by the Authorization Server) at least one of the registered ones of the Client. Thus, after successfully interacting with the Authorization Server, the Resource Owner can be redirected to the Redirection Endpoint of the Client. Within this redirect the beforehand issued Authorization Grant is passed as request parameter from the Authorization Server to the Client.

2.4.3.3. Token Endpoint

To retrieve a valid Access Token (used for the actual resource gathering, see Section 2.4.2) the Client has to send a HTTP POST request with the received Authorization Grant as parameter to the Token Endpoint of the Authorization Server. Depending on the application scenario the Client has the ability to authenticate itself to the Authorization Server within this request.

2.4.4. Abstract Protocol Flow

To clarify the interaction between the previously described protocol endpoints, Figure 2.5 depicts an abstract protocol flow including the four roles mentioned in Section 2.4.1.

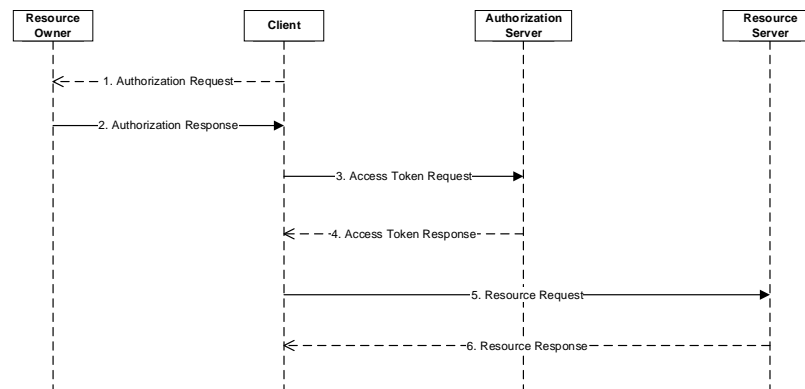


Figure 2.5.: OAuth 2.0 Abstract Protocol Flow [5]

1. Authorization Request: To access a protected resource of the Resource Owner, the Client at first sends an Authorization Request directly, or alternatively via the Authorization Server, to the Resource Owner.
2. Authorization Response: If the Resource Owner consents the Authorization Request a Authorization Grant is sent back to the Client.
3. Access Token Request: After receiving the Authorization Grant, the Client uses the latter to request an Access Token directly at the Authorization Server.
4. Access Token Response: After successfully validating the received Authorization Grant and possibly the Client authentication, the Authorization Server issues an Access Token and sends it back to the Client.
5. Resource Request: To request the protected resources of the Resource Owner, the Client sends a request containing the previously received Access Token to the Resource Server.
6. Resource Response: When receiving a valid Access Token, the Resource Server responds with the requested resources.

2.4.5. Authorization Grant Types

To cover multiple application scenarios / types (see Client role description of Section 2.4.1) the specification defines four different protocol flows by redefining the Authorization Grant Type. The proposed grant types are:

- Authorization Code [5, Section 4.1] (Client as server-sided web-application),

2. Foundations

- Implicit [5, Section 4.2] (Client as End-User-sided application running within the browser of the Resource Owner),
- Resource Owner Password Credentials [5, Section 4.3] (Client as native application installed on the device of the Resource Owner), and
- Client Credentials [5, Section 4.4] (Client takes over the roles of the Client as well as the Resource Owner - server-to-server communication).

As the Authorization Code Grant Type is the most prevalent used grant type and describing all others would bust the extend of a foundations chapter, the following section will concentrate on a detailed description, alongside its various requests and responses, of the Authorization Code Grant Type.

2.4.5.1. Authorization Code Grant Type

The Authorization Code Grant Type is used for server-sided applications fulfilling the Client role of the protocol, for example, a server-sided web application for managing contact details which wants to access a Resource Owner's stored contact list on Google Contacts. Figure 2.6 depicts the protocol flow of the Authorization Code Grant Type with its most important parameters (second row of each message) together with mention-worth optional ones (enclosed in square brackets).

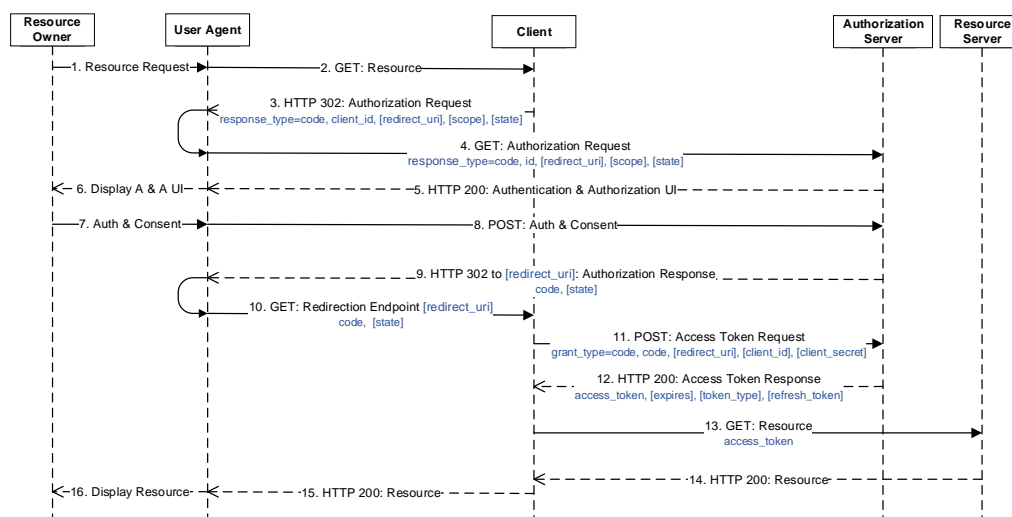


Figure 2.6.: OAuth 2.0 Authorization Code Grant [2]

2. Foundations

1. - 2. By using a UA to perform access delegation via the OAuth 2.0 protocol, the Resource Owner starts the protocol flow by telling the Client to request some resources of herself on another server.
3. After determining the location of the Authorization Server (through an out-of-band mechanism because it is not exactly specified within [5]) responsible for the requesting Resource Owner, the Client responds with a HTTP redirect, called the Authorization Request, to the Authorization Endpoint of the Authorization Server. This redirect has to contain several mandatory and optional parameters like the Redirect URI (**redirect_uri**) to which the Resource Owner is redirected after successful authentication to the Authorization Server (Step 9.), a Client ID (**client_id**) which the Authorization Server uses to identify the Client, a scope (**scope**) to specify what access privileges are being requested, and for the Authorization Code Grant Type the **response_type** value **code**. Other optional parameters, like **state**, which value is commonly used to prevent Cross-Site Request Forgery attacks, can also be sent within this redirect.
4. The UA executes the received redirect described in the previous step.
5. - 8. After successfully validating the received Authorization Request (see Section 4.1.1 of [5]), the Authorization Server attempts to authenticate the Resource Owner or determine whether the Resource Owner is already authenticated. The methods used within this process are beyond the scope of the OAuth 2.0 specification. After successfully authenticating the Resource Owner, the Authorization Server must prompt the Resource Owner with an authorization decision displaying all access privileges which are being requested by the Client. This dialogue can usually only be consented or denied.
9. Once the Resource Owner is authenticated and has given her consent, the Authorization Server responds with a HTTP redirect, called the Authorization Response, to the Redirect URI retrieved from the Authorization Request. This redirect has to contain the mandatory **code** parameter containing the actual Authorization Grant to be used by the Client. If a **state** value was present within the received Authorization Request it also has to be returned in this response.
10. The UA executes the received redirect described in the previous step.
11. After successfully validating the received Authorization Response (see Section 4.1.2 of [5]), the Client directly communicates with the Token Endpoint of the Authorization Server. Within this so called Token Request the Client has to

2. Foundations

submit several optional and mandatory parameters: The `code` parameter is used to present the Authorization Grant (received within the previous step) to the Authorization Server, the `grant_type` parameter containing the value `authorization_code`, the same Redirect URI as sent within Step 4. for security reasons, and if applicable (if the Client is a confidential Client to the Authorization Server) its `client_id` and `client_secret`.

12. After successfully validating the received Token Request (see Section 4.1.3 of [5]) and, if applicable, verifying the identity of the Client (with the received `client_id` and `client_secret` parameters), the Authorization Server issues an Access Token (as described in Section 2.4.2). Along with this parameter the Token Response of the Authorization Server has to contain several other mandatory parameters: `token_type` describing the type of the issued Access Token (its value must be Bearer, as specified within [5], unless another token type has been negotiated with the Client), and `expires_in` defining the lifetime in seconds of the Access Token. Other optional parameters, like the `refresh_token` which can be used to obtain new Access Tokens, can also be sent within this response.
13. After successfully validating the received Token Response (see Section 4.1.4 of [5]), the Client requests the protected resources of the Resource Owner at the Resource Server. This is done via a simple GET request containing the Access Token, received in Step 12., within the Authorization header (as described in [23]).
14. After checking the validity of the received Access Token, the Resource Server responds with the resources which were requested during the Authorization Request (specified within the `scope` parameter).
15. - 16. When receiving the requested resources the Client displays the latter to the Resource Owner.

3. OpenID Connect

Within this chapter we are giving a detailed description of the Single Sign-On protocol OpenID Connect as well as its extensions Discovery and Dynamic Client Registration. To produce comprehensibility we will at first outline possible objectives and use cases of the protocol. Later on, after introducing the involved parties of the protocol, we will elaborate the protocols underlying concepts with the help of abstract protocol flows. To do so we will introduce the protocols Authentication Flows together with their intended fields of application. To prepare our work for the following chapter, Security Analysis, we will also outline several security-related validation steps to be performed by the protocols involved parties.

3.1. Preliminaries

OpenID Connect provides SPs (from here on called Client or Relying Party (RP)) the ability to verify the identity of an End-User due to her authentication to an IdP (from here on called OpenID Provider (OP)). In addition to that, as the protocol extends the OAuth 2.0 framework, it allows a Client to access protected resources of an End-User (e.g. profile information) via delegated authorization. OpenID Connect thus implements authentication services on top of the authorization process of OAuth 2.0. To cover multiple client applications (web-based, mobile, JavaScript), the protocol defines three different Authentication Flows resulting in three different protocol variants. The main differences of these flows are within the way of communication between the Client and the OP and thus indirectly affecting the security of the protocol. The "OpenID Connect Protocol Suite" [8] consists, beyond the "Core" specification [10], of multiple optional specifications: "Discovery" [11] is a specification which can be used by OpenID Connect Clients to utilize the WebFinger protocol [24] to discover, for the Core protocol flow, important (configuration-) information (like for example the public key of the OP used for signature verification purposes) of an OP responsible for authenticating a specific End-User. "Dynamic Client Registration" [12] specifies a way for Clients to dynamically register themselves with a specific OP in order to ensure proper Client authentication during the Core protocol.

3.2. Use Cases and Objectives

Since OpenID Connect is based on the OAuth 2.0 family of specifications, OAuth 2.0 capabilities are integrated with the protocol itself [7]. This fact alongside its ability to verify the identity of an End-User [8] is making OpenID Connect a combination of a SSO and a delegated Authorization protocol. The OpenID Foundation calls it "a simple identity layer on top of the OAuth 2.0 protocol" [7]. Besides its architectural similarities to OpenID 2.0, OpenID Connect does support more than only one application type, thus extending its use cases from only web-based applications to native- and mobile applications as well. In addition to that, OpenID Connect is designed to raise interoperability by making it easier for developers to implement it than OpenID 2.0. This is mostly done by using Representational State Transfer (REST) / JavaScript Object Notation (JSON) message flows together with TLS infrastructure and JSON Web Token (JWT) data structures for cryptography instead of XML and custom message signature schemes as used in OpenID 2.0. As OpenID 2.0, OpenID Connect is also a decentralized protocol, meaning that literally everybody can become an OP and provide authentication services for RPs.

With OAuth 2.0 designed for giving Clients access to protected resources, a Client might be asking why not just use the assigned Access Token to discover the identity of the corresponding End-User? As the Access Token of OAuth 2.0 is a Bearer token, a Client receiving the token has no way of knowing to whom it was actually issued to [25]. This is however a crucial characteristic when trying to authenticate a user via a token and right at this point OpenID Connect comes into play. By adding an additional layer to the protocol, a so called ID Token, OpenID Connect provides a verifiable method to prove the identity of an End-User to a Client.

3.3. Roles

Within the OpenID Connect protocol three different parties implying three different roles can be found. The relationship between the different roles can be seen in Figure 3.1.

3. OpenID Connect

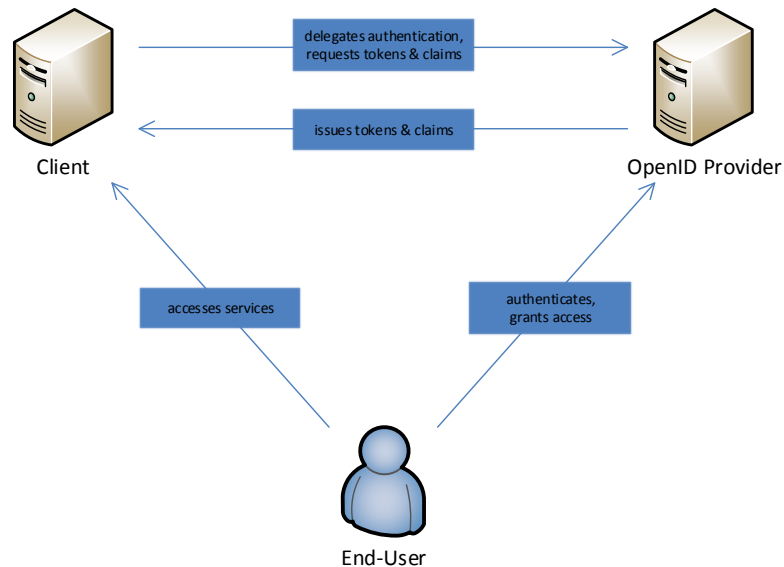


Figure 3.1.: Role Relationship within the OpenID Connect protocol [25]

- **End-User:** The End-User, represented by her UA, wants to access selected services of a Client. Therefore, she needs to prove her identity to the Client. Additionally, the End-User has the possibility to authorize the Client to access a specific set of her resources, defined via the **scope** and **claims** parameters, in her name.
- **Client:** The Client is an application which delegates the authentication of an End-User, wanting to access its services, to the corresponding OP. Therefore, the Client has to request an ID Token which proves the identity of the End-User and an optional OAuth 2.0 Access Token to access protected resources of the End-User. Depending on the application scenario the Client has the ability to authenticate itself to the OP.
- **OpenID Provider:** The OP issues an ID Token, containing a specific set of claims proving the identity of the End-User, and an optional OAuth 2.0 Access Token for the requested resources to the Client right after successful authentication and authorization by the End-User.

3.4. Protocol Endpoints

Within the OpenID Connect Core specification [10] together with its extensions Discovery [11] and Dynamic Client Registration [12] the following endpoints are defined:

3.4.1. Authorization Endpoint

In order to consent the Authentication Request (together with optionally requested resources) of the Client, the End-User has to be redirected to the Authorization Endpoint of the OP. Prior to the proper consent, the End-User has to authenticate herself to the OP. The actual authentication process is left unspecified within the specification. All communication with the Authorization Endpoint must, per specification, utilize TLS.

3.4.2. Redirection Endpoint

Each Client has to register one or more Redirect URI(s), building its Redirection Endpoint, with the OP it wants to communicate with. This registration process is left unspecified within the Core specification but a proposed specification for dynamically registering OpenID Connect Clients with OPs can be found at [12]. To specify the, in a specific protocol flow, to-be-used Redirect URI, the Client has to add the URI as request parameter to the initial Authentication Request. This specific URI has to match (to be validated by the OP, see Section 3.8.1) at least one of the registered ones of the Client. Thus, after successfully interacting with the OP, the End-User can be redirected to the Redirection Endpoint of the Client. Depending on the used protocol flow, this endpoint has to be able to receive an Authorization Code (see Section 3.5.2), an ID Token, an Access Token, or a combination of the three (see Section 3.5.3 and Section 3.5.4) as request parameters.

3.4.3. Token Endpoint

When using the Authorization Code or Hybrid -flow of the protocol, the Client has to communicate with the Token Endpoint of the OP in order to obtain an Access Token, an ID Token (described in Section 3.5.1), or a combination of both. This is done via direct communication (without involving the End-User) between the Client and the OP. In order for the OP to be able to issue the requested token(s), the Client has to append the beforehand received Authorization Code as request parameter.

3. OpenID Connect

Depending on the application scenario the Client has the ability to authenticate itself to the OP within this request. All communication with the Token Endpoint must, per specification, utilize TLS.

3.4.4. JSON Web Key Set Endpoint

When using asymmetric cryptography to digitally sign and optionally encrypt the ID Token, the Client needs the possibility to validate the signature and optionally decrypt a received ID Token from the OP. The JSON Web Key Set Endpoint of an OP contains a JSON Web Key Set (JWKS) [26] enlisting the, for the above described purpose, needed public key(s).

3.4.5. UserInfo Endpoint

To obtain additionally requested claims about the End-User, a Client has the possibility to make a request to the UserInfo Endpoint of the corresponding OP. This request however has to contain an Access Token (as defined in [23]) obtained through OpenID Connect Authentication to prove the validity of the access delegation by the End-User. All communication with the UserInfo Endpoint must, per specification, utilize TLS.

3.4.6. Dynamic Registration Endpoint

With the help of the Dynamic Registration Endpoint [12] of an an End-User's OP, a Client is able to dynamically register with the latter. This process is needed to provide the OP with information about the Client (e.g. its Redirect URI(s), see above) and in return get a unique Client ID and an optional Client Secret used to authenticate the Client to the OP. All communication with the Dynamic Registration Endpoint must, per specification, utilize TLS.

3.4.7. Discovery Endpoint(s)

In order for an OpenID Connect RP to discover the OP of a specific End-User and also obtain its configuration information (required for successfully interacting with it), the OP has the possibility to provide two endpoints with well-known locations serving the described purpose: With the help of the OpenID Provider Issuer Discovery Endpoint [11, Chapter 2], the Client is able to determine the location of the OP and with the OpenID Provider Configuration Information Endpoint [11,

Chapter 4] it retrieves the OP's configuration information (including its endpoint locations described above). All communication with the Discovery Endpoint(s) must, per specification, utilize TLS.

3.5. OpenID Connect Core Specification

This section is entitled to give a detailed description of the core OpenID Connect functionality as specified in [10]. Therefore, we will introduce OpenID Connect's most important extension to the OAuth 2.0 protocol, the ID Token, together with its three Authentication Flows. The to-be-used Authentication Flow is defined by the `response_type` parameter (sent within the initial Authentication Request of the Client) and determines how the ID Token and Access Token are returned to the Client.

3.5.1. ID Token

The ID Token [10, Chapter 2] is a security token containing claims about the authentication of an End-User by an OP, proving the identity of the End-User to a Client. Its data structure is represented as a JWT [27]. In order to provide authenticity as well as integrity of the token, the OP is responsible for signing it using JSON Web Signature (JWS) [28]. To provide an additional layer of security, the token can also be encrypted using JSON Web Encryption (JWE) [29]. The following list provides an excerpt of mandatory and optional claims used within an ID Token when performing authentication via OpenID Connect:

- **iss:** The issuer identifier is a mandatory claim identifying the issuer (the OpenID Provider) of the ID Token, represented by a case sensitive URL using the Hypertext Transfer Protocol Secure / HTTP Over TLS (HTTPS) scheme (e.g. `https://www.myOpenIDProvider.com`).
- **sub:** `sub` is a mandatory claim specifying an identifier for the End-User to be consumed by the Client. Issued by the OpenID Provider (`iss`), it has to be locally unique and never reassigned (e.g. `alice@myOpenIDProvider.com`).
- **aud:** `aud` is a mandatory claim specifying an array of case sensitive strings defining the audience(s) that this ID Token is intended for. It must, at least, contain the OAuth 2.0 `client_id` of the Client which requested the token.
- **iat:** Issued at is a mandatory claim specifying the time at which the ID Token was issued. When a Client receives a token with an `iat` claim valuing a time

3. *OpenID Connect*

after the current time, the ID Token is issued in the future, which is evidently not possible, and must not be used.

- **exp**: Expires is a mandatory claim specifying the time at which the ID Token will expire. When a Client receives a token with an **exp** claim valuing a time before the current time, the ID Token is expired and must not be used.
- **nonce**: The **nonce** is a randomly chosen String value, sent by the Client within the Authentication Request and passed through unmodified to the ID Token, used to mitigate replay attacks. It is a mandatory claim if it is present in the Authentication Request sent by the Client.
- **at_hash**: **at_hash** is a, depending on the used Authentication Flow, either optional or mandatory claim valuing the left-most half of the hash of the corresponding sent Access Token.
- **c_hash**: **c_hash** is a, depending on the used Authentication Flow, either optional or mandatory claim valuing the left-most half of the hash of the corresponding sent Authorization Code.

3.5.2. Authentication using the Authorization Code Flow

The Authorization Code Flow [10, Section 3.1] is used by server-sided applications fulfilling the Client role of the protocol. In this flow all tokens are returned from the Token Endpoint and thus not revealed to the End-User. Figure 3.2 depicts the Authorization Code Flow with its most important parameters (second row of each message) together with mention-worthy optional ones (enclosed in square brackets).

3. OpenID Connect

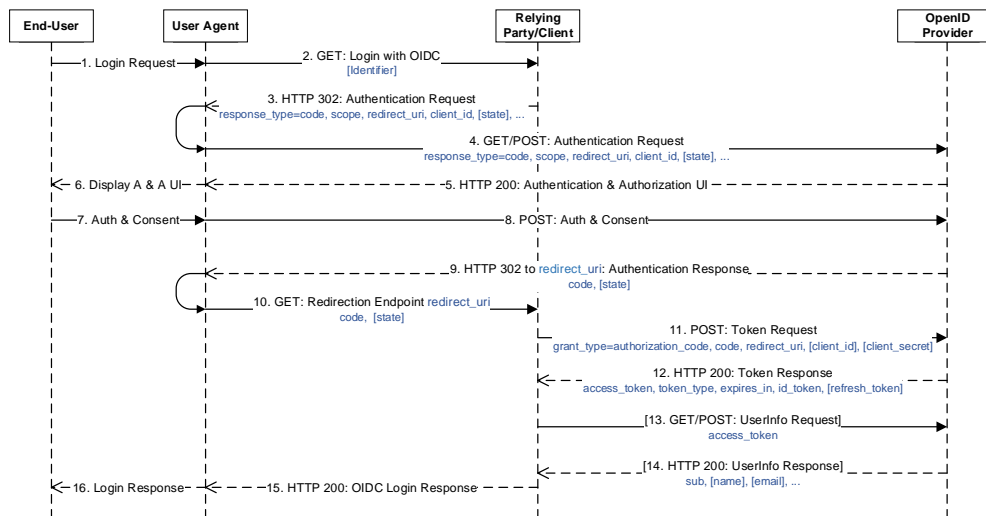


Figure 3.2.: OpenID Connect Authorization Code Flow

1. - 2. By using a UA to perform OpenID Connect based authentication, the End-User usually supplies the "Login with OIDC" request to the RP with an Identifier (needed when the RP wants to perform OpenID Provider Issuer Discovery). In some cases the RP might know the OP's Issuer location through an out-of-band mechanism and thus the End-User does not have to submit a special Identifier (e.g. "Login with Google").
3. After determining the location of the OP (through OpenID Provider Issuer Discovery or any other out-of-band mechanism) responsible for the requesting End-User, the RP responds with a HTTP redirect, called the Authentication Request, to the Authorization Endpoint of the OP. This redirect has to contain several mandatory parameters like the Redirect URI (**redirect_uri**) to which the End-User is redirected after successful authentication to the OP (Step 9.), a Client ID (**client_id**) which the OP uses to identify the RP, a scope (**scope**) to specify what access privileges are being requested (has to contain the **openid scope** value to make sure that the OP utilizes the OpenID Connect protocol instead of solely OAuth 2.0), and for the Authorization Code Flow the **response_type** value **code**. Other optional parameters, like **state**, which value is usually used to prevent Cross-Site Request Forgery attacks, can also be sent within this redirect.

3. OpenID Connect

```
1 HTTP/1.1 302 Found
2 Location: https://example.openIDProvider.com/authorize?response_type=code&scope=op
  enid%20profile%20email&client_id=s6BhdRkqt3&state=af0ifjsldkj&redirect_uri=htt
  ps%3A%2F%2Fexample.client.com%2Fcallback
```

Listing 3.1: Non-Normative Example Authentication Request

4. The UA executes the received redirect described in the previous step.
5. - 8. After successfully validating the received Authentication Request (see Section 3.8.1), the OP attempts to authenticate the End-User or determine whether the End-User is already authenticated. The methods used within this process are beyond the scope of the OpenID Connect specification. After successfully authenticating the End-User, the OP must prompt the End-User with an authorization decision displaying all access privileges which are being requested by the RP. This dialogue can usually only be consented or denied.
9. Once the End-User is authenticated and has given her consent, the OP responds with a HTTP redirect, called the Authentication Response, to the Redirect URI retrieved from the Authentication Request. This redirect has to contain the mandatory `code` parameter. The parameter values the actual authorization code to be used by the RP. If a `state` value was present within the received Authentication Request it also has to be returned in this response.

```
1 HTTP/1.1 302 Found
2 Location: https://example.client.com/callback?code=SplxlOBeZQQYbYS6WxSbIA&stat
  e=af0ifjsldkj
```

Listing 3.2: Non-Normative Example Authentication Response

10. The UA executes the received redirect described in the previous step.
11. After successfully validating the received Authentication Response (see Section 3.8.2), the RP directly communicates with the Token Endpoint of the OP. Within this so called Token Request the RP has to submit several mandatory parameters: The `code` parameter is used to present the authorization code (received within the previous step) to the OP, the `grant_type` parameter containing the value `authorization_code` (as described within [5]), the same Redirect URI as sent within Step 4. for security reasons, and if applicable (if the RP is a confidential client to the OP) its `client_id` and `client_secret`.

```
POST /token HTTP/1.1
Host: example.openIDProvider.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

grant_type=authorization_code&code=Sp1xIOBeZQqYbYS6WxSbIA&redirect_uri=http
s%3A%2F%2Fexample.client.com%2Fcallback
```

12. After successfully validating the received Token Request (see Section 3.8.3) and, if applicable, verifying the identity of the RP (with the received `client_id` and `client_secret` parameters), the OP issues an OAuth 2.0 Access Token (as described within [5]) and an ID Token (see Section 3.5.1) to prove the identity of the End-User to the RP. Along with those two parameters the Token Response of the OP has to contain several other mandatory parameters: `token_type` describing the type of the issued Access Token (its value must be Bearer, as specified within [23], unless another token type has been negotiated with the Client), and `expires_in` defining the lifetime in seconds of the Access Token. Other optional parameters, like `refresh_token` which can be used to obtain new Access Tokens (as described within [5]), can also be sent within this response.

[illegible]

3. *OpenID Connect*

13. After successfully validating the received Token Response (see Section 3.8.4), the RP has the opportunity to request additional claims about the End-User at the Userinfo Endpoint of the OP. This is done via a simple GET request containing the Access Token, received in Step 12., within the Authorization header (as described in [23]).
14. After checking the validity of the received Access Token, the OP responds with the claim values of the End-User which were requested during the Authentication Request (specified within the `scope` and `claims` parameter). Additionally, the `sub` claim (which is the OPs unique Subject Identifier for the End-User) has to be sent with this response.
15. - 16. After successfully validating the received Token Response (see Section 3.8.4), the identity of the End-User is now proven to the RP and the End-User should be notified that she has successfully logged in.

3.5.3. Authentication using the Implicit Flow

The Implicit Flow [10, Section 3.2] is mainly used by End-User-sided applications, implemented in a browser using a scripting language, fulfilling the Client role of the protocol. In this flow all tokens are returned from the Authorization Endpoint and thus revealed to the End-User. By avoiding the Token Endpoint, Clients are not able to authenticate to the OP. Figure 3.3 depicts the Implicit Flow with its most important parameters (second row of each message) together with mention-worth optional ones (enclosed in square brackets).

3. OpenID Connect

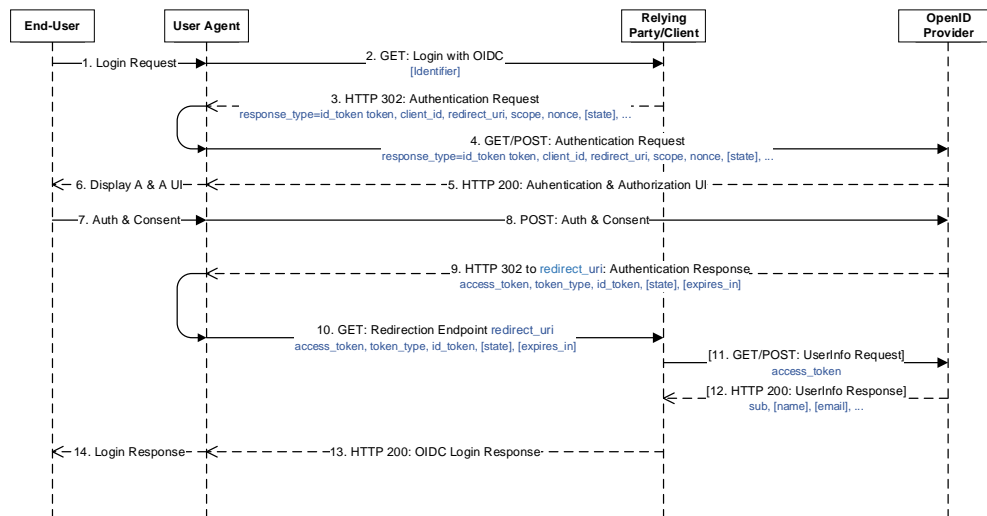


Figure 3.3.: OpenID Connect Implicit Flow

1. - 2. Analogous to Steps 1. and 2. of the Authorization Code Flow.
3. Analogous to Step 3. of the Authorization Code Flow except that the value of the **response_type** parameter has to be one of the following:
 - **id_token token**: When both, ID Token and Access Token, are requested.
 - **id_token**: When only an ID Token is requested.

Additionally, a **nonce** parameter (containing a value to associate a client session with an ID Token), in order to mitigate replay attacks, has to be appended to the request.
4. The UA executes the received redirect described in the previous step.
5. - 8. Analogous to Steps 5. - 8. of the Authorization Code Flow.
9. Once the End-User is authenticated and has given her consent, the OP issues an ID Token (see Section 3.5.1) to prove the identity of the End-User to the RP and, if applicable (see Step 3.), an OAuth 2.0 Access Token (as described within [5]) and appends them to its HTTP redirect, called the Authentication Response, to the Redirect URI retrieved from the Authentication Request.
10. The UA executes the received redirect described in the previous step.
11. - 12. Analogous to Steps 13. - 14. of the Authorization Code Flow.

3. OpenID Connect

13. - 14. After successfully validating the received Authentication Response (see Section 3.8.4), the identity of the End-User is now proven to the RP and the End-User should be notified that she has successfully logged in.

3.5.4. Authentication using the Hybrid Flow

The Hybrid Flow [10, Section 3.3] is mainly used by native applications fulfilling the Client role of the protocol. In this flow both, the Authorization Endpoint as well as the Token Endpoint, are used to retrieve tokens. Figure 3.4 depicts the Hybrid Flow with its most important parameters (second row of each message) together with mention-worthy optional ones (enclosed in square brackets).

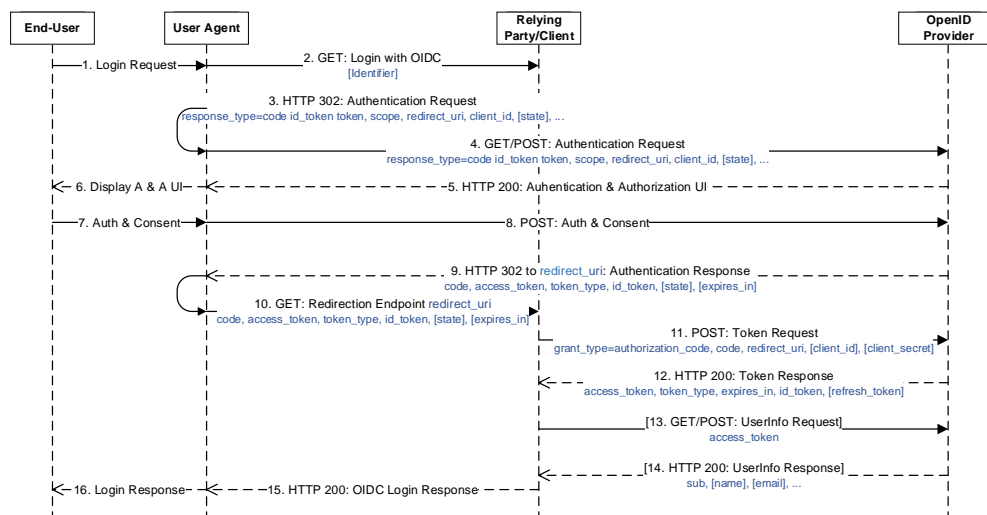


Figure 3.4.: OpenID Connect Hybrid Flow

1. - 2. Analogous to Steps 1. and 2. of the Authorization Code Flow.
3. Analogous to Step 3. of the Authorization Code Flow except that the value of the **response_type** parameter has to be one of the following:
 - **code id_token token**: When alongside the authorization code both, ID Token and Access Token, are requested.
 - **code token**: When alongside the authorization code only the Access Token is requested.
 - **code id_token**: When alongside the authorization code only the ID Token is requested.

3. *OpenID Connect*

4. The UA executes the received redirect described in the previous step.
5. - 8. Analogous to Steps 5. - 8. of the Authorization Code Flow.
9. Once the End-User is authenticated and has given her consent, the OP responds with a HTTP redirect, called the Authentication Response, to the Redirect URI retrieved from the Authentication Request. This redirect has to contain the mandatory **code** parameter containing the actual authorization code to be used by the RP and, if applicable (see Step 3.), an ID Token and / or an OAuth 2.0 Access Token. If a **state** value was present within the received Authentication Request it also has to be returned in this response.
10. The UA executes the received redirect described in the previous step.
11. - 16. Analogous to Steps 11. - 16. of the Authorization Code Flow.

3.6. OpenID Connect Discovery Specification

In order for an OpenID Connect RP to discover the OP of a specific End-User and also obtain its configuration information (required for successfully interacting with it), both parties have to communicate with each other before initiating the Core protocol. Within the Core specification, this interaction is presumed and thus not specified. The OpenID Connect Protocol Suite however contains an additional specification, called OpenID Connect Discovery [11], defining "how Clients dynamically discover information about OpenID Providers" [8]. The protocol flow (as defined in the specification) can be separated into two main tasks:

- Location of the OP for an End-User via WebFinger [24], from now on called Issuer Discovery.
- Retrieval of the OP's configuration information via a well-known location of the OP, from now on called Configuration Discovery.

Figure 3.5 depicts the Discovery Flow (containing the above mentioned tasks: Issuer Discovery: Steps 1. - 4.; Configuration Discovery: Steps 5. - 6.) with its most important parameters (second row of each message) together with mention-worth optional ones (enclosed in square brackets).

3. OpenID Connect

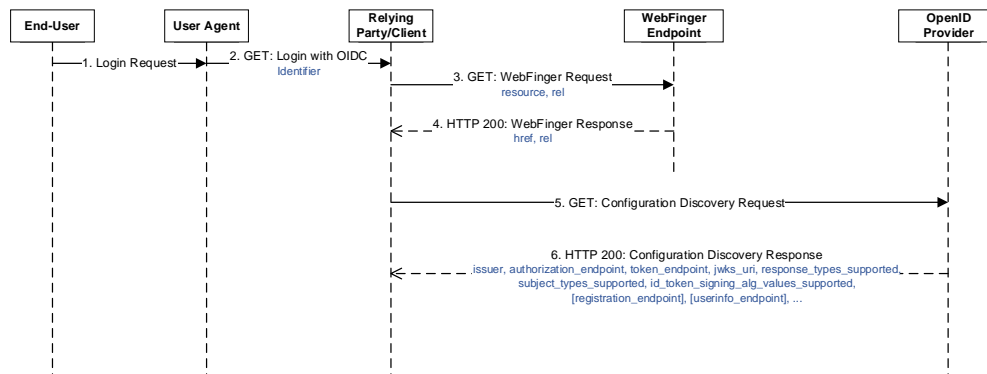


Figure 3.5.: OpenID Connect Discovery Flow

1. - 2. By using a UA to perform OpenID Connect based authentication, the End-User initially supplies the "Login with OIDC" (see for example Figure 3.2) request to the RP with an Identifier in order to help the RP locate the OP for this specific End-User. The supplied Identifier is usually a URL or URI relative reference (as defined in [30]) containing a host value (server where a WebFinger service is hosted) and a **resource** value (identifier for the target End-User that is the subject of the discovery request).
3. After normalizing the received Identifier (see section 2.1 of [11]) and thus determining its **resource** and host, the RP sends a WebFinger request (as defined in [24]), called the Issuer Discovery Request, containing the given **resource** and another parameter called **rel** (URI identifying the type of service whose location is being requested; here it is: <http://openid.net/specs/connect/1.0/issuer> for OpenID Connect Issuer) to the Issuer Discovery Endpoint of the host.

```

1 GET /.well-known/webfinger?resource=https%3A%2F%2FopenidProvider.com%3A8080%2F&rel=http%3A%2F%2Fopenid.net%2Fspecs%2Fconnect%2F1.0%2Fissuer HTTP/1.1
2 Host: openidProvider.com:8080
    
```

Listing 3.5: Non-Normative Example Issuer Discovery Request

4. The location of the requested service, which is in this case the OpenID Provider issuer location, is then returned in the corresponding WebFinger response, called the Issuer Discovery Response, as the value of the **href** member of a links array element with the same **rel** member value as in the request.

```

1 HTTP/1.1 200 OK
2 Content-Type: application/jrd+json
    
```

3. OpenID Connect

```
3 {  
4   "subject": "https://openIDProvider.com:8080/",  
5   "links":  
6   [  
7     {  
8       "rel": "http://openid.net/specs/connect/1.0/issuer",  
9       "href": "https://example.openIDProvider.com"  
10    }  
11  ]  
12 }
```

Listing 3.6: Non-Normative Example Issuer Discovery Response

5. In order to retrieve the configuration information, including the OAuth 2.0 endpoint locations, of the previously identified OP, the received issuer location concatenated with a well-known path (`/.well-known/openid-configuration`) is requested.
6. When receiving a Configuration Discovery Request, the OP responds with a set of claims about its configuration: `issuer` being the URL which the OP claims as its issuer identifier (has to be identical to the `href` value returned in the Issuer Discovery Response from Step 4.), `authorization_endpoint` being the URL of the OP's OAuth 2.0 Authorization Endpoint, `token_endpoint` being the URL of the OP's OAuth 2.0 Token Endpoint, `jwks_uri` being the URL of the OP's JWKS document (containing public key material to verify signatures of the OP), `response_types_supported` being a list of the OAuth 2.0 `response_type` values that this OP supports, `subject_types_supported` being a list of the Subject Identifier types that this OP supports, `id_token_signing_alg_values_supported` being a list of the JWS signing algorithms supported by the OP, and possibly other optional ones.

```
1 HTTP/1.1 200 OK  
2 Content-Type: application/json  
3 {  
4   "issuer": "https://example.openIDProvider.com",  
5   "authorization_endpoint": "https://example.openIDProvider.com/authorize",  
6   "token_endpoint": "https://example.openIDProvider.com/token",  
7   "jwks_uri": "https://example.openIDProvider.com/jwks.json",  
8   "response_types_supported": ["code", "code id_token", "id_token", "token id_token"],  
9   "subject_types_supported": ["public", "pairwise"],  
10  "id_token_signing_alg_values_supported": ["RS256", "ES256", "HS256"],  
11  "registration_endpoint": "https://example.openIDProvider.com/register",  
12  "userinfo_endpoint": "https://example.openIDProvider.com/userinfo"  
13 }
```

Listing 3.7: Non-Normative Example Configuration Discovery Response

3.7. OpenID Connect Dynamic Client Registration Specification

In order for an OpenID Connect RP to commit client-specific metadata (like the `redirect_uri` or the `application_type`) to an OP of a specific End-User as well as to establish a trust relationship with the latter, both parties have to communicate with each other before initiating the Core protocol. Within the Core specification, this interaction is presumed and thus not specified. The OpenID Connect Protocol Suite however contains an additional specification, called OpenID Connect Dynamic Client Registration [12], defining "how clients dynamically register with OpenID Providers" [8].

Figure 3.6 depicts the Dynamic Client Registration Flow (defined in the above mentioned specification) with its most important parameters (second row of each message) together with mention-worthy optional ones (enclosed in square brackets).



Figure 3.6.: OpenID Connect Dynamic Client Registration Flow

1. In order for a RP to dynamically register with the End-User's OP, it needs to provide information about itself to the OP and in return obtain information needed to use it during the OpenID Connect Core protocol. This is accomplished by sending a Client Registration Request to the Registration Endpoint of the OP. This request has to contain at least one mandatory client metadata parameter: `redirect_uris` being an array of Redirect URI values to be used during the OpenID Connect Core protocol. The `redirect_uri` parameter of each Authentication Request (of the Core protocol) has to exactly match one of the registered ones. Other optional request parameters containing additional

3. OpenID Connect

client metadata can also be sent with this request. For OPs that only support authorized registration requests, it is also possible to append an optional Authorization header (as described in [23]), to the registration request. The therefore required initial Access Token is provisioned out-of-band.

```
1 POST /register HTTP/1.1
2 Content-Type: application/json
3 Accept: application/json
4 Host: example.openIDProvider.com
5 {
6   "redirect_uris": ["https://example.client.com/callback", "https://example.client.com/c
   allback2"],
7   "client_name": "Example Client",
8   "token_endpoint_auth_method": "client_secret_basic"
9 }
```

Listing 3.8: Non-Normative Example Client Registration Request

2. When receiving a Client Registration Request, the OP assigns the Client an unique Client Identifier (`client_id`), optionally assigns a Client Secret (`client_secret`), and associates the metadata given in the request with the issued Client Identifier. The OP then responds with the beforehand chosen values and, if applicable, other optional response parameters.

```
1 HTTP/1.1 201 Created
2 Content-Type: application/json
3 Cache-Control: no-store
4 Pragma: no-cache
5 {
6   "client_id": "s6BhdRkqt3",
7   "client_secret": "ZJYCqe3GGRvdudKyZS0XhGv_Z45DuKhCUk0gBR1vZk",
8   "client_secret_expires_at": 1577858400,
9   "client_name": "Example Client",
10  "token_endpoint_auth_method": "client_secret_basic",
11  "redirect_uris": ["https://example.client.com/callback", "https://example.client.com/c
   allback2"]
12 }
```

Listing 3.9: Non-Normative Example Client Registration Response

3.8. Validation Steps

As the security of SSO protocols like OpenID Connect heavily depends on the confidentiality, integrity and authenticity of the, within the protocol, exchanged data, it is of utmost importance for a developer to implement security-relevant validation

3. OpenID Connect

steps as prescribed in the specification. In order to be able to understand the attack-scenarios, to be described in the following chapter, this section enlists validation steps to be performed by the protocols involved parties to guaranty the mentioned attributes. In addition to the following validation steps, each party must validate the correctness and reliability of the certificate presented by their communication partner when the use of TLS is prescribed (see Section 3.4).

3.8.1. Authentication Request Validation

The Authentication Request, received by the OP, must be validated as follows (only security-relevant validation steps are enlisted):

1. The value of the `client_id` parameter (received within the request) must belong to a beforehand registered Client of the OP.
2. The value of the `redirect_uri` parameter (received within the request) has to exactly match at least one of the beforehand registered (of the Client identified by the submitted `client_id`, see above).
3. If the Authentication Response should contain an Access Token, ID Token or both, the value of the `redirect_uri` parameter (received within the request) must use the HTTPS scheme.
4. If the Authentication Response should contain an ID Token, the request has to contain a `nonce` parameter.
5. If the `sub` claim is requested with a specific value for the ID Token (either by the `claims` parameter itself or by usage of the `request` or `request_uri` parameter), the End-User must successfully authenticate herself to the OP resulting in exactly that requested `sub`.

3.8.2. Authentication Response Validation

The Authentication Response, received by the RP, must be validated as follows (only security-relevant validation steps are enlisted):

1. If the `state` parameter was present in the corresponding sent Authentication Request, the value of the `state` parameter (received within the response) must exactly match the one of the request.
2. If the response contains an ID Token, additional validation steps from Section 3.8.4 have to be applied.

3.8.3. Token Request Validation

The Token Request, received by the OP, must be validated as follows (only security-relevant validation steps are enlisted):

1. The value of the `client_id` parameter (received within the request) must belong to a beforehand registered Client of the OP.
2. If agreed upon beforehand, the received Client authentication has to be validated.
3. The value of the `code` parameter (received within the request) must belong to a beforehand, to that Client (identified by the submitted `client_id`, see above), issued one.
4. The value of the `code` parameter (received within the request) must be used the first time within a Token Request.
5. The value of the `redirect_uri` parameter (received within the request) must exactly match the one included in the initial Authentication Request of the Client.

3.8.4. Token Response Validation

The Token Response, received by the RP, must be validated as follows (only security-relevant validation steps are enlisted):

1. If agreed upon beforehand, the received ID Token has to be decrypted using a public key of the OP.
2. The signature of the received ID Token has to be validated using either the `client_secret` or a public key of the OP depending on the used signature algorithm.
3. The `iss` claim of the ID Token (received within the response) has to exactly match the `issuer` identifier of the issuing OP (typically obtained during Discovery, see Section 3.6).
4. The `aud` claim of the ID Token (received within the response) has to contain at least (as it can be an array with more than one element) the Client's `client_id` registered at the OP identified by the `iss` claim.
5. If the received `aud` claim contains multiple Client IDs, the `azp` claim of the ID Token (received within the response) has to exactly match the `client_id` described in the previous step.

3. *OpenID Connect*

6. The **exp** claim of the ID Token (received within the response) must be a time after the current.
7. The **iat** claim of the ID Token (received within the response) must be a time before the current.
8. If the **nonce** parameter was present in the corresponding sent Authentication Request, the **nonce** claim of the ID Token (received within the response) must exactly match the one of the request.
9. If the **response_type** parameter of the corresponding sent Authentication Request valued **id_token token** or **code id_token token**, the left-most half of the hash of the corresponding sent Access Token has to exactly match the **at_hash** claim of the ID Token (received within the response).
10. If the **response_type** parameter of the corresponding sent Authentication Request valued **code id_token** or **code id_token token**, the left-most half of the hash of the corresponding sent Authorization Code has to exactly match the **c_hash** claim of the ID Token (received within the response).

4. Security Analysis

This chapter is intended to outline the general concept of our security analysis of the Single Sign-On protocol OpenID Connect, give references to related work, as well as reveal several custom designed attack-scenarios. In addition to this, we will demonstrate the applicability of such scenarios by introducing two self-developed proof-of-concept Java pentest applications for auditing OpenID Connect implementations. Furthermore, we will introduce a custom designed security aspect catalog and reveal the results of our practical analysis with the help of the latter.

4.1. Security Model

This section will give a detailed description of the security model used in the proper analysis of the protocol. To do so, we will at first outline the objectives of an attacker together with mention-worth assumptions when analyzing a Single-Sign On protocol. Later on we will define the capabilities of an attacker as well as the behavior of a victim.

4.1.1. Objectives of the Attacker

The essential goal of the attacker is to impersonate her victim when accessing a service of a RP. This subsequently leads to an unauthorized access of protected resources of the victim. Within the context of OpenID Connect, this could either mean acquiring an ID Token issued for the victim and redeeming it at the proper RP or tricking a RP into accepting a manipulated ID Token resulting in the impersonation of the victim.

4.1.2. Assumptions

As a huge proportion of the security of OpenID Connect is based on the utilization of TLS to secure the communication between the involved parties, we consider, if TLS is used, TLS to be secure. For instance, the implemented version of TLS is up-to-date and not vulnerable to some known attack, for example, BEAST [31], which

could decrypt or manipulate data sent from one party to another. Deceiving an End-User, via for example, Cross-Site Scripting, Phishing or false TLS certificates, is also considered out of scope. Basically all software used by the End-User, for example, her User Agent, and furthermore her Operating System is assumed to be not compromised. Thus the attacker has no way of exploiting vulnerabilities within the UA or the Operating System of the End-User. Furthermore, all tests on live implementations of the protocol can be regarded as black-box tests, due to the lack of the source code or any documentation of the implemented libraries. Beyond that, when solely acting as the End-User, the attacker cannot see or influence any data which is directly sent from the RP to the OP or vice versa.

4.1.3. Capabilities of the Attacker

The attacks to-be-introduced in this thesis have been strictly verified in the "web attacker" model [32]. In contrast to the network-based attacker model (cryptographic attacker model), the web attacker does not need full control over the network and thus is not able to eavesdrop or manipulate network connections. She is however able to use an UA or a custom HTTP client to send HTTP requests to every publicly available web application in the web and subsequently receive its response. HTTP parameters (to-be-sent within a request) as well as headers can be freely chosen or manipulated. Requests of the attacker can also be delayed or aborted and responses do not need to be handled in a standard-compliant way, for example, HTTP redirect responses do not need to be followed. For tests within live implementations the attacker is able to register as many accounts on a specific RP or OP as she wishes. Furthermore, links (e.g. sent via email) or web-blog commentaries can be used to lure the victim into opening a (manipulated) URI to, for example, conduct Cross-Site Request Forgery attacks. Other attacks on the web application part not directly handling OpenID Connect, like SQL Injections or Cross-Site Scripting attacks, are considered out of scope. In addition to that, an attacker may set up her own web application(s) extending her role from End-User only to RP and OP as well. With that capability the attacker is also able to influence data which is directly sent from the RP to the OP or vice versa.

4.1.4. Behavior of the Victim

Within our security model, the victim is assumed to visit every publicly available web application she wants to or is directed to (e.g. by sending the victim a link to a web application controlled by the attacker) [33]. HTTP parameters within such requests, made by the victim, are not checked for sanity or validity. Phishing attempts, like

reconstructing a known website for example, are however discovered by the victim and thus not leading to the exposure of sensitive information of the latter.

4.2. Related Work

Writing this thesis, there is still a lack of security-relevant research work on OpenID Connect itself. Therefore, this section refers to related work about the protocols OAuth 2.0 and OpenID 2.0 as OpenID Connect is based on the concepts of both protocols.

Formal Verification of OAuth 2.0 using Alloy Framework. In June 2011, Suhas Pai et al. published a paper called "Formal Verification of OAuth 2.0 using Alloy Framework" [34]. Within this paper the authors conducted a knowledge flow analysis [35] for formal specification of OAuth 2.0. Using the Alloy modeling language [36] for specification and the Alloy Analyzer for verification they were able to discover a known security vulnerability [37] of the protocol.

Universally Composable Security Analysis of OAuth v2.0. In September 2011, Suresh Chari, Charanjit Jutla and Arnab Roy published a paper called "Universally Composable Security Analysis of OAuth v2.0" [38]. Intent of this paper was to conduct a security analysis of an universally-composable (UC) [39] realization of the Authorization Code Grant Type of OAuth 2.0. Unlike common UC settings, the implementation of Chari et al. does not require the involved parties of the protocol to decide on a session identifier in advance.

The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. The paper "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems" [40], published by San-Tsai Sun and Konstantin Beznosov in 2012, gives a thorough examination of the security of OAuth 2.0 deployed in real-life implementations. By analyzing HTTP traffic and conducting known web application attacks like XSS or XSRF against multiple SPs using the OAuth protocol, Sun and Beznosov uncovered several vulnerabilities resulting in unauthorized access of protected resources or user impersonification. Altogether they analyzed OAuth (-related) implementations of 96 SPs as well as the three IdPs: Facebook, Microsoft, and Google. As most of the found vulnerabilities were "caused by a set of design decisions that trade security for implementation simplicity" they

4. Security Analysis

furthermore gave several design and implementation suggestions to improve the security of OAuth 2.0.

Sicherheitsanalyse von OAuth 2.0 mittels Web Angriffen auf bestehende Implementierungen. Within his master thesis "Sicherheitsanalyse von OAuth 2.0 mittels Web Angriffen auf bestehende Implementierungen" [2], released in December 2013, Christoph Nickel made a practical security analysis of several live implementations of the OAuth 2.0 protocol. While using the web attacker model, the thesis analyzes common protocol features as well as reactions to protocol-parameter-changes of the tested Client implementations. The author discovered that, in many cases, Authorization Servers poorly validated the received Redirect URI of a Client enabling malicious users to access the Access Token of another user. Additionally, Nickel introduced a concept as well as a prototypal implementation for automated testing of the OAuth 2.0 protocol.

SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In the beginning of 2014, Yuchen Zhou and David Evans published a paper called "Automated Testing of Web Applications for Single Sign-On Vulnerabilities" [41]. Within their paper they presented five novel attacks on applications using Facebook Single Sign-On (SSO) Application Programming Interfaces(APIs). Facebook SSO is using a variant of the OAuth 2.0 protocol to give applications (implementing it) the possibility for delegated authentication and authorization of their users. All presented attacks are stemming from implementation flaws of the Facebook SSO APIs and result in user impersonification. They additionally introduced their implementation of an automatic vulnerability checker utilizing the described attacks and furthermore presented a large-scale study on applications using Facebook SSO discovering that over 20% of them suffered from at least one of their attacks.

The FAT Attack. Facebook Social Login Session Hijacking. In July 2014, MetaIntell made a press release [42] disclosing a security vulnerability in the Facebook SDK for IOS and Android. Facebook SDK, used by 71 of the top 100 free iOS apps together with 31 of the top 100 Android apps [42], can be used to utilize a variant of the OAuth 2.0 protocol to obtain an Access Token providing access to the user's Facebook APIs. Furthermore, it can be used to confirm a person's identity for registration and sign-in purposes. MetaIntell discovered that the Access Token of a user is insecurely cached on the device and thus vulnerable to session hijacking.

Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. In March 2012, San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov published a paper called "Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures" [43]. Within their paper, the authors summarized their formal analysis of OpenID 2.0 as well as an empirical evaluation of 132 popular websites that support OpenID. Their formal analysis revealed "that the protocol does not guarantee the authenticity and integrity of the authentication request". Their empirical evaluation revealed that 81% of the tested OpenID-enabled websites were prone to Cross-Site Request Forgery (CSRF) attacks allowing "an attacker to stealthily force a victim user to sign into the OpenID supporting website and launch subsequent CSRF attacks". Other CSRF vectors as well as extension-parameter-forgery attacks resulting in the impersonation of a victim were also introduced.

Untrusted Third Parties: When IdPs Break Bad. Within their paper "Untrusted Third Parties: When IdPs Break Bad" [1], Vladislav Mladenov and Christian Mainka summarized their research work about analyzing OpenID 2.0 from the RP point of view. While using the web attacker model, they designed four novel attacks on OpenID resulting in the impersonation ("Identity Theft") of a rightful user by an attacker. All attacks are based on insufficient validation of security-aspects of the protocol on the RP- (token verification) or OP -side. To test the applicability of the designed attacks, 16 OpenID implementations were audited and 11 of them found vulnerable. Additionally, a free and open source tool, called OpenID Attacker, capable of executing the attacks was developed by the authors within their research.

4.3. OpenID Connect Pentest Applications

In the beginning of the protocol analysis, we decided to use the open source reference implementation of OpenID Connect and OAuth 2.0, "MITREid Connect" [44], from the MITRE Corporation and MIT Kerberos and Internet Trust (KIT), to get a detailed understanding of the concepts as well as the communication flows of the protocol. MITREid Connect offers a Java-based implementation of the OP- as well as the RP -side of OpenID Connect. In addition to the Core specification, it implements its extensions Discovery and Dynamic Client Registration. After using the implementation to connect to several OP test-deployments, found via the Fifth OpenID Connect Interop (OC5) [9], we decided to implement our own OpenID Connect implementation of the OP- as well as the RP -side. This decision was mostly

4. Security Analysis

influenced by the wish to analyze the protocol step-by-step being able to influence all parameters of each step, which the reference implementation MITREid Connect could not provide (without having to rebuild the web application each time). Both developed applications are written in Java and utilize the following third party libraries to add certain functionality:

- **NanoHttpd:** NanoHttpd [45], developed by Paul Hawke, is an open source implementation of a light-weight HTTP server designed to be embedded into other Java applications. In order to enable TLS support for our server, we used a slightly customized version of this library.
- **jose.4.j:** jose.4.j [46], developed by Brian Campbell, is an open source implementation of the Javascript Object Signing and Encryption (JOSE) specification suite. The library can be used to access functionality defined in the [28, 29, 26] specifications and subsequently to build JWTs [27].
- **JSON.simple:** JSON.simple [47], developed by Yidong Fang, is an open source library for encoding and decoding JSON text.
- **Apache HttpClient:** Apache HttpClient [48], as a module of the Apache HttpComponents project [49], is a HTTP/1.1 compliant HTTP agent implementation to provide client-sided HTTP services extending the basic functionality of the java.net package.

4.3.1. The Relying Party

Our developed Java GUI "OpenID Connect Provider TestSuite" enables a user to feature-test and/or security-audit OpenID Connect Provider implementations. It therefore acts as an OpenID Connect RP being able to conduct each protocol flow of OpenID Connect step-by-step. The tool can be configured manually by providing the Provider EndPoints view (see Figure 4.1) with a set of endpoint locations of the to-be-tested OP or automated by providing the Metadata Discovery view (see Figure 4.2) with the Configuration Information Endpoint of the OP: Once clicked on the "Send Metadata Discovery Request" button, the tool automatically sends a Configuration Discovery Request (as described in Section 3.6) to the OP and fills the Provider EndPoints view with the corresponding received data.

4. Security Analysis

The screenshot shows a web interface titled "Provider EndPoints". It contains five input fields, each with a label and a text value:

- registration_endpoint**: `https://mitreid.org/register`
- authorization_endpoint**: `https://mitreid.org/authorize`
- token_endpoint**: `https://mitreid.org/token`
- jwtks_uri**: `https://mitreid.org/jwk`
- userinfo_endpoint**: `https://mitreid.org/userinfo`

Figure 4.1.: OpenID Connect Provider TestSuite - Provider EndPoints

The screenshot shows a web interface titled "Metadata Discovery". It contains one input field with the label "Well-Known URI" and the text value `https://mitreid.org/well-known/openid-configuration`. Below the input field is a button labeled "Send Metadata Discovery Request".

Figure 4.2.: OpenID Connect Provider TestSuite - Metadata Discovery

If the Configuration Information Endpoint of the OP is not known or support for Issuer Discovery has to be tested, the WebFinger Discovery view, see Figure 4.3, can be provided with an Identifier: Once clicked on the "Send WebFinger Request" button, the tool automatically sends a Issuer Discovery Request (as described in Section 3.6) to the OP and fills the Metadata Discovery view with the corresponding received data.

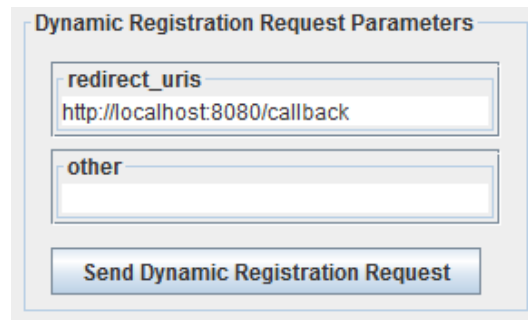
The screenshot shows a web interface titled "WebFinger Discovery". It contains one input field with the label "identifier" and the text value `user@mitreid.org`. Below the input field is a button labeled "Send WebFinger Request".

Figure 4.3.: OpenID Connect Provider TestSuite - WebFinger Discovery

In order to register our Client with a specific OP or just test its support for Dynamic Client Registration, the Dynamic Registration Request Parameters view, see Figure 4.4, can be provided with arbitrary parameters: Once clicked on the "Send Dynamic Registration Request" button, the tool automatically sends a Client Regis-

4. Security Analysis

tration Request (as described in Section 3.7) to the Registration Endpoint (retrieved from the Provider EndPoints view) of the OP.



Dynamic Registration Request Parameters

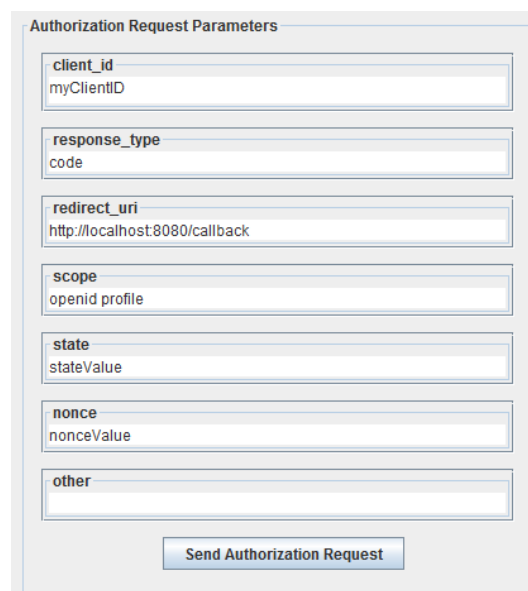
redirect_uris
http://localhost:8080/callback

other

Send Dynamic Registration Request

Figure 4.4.: OpenID Connect Provider TestSuite - Dynamic Registration Request Parameters

To submit an actual Authentication Request, the Authorization Request Parameters view, see Figure 4.5, can be provided with arbitrary parameters: Once clicked on the "Send Authorization Request" button, the tool opens the UA of the user and submits a request (as described in Section 3.5.2, Section 3.5.3 or Section 3.5.4) to the Authorization Endpoint (retrieved from the Provider EndPoints view) of the OP with the beforehand specified parameters. If the tool subsequently receives an Authentication Response, it is parsed and displayed to the user.



Authorization Request Parameters

client_id
myClientID

response_type
code

redirect_uri
http://localhost:8080/callback

scope
openid profile

state
stateValue

nonce
nonceValue

other

Send Authorization Request

Figure 4.5.: OpenID Connect Provider TestSuite - Authorization Request Parameters

4. Security Analysis

cific domain and all accessible endpoints are bound to a specific path as seen in Figure 4.8.

```
Started server for URL: http://alpha.cloud.nds.rub.de; Port: 8080
Accessible endpoints are:
http://alpha.cloud.nds.rub.de/.well-known/webfinger
http://alpha.cloud.nds.rub.de/.well-known/openid-configuration
http://alpha.cloud.nds.rub.de/register
http://alpha.cloud.nds.rub.de/auth
http://alpha.cloud.nds.rub.de/token
http://alpha.cloud.nds.rub.de/jwk
http://alpha.cloud.nds.rub.de/userinfo
```

Figure 4.8.: OpenID Connect Client TestSuite - Accessible Endpoints

All incoming requests are then handled upon their path: If a specific endpoint of the Provider is requested (e.g. `http://alpha.cloud.nds.rub.de/auth`) the tool invokes the corresponding response handler and creates a customized response based on several input parameters provided by the user:

If the Configuration Information Endpoint is requested, response parameters of the Metadata Discovery Response Data view, see Figure 4.9, are gathered, upon these a response is created and subsequently sent back to the Client.

Metadata Discovery Response Data	
issuer	<input type="text" value="http://alpha.cloud.nds.rub.de"/>
other	<input example"}"="" type="text" value='{"scopes_supported":["openid", "profile"]}"/></td></tr></tbody></table></div><div data-bbox="161 599 833 616" data-label="Caption"><p>Figure 4.9.: OpenID Connect Client TestSuite - Metadata Discovery Response Data</p></div><div data-bbox="161 644 833 701" data-label="Text"><p>If the Dynamic Registration Endpoint is requested, response parameters of the Dynamic Client Registration Response Data view, see Figure 4.10, are gathered, upon these a response is created and subsequently sent back to the Client.</p></div><div data-bbox="328 711 660 788" data-label="Form"><table border="1"><thead><tr><th colspan="2">Dynamic Client Registration Response Data</th></tr></thead><tbody><tr><td>client_id</td><td><input type="text" value="TestClient"/></td></tr><tr><td>client_secret</td><td><input type="text" value="TestPassword"/></td></tr><tr><td>other</td><td><input type="text" value=' {"client_name":"my=""/>

Figure 4.10.: OpenID Connect Client TestSuite - Dynamic Client Registration Response Data

If the Authorization Endpoint is requested, response parameters of the Authentication Response Data view, see Figure 4.11, are gathered, upon these a response is created and subsequently sent back to the Client.

Figure 4.11.: OpenID Connect Client TestSuite - Authentication Response Data

Figure 4.12.: OpenID Connect Client TestSuite - Token Response Data

Figure 4.13.: OpenID Connect Client TestSuite - Token Request-Response Pair

4.4. Attacks / Attack-Scenarios

This section is intended to enumerate several attack-scenarios corresponding to our, in Section 4.1, defined security model. There are two basic requirements which apply to all to-be-described attack-scenarios:

1. As within the context of OpenID Connect, the identity of an End-User is proven to a RP via a combination of an ID Token's **sub** and **iss** claim (e.g. $ID = sub : iss$), an attacker has to control both values in order to impersonate her victim.
2. As the **sub** claim is just a unique identifier, it cannot be verified by a RP.

4.4.1. ID Spoofing

ID Spoofing (IDS) is an attack which targets the ID Token verification part of a RP. If the **iss** claim verification (see Step 3. of Section 3.8.4) by a RP is not handled correctly, an attacker may be able to log in as an arbitrary End-User of this application: To perform an IDS attack an attacker has to act as an End-User and an OP simultaneously. Let the identity of the victim be represented by $ID_V = sub_V : iss_V$ and the identity of the attacker by $ID_A = sub_A : iss_A$ with iss_V belonging to OP_V and iss_A belonging to OP_A . In theory, OP_A should not be able to issue a valid ID Token t^* containing iss_V . In the attack however, the attacker uses her OP OP_A to send exactly t^* to a RP with which her victim is registered. If the RP accepts t^* the attack is successful (and the attacker should be logged in with ID_V).

4.4.2. Issuer Confusion

Issuer Confusion (IC) is an attack which targets the Configuration Discovery verification part of a RP. If the **issuer** claim verification (see Step 6. of Section 3.6) by a RP is not handled correctly, an attacker may be able to log in as an arbitrary End-User of this application: To perform an IC attack an attacker has to act as an End-User and an OP simultaneously. Let the identity of the victim be represented by $ID_V = sub_V : iss_V$ and the identity of the attacker by $ID_A = sub_A : iss_A$ with iss_V and $issuer_V$ (being the **issuer** claim of the Provider's Configuration Discovery Response) belonging to OP_V and iss_A and $issuer_A$ belonging to OP_A . In theory, OP_A should not be able to send a valid Configuration Discovery Response cdr^* containing $issuer_V$. In the attack however, the attacker uses her OP OP_A to send exactly cdr^* to a RP with which her victim is registered. If the RP accepts cdr^* and later on

4. Security Analysis

compares the `iss` claim of the ID Token (which also contains `issV`) to `issuerv` from `cdr*` the attack is successful (and the attacker should be logged in with `IDV`).

4.4.3. Signature Manipulation

Signature Manipulation (SM) is an attack which targets the ID Token verification part of a RP. If the signature verification (see Step 2. of Section 3.8.4) by a RP is not handled correctly, an attacker may be able to log in as an arbitrary End-User of this application: To perform a SM attack an attacker has to act as an End-User only. Let the ID Token of the victim be represented by $t_V = ID_V || \sigma_V$ where $ID_V = sub_V : iss_V$ and σ_V is the signature or HMAC of ID_V . In theory, an attacker should not be able to alter the contents of her ID Token $t_A = ID_A || \sigma_A$ to, for example, $t^* = ID_V || \sigma$ without being noticed by the RP verifying the token. In the attack however, the attacker uses an Authentication Flow (e.g. Implicit) where she has direct access to the issued ID Token and alters the intercepted token as described above. If the RP accepts t^* the attack is successful (and the attacker should be logged in with `IDV`).

4.4.4. Sub Claim Spoofing

Sub Claim Spoofing (SCS) is an attack which targets the Authentication Request verification part of an OP. If the `sub` claim verification (see Step 5. of Section 3.8.1) by an OP is not handled correctly, an attacker may be able to log in as any End-User registered with this OP at a RP of her choice: Within the Authentication Request of OpenID Connect, the RP has the possibility to request individual claims about the End-User either by using the `claims` parameter or by using an additional JWT containing a whole OpenID Connect request via the `request` or `request_uri` parameter. An example of a decoded `claims` parameter value can be seen in Listing 4.1.

```
1 {
2   "userinfo":
3   {
4     "given_name": {"essential": true},
5     "nickname": null,
6     "email": {"essential": true},
7     "email_verified": {"essential": true},
8     "picture": null
9   },
10  "id_token":
11  {
12    "auth_time": {"essential": true},
```

4. Security Analysis

```

13   "acr": {"values": ["urn:mace:incommon:iap:silver"]}
14   }
15 }
    
```

Listing 4.1: Non-Normative Example Claims Request [10]

Within our example the RP requests the additional claims `auth_time` and `acr` (with the value `"urn:mace:incommon:iap:silver"`) to be added to the default claims in the ID Token. As in the example (for the `acr` claim) a RP has the possibility to request that an individual claim is returned with a particular value. For instance the decoded `claims` parameter value:

```

1 {"id_token":{"sub":{"value": "subOfTheVictim"}}}
    
```

Listing 4.2: Sub Claim Request

can be used to specify that the request applies to the End-User with Subject Identifier `"subOfTheVictim"`. To perform an SCS attack an attacker has to act as an End-User only. Let the identity of the victim be represented by $ID_V = sub_V : iss^*$ and the identity of the attacker by $ID_A = sub_A : iss^*$. In theory, if the `sub` claim is requested with a specific value for the ID Token, the OP must only send a positive response if the End-User identified by that `sub` value has an active session with the OP or has been authenticated as a result of the request. In the attack however, the attacker appends a `claims` parameter valuing `{"id_token": {"sub": {"value": sub_V }}}` to the Authentication Request to the OP identified by iss^* . If the OP subsequently issues an ID Token t^* containing ID_V , although the attacker did not authenticate with the credentials resulting in sub_V , the attack is successful (and the attacker should be logged in with ID_V).

4.4.5. Redirect URI Manipulation

Redirect URI Manipulation (RUM) is an attack which targets the Authentication Request verification part of an OP. If the `redirect_uri` verification (see Step 2. of Section 3.8.1) by an OP is not handled correctly, an attacker may be able to log in as any End-User registered with this OP at a RP of her choice. As RUM is applicable to Authentication using the Authorization Code Flow and Authentication using the Implicit Flow, there are actually two variants of the attack:

1. Redirect URI Manipulation using the Code Flow (RUM #1) Within this attack-scenario the Authentication Response of a victim is redirected to a website

4. Security Analysis

controlled by the attacker. The thereby obtained authorization code is then used within a separate protocol flow, initiated by the attacker, to redeem it for an ID Token of the victim. Figure 4.14 depicts the attack-procedure of the attacker using the Authorization Code Flow (all manipulated request parameters are highlighted in red).

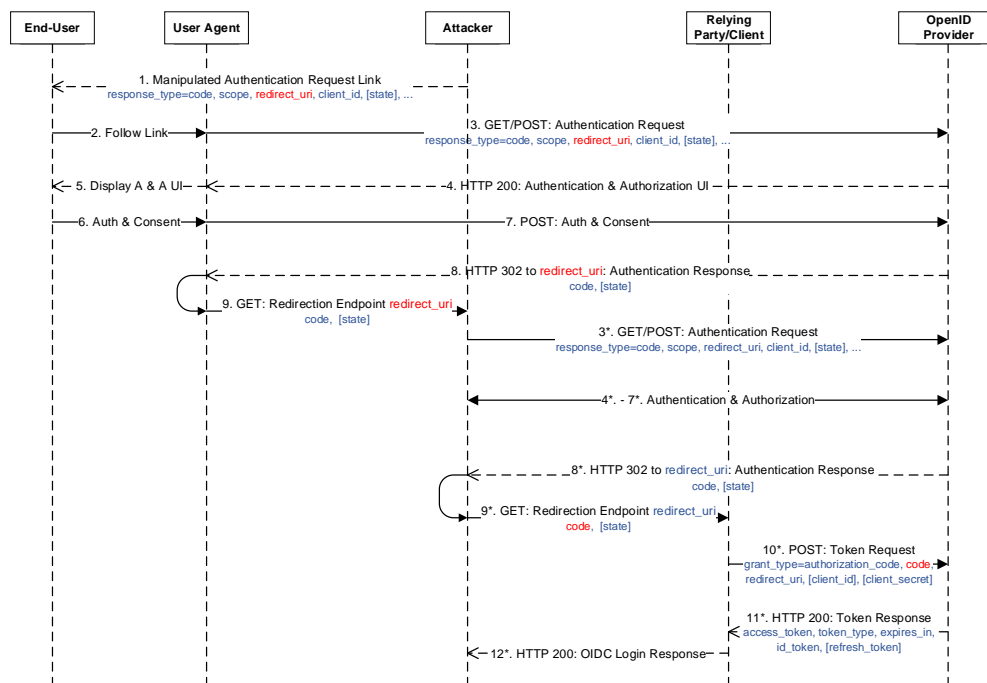


Figure 4.14.: RUM using Authorization Code Flow

1. The attacker sends her victim a manipulated Authentication Request (e.g. via a link) containing a Redirect URI pointing to a website controlled by the attacker.
2. - 7. The victim follows the link and thus starts the Authorization Code Flow of OpenID Connect. In the following steps she authenticates to the OP and consents the Authentication Request of the Client.
8. - 9. When the UA of the victim receives the Authentication Response, it is redirected to the server of the attacker, thus sending her the authorization code.
- 3*. - 7*. The attacker initiates her own protocol flow with the same Client. The Redirect URI in this case is however not manipulated.

4. Security Analysis

8*. - 9*. When receiving the Authentication Response via her UA, the attacker at first substitutes the received authorization code with the one received in Step 9. and then follows the redirect.

10*. - 11*. The Client redeems the received authorization code of the attacker for an ID Token.

12*. The attacker is notified that she is now logged in with the identity of the victim.

2. Redirect URI Manipulation using the Implicit Flow (RUM #2) Within this attack-scenario the Authentication Response of a victim is redirected to a website controlled by the attacker. The thereby obtained ID Token of the victim is then used within a separate protocol flow, initiated by the attacker, to send it to the Client. Figure 4.15 depicts the attack-procedure of the attacker using the Implicit Flow (all manipulated request parameters are highlighted in red).

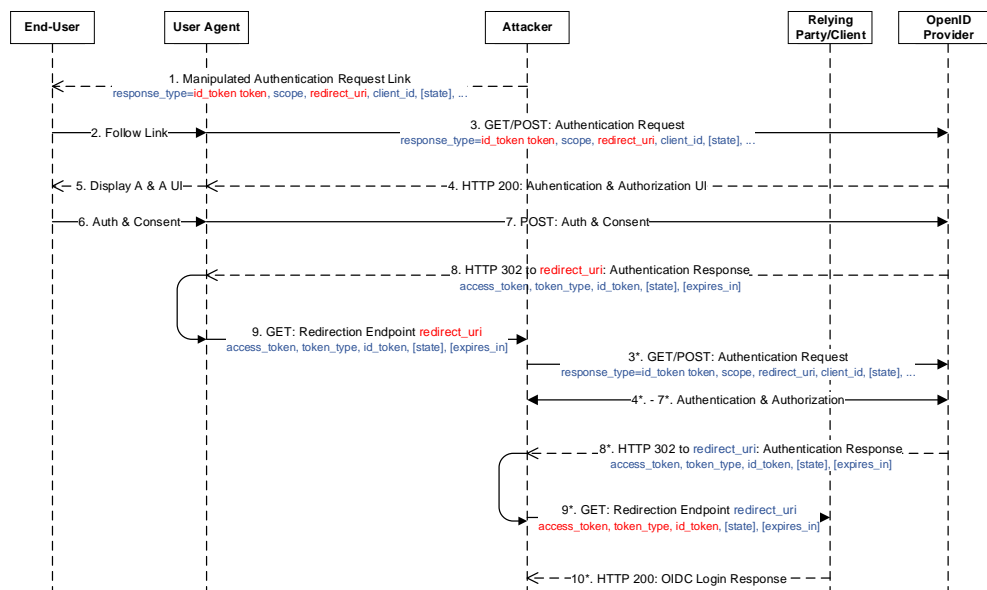


Figure 4.15.: RUM using Implicit Flow

1. The attacker sends her victim a manipulated Authentication Request (e.g. via a link) containing a Redirect URI pointing to a website controlled by the attacker and a Response Type indicating the usage of the Implicit Flow.

4. Security Analysis

2. - 7. The victim follows the link and thus starts the Implicit Flow of OpenID Connect. In the following steps she authenticates to the OP and consents the Authentication Request of the Client.
8. - 9. When the UA of the victim receives the Authentication Response, it is redirected to the server of the attacker, thus sending her the ID Token issued for the victim.
- 3*. - 7*. The attacker initiates her own protocol flow with the same Client. The Redirect URI in this case is however not manipulated.
- 8*. - 9*. When receiving the Authentication Response via her UA, the attacker at first substitutes the received ID Token with the one received in Step 9. and then follows the redirect.
- 10*. The attacker is notified that she is now logged in with the identity of the victim.

4.5. Provider / Library Selection

To arrange a list of libraries implementing either the OP part of OpenID Connect 1.0 (and optionally its extensions Discovery and Dynamic Client Registration) or the RP part of such, we used the official OpenID website [50] alongside the official feature test initiative of individual deployed OpenID Connect implementations [9]. Since implementation flaws are independent of programming languages, we tried to cover every available language. Our target list contains implementations written in the languages Java, C, PHP, Python and Ruby.

4.5.1. Relying Party Implementations

Table 4.1 shows the selected RP libraries, described by their name, URL (of the project page), tested version and a check-box defining if the library contains a working test-implementation of the code (out of the box - checkmark in column Box) or if it is just a framework for building OpenID Connect services.

As things turned out, mostly due to the, at the time of this writing, just recently released final version of the Core specification, we were not able to find live (meaning running in a productive environment) services implementing the RP part of OpenID Connect 1.0 worth testing.

4. Security Analysis

Name	URL	Language	Tested Version	Box
MITREid Connect	http://kit.mit.edu/projects/mitreid-connect	Java	1.1.2	✓
mod_auth_openidc	https://github.com/pingidentity/mod_auth_openidc	C	1.3	✓
Nimbus OAuth 2.0 SDK with OpenID Connect extensions	https://bitbucket.org/connect2id/oauth-2.0-sdk-with-openid-connect-extensions	Java	3.3	✓
Google OAuth Client Library for Java	https://code.google.com/p/google-oauth-java-client/	Java	1.18.0-rc	×
Apache Oltu	https://oltu.apache.org/	Java	0.31 / 1.0.0	×
phpOIDC	https://bitbucket.org/PEOFIAMP/phpoidc	PHP	Commit 0d7944b	✓
Drupal OpenID Connect Module	https://www.drupal.org/project/openid_connect	PHP	7.x-1.0-beta 1	✓
pyoidc	https://github.com/rohe/pyoidc	Python	0.5.0beta	✓
Ruby OpenIDConnect	https://github.com/nov/openid_connect	Ruby	0.8.0	✓

Table 4.1.: Relying Party Libraries

4.5.2. OpenID Provider Implementations

Table 4.2 shows the selected OpenID Provider libraries, described by their name, URL (of the project page), tested version and a check-box defining if the library contains a working test-implementation of the code (out of the box - checkmark in column Box) or if it is just a framework for building OpenID Connect services. Table 4.3 shows a list of selected live (meaning running in a productive environment) services implementing the OP part of OpenID Connect, described by their name, URL (of the service page or developers page), and a description of the service.

4. Security Analysis

Name	URL	Language	Tested Version	Box
Thinkecture IdentityServer v3	https://github.com/thinktecture/Thinkecture.IdentityServer.v3	C#	3	✓
Nimbus OAuth 2.0 SDK with OpenID Connect extensions	https://bitbucket.org/connect2id/oauth-2.0-sdk-with-openid-connect-extensions	Java	3.4	×
MITREid Connect	http://kit.mit.edu/projects/mitreid-connect	Java	1.1.8	✓
oxAuth	http://www.gluu.org/open-source/open-source-vs-on-demand/	Java	1.1.0.Final	✓
Apache Oltu	https://oltu.apache.org/	Java	0.31 / 1.0.0	×
phpOIDC	https://bitbucket.org/PEOFIAMP/phpoidc	PHP	Commit e6e82c4	✓
oauth2-server-php	https://github.com/bshaffer/oauth2-server-php	PHP	1.4	✓
pyoidc	https://github.com/rohe/pyoidc	Python	0.5.0beta	✓
Ruby OpenIDConnect	https://github.com/nov/openid_connect	Ruby	0.8.1	✓

Table 4.2.: OpenID Provider Libraries

Name	URL	Description
Google+ Sign-In	https://developers.google.com/accounts/docs/OAuth2Login	Search Engine, Identity Services, ...
Log In with PayPal	https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/	E-Commerce, Identity Services, ...
Salesforce Identity	https://developer.salesforce.com/page/Inside_OpenID_Connect_on_Force.com	Cloud Computing, Identity Services, ...

Table 4.3.: OpenID Provider Live Implementations

4.6. Security Aspect Catalog

In order to be able to conduct a thorough practical analysis of each selected implementation, this section introduces two custom designed security aspect catalogs to be utilized when classifying certain security-related aspects of an OP or RP implementation. With the help of the defined catalogs, testing implementations for

4. Security Analysis

the attack-scenarios introduced in Section 4.4 can be facilitated. Both catalogs are segmented into three parts: 1. The name or the class of the security aspect; 2. A description of the latter or a question exemplifying the to-be-tested security aspect; 3. A statement of the applicability of the security aspect. The first catalog, see Table 4.4, is intended to reveal important general information of a tested RP or OP. Therefore, it defines several protocol-specific questions alongside security-related and non-security-related characteristics of the implementation itself. The second catalog, see Table 4.5, is intended to gather information about the implementation of the most important security-related validation steps (see Section 3.8) of the protocol. It is segmented by the certain messages sent during an Authentication Flow.

Security Aspect	Description	Applicable to
Authentication Flow		
Response Type(s)	Which Response Type(s) are supported?	RP & OP
Additional-Specifications		
Discovery	Is OpenID Connect Discovery supported?	RP & OP
Dynamic Client Registration	Is OpenID Connect Dynamic Client Registration supported?	RP & OP
TLS Security		
Non-HTTPS #1	Are non-HTTPS OpenID Providers accepted?	RP
Non-HTTPS #2	Are non-HTTPS Relying Parties accepted?	OP
Client Authentication		
Client Authentication	Which Client Authentication methods are supported?	RP & OP
Authentication-Request Parameters		
claims_parameter_-supported	Can claims be requested using the claims request parameter?	OP
request_parameter_-supported	Can request objects be passed by value using the request request parameter?	OP
ID Token		
Validity	How long is an issued ID Token valid?	OP
Miscellaneous		
Account Separation	Are OpenID Connect user-accounts separated from "normal" ones?	RP
Multiple OPs	Are multiple OpenID Providers supported?	RP

Table 4.4.: Security Aspect Catalog - General Information

4. Security Analysis

Security Aspect	Description	Applicable to
Configuration-Discovery Response		
issuer claim	Verification if the received issuer claim value is identical to the href value received in the corresponding Issuer Discovery Response.	RP
Authentication-Request		
redirect_uri Parameter	Verification if the received redirect_uri parameter matches at least one of the of the Clients, identified by the submitted client_id , beforehand registered.	OP
sub Claim	Verification if the sub claim is requested with a specific value which does not match the of the authenticated End-User.	OP
Authentication-Response		
state Parameter	Verification if the received state parameter matches the one sent in the corresponding Authentication Request.	RP
Token Request		
redirect_uri Parameter	Verification if the received redirect_uri parameter matches the one of the initial Authentication Request.	OP
Token Response		
iss Claim	Verification if the received iss claim value matches the issuer identifier of the issuing OpenID Provider	RP
aud Claim	Verification if the received aud claim contains the Clients client_id registered at the OpenID Provider identified by the iss claim.	RP
nonce Claim	Verification if the received nonce claim matches the one sent in the corresponding Authentication Request.	RP
iat Claim	Verification if the received iat claim values a time before the current.	RP
exp Claim	Verification if the received exp claim values a time after the current.	RP
Signature	Verification if the signature of the received ID Token is valid.	RP
UserInfo Response		
sub Claim	Verification if the received sub claim matches the one of the corresponding ID Token.	RP

Table 4.5.: Security Aspect Catalog - Validation

4.7. Practical Analysis

Table 4.6 and Table 4.7 depict the results of our practical analysis of the selected RP implementations mentioned in Section 4.5.1. Table 4.8 and Table 4.9 depict the results of our practical analysis of the selected OP libraries mentioned in Section 4.5.2. All tables strictly correspond to our defined security aspect catalogs from Section 4.6. Almost all results of the analysis were produced by testing out of the box test-implementations (developed by the library developers) of the library code itself. Thus all libraries which did not provide a working test-implementation of their code could not be tested for certain security aspects (as they heavily depend on the exact implementation of the framework). Beneath the tables we will summarize several aspects of the catalogs and, if need be, exemplify some of the results of certain RP and OP implementations. Furthermore, Table 4.12, Table 4.13 and Table 4.14 show the applicability of our in Section 4.4 defined attack-scenarios on the selected RP libraries, OP libraries as well as the selected OP live implementations.

4. Security Analysis

	MITREid Connect	mod_auth_openidc	Nimbus OAuth 2.0 SDK with OpenID Connect extensions	Google OAuth Client Library for Java	Apache Oltu	phpOIDC	Drupal OpenID Connect Module	pyoidc	Ruby OpenIDConnect
Authentication Flow									
Response Type(s)	AC	AC, I, H*	AC, I, H**	AC	n/a	AC, I, H*	AC	AC, I*	AC
Additional-Specifications									
Discovery	✓	✓	✓	×	×	✓	×	✓	✓
Dynamic Client Registration	✓	✓	✓	×	×	✓	×	✓	✓
TLS Security									
Non-HTTPS #1	✓	✓	✓	✓	n/a	×	✓	×	✓
Client Authentication									
Client Authentication	B,P, JWT	B,P	B,P, JWT	n/a	n/a	B,P, JWT	P	B,P, JWT	B,P
Miscellaneous									
Account Separation	n/a	n/a	n/a	n/a	n/a	n/a	✓	n/a	×
Multiple OPs	✓	✓	✓	×	×	✓	✓	✓	✓

Table 4.6.: Relying Party Libraries - General Information

Legend: AC = code; I = id_token, id_token token; H = code id_token, code token, code id_token token; I* = id_token token; H* = code id_token, code id_token token; H** = code id_token; B = client_secret_basic; P = client_secret_post; JWT = client_secret_jwt and/or private_key_jwt

4. Security Analysis

	MITREid Connect	mod_auth_openidc	Nimbus OAuth 2.0 SDK with OpenID Connect extensions	Google OAuth Client Library for Java	Apache Oltu	phpOIDC	Drupal OpenID Connect Module	pyoidc	Ruby OpenIDConnect
Configuration-Discovery Response									
issuer claim	✓	✓	×	n/a	n/a	×	n/a	✓	×
Authentication-Response									
state Parameter	✓	✓	✓	n/a	n/a	×	✓	×	×
Token Response									
iss Claim	✓	✓	×	✓*	✓*	×	×	✓	✓
aud Claim	✓	✓	×	✓*	✓*	×	×	×	✓
nonce Claim	✓	✓	×	×	×	×	×	×	✓
iat Claim	✓	✓	×	✓*	×	×	×	×	×
exp Claim	✓	✓	×	✓*	×	×	×	×	✓
Signature	✓*	✓	✓	×	×	✓	×	×	✓
UserInfo Response									
sub Claim	✓	×	×	n/a	n/a	×	×	×	×

Table 4.7.: Relying Party Libraries - Validation

As seen in Table 4.6 most of the examined libraries support key features of the Core specification like authentication using the Authorization Code Flow (denoted with an "AC" in line Response Type(s)) or Client authentication using the HTTP Basic authentication scheme (denoted with a "B" in line Client Authentication). Additional features, like the support for the Implicit or Hybrid Flow or the implementation of the optional specifications Discovery and Dynamic Client Registration, are however not that common. Disturbingly almost all libraries allow non-HTTPS communication with OPs, thus exposing sensitive information, for example, `client_secret`, to potential man-in-the-middle attackers. Most of the analyzed test-implementations of the libraries do not provide any user management within their application and thus could not be classified according to their account separation capabilities. Table 4.7 shows that the `sub` claim of the UserInfo Response is, in most cases, left unvalidated, thus relying that the response is guaranteed to be about the End-User identified by the `sub` claim of the ID Token.

MITREid Connect Although MITREid Connect does offer a comparatively good overall-security concerning the distinct validations steps during the Authorization Code Flow, its signature verification is erroneous: If the contents received via the stored `jwks_uri` (which usually points to its JSON Web Key Set Endpoint) of an OP is no valid JWKS document the signature verification of a received ID Token is skipped. In addition to this, a received UserInfo Response is only validated for its `sub` claim if it is indeed present within the response; if not the response is still accepted and not rejected as the Core specification demands.

Nimbus OAuth 2.0 SDK with OpenID Connect extensions The test-implementation OpenID Connect Dev Client (accessible via <https://bitbucket.org/connect2id/openid-connect-dev-client/>) as well as the commercial product Connect2id server (accessible via <http://connect2id.com/products/server/download>) are both using the library Nimbus OAuth 2.0 SDK with OpenID Connect extensions. Both implement almost none of the validation steps described in Section 3.8, enabling an attacker to use the ID Spoofing attack to impersonate any user of the application. Connect2id however describes it as only "Designed for testing and development purposes".

Google OAuth Client Library for Java Google OAuth Client Library for Java is only a framework for building OpenID Connect services as it does not provide a test-implementation of its code. It however does provide several built-in procedures for initiating the Authorization Code Flow. It furthermore does provide a method for validating a received ID Token. Within the method, only the `iss` claim, `exp` claim, `iat` claim and the `aud` claim are validated. Other validation steps (like signature verification) are not provided. Above this, the mentioned method can be initialized without passing any reference-values of `iss` and `aud` to it without throwing an exception, thus circumventing two critical validation steps.

Apache Oltu Apache Oltu is only a framework for building OAuth 2.0 (and additional OpenID Connect) services as it does not provide a test-implementation of its code. In version 0.31 of the library the only support for OpenID Connect was provided by a built-in method for validating a received ID Token. Within the method, only the `iss` claim, `exp` claim and the `aud` claim were validated. Other validation steps (like signature verification) were not provided. Above this, the implemented `exp` claim validation was incorrect as it required the `exp` claim value to be a time before the current and not after. Within the current version, 1.0.0, of the library

4. Security Analysis

the mentioned method and all its corresponding classes and packages and thus all support for building OpenID Connect services is however gone.

phpOIDC The test-implementation of phpOIDC does provide almost none of the validation steps described in Section 3.8, enabling an attacker to use the ID Spoofing attack to impersonate any user of the application. The developers however explicitly state that "less focus was given to the security issues around the implementation".

Drupal OpenID Connect Module The OpenID Connect module for Drupal is the only tested implementation which provides the capability to separate End-Users authenticated with the help of OpenID Connect from otherwise authenticated users. Despite of not implementing one validation step as described within the Core specification, the module is not vulnerable to any of our defined attack-scenarios. This is mostly achieved by restricting the authentication to the Authorization Code Flow only and by not supporting arbitrary OPs for authentication of its users.

pyoidc The analyzed test-implementation (accessible via https://github.com/rohe/pyoidc/tree/master/oidc_example/rp3) of the pyoidc library does barely provide that much validation steps as described in Section 3.8, to prevent ID Spoofing and Issuer Confusion attacks. Unfortunately its signature verification is erroneous: If a signature of a given ID Token is manipulated, the ID Token is accepted anyway, enabling an attacker to use the Signature Manipulation attack to impersonate any user of the application.

4. Security Analysis

	Thinkecture IdentityServer v3	Nimbus OAuth 2.0 SDK with OpenID Connect extensions	MITREid Connect	oxAuth	Apache Oltu	phpOIDC	oauth2-server-php	pyoidc	Ruby OpenIDConnect
Authentication Flow									
Response Type(s)	AC, I	AC, I, H*	AC	AC, I, H*	n/a	AC, I, H	AC, I	AC, I, H	AC, I, H
Additional-Specifications									
Discovery	✓*	✓*	✓	✓	×	✓	×	✓	✓
Dynamic Client Registration	×	✓	✓	✓	×	✓	×	✓	✓
TLS Security									
Non-HTTPS #2	✓	✓	✓	×	n/a	✓	✓	✓	✓
Client Authentication									
Client Authentication	B,P	B,P, JWT	B,P, JWT	B,P, JWT	n/a	B,P, JWT	B,P	B,P, JWT	B,P
Authentication-Request Parameters									
claims_parameter_-supported	×	✓	×	×	×	✓	×	✓	×
request_parameter_-supported	×	✓	✓	✓	×	✓	×	✓	✓
ID Token									
Validity	1h	n/a	10min	1h	n/a	30min	1h	24h	6h

Table 4.8.: OpenID Provider Libraries - General Information

Legend: AC = code; I = id_token, id_token token; H = code id_token, code token, code id_token token; H* = code id_token, code id_token token; B = client_secret_basic; P = client_secret_post; JWT = client_secret_jwt and/or private_key_jwt

4. Security Analysis

	Thinkecture IdentityServer v3	Nimbus OAuth 2.0 SDK with OpenID Connect extensions	MITREid Connect	oxAuth	Apache Oltu	phpOIDC	oauth2-server-php	pyoidc	Ruby OpenIDConnect
Authentication-Request									
redirect_uri Parameter	✓	×*	✓	✓	×*	✓	✓	✓	✓
sub Claim	n/a	×*	✓	✓	n/a	✓	n/a	✓	✓
Token Request									
redirect_uri Parameter	✓	×*	✓	×	×*	×	✓	✓	✓

Table 4.9.: OpenID Provider Libraries - Validation

As seen in Table 4.8 most of the examined libraries implement most of the described Authentication Flows as well as the additional specifications Discovery and Dynamic Client Registration. Most libraries also offer a wide range of Client authentication mechanisms. Mechanisms to handle requests for additional claims using the `claims` or `request` parameter are however rarely implemented. The validity of the issued ID Tokens also varies in a wide range between leastwise 10 minutes and 24 hours at the maximum. Disturbingly, like in the case of the RP libraries, almost all libraries allow non-HTTPS communication with RPs by accepting non-HTTPS Redirect URIs during registration. Table 4.9 however shows that, among other factors, due to recent publications, like [2], the security-awareness of developers implementing OAuth-like protocols on the Authorization Server side has evolved: Almost all of the examined libraries verify that the, with the Authentication Request received, Redirect URI matches at least one of the Clients registered.

Thinkecture IdentityServer v3 Thinkecture IdentityServer v3 is a .NET-based open source implementation of an OP and OAuth 2.0 Authorization Server. It supports authentication using the Authorization Code- and the Implicit Flow. Additional specifications are only partly supported: While the implementation provides a Configuration Information Endpoint for retrieving the OP's configuration information, it lacks the possibility to determine the location of the OP via an Issuer Discovery Endpoint. Support for Dynamic Client Registration is also not provided.

4. Security Analysis

With an ID Token validity of one hour and adequate implemented `redirect_uri` validation mechanisms, Thinktecture IdentityServer v3 does however offer a comparatively good overall-security.

Nimbus OAuth 2.0 SDK with OpenID Connect extensions Although it does not provide a test-implementation of its code, Nimbus OAuth 2.0 SDK with OpenID Connect extensions is an almost fully-featured framework for building OpenID Connect services: The implementation does provide ready-to-use Java classes and methods for all three Authentication Flows of the Core specification, multiple Client authentication mechanisms as well as support for Configuration Discovery and Dynamic Client Registration. Only Issuer Discovery is not supported. As the focus of the implementation is obviously on interoperability and not security, it lacks however of built-in validation mechanisms and secure default values of specific parameters (e.g. ID Token validity).

oxAuth oxAuth is an OP implementation by Gluu as a component of their open source "OX Server Stack" as well as their on demand "Gluu Server Stack" [51]. It supports almost all OpenID Connect Response Types (defined in the Core specification) and additional support for Discovery and Dynamic Client Registration. Although the implementation's default values of the OP's configuration information stated the support for handling requests for additional claims using the `claims` parameter (`claims_parameter_supported`), a closer look at the code revealed that it only supports the `request` parameter (`request_parameter_supported`). Nonetheless sufficient mechanisms for preventing Sub Claim Spoofing attacks are in place. Surprisingly, while on the one hand validating the received `redirect_uri` parameter of a given Authentication Request and thus raising the security of the implementation, on the other previous security-considerations are disregarded by leaving the `redirect_uri` parameter of a given Token Request unvalidated.

Apache Oltu Apache Oltu is only a framework for building OAuth 2.0 (and additional OpenID Connect) services as it does not provide a test-implementation of its code. In version 0.31 of the library the only support for OpenID Connect was provided by a built-in method for validating a received ID Token. Classes of the OAuth 2.0 implementation, for example, for handling the Authorization Code Grant Type, could however be extended to build up a working OP implementation. Within the current version, 1.0.0, of the library the mentioned method and all its corresponding classes and packages and thus all support for building OpenID Connect services is however gone.

4. Security Analysis

pyoidc The analyzed test-implementation (accessible via https://github.com/rohe/pyoidc/tree/master/oidc_example/op2) of the pyoidc library does support all OpenID Connect Response Types, Discovery and Dynamic Client Registration, multiple Client authentication mechanisms as well as handling requests for additional claims using the `claims` and `request` parameter. It does also provide sufficient `sub` claim and `redirect_uri` validation mechanisms. Unfortunately the implementation ships with a downright ridiculously high default value for the validity of its issued ID Tokens.

	Google+ Sign-In	Log In with PayPal	Salesforce Identity
Authentication Flow			
Response Type(s)	AC, I, H	AC, I*, H*	AC, I**
Additional-Specifications			
Discovery	✓*	×	✓*
Dynamic Client Registration	×	×	×
TLS Security			
Non-HTTPS #2	✓	✓	×
Client Authentication			
Client Authentication	P	B, P	P, JWT
Authentication-Request Parameters			
claims_parameter_supported	×	×	×
request_parameter_supported	×	×	×
ID Token			
Validity	1h 5min	8h	2min

Table 4.10.: OpenID Provider Live Implementations - General Information

Legend: AC = code; I = id_token, id_token token; H = code id_token, code token, code id_token token; I* = id_token; I** = id_token token; H* = code id_token; B = client_secret_basic; P = client_secret_post; JWT = client_secret_jwt and/or private_key_jwt; ✓* = only Configuration Discovery

4. Security Analysis

	Google+ Sign-In	Log In with PayPal	Salesforce Identity
Authentication-Request			
redirect_uri Parameter	✓	✓*	✓
sub Claim	n/a	n/a	n/a
Token Request			
redirect_uri Parameter	✓	✓	✓

Table 4.11.: OpenID Provider Live Implementations - Validation

Legend: ✓* = except for the path, query and fragment part of the URI

Log In with PayPal Log In with PayPal (formerly PayPal Access) is a commerce, OpenID Connect based identity solution. Acting as OpenID Provider, PayPal provides a selected set of Response Types for all three Authentication Flows of OpenID Connect. Optional specifications like Discovery or Dynamic Client Registration as well as handling requests for additional claims using the `claims` and `request` parameter are not supported. Clients have to be registered with the help of the PayPal Developer Console. The `redirect_uri` verification (see Step 2. of Section 3.8.1) of PayPal is implemented incorrectly as it only uses the scheme and authority part of a registered Redirect URI of a Client to match the sent `redirect_uri` of an Authentication Request. Possible path, query or fragment parts of the sent `redirect_uri` are left unvalidated. For instance, if a Client's registered Redirect URI is `https://goodRP.com/callback` and PayPal receives an Authentication Request with the Client's `client_id` and a `redirect_uri` parameter valuing, for example, `https://goodRP.com/anotherPath`, the End-User (after successful authentication to PayPal) is redirected to `https://goodRP.com/anotherPath` and not to the registered one. As the authority part is still validated and the Redirect URI thus cannot be changed to, for example, `https://badRP.com/callback`, one might be questioning the security-relevancy of this observation. Acting in the defined security model (see Section 4.1), it is truly irrelevant as RUM is (due to the authority part validation) not feasible. Extending the capabilities of our attacker to be able to conduct Cross-Site Scripting attacks as well, our observation might however make a variant of RUM applicable: If, for example, the query part of `https://goodRP.com/anotherPath` can be misused to inject malicious script-code (via Cross-Site Scripting) into the site, a precisely chosen payload can be used to

4. Security Analysis

send the authorization code (in case of the Authorization Code Flow) or the ID Token issued for the victim (in case of the Implicit Flow) to the attacker. In addition to the described faulty implementation of the `redirect_uri` verification part, some claims of the ID Tokens issued by PayPal are not compliant to the Core specification of OpenID Connect: Instead of the mandatory `sub` claim of the ID Token, another claim called `user_id` valuing the identifier of the End-User to be consumed by the Client is used; Instead of valuing a timestamp with the time at which the ID Token will expire, the `exp` claim of PayPal values a timespan specifying the validity of the issued ID Token. Both observations alongside a deprecated ID Token signature verification-logic are not directly security-relevant but could cause interoperability problems on the side of the Clients trying to communicate with PayPal in a OpenID Connect specification compliant way.

	MITREid Connect	mod_auth_openidc	Nimbus OAuth 2.0 SDK with OpenID Connect extensions	Google OAuth Client Library for Java	Apache Oltu	phpOIDC	Drupal OpenID Connect Module	pyoidc	Ruby OpenIDConnect
ID Spoofing	×	×	✓	n/a	n/a	✓	×	×	×
Issuer Confusion	×	×	×	n/a	n/a	×	×	×	×
SM	×	×	×	n/a	n/a	×	×	✓	×

Table 4.12.: Relying Party Libraries - Applicable Attack-Scenarios

4. Security Analysis

	Thinkecture IdentityServer v3	Nimbus OAuth 2.0 SDK with OpenID Connect extensions	MITREid Connect	oxAuth	Apache Oltu	phpOIDC	oauth2-server-php	pyoidc	Ruby OpenIDConnect
Sub Claim Spoofing	n/a	n/a	×	×	n/a	×	n/a	×	×
RUM #1	×	n/a	×	×	n/a	×	×	×	×
RUM #2	×	n/a	×	×	n/a	×	×	×	×

Table 4.13.: OpenID Provider Libraries - Applicable Attack-Scenarios

	Google+ Sign-In	Log In with PayPal	Salesforce Identity
Sub Claim Spoofing	n/a	n/a	n/a
RUM #1	×	×	×
RUM #2	×	×	×

Table 4.14.: OpenID Provider Live Implementations - Applicable Attack-Scenarios

5. Conclusion

5.1. Summary

In this thesis, we gave a comprehensive description of the SSO protocol OpenID Connect. We therefore provided insight into the general concept of (web-based) Single Sign-On as well as OpenID Connect's related protocols OAuth 2.0 and OpenID 2.0. By introducing the use cases as well as the involved parties of OpenID Connect we lay the foundations for describing the different endpoints of the protocol. With an understanding of the latter, we presented the protocol's Authentication Flows and its extensions Discovery and Dynamic Client Registration. Furthermore, to be able to present a thorough security analysis of OpenID Connect, we listed the (within the specification of the protocol) proposed security-relevant validation steps.

In the main part of this thesis, we focused on the security analysis of OpenID Connect. We therefore presented our used security model and gave reference to related work. Furthermore, we introduced five novel attacks / attack-scenarios on the protocol all resulting in unauthorized access of a victim's protected resources. All of the defined attacks target implementation flaws on either the RP or the OP side. Three of them target RP implementations and the other two target OP implementations. To ease the work when testing an OpenID Connect implementation for the applicability of such attacks, we additionally introduced two self-developed proof-of-concept Java pentest applications. For a selection of nine Relying Party libraries, nine OpenID Provider libraries as well as three OP live implementations of the protocol, we developed a security aspect catalog to summarize the implementations' feature and validation variety. We finally concluded with a summary of the applicability of our defined attacks on the selected implementations. With the help of our developed applications, we were able to exploit three out of nine RP libraries using two of our defined attacks. Although all tested OP implementations (libraries and live ones) proved to be not vulnerable to our introduced attacks, the implementations never proved to be flawless regarding all security-relevant validation steps, leaving space for security improvements. Additionally, most of the tested implementations (on the RP or the OP side) did not support all features of the OpenID Connect Core specification and its extensions Discovery and Dynamic Client Registration. Furthermore,

5. Conclusion

several libraries seemed to be proof-of-concept implementations mostly focusing on the operability and not the security of the protocol. Thus, additional security flaws may arise when integrating such libraries into 3rd party products.

Our evaluation results show that security-relevant validation steps, as defined in the specification of the protocol, are either not regarded as important enough or not well enough understood to be completely implemented in the tested libraries. As a whole section of the Core specification is dedicated to security considerations and variants of some of our defined attacks [1, 2, 41] are known to be applicable to related protocols like OAuth or OpenID, this observation gives reason for concern.

5.2. Further Studies

As the development of most of our tested libraries is still in progress and features as well as validation mechanisms are constantly added, in combination with the few found live implementations of OpenID Connect, the security analysis presented in this thesis cannot be regarded as final. On the contrary, we are thinking about starting a large scale study on multiple RP and OP live and library implementations to raise the awareness of developers regarding SSO security in general and OpenID Connect security in specific. While our presented pentest applications might help trained security auditors to test OpenID Connect implementations for certain vulnerabilities, it is not yet suited for not security-aware application developers. To enable such functionality, fully-automated vulnerability tests (comparable to the ones implemented in the OAuth 2.0 vulnerability checker "SSOScan" [41]) have to be integrated into the applications. Test-automation however implies coping with certain implementation difficulties like:

- Handling and interpretation of dynamic responses due to parameter changes.
- Handling of a great variety of ID Token permutations.
- Handling of generic and customized error messages.
- Simulating user interactions.

Some of the mentioned difficulties arise due to the developers' interpretation variety of the protocol. This variety may sometimes cause an unpredictable behavior and thus make coping with this difficulty unachievable. Nonetheless we will try to extend our proof-of-concept applications into a single fully-automated OpenID Connect pentest application capable of testing implementations for our defined attacks. By making the source code of our to-be-developed application publicly available, we like

5. Conclusion

to encourage developers and pentesters to improve the security of OpenID Connect based SSO systems.

Bibliography

- [1] V. Mladenov and C. Mainka, “Untrusted Third Parties: When IdPs Break Bad,” 2014, Unpublished. 1, 4.2, 5.1
- [2] C. Nickel, “Sicherheitsanalyse von OAuth 2.0 mittels Web Angriffen auf bestehende Implementierungen,” Master’s thesis, Ruhr-Universität Bochum, 2013. 1, 2.6, 4.2, 4.7, 5.1
- [3] *2013 Consumer Research: The Value of Social Login*. Blue Research, 2013. [Online]. Available: <http://janrain.com/resources/industry-research/2013-consumer-research-value-of-social-login/> 1
- [4] *OpenID Usage Statistics*. BuiltWith, 2014. [Online]. Available: <http://trends.builtwith.com/docinfo/OpenID> 1
- [5] D. Hardt, *The OAuth 2.0 Authorization Framework*, Internet Engineering Task Force (IETF) Std. RFC 6749, October 2012. [Online]. Available: <http://tools.ietf.org/html/rfc6749> 1, 2.4, 2.4.1, 2.4.2, 2.4.3, 2.5, 2.4.5, 2.4.5.1, 3.5.2, 3.5.2, 3.5.3
- [6] The OpenID Foundation (OIDF), *OpenID Authentication 2.0 - Final*, The OpenID Foundation (OIDF) Std., December 2007. [Online]. Available: http://openid.net/specs/openid-authentication-2_0.html 1, 2.3
- [7] *OpenID Connect FAQ and Q&As*. The OpenID Foundation (OIDF), 2014. [Online]. Available: <http://openid.net/connect/faq/> 1, 3.2
- [8] *What is OpenID Connect?* The OpenID Foundation (OIDF), 2014. [Online]. Available: <http://openid.net/connect/> 1, 3.1, 3.2, 3.6, 3.7
- [9] *OC5:OpenID Connect Interop 5*. OSIS Open Source Identity, 2013. [Online]. Available: http://osis.idcommons.net/wiki/OC5:OpenID_Connect_Interop_5 1, 4.3, 4.5
- [10] The OpenID Foundation (OIDF), *OpenID Connect Core 1.0*, The OpenID Foundation (OIDF) Std., February 2014. [Online]. Available: http://openid.net/specs/openid-connect-core-1_0.html 1, 3.1, 3.4, 3.5, 3.5.1, 3.5.2, 12, 3.5.3, 3.5.4, 16

Bibliography

- [11] —, *OpenID Connect Discovery 1.0*, The OpenID Foundation (OIDF) Std., February 2014. [Online]. Available: http://openid.net/specs/openid-connect-discovery-1_0.html 1, 3.1, 3.4, 3.4.7, 3.6, 3.6
- [12] —, *OpenID Connect Dynamic Client Registration 1.0*, The OpenID Foundation (OIDF) Std., February 2014. [Online]. Available: http://openid.net/specs/openid-connect-registration-1_0.html 1, 3.1, 3.4, 3.4.2, 3.4.6, 3.7
- [13] D. DiNucci, “Fragmented Future,” *Print*, vol. 53, no. 4, pp. 32, 221 – 222, April 1999. [Online]. Available: http://darcy.com/fragmented_future.pdf 2.1
- [14] *Kerberos Papers and Documentation*. MIT, 2004. [Online]. Available: <http://web.mit.edu/kerberos/papers.html> 2.2
- [15] R. M. Needham and M. D. Schroeder, “Using encryption for authentication in large networks of computers,” *Magazine Communications of the ACM*, vol. 21, no. 12, pp. 993 – 999, December 1978. 2.2
- [16] J. T. Kohl, B. C. Neuman, and T. Y. Ts’o, “The Evolution of the Kerberos Authentication Service,” *Distributed Open Systems*, pp. 78 – 94, 1994. [Online]. Available: http://www.isi.edu/div7/publication_files/rs-94-412.pdf 2.2, 2.1
- [17] M. Amon, “The Strong Locked Same Origin Policy in Firefox and its Application in Single Sign-On Schemes,” Master’s thesis, Ruhr-Universität Bochum, November 2010. 2.2
- [18] E. Hammer-Lahav, *The OAuth 1.0 Protocol*, Internet Engineering Task Force (IETF) Std. RFC 5849, April 2010. [Online]. Available: <http://tools.ietf.org/html/rfc5849> 2.4
- [19] —, “Introducing OAuth 2.0,” *hueniverse*, May 2010. [Online]. Available: <http://hueniverse.com/2010/05/15/introducing-oauth-2-0/> 2.4, 2.4.2
- [20] *OAuth Security Advisory: 2009.1*. The OAuth Community, April 2009. [Online]. Available: <http://oauth.net/advisories/2009-1/> 2.4
- [21] R. Boyd, *Getting Started with OAuth 2.0*. O’Reilly Media, February 2012. 2.4, 2.4.1
- [22] J. Richer, M. Jones, J. Bradley, M. Machulak, and P. Hunt, *OAuth 2.0 Dynamic Client Registration Protocol draft-ietf-oauth-dyn-reg-17*, OAuth Working Group Std., May 2014. [Online]. Available: <http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg-17> 2.4.3.2

- [23] M. Jones and D. Hardt, *The OAuth 2.0 Authorization Framework: Bearer Token Usage*, Internet Engineering Task Force (IETF) Std. RFC 6750, October 2012. [Online]. Available: <http://tools.ietf.org/html/rfc6750> 2.4.5.1, 3.4.5, 3.5.2, 3.5.2, 1
- [24] P. Jones, G. Salgueiro, M. Jones, and J. Smarr, *WebFinger*, Internet Engineering Task Force (IETF) Std. RFC 7033, September 2013. [Online]. Available: <http://tools.ietf.org/html/rfc7033> 3.1, 3.6, 3.6
- [25] *Inside OpenID Connect on Force.com*. Salesforce.com, inc., 2014. [Online]. Available: https://developer.salesforce.com/page/Inside_OpenID_Connect_on_Force.com 3.2, 3.1
- [26] M. Jones, *JSON Web Key (JWK) draft-ietf-jose-json-web-key-30*, JOSE Working Group Std., July 2014. [Online]. Available: <http://tools.ietf.org/html/draft-ietf-jose-json-web-key-30> 3.4.4, 4.3
- [27] M. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT) draft-ietf-oauth-json-web-token-25*, OAuth Working Group Std., July 2014. [Online]. Available: <http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-25> 3.5.1, 4.3
- [28] —, *JSON Web Signature (JWS) draft-ietf-jose-json-web-signature-30*, JOSE Working Group Std., July 2014. [Online]. Available: <http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-30> 3.5.1, 4.3
- [29] M. Jones and J. Hildebrand, *JSON Web Encryption (JWE) draft-ietf-jose-json-web-encryption-31*, JOSE Working Group Std., July 2014. [Online]. Available: <http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-31> 3.5.1, 4.3
- [30] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, Network Working Group Std. RFC 3986, January 2005. [Online]. Available: <http://www.ietf.org/rfc/rfc3986> 3.6
- [31] T. Duong and J. Rizzo, Eds., *Here Come The XOR Ninjas*, May 2011. [Online]. Available: http://www.infoworld.com/sites/infoworld.com/files/pdfe/BEAST_Duong_Rizzo.pdf 4.1.2
- [32] A. Barth, C. Jackson, and J. C. Mitchell, “Securing frame communication in browsers,” *Communications of the ACM - One Laptop Per Child: Vision vs. Reality*, vol. 52, pp. 83–91, June 2009. [Online]. Available: <http://seclab.stanford.edu/websec/frames/post-message.pdf> 4.1.3

- [33] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, “Towards a Formal Foundation of Web Security,” *CSF '10 Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, pp. 290–304, 2010. 4.1.4
- [34] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, “Formal Verification of OAuth 2.0 Using Alloy Framework,” *Communication Systems and Network Technologies (CSNT), 2011 International Conference*, pp. 655–659, June 2011. [Online]. Available: http://www.researchgate.net/profile/Sanjay_Singh7/publication/214826971_Formal_Verification_of_OAuth_2.0_using_Alloy_Framework/file/32bfe50d18aab86c40.pdf 4.2
- [35] E. Torlak, M. van Dijk, B. Gassend, D. Jackson, and S. Devadas, Eds., *Knowledge Flow Analysis for Security Protocols*, August 2005. [Online]. Available: <http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-1007.pdf> 4.2
- [36] D. Jackson, Ed., *alloy: a language & tool for relational models*, 2012. [Online]. Available: <http://alloy.mit.edu/alloy/> 4.2
- [37] R. Paul, “Compromising Twitter’s OAuth security system,” *Ars Technica*, September 2010. [Online]. Available: <http://arstechnica.com/security/2010/09/twitter-a-case-study-on-how-to-do-oauth-wrong/> 4.2
- [38] S. Chari, C. Jutla, and A. Roy, “Universally Composable Security Analysis of OAuth v2.0,” *Cryptology ePrint Archive, Report 2011/526*, September 2011. [Online]. Available: <http://eprint.iacr.org/2011/526.pdf> 4.2
- [39] R. Canetti, “Universally Composable Security: A New Paradigm for Cryptographic Protocols,” *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium*, pp. 136–145, October 2001. [Online]. Available: <http://eprint.iacr.org/2000/067.pdf> 4.2
- [40] S.-T. Sun and K. Beznosov, “The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems,” *CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 378–390, 2012. 4.2
- [41] Y. Zhou and D. Evans, “SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities,” *23rd USENIX Security Symposium*, 2014. [Online]. Available: <http://www.ssoscan.org/SSOScan.pdf> 4.2, 5.1, 5.2
- [42] *The FAT Attack. Facebook Social Login Session Hijacking*. MetaIntell, July 2014. [Online]. Available: <http://metaintell.com/blog/2014/06/24/the-fat-attack-facebook-social-login-session-hijacking/> 4.2

- [43] S.-T. Sun, K. Hawkey, and K. Beznosov, “Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures,” *Computers & Security*, vol. 31, no. 4, pp. 465–483, June 2012. 4.2
- [44] *MITREid Connect*. MITRE Corporation, MIT Kerberos, Internet Trust (KIT), 2014. 4.3
- [45] P. Hawke, “NanoHttpd,” 2014. [Online]. Available: <https://github.com/NanoHttpd/nanohttpd> 4.3
- [46] B. Campbell, “jose.4.j,” 2014. [Online]. Available: https://bitbucket.org/b_c/jose4j/wiki/Home 4.3
- [47] Y. Fang, “JSON.simple,” 2014. [Online]. Available: <https://code.google.com/p/json-simple/> 4.3
- [48] *Apache HttpClient*. The Apache Software Foundation, 2014. [Online]. Available: <http://hc.apache.org/httpcomponents-client-ga/> 4.3
- [49] *Apache HttpComponents*. The Apache Software Foundation, 2014. 4.3
- [50] *Libraries, Products, and Tools*. The OpenID Foundation (OIDF), 2014. [Online]. Available: <http://openid.net/developers/libraries/> 4.5
- [51] *Open Source vs On Demand / The Gluu Server for SSO, 2FA & WAM*. Gluu, Inc., 2014. [Online]. Available: <http://www.gluu.org/open-source/open-source-vs-on-demand/> 4.7

A. Appendix

The attached DVD contains the following:

- The OpenID Connect Provider TestSuite application compiled as
OpenIDConnectProviderTestSuite-1.0-SNAPSHOT-jar-with-dependencies.jar
- The source code of the OpenID Connect Provider TestSuite application
- The OpenID Connect Client TestSuite application compiled as
OpenIDConnectClientTestSuite-1.0-SNAPSHOT-jar-with-dependencies.jar
- The source code of the OpenID Connect Client TestSuite application
- This thesis as PDF file