OpenID Connect
Security Considerations

Vladislav Mladenov Christian Mainka

January 13, 2017

Contents

Chapter 1

Attacker Model

This chapter is intended to outline the general concept of our security analysis of the Single Sign-On protocol OpenID Connect, as well as reveal several custom designed attack-scenarios.

1.1 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.

1.1.1 Objectives of the Attacker

The essential goal of the attacker is to impersonate her victim when accessing a service of a Relying Party (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.

1.1.2 Assumptions

As a huge proportion of the security of OpenID Connect is based on the utilization of Transport Layer Security (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, 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, user agent, and furthermore her Operating System is assumed to be not compromised. Thus the attacker has no way of exploiting vulnerabilities within the user agent (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 OpenID Provider (OP) or vice versa.

1.1.3 Capabilities of the Attacker

The attacks to-be-introduced in this technical report have been strictly verified in the *web attacker* model [2]. 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) Uniform Resource Identifier (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.

1.1.4 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). 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 exposure of sensitive information of the latter.

Chapter 2

Specification Flaws

2.1 Malicious Endpoints Attacks

This section describes four different attacks, which belong to the class of Malicious Endpoints attacks. All attacks use the *malicious Discovery service* and influence the OpenID Connect flow. Since each attack pursues different goals, we describe for each attack the main goal, the setup including the attacker model and the attack itself. Please note that all following attacks are based on the fact that an attacker can enforce the Client to use a malicious endpoint. A developer or customer should thus be aware of these security issues, although the concrete attack impact may depend on the service deployment (see Sections 2.1.2, 2.1.3, 2.1.4 for details). The impact of the attack presented in Section 2.1.1 is independent of its deployment.

2.1.1 Broken End-User Authentication

This attack has been reported by to the authors of OpenID Connect in October 2014, and was adopted to OAuth in 2015 by Fett et. al (known as *Mixup* Attack). The idea behind the attack is to influence the information flow in the Discovery and Dynamic Registration Phase in such a way that the attacker gains access to sensitive information[1]. The attacker pursues the theft of sensitive information like the Authorization Code (`code`), Access Token (`token`), ID Token and/or Client credentials used for authentication.

Setup. The basic setup for the attack is as follows:

- ▶ The End-User (*victim*) has an active account on the genuine *honest Client*. We assume that the End-User trusts this Client and the Client follows the OpenID Connect protocol rules.
- ▶ The End-User is registered at the *Honest OP* on the domain *`https://honestOP.com`*. The End-User trusts this OP and the OP also follows the OpenID Connect protocol rules.
- ▶ To perform the attack, the attacker has to set up his own Discovery service running on the domain *`http://malicious.com`*. This Discovery Service acts *maliciously*, which means, that it deviates from the *normal* OpenID Connect protocol flow. Note that there is no need to disguise *`http://malicious.com`* as the regular Discovery service belonging to the Honest OP in any way.

[1]Please note that the attack is also possible if the Client does not support Discovery and Dynamic registration, but allows to register a custom OP manually.

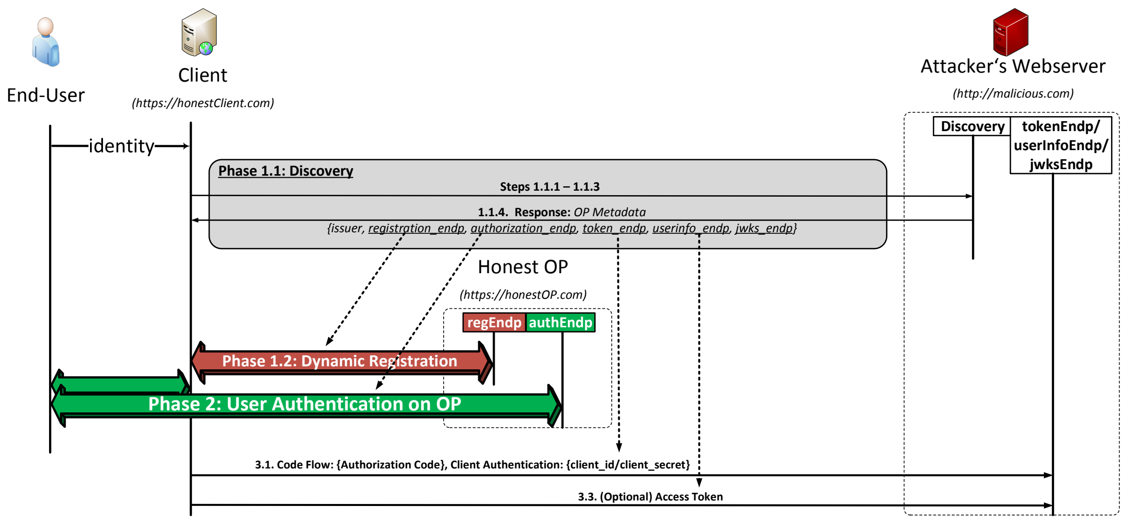

End-User
Client
(https://honestClient.com)
Attacker's Webserver
(http://malicious.com)
Discovery
tokenEndp/ userInfoEndp/ jwksEndp
identity
Phase 1.1: Discovery
Steps 1.1.1 – 1.1.3
1.1.4. Response: OP Metadata
{issuer, registration_endp, authorization_endp, token_endp, userinfo_endp, jwks_endp}
Honest OP
(https://honestOP.com)
regEndp
authEndp
Phase 1.2: Dynamic Registration
Phase 2: User Authentication on OP
3.1. Code Flow: {Authorization Code}, Client Authentication: {client_id/client_secret}
3.3. (Optional) Access Token

Figure 2.1: Malicious Endpoints attack: Attacker's Discovery service sets the endpoint variables in a specific way, such that the secret tokens sent in the third phase are seamlessly distributed to the attacker's server.

- ▶ According to the attacker model, the attacker does not have any control over the honest Client, the End-User, the Honest OP or the network traffic between these instances. The attacker is able to send an HTTP request through End-User's browser, for example, by embedding an image in a benign HTML website that causes the browser to automatically issue a request when the website is viewed.

Attack description. In the following, we describe the attack protocol flow, which we depicted in Figure 2.1.

Phase 1.1 - Injecting malicious endpoints The attacker's intention in the first phase is to force a valid Client to use the attacker's malicious Discovery service. For this purpose, he constructs a malicious link and stores it on a benign website, e.g. in a web forum. For example, this can be a link to the valid Client containing an identity *alice@malicious.com*.

By visiting the website containing the malicious link, an HTTP request will be sent to the Client through the End-User's (victim's) browser. Consequentially, the Client starts a Discovery Phase with the malicious Discovery service `http://malicious.com`. The Client sends a request to determine the corresponding endpoints. The attacker's Discovery service responds with the following values, initiating the actual attack:

```
issuer:    http://malicious.com
regEndp:   https://honestOP.com/register
authEndp:  https://login.honestOP.com/
tokenEndp:  http://malicious.com
userInfoEndp: http://malicious.com
```

Listing 2.1: Endpoints returned by the malicious Discovery service

Phase 1.2 – Dynamic Registration In the next step, the Client accesses *regEndp* for the Dynamic Registration. It sends a registration request to `https://honestOP.com/register` and receives a *client_id* and *client_secret* in the response.

Note: The Client automatically starts the Dynamic Registration, even if it is already registered on the Honest OP. The reason for this behavior is that the Client believes that `http:`

`//malicious.com` is the responsible OP, since it is not known from previous authentication procedures. Thus, `http://malicious.com` is a new OP for the Client and it starts the registration procedure.

Phase 2 – End-User Authentication and Authorization In the next phase, the Client redirects the End-User to *authEndp*, `https://login.honestOP.com/`, where the End-User has to authenticate himself and authorize the Client. The End-User is not able to detect any abnormalities in the protocol flow: Phase 1.1 and Phase 1.2 cannot be observed by the End-User, and in Phase 2 the End-User will be prompted to authenticate to the Honest OP and authorize the honest Client, both of which he knows and trusts. Thus, the End-User authorizes the Client and the OP generates the `code`, which is sent to the Client.

Note: Phase 2 exactly follows the original OpenID Connect protocol flow – there are no parameter manipulations, no redirects to malicious websites and no observation of the network traffic between the End-User, the Honest OP and the Client. Thus, the attack started at the beginning of the protocol flow can be neither detected nor prevented by any of the participants at this point.

Phase 3 – The Theft In dependence of the protocol flow, Code or Implicit, the messages sent to the attacker differ.

Within the *Code flow* the Client redeems the received `code` from the previous phase: It sends the `code` together with the corresponding Client's credentials received during the Dynamic Registration (*client_id/ client_secret*) to the *tokenEndp* originally specified by the malicious Discovery service – in this example `http://malicious.com`, see Listing 2.1.

Since the *Implicit flow* does not use the *tokenEndp*, the attacker is not able to receive the information send in Phase 2. However, he can use another malicious endpoint – *userInfoEndp* used in Step 3.3 in Figure 2.1 to retrieve further information about the authenticated user. In the request, the Client sends a freshly generated Access Token. As a result, the attacker receives this Access Token and is able to access the authorized resources on the OP.

2.1.2 Server Side Request Forgery (SSRF)

A Server Side Request Forgery (SSRF) attack describes the ability of an attacker to create requests from a vulnerable web application to the application's Intranet and the Internet. Usually, SSRF is used to attack internal services placed behind a firewall and not accessible from Internet. According to *Acunetix*, SSRF is a severity level 3 vulnerability (classification: *high*).[2].

In context of OpenID Connect, the malicious Discovery service can be used to start such attacks in order to (1.) gather information about the Intranet infrastructure of the Client, and (2.) disseminate attack vectors.

Setup. The attacker sets up a malicious Discovery service returning endpoints called by the Client during the protocol flow. The endpoints are URLs specifying protocol (http(s), ftp, smb etc.), port, path, and parameters. Since there are no restrictions regarding the URLs, these can point to the Intranet infrastructure of the Client. The Client will use these URLs and performs HTTP GET requests on them. In this manner, the Client can, for example, be enforced to invoke internal REST-based web services. This capability of the attacker is considered by the attacker model, since the attacker is able to use his UA and send arbitrary HTTP requests to every publicly available domain. Thus, he can cause the Client to establish connection with the malicious Discovery service.

Attack description. In comparison to the Malicious Endpoint attack, now the attacker initiates the OpenID Connect authentication on the Client by entering his identity (e.g. *os-*

[2]`https://acunetix.com/vulnerabilities/web/server-side-request-forgery`

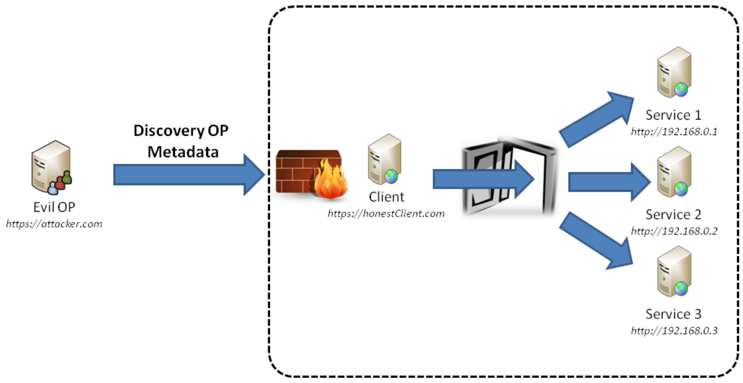


Figure 2.2: Abstract overview of Server-Side Request Forgery

kar@malicious.com). Thus, no CSRF attack is needed. In the end of the Discovery phase, the malicious Discovery service returns the malicious endpoints called during the different phases of the protocol. Previous researches reveal how the execution of URLs can be used to (1.) connect and execute commands on different services like Memcached, (2.) Port scanning and (3.) data retrieving [1, 3].

2.1.3 Code Injection Attacks

User's input sent through the web interface of the Client is usually treated as untrusted and thus filtered to prevent attacks like Cross-Site-Scripting (XSS) and SQL-Injection. In order to bypass the existing filter an attacker can use other channels to inject the attacks vectors – for instance within the server-to-server communication in Phase 3.

Setup. The attacker configures his server to inject malicious content in the messages returned in Step 3.2 (e.g. in the ID Token) or in Step 3.4 (informations about the authenticated user), which are sent to Client within the ID Token and Access Token. Please note that the ID Token and Access Token returned by the malicious server are valid according to the specification, since there are no restrictions regarding the values of parameters like "sub", "name" or "preferred_username".

Attack description. Initially, the attacker starts the OpenID Connect authentication on the Client by entering his identity (e.g. *oskar@malicious.com*). He proceeds with the protocol execution until Steps 3.1 and 3.3. The malicious server then responds with valid tokens (ID Token and Access Token) containing the attack vectors. A toy example of such attack vector is shown in Listing 2.2 where an XSS attack vector is injected into the field presenting the name of an authenticated user in Step 3.4.

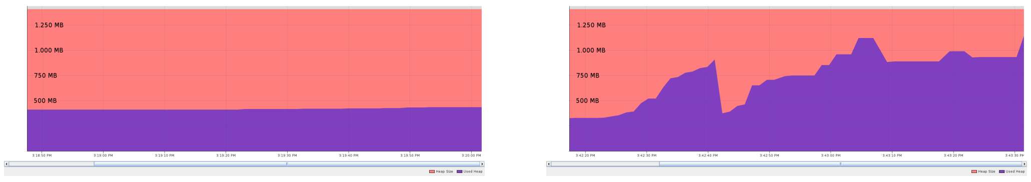
```
{
  "sub":"90342.ASDFJWFA",
  "name":"<script>alert(1)</script>",
  "preferred_username":"admin",
  "email":"bob@malicious.com",
  "email_verified":true
}
```

Listing 2.2: An example of an XSS attack vector hidden in the "name" filed within an Access Token.

Now, the placed XSS attack vector is stored in the web application (*persistent XSS*). Other webpages on the client will use it, for example on a guestbook page, and embed the code, so that other page visitors get harmed. The same schema can be used to place SQL-Injection attack vectors.

2.1.4 Denial-of-Service (DoS) Attacks

By applying Denial-of-Service (DoS) attacks the attacker allocates resources on a Client and negatively affects its workflow. Such resources are CPU usage, network traffic or memory. The attack can target one or multiple of these resources during the execution of DoS attack.



(a) Memory usage on the Client within 5 parallel OpenID Connect authentication flows to an Honest OP.

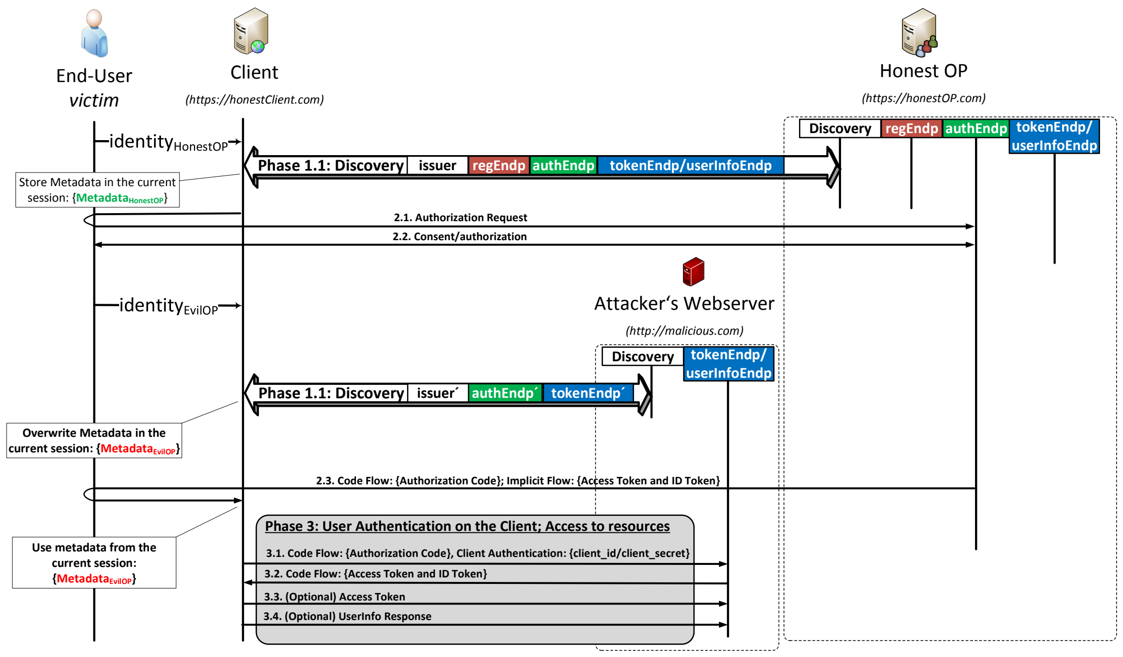
(b) Memory usage on the Client within 5 parallel OpenID Connect authentication flows to a malicious Discovery service pointing to a large file (in this case, we used a Debian Linux image file with 3.7GB).

Figure 2.3: Direct comparison between the memory usage on the Client using (a) an Honest OP and a (b) malicious Discovery service.

Setup. The setup is similar to the SSRF attack – the attacker sets up a malicious Discovery service returning endpoints called by the Client during the protocol flow. The attacker is able to use his UA and send HTTP request to the Client causing the Client to establish connection with the malicious Discovery service.

Attack description. An attack can be started by using a malicious endpoint pointing to a large data file, which will be downloaded. The Client calls later on the malicious endpoint URL, allocates network resources as well as large amount of the memory, which will be unnecessarily used.

We provide a measurement shown in Figure 2.3 on an Apache Tomcat server with 1280 MB memory and 4x2.4 Ghz CPU. In Figure 2.3a we first measured the memory usage on the Client within five parallel OpenID Connect protocol runs with an Honest OP. Once can say that almost imperceptible changes in the memory consumption occur. In Figure 2.3b we repeated the same tests, but this time we used our malicious Discovery service pointing to a large file. After few seconds, the memory usage increased almost threefold. After 60 seconds, the Client was not accessible for any incoming requests.



End-User
victim
Client
(https://honestClient.com)
Honest OP
(https://honestOP.com)
Discovery
regEndp
authEndp
tokenEndp/
userInfoEndp
identity$_{HonestOP}$
Phase 1.1: Discovery
issuer
regEndp
authEndp
tokenEndp/userInfoEndp
Store Metadata in the current session: {Metadata$_{HonestOP}$}
2.1. Authorization Request
2.2. Consent/authorization
Attacker's Webserver
(http://malicious.com)
Discovery
tokenEndp/
userInfoEndp
identity$_{EvilOP}$
Phase 1.1: Discovery
issuer´
authEndp´
tokenEndp´
Overwrite Metadata in the current session: {Metadata$_{EvilOP}$}
2.3. Code Flow: {Authorization Code}; Implicit Flow: {Access Token and ID Token}
Phase 3: User Authentication on the Client; Access to resources
3.1. Code Flow: {Authorization Code}, Client Authentication: {client_id/client_secret}
3.2. Code Flow: {Access Token and ID Token}
3.3. (Optional) Access Token
3.4. (Optional) UserInfo Response
Use metadata from the current session: {Metadata$_{EvilOP}$}

Figure 2.4: Malicious Endpoints attack: Attacker's Discovery service sets the endpoint variables in a specific way, such that the secret tokens sent in the third phase are seamlessly distributed to the attacker's server.

2.2 Session Overwriting

Almost all web applications need to save some objects between different requests. In this manner the web application preserves certain data across subsequent accesses. A End-User accessing the web application is assigned a unique session ID, which is usually stored in a cookie.

This section describes a novel attack, which leads to *Broken End-User Authentication*. The idea of the attack is to overwrite objects stored in the session leading to security issues.

Setup. The basic setup for the attack is as follows:

- The End-User (*victim*) has an active account on the genuine *honest Client*. We assume that the End-User trusts this Client and the Client follows the OpenID Connect protocol rules.

- The End-User is registered at the *Honest OP* on the domain *`https://honestOP.com`*. The End-User trusts this OP and the OP also follows the OpenID Connect protocol rules.

- To perform the attack, the attacker has to set up his own Discovery service running on the domain *`http://malicious.com`*. This Discovery Service acts does not deviate from the normal OpenID Connect protocol flow.

- According to the attacker model, the attacker does not have any control over the honest Client, the End-User, the Honest OP or the network traffic between these instances. The attacker is able to send an HTTP request through End-User's browser, for example, by embedding an image in a benign HTML website that causes the browser to automatically issue a request when the website is viewed.

Attack description. In the following, we describe the attack protocol flow, which we depicted in Figure 2.4.

Storing $Metadata_{Honest}$ *in the Session* The attacker's intention in the first phase is to force a valid Client to use the Honest OP. For this purpose, he constructs a malicious link and stores it on a benign website, e.g. in a web forum. For example, this can be a link to the valid Client containing an identity *alice@honestOP.com*.

By visiting the website containing the malicious link, an HTTP request will be sent to the Client through the End-User's (victim's) browser. Consequentially, the Client starts a Discovery Phase with the malicious Discovery service `http://honestOP.com`. The Client sends a request, determines the corresponding endpoints and stores them in the current session – $Metadata_{Honest}$.

End-User Authentication and Authorization In the next phase, the Client redirects the End-User to *authEndp*, `https://login.honestOP.com/`, where the End-User has to authenticate himself and authorize the Client. Phase 2 exactly follows the original OpenID Connect protocol flow – there are no parameter manipulations, no redirects to malicious websites and no observation of the network traffic between the End-User, the Honest OP and the Client.

Overwrite the Metadata stored in the Session The attacker's intention is to force the Client to use the attacker's malicious Discovery service. For this purpose, he initiates a second HTTP request to the Client containing now the identity *alice@malicious.com. Note:* The attacker enforces the browser of the End-User to send two HTTP requests. This can be done by loading two HTML IFrames time-shifted.

As result, the Client discovers the malicious Discovery service, determines the corresponding endpoints and overwrites the old metadata with the new one – $Metadata_{EvilOP}$.

Broken End-User Authentication In dependence of the protocol flow, Code or Implicit, the messages sent to the attacker differ.

Within the *Code flow* the Client redeems the received `code` from the previous phase: It sends the `code` together with the corresponding Client's credentials (*client_id/ client_secret*) to the *tokenEndp* stored in the metadata within the session – in this example `http://malicious.com`.

Since the *Implicit flow* does not use the *tokenEndp*, the attacker is not able to receive the information send in Phase 2. However, he can use another malicious endpoint – *userInfoEndp* used in Step 3.3 in Figure 2.1 to retrieve further information about the authenticated user. In the request, the Client sends a freshly generated Access Token. As a result, the attacker receives this Access Token and is able to access the authorized resources on the OP.

2.3 IdP Confusion

Setup. We assume that the Service Provider (SP) allows the usage of custom OPs. Additionally, we assume that during the registration the SP receives the same *client_id* from the Attacker OP as on the Honest OP. In other words, the SP has the same *client_id* on two different OPs.

Attack description.

`Attack preparation`

- ▶ In step 0.1.1 the SP registers at the Honest OP, and
- ▶ in step 0.1.2 it receives an unique *client_id* and *client_password* combination.
- ▶ In step 0.1.3 the SP registers at the Attacker OP, and
- ▶ in step 0.1.4 it receives the same *client_id* and new *client_password'* combination.

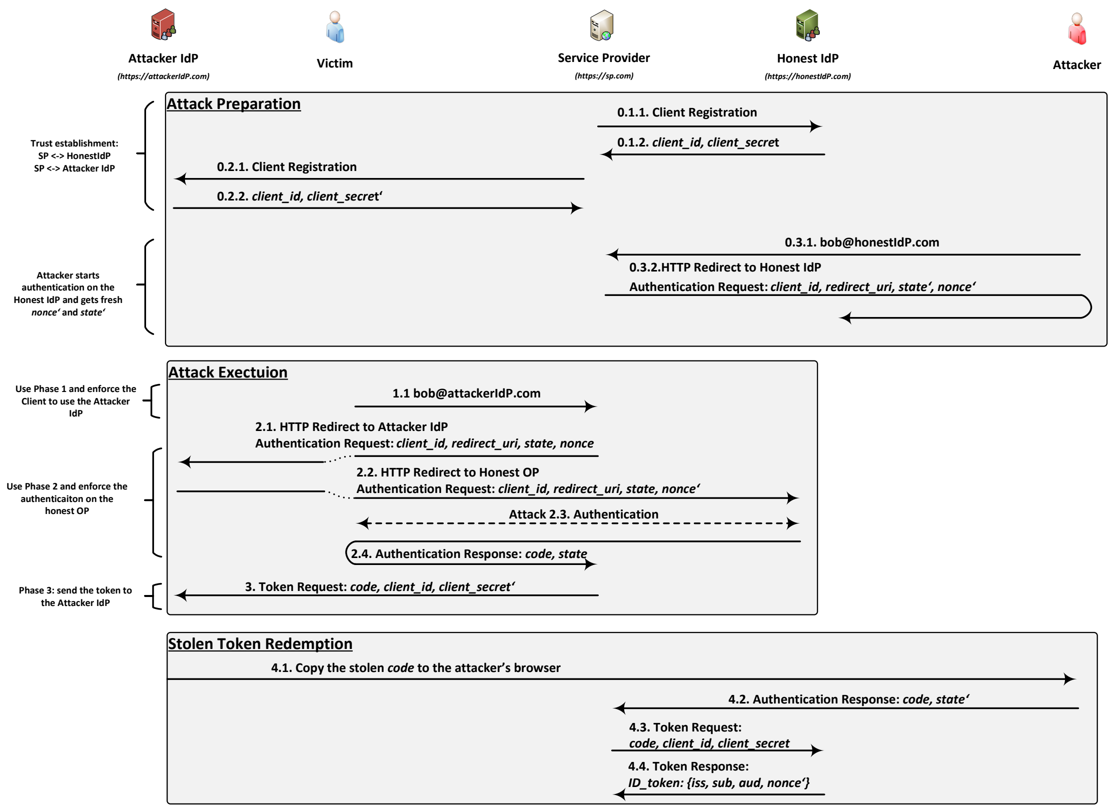

Attacker IdP
(https://attackerIdP.com)
Victim
Service Provider
(https://sp.com)
Honest IdP
(https://honestIdP.com)
Attacker
Attack Preparation
Trust establishment: SP <-> HonestIdP SP <-> Attacker IdP
0.1.1. Client Registration
0.1.2. client_id, client_secret
0.2.1. Client Registration
0.2.2. client_id, client_secret'
Attacker starts authentication on the Honest IdP and gets fresh nonce' and state'
0.3.1. bob@honestIdP.com
0.3.2.HTTP Redirect to Honest IdP
Authentication Request: client_id, redirect_uri, state', nonce'
Attack Exectuion
Use Phase 1 and enforce the Client to use the Attacker IdP
1.1 bob@attackerIdP.com
2.1. HTTP Redirect to Attacker IdP
Authentication Request: client_id, redirect_uri, state, nonce
Use Phase 2 and enforce the authenticaiton on the honest OP
2.2. HTTP Redirect to Honest OP
Authentication Request: client_id, redirect_uri, state, nonce'
Attack 2.3. Authentication
2.4. Authentication Response: code, state
Phase 3: send the token to the Attacker IdP
3. Token Request: code, client_id, client_secret'
Stolen Token Redemption
4.1. Copy the stolen code to the attacker's browser
4.2. Authentication Response: code, state'
4.3. Token Request:
code, client_id, client_secret
4.4. Token Response:
ID_token: {iss, sub, aud, nonce'}

Figure 2.5: OP Confusion abuses a logical flaw in the current OpenID Connect specification resulting into token theft and consequentially *broken authentication*. The attack consists of three parts: (1) the attack preparation, (2) the attack execution, and (3) the stolen token redemption.

- In step 0.3.1 the attacker initiates an authentication on the SP with his identity on the Honest OP.
- In step 0.3.2 the attacker will be redirected to the Honest OP. Now, the attacker possess valid *state* and *nonce* values.

`Attack execution`

(1.) In the first step of the attack the victim clicks on a malicious link or visits an attacker controlled website. Unintentionally an identity managed by the Attacker OP is sent to the SP – *bob@attackerIdP.com*.

(2.) Optionally the SP retrieves in step 1.2 the metadata of the OP from the database or via Discovery and uses it during the protocol flow.

(3.) In step 2.1 the SP redirects the End-User to the authorization endpoint of the Attacker OP responsible for authentication.

(4.) In step 2.2 however the Attacker OP redirects the End-User to the Honest OP. Additionally, it replaces the *nonce* parameter. This manipulation is needed in order to successfully impersonate the victim on the SP. Please note that all steps till now does not require any interaction of the End-User and are transparent for him. Thus, he is not able to detect the attack.

(5.) In step 2.3 the End-User has to authenticate on the OP. In case that he is already authenticated, this step will be skipped. Only during this step it is possible to detect the attack. However, since the SP is already trusted on the Honest OP and the user is already authenticated, usually this step is transparent for the End-User.

(6.) The OP generates a valid *code* and returns it together with the *state* parameter back to the SP.

(7.) The SP still believes that it is communicating with the Attacker OP due to step 1.2. For this reason, it redeems the received *code* on the Attacker OP and additionally sends its client_id and client_secret.

(8.) As a result, the attacker now has a valid *code*, which can he now redeem through his browser on the SP.

`Token redemption`

- ▶ The attacker retrieves in step 4.1. the stolen *code* from his server (Attacker OP) and
- ▶ proceeds with his authentication on the SP by sending the stolen *code* together with the previously generated *state'*.
- ▶ Since the *state'* value is correct, the SP redeems the received *code* on the Honest OP in step 4.3 and
- ▶ receives the corresponding *ID Token* in step 4.4.

The received ID Token has the following information:

```
Header:  { "alg": "HS256" }
Body:    {
    "iss": "https://honestIdP.com/",
    "sub": "victim",
    "exp": 1444148908,
    "iat": 1444148308,
    "nonce": "{nonce'}",
    "aud": "{client_id}",
    }
Signature: AF45JF93LKD76D....
```

Listing 2.3: Received ID Token by the SP within the IdP Confusion attack

A correctly implemented SP will first verify the *iss* parameter, which is correct (the SP communicates with the Honest OP). The timestamps are freshly generated and still valid. The nonce parameter contains the value of *nonce'* and will be verified successfully. Since the *aud* contains the *client_id* and is valid too. Finally, the signature will be verified successfully since it is properly signed by the Honest OP.

Chapter 3

Implementation Flaws

This chapter enumerates several attack-scenarios corresponding to our security model (cf. section 1.1).

The following security issues are caused by implementation flaws, but exist in numerous OpenID Connect implementations. The OpenID Connect specification [4] states how to prevent these attacks, but lacks of details why these validation steps are necessary. We address this and give detailed descriptions on the impact of the attacks, and how to counter them by pointing to the OpenID Connect specification.

3.1 Client Flaws

This section addresses flaws that are caused due to improper implementations of OpenID Connect Clients (Service Providers). For OP flaws, the section 3.2

3.1.1 Replay Attacks

```
Header:  { "alg": "HS256" }
Body:    {
    "iss": "http://openidConnectProvider.com/",
    "sub": "user1",
    "exp": 1444148908,
    "iat": 1444148308,
    "nonce": "40c6b33b9a2e",
    "aud": "http://client.com/",
    }
Signature: AF45JF93LKD76D....
```

Listing 3.1: Relevant parameters within the ID Token preventing Replay Attacks.

Attack Scenario & Impact. Replay attacks allow to reuse an ID token in order to authenticate the attacker as a victim. As a prerequisite, the attacker needs to get in possession of an old token and submit it to the Client. One example of retrieving old tokens is by crawling support forums or using a web search engine like Google.

Attack Defense. Listing 3.1 highlights three parameters that are used to prevent replay attacks. The parameter `iat` (issued at) indicates the time on which the ID token is created. The parameter `exp` (expires) indicates the latest time on which the ID token is valid. A Client implementation must if the current time is after `iat` but before `exp`.

Attack Defense: Code Flow. The parameters `iat` and `exp` MUST be checked according to [4, Section 3.1.3.7, Steps 9-10]. The specification also advices to check the `nonce` parameter to prevent replay attacks [4, Section 3.1.3.7, Step 11], but lacks in giving details how to do this properly.

Attack Defense: Implicit Flow. The parameters `iat` and `exp` do not directly prevent replay attacks, since the token can be submitted multiple times during the valid time frame. To get real reply protection, the OpenID Connect specification offers the use of the `nonce` parameter [4, Section 3.2.2.11]. Unfortunately, the specification states checking the `nonce` as a *SHOULD* (although the parameter *MUST* be present [4, Section 3.2.2.10]).

Checking nonces is a non-trivial problem. A very short hint on implementing `nonce` validation is given in [4, Section 15.5.2].

Summary: Replay Attacks. Although the specification gives clear instructions to prevent replay attacks, checking nonces and token expiration parameters is still a problem due to our research in investigating OpenID Connect implementations. Detailed evaluation results will be provided soon.

3.1.2 Signature Manipulation

Signature Manipulation (SM) is an attack which targets the ID Token verification part of a Client. If the signature verification by a Client is not handled correctly, an attacker may be able to login 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 content of her ID Token $t_A = ID_A || \sigma_A$ to, for example, $t^* = ID_V || \sigma$. If 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 and if the Client accepts t^*, the attack is successful (and the attacker should be logged in with ID_V).

There are different possibilities to achieve this goal.

No Signature Validation at all. If the Client *does not* to validate the signature at all, the attacker can inject arbitrary content in the ID token. The signature validation is enforced in [4, Section 3.2.2.11, Step 1].

The "none" algorithm. It is possible to create a valid JWT token by setting the `alg` parameter in the JWT header to `none`. The algorithm is not allowed in the implicit flow [4, Section 2], but if the implementation just uses a `validateToken()` method without checking if the `none` algorithm is used (and allowed!), this leads to serious security problems – the attacker can impersonate any End-User on the Client.

From a developer perspective, using the same code for validating a token is very useful. However, in OpenID Connect, there must be a clear distinction between the implicit and the code flow.

3.1.3 Token Recipient Confusion

Each ID token is intended to be used for a specific Client. This is indicated by the `aud` (audience) parameters as shown in Listing 3.2. The `aud` parameter contains the `client_id` of the recipient Client and must be checked according to [4, Section 3.1.3.7, Step 3].

```
Header:  { "alg": "HS256" }
```

```
Body:    {
    "iss": "http://openidConnectProvider.com/",
    "sub": "user1",
    "exp": 1444148908,
    "iat": 1444148308,
    "nonce": "40c6b33b9a2e",
    "aud": "theClientId",
    }
Signature: AF45JF93LKD76D....
```

Listing 3.2: Relevant parameters within the ID Token preventing Token Recipient Confusion.

Attack Scenario & Impact. If this check is missing, an attacker can reuse tokens that are intended to be used on a different Client. He can set up his own malicious Client, for example, a harmless weather forecast service and lure the victim to login. The attacker then receives the ID token that is intended to be used on that weather forecast service, but reuse it on a different Client. As a result, the attacker will get access on the targeted Client in context of the victim.

3.1.4 ID Spoofing

ID Spoofing in the ID Token

ID Spoofing (IDS) is an attack which targets the ID Token verification part of a Client. If the `issuer` claim verification by a Client is not handled correctly, an attacker is able to login as an arbitrary End-User of this application.

Attack Scenario & Impact. To perform an IDS attack an attacker has to act as an End-User and an OP simultaneously. The attacker's OP however, issues tokens in the name of other Honest OP like Google.

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 Attacker OP OP_A to send exactly t^* to a Client with which her victim is registered. If the Client accepts t^* the attack is successful (and the attacker should be logged in with ID_V).

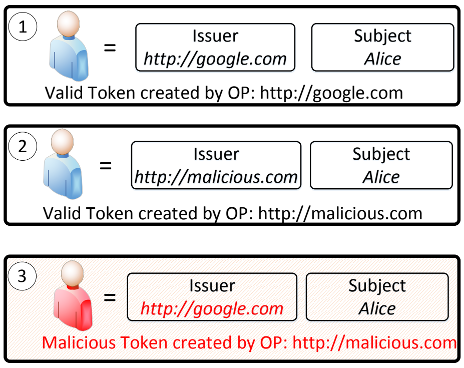

1
=
Issuer
http://google.com
Subject
Alice
Valid Token created by OP: http://google.com
2
=
Issuer
http://malicious.com
Subject
Alice
Valid Token created by OP: http://malicious.com
3
=
Issuer
http://google.com
Subject
Alice
Malicious Token created by OP: http://malicious.com

Figure 3.1: ID Spoofing: (1) Valid Token created by an Honest OP *http://google.com* about the End-User *Alice*. (2) Valid Token created by the Attacker OP about the End-User *Alice*. Please note that both tokens represent different End-User identities. (3) Malicious ID Token issued by the Attacker OP containing the End-User identity controlled by the *Honest OP*.

Attack Defense. Listing 3.1 highlights two parameters that are used to prevent the attack. The Client MUST verify that the URL specified by the `iss` parameter in ID Token is equal with the URL of the corresponding OP controlling the identity of the End-User. For instance, the `iss` parameter in the ID Token MUST exactly match the `issuer` discovered during the Discovery phase.

```
Header:  { "alg": "HS256" }
Body:    {
    "iss": "http://openidConnectProvider.com/",
    "sub": "user1",
    "exp": 1444148908,
    "iat": 1444148308,
    "nonce": "40c6b33b9a2e",
    "aud": "http://client.com/",
    }
Signature: AF45JF93LKD76D....
```

Listing 3.3: Relevant parameters within the ID Token preventing ID Spoofing.

The specification states that “The Issuer Identifier for the OpenID Provider (which is typically obtained during Discovery) MUST exactly match the value of the iss (issuer) Claim." However is unclear how this verification has to be done in case that no Discovery is provided.

ID Spoofing in the ID Token with E-Mail

```
Header:  { "alg": "HS256" }
Body:    {
    "iss": "http://openidConnectProvider.com/",
    "sub": "user1",
    "exp": 1444148908,
    "iat": 1444148308,
    "nonce": "40c6b33b9a2e",
    "aud": "http://client.com/",

    "name": "Jane Doe",
    "given_name": "Jane",
    "family_name": "Doe",
    "gender": "female",
    "birthdate": "0000-10-31",
    "email": "janedoe@google.com",
    }
Signature: AF45JF93LKD76D....
```

Listing 3.4: An ID Token containing further information (sub-claims) about the authenticated user.

The OpenID Connect specification states that the combination of `sub` and `iss` are the only claims that the Client can rely upon as a stable identifier [4, Section 5.7]. It is however possible to add additional info into the ID token. One of this is adding an `email` address.

It is of essential importance that the Client *MUST NOT* use the `email` parameter to identify the End-user, because the Client does not know if the *email* address contained in the token is under the control of the End-User.

Attack Scenario & Impact. To perform an IDS attack an attacker has to act as an End-User and an simultaneously. The Attacker OP however, issues tokens containing the email address of the victim in order to log in his/her account on the Client.

Attack Defense. It must be stressed, that the `email` parameter *MUST NOT* for anything related to identify the End-user. This is contra-intuitive, since most websites use an email address to identify the End-user. This is not possible in OpenID Connect.

The same problem appeared, for example, in OpenID[1] – it must be taken seriously and developers must be warned not to do the same in OpenID Connect.

Issuer Confusion

Issuer Confusion (IC) is an attack which targets the Discovery phase of the protocol and bypasses the verification check specified in [4, Section 3.1.3.7.] – "The Issuer Identifier for the OpenID Provider (which is typically obtained during Discovery) MUST exactly match the value of the iss (issuer) Claim.".

Attack Scenario & Impact. To perform an IDS attack an attacker has to act as an End-User and an OP simultaneously. The Attacker OP however, issues tokens containing the email address of the victim in order to log in his/her account on the Client.

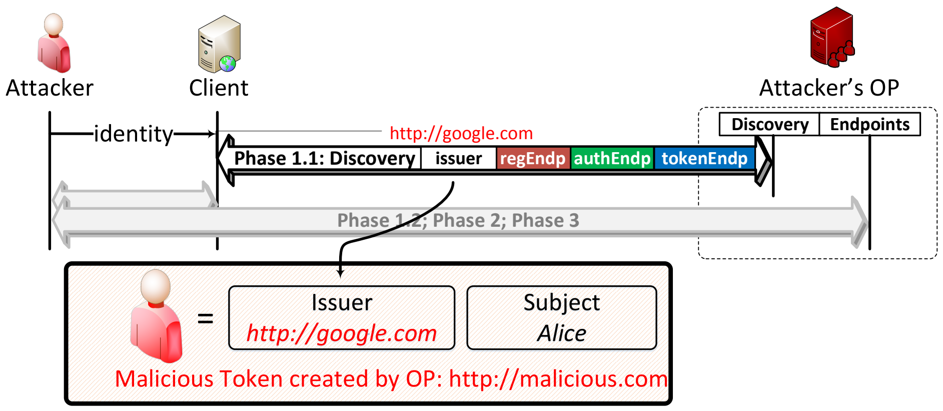


Figure 3.2: Issuer Confusion: The Attacker OP sets the *issuer* parameter to an Honest OP during the Discovery phase. As a result the Client can be confused that the Attacker OP controls the identities of the Honest OP.

If the `issuer` claim verification by a Client is not handled correctly, an attacker may be able to login 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 Attacker OP OP_A to send exactly cdr^* to a Client with which her victim is registered. If the Client accepts cdr^* and later on compares the `issuer` claim of the ID Token (which also contains iss_V) to $issuer_V$ from cdr^* the attack is successful (and the attacker should be logged in with ID_V).

Attack Defense. [5, Section 3] clearly states that "If Issuer discovery is supported (see Section 2), this value MUST be identical to the issuer value returned by WebFinger. This also MUST be identical to the iss Claim value in ID Tokens issued from this Issuer."

[1]`http://wiki.oxidforge.org/index.php?title=Security_bulletins/2015-001`

3.1.5 Key Confusion

KC introduces a class of attacks forcing the Client to use a key of the attacker's choice for the verification of tokens. The enforced key is a legit key that is shared between the target Client and the Attacker OP. However, during the KC attack, the Client is convinced to believe, that the key belongs to the Honest OP (instead of the Attacker OP) .

To execute KC, the attacker may follow one of two strategies to succeed.

Key Confusion with wrong references

Since the ID Token is a JWT, it can contain in the Header a reference to the key material used for verification. This information can be stored in one of the following fields: x5u, x5c, jku, or jwk. However, the Client must verify that the referenced key belongs to the corresponding OP and is trusted.

Attack Scenario & Impact. In case that the Client does not verify the trustiness of the key used to sign ID Token, the attacker can manipulate it within the implicit flow and inject wrong identities. This attack results in broken End-User authentication.

Attack Defense. The specification states clearly that "ID Tokens SHOULD NOT use the JWS or JWE x5u, x5c, jku, or jwk Header Parameter fields. Instead, references to keys used are communicated in advance using Discovery and Registration parameters, per Section 10."

Key Confusion with Session Overwriting

During this attack, the Client is convinced to believe, that the ID Token is issued by the Honest OP, but it is issued by the Attacker OP and signed with its key.

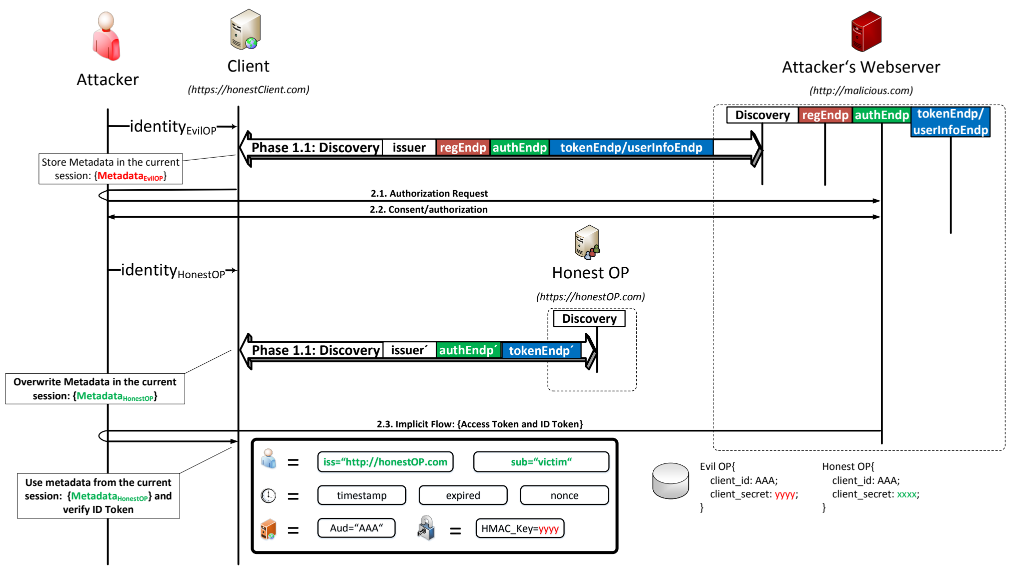

Attacker
Client
(https://honestClient.com)
Attacker's Webserver
(http://malicious.com)
Discovery
regEndp
authEndp
tokenEndp/ userInfoEndp
identity$_{EvilOP}$
Phase 1.1: Discovery
issuer
regEndp
authEndp
tokenEndp/userInfoEndp
Store Metadata in the current session: {Metadata$_{EvilOP}$}
2.1. Authorization Request
2.2. Consent/authorization
identity$_{HonestOP}$
Honest OP
(https://honestOP.com)
Discovery
Phase 1.1: Discovery
issuer´
authEndp´
tokenEndp´
Overwrite Metadata in the current session: {Metadata$_{HonestOP}$}
2.3. Implicit Flow: {Access Token and ID Token}
Use metadata from the current session: {Metadata$_{HonestOP}$} and verify ID Token
=
iss="http://honestOP.com
sub="victim"
timestamp
expired
nonce
Aud="AAA"
HMAC_Key=yyyy
Evil OP{ client_id: AAA; client_secret: yyyy; }
Honest OP{ client_id: AAA; client_secret: xxxx; }

Figure 3.3: Key Confusion with Session overwriting

Attack Scenario & Impact. To perform the attack an attacker has to act as an End-User and an simultaneously. The attacker's however, issues tokens in the name of other honest like Google.

In the following, we describe the attack protocol flow, which we depicted in Figure 3.3.

Authentication on the Attacker OP The attacker's intention in the first phase is to force a valid Client to use the Attacker OP. The attacker enters his identity *alice@malicious.com* and enforces the Client to Discover the Attacker OP, if needed to register on it, and redirects the End-User in order to authenticate. Please note that the End-User is the attacker using his browser.

During the Discovery phase the Client determines the corresponding endpoints and stores them in the current session – $Metadata_{EvilOP}$.

End-User Authentication and Authorization In the next phase, the Client redirects the End-User to the Attacker OP, where the End-User has to authenticate himself and authorize the Client. Now, the attacker waits with the next step – the redirect back to the Client together with the ID Token and Access Token.

Overwrite the Metadata stored in the Session The attacker's intention is to force the Client to load the metadata of the honest , by entering an identity controlled by the Honest OP. As a result, the Honest OP will be discovered and its metadata will be stored in the current session – $Metadata_{Honest}$. Now, the Client believes to communicate with the Honest OP.

Broken End-User Authentication Now the Attacker OP generates the ID Token and sends it back to the Client via an HTTP redirect- Consequentially, the Client verifies the ID Token according the specification.

- (✓) The Issuer Identifier for the OpenID Provider (which is typically obtained during Discovery) MUST exactly match the value of the iss (issuer) Claim.
- (✓) The Client MUST validate that the aud (audience) Claim contains its client_id value registered at the Issuer identified by the iss (issuer) Claim as an audience.
- (✓) ... further checks like timestamps, nonces etc.
- (?!?)If the JWT alg Header Parameter uses a MAC based algorithm such as HS256, HS384, or HS512, the octets of the UTF-8 representation of the client_secret corresponding to the client_id contained in the aud (audience) Claim are used as the key to validate the signature.

According to the specification the Client should use its client_id to find the corresponding key. Unfortunately the Client has two identical client_ids on both OPs, see Figure 3.3. Now, it is unclear which key will be used for verification. In case that the client uses the key "yyyy" the attack is successful an the attacker can impersonate any End-User on the Client.

Attack Defense. The specification should clearly state how the correct verification key has to be fetched – via the client_id **and** the corresponding issuer. We propose a minimal change of the validation check in order to address the attack:" If the JWT alg Header Parameter uses a MAC based algorithm such as HS256, HS384, or HS512, the octets of the UTF-8 representation of the **`client_secret`** corresponding to the **`client_id`** and **`issuer`** contained in the ID Token are used as the key to validate the signature."

3.1.6 Subclaim Spoofing within the Access Token

According to the specification "... the UserInfo Response is not guaranteed to be about the End-User identified by the sub (subject) element of the ID Token. The sub Claim in the UserInfo Response MUST be verified to exactly match the sub Claim in the ID Token; if they do not match, the UserInfo Response values MUST NOT be used." [4, Section 16.11].

3.2 Identity Provider Flaws

3.2.1 Sub Claim Spoofing

Sub Claim Spoofing (SCS) is an attack which targets the Authentication Request verification part of an OP. If the `subject` claim verification by an OP is not handled correctly, an attacker may be able to login as any End-User registered with this OP at a Client of her choice: Within the Authentication Request of OpenID Connect, the Client has the possibility to request individual claims about the End-User either by using the `claims` parameter or by using an additional JSON Web Token (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 3.5.

```
{
  "userinfo":
  {
    "given_name": {"essential": true},
    "nickname": null,
    "email": {"essential": true},
    "email_verified": {"essential": true},
    "picture": null
  },
  "id_token":
  {
    "auth_time": {"essential": true},
    "acr": {"values": ["urn:mace:incommon:iap:silver"]}
  }
}
```

Listing 3.5: Non-Normative Example Claims Request [4]

Within our example the Client 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.

Attack Scenario & Impact. To perform the attack an attacker has to act as an End-User manipulating the Authentication Request. Since the Authentication Request is sent trough the attacker's browser, it can be modified.

As in the example (for the `acr` claim) a Client has the possibility to request that an individual claim is returned with a particular value. Thus, the Client can make statements, which has to be verified by the OP. For instance the decoded `claims` parameter value can be used to specify that the request applies to the End-User with Subject Identifier *"subOfTheVictim"*.

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 `subject` 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 `subject` 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^*, see Listing 3.6.

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

Listing 3.6: Sub Claim Request

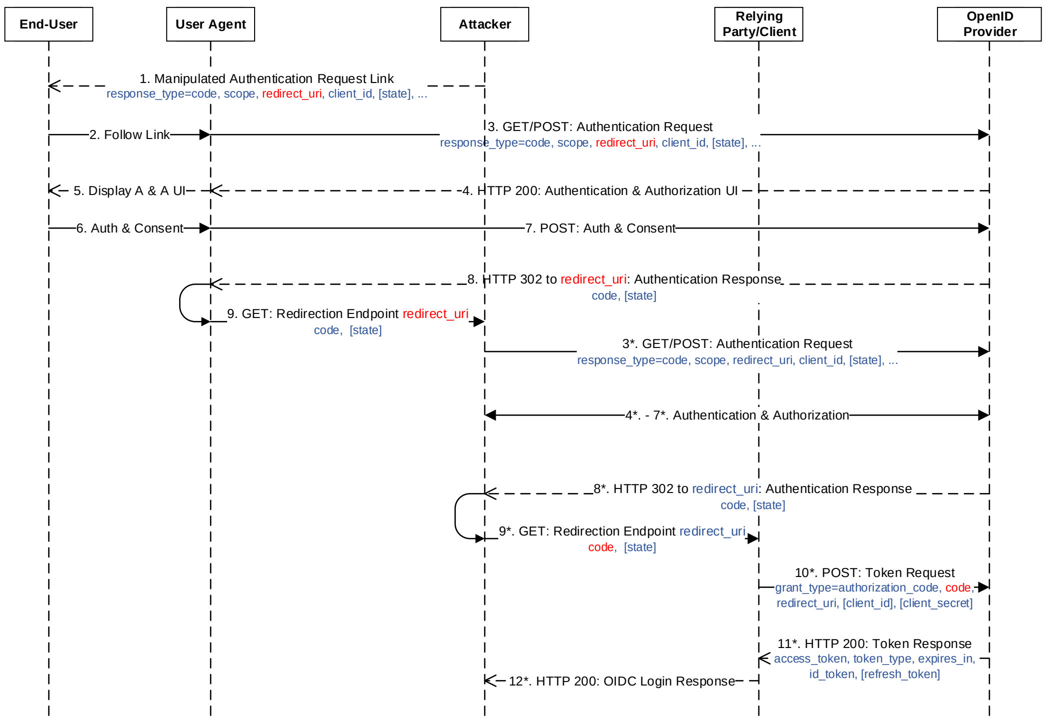
If the subsequently issues an ID Token t^* containing ID_V, although the attacker did not authenticate with the credentials resulting in sub_V, the attack is successful (and the attacker should be logged in with ID_V).

Attack Defense. Statements made within the Authentication Request MUST be considered untrusted and MUST be verified. Unverified statements MUST NOT be included in the ID Token and Access Token issued by the OP.

3.2.2 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 [4, Section 3.1.2.1.] by an OP is not handled correctly, an attacker may be able to login as any End-User registered with this OP at a Client of her choice. RUM is applicable to Authentication using the Authorization Code Flow and Authentication using the Implicit Flow.

Attack Scenario & Impact. Within this attack-scenario the Authentication Response of a victim is redirected to a website controlled by the attacker. The thereby obtained authorization code or the ID Token is then used within a separate protocol flow, initiated by the attacker, to redeem it for an ID Token of the victim. Figure 3.4 depicts the attack-procedure of the attacker using the Authorization Code Flow (all manipulated request parameters are highlighted in red).



End-User
User Agent
Attacker
Relying Party/Client
OpenID Provider
1. Manipulated Authentication Request Link
response_type=code, scope, redirect_uri, client_id, [state], ...
2. Follow Link
3. GET/POST: Authentication Request
response_type=code, scope, redirect_uri, client_id, [state], ...
5. Display A & A UI
4. HTTP 200: Authentication & Authorization UI
6. Auth & Consent
7. POST: Auth & Consent
8. HTTP 302 to redirect_uri: Authentication Response
code, [state]
9. GET: Redirection Endpoint redirect_uri
code, [state]
3*. GET/POST: Authentication Request
response_type=code, scope, redirect_uri, client_id, [state], ...
4*. - 7*. Authentication & Authorization
8*. HTTP 302 to redirect_uri: Authentication Response
code, [state]
9*. GET: Redirection Endpoint redirect_uri
code, [state]
10*. POST: Token Request
grant_type=authorization_code, code,
redirect_uri, [client_id], [client_secret]
11*. HTTP 200: Token Response
access_token, token_type, expires_in,
id_token, [refresh_token]
12*. HTTP 200: OIDC Login Response

Figure 3.4: 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 Con-

nect. 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.

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.

Attack Defense. It is essential that the OP provides the following verification step: “This URI MUST exactly match one of the Redirection URI values for the Client pre-registered at the OpenID Provider” [4, Section 3.1.2.1.].

Bibliography

[1] Dmitry Chastukhin Alexander Polyakov. Ssrf vs. business-critical applications: Xxe tunneling in sap. BlackHat, 2012. URL `https://media.blackhat.com/bh-us-12/Briefings/Polyakov/BH_US_12_Polyakov_SSRF_Business_Slides.pdf`. 3

[2] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. *Communications of the ACM - One Laptop Per Child: Vision vs. Reality*, 52:83–91, June 2009. URL `http://seclab.stanford.edu/websec/frames/post-message.pdf`. 1.1.3

[3] ONsec Lab. Ssrf bible. cheatsheet, August 2014. URL `https://docs.google.com/document/d/1v1TkWZtrhzRLy0bYXBcdLUedXGb9njTNIJXa3u9akHM/edit?pli=1`. 3

[4] The OpenID Foundation (OIDF). OpenID Connect Core 1.0, February 2014. URL `http://openid.net/specs/openid-connect-core-1_0.html`. 3, 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.4, 3.1.6, 16, 3.2.2, 3.2.2

[5] The OpenID Foundation (OIDF). OpenID Connect Discovery 1.0, February 2014. URL `http://openid.net/specs/openid-connect-discovery-1_0.html`. 3.1.4